علوم الحاسوب

من الألف إلى الياء

تأليف

Ian Wienand

ترجمة

علا عباس

زينب الزعيم

أكاديمية حسوب

علوم الحاسوب من الألف إلى الياء

مرجع شامل لتعلم علوم الحاسوب

اسم الكتاب: علوم الحاسوب من الألف إلى الياء

Book Title: Computer Science from the Bottom Up

المؤلف: إيان ويناند

Author: Ian Wienand

المترجم: علا عباس - زينب الزعيم

Translator: Ola Abbas - Zainab Al Zaeem

المحرر: آيات اليطقان - جميل بيلوني

Editor: Ayat Alyatakan - Jamil Bailony

تصميم الغلاف: أحمد عمارة

Cover Design: Ahmed Emara

سنة النشر: 2024

Publication Year: 2024

رقم الإصدار: 1.0

Edition: 1.0

https://academy.hsoub.com

academy@hsoub.com

عن الناشـر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

تهدف أكاديمية حسوب إلى تعليم البرمجة باللغة العربية وإثراء المحتوى البرمجي العـربي عـبر توفـير دورات برمجة وكتب ودروس عالية الجودة من متخصصين في مجال البرمجة والمجالات التقنية الأخـرى، بالإضـافة إلى توفير قسم للأسئلة والأجوبة للإجابة على أي سؤال يواجه المتعلم خلال رحلته التعليمية لتكون معه وتؤهلـه حـتى دخول سوق العمل.

حسوب شركة تقنية في مهمة لتطوير العالم العـربي. تبـني حسـوب منتجـات تركِّـز على تحسـين مسـتقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العـالم العـربي، مسـتقل وخمسـات ويعمـل فيها فريق شاب وشغوف من مختلف الدول العربية.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

المحتويات باختصار

جدول المحتويات

فهرس الأشكال

فهرس الجداول

دورة تطوير التطبيقات
باستخدام لغة بايثون

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

تمهيد

يوفر كتاب (علوم الحاسوب من الألف إلى الياء) معلومات شاملة حول علوم الحاسوب، ويشرح المواضيع الأساسية لفهم آلية عمل عتاد الحاسوب ونظام تشغيله بأسلوب تصاعدي يبدأ من شرح التفاصيل ذات المستوى المنخفض، ثم ينتقل تدريجيًا إلى مفاهيم أكثر تقدمًا كي يساعدك على فهمها بسهولة أكبر.

حيث يبدأ الكتاب بشرح المفاهيم الأساسية التي تبنى عليها أجهزة الحاسوب مثل طريقة تمثيل البيانات باستخدام النظام الثنائي والست عشري ويشرح أهم العمليات البوليانية التي تنفذ عليها، ثم يتعمق في الفصول اللاحقة في موضوعات ومفاهيم متقدمة كشرح الذاكرة الوهمية virtual memory وآلية عملها وطريقة عمل أنظمة التشغيل وتنظيمها وطريقة إدارتها لعتاد الحاسوب وبرامجه.

كما يشرح الكتاب العديد من المواضيع التي تهم المبرمجين ويوضح طريقة عمل سلسلة الأدوات Toolchain التي تتعامل مع البرامج الحاسوبية، وأهم الاختلافات بين اللغات المُصرَّفة compiled واللغات المُفسَّرة interpreted إلى جانب توضيح مجموعة واسعة من المفاهيم الأخرى المتعلقة بعلوم الحاسوب والمفيدة للمبتدئين والمحترفين على حد سواء.

لا تحتاج إلى أن تكون مبرمجًا خبيرًا لفهم المواضيع الواردة في هذا الكتاب، لكنك تحتاج لامتلاك معرفة أساسية بأجهزة الحاسوب ومكوناته ومفهوم نظام التشغيل Operating System ومعرفة بأساسيات البرمجة.

حول الكتاب

هذا الكتاب هو ترجمة لكتاب Computer Science from the Bottom Up لكاتبه إيان ويناند Ian Wienand ويوضح كافة المفاهيم التي يحتاج القارئ لمعرفتها حول عتاد وبرمجيات الحاسوب ونظام تشغيله وطريقة عمله بالتفصيل من المستوى المبتدئ للمتقدم.

يوفر الفصل الأول عن نظرة متقدمة على نظام التشغيل يونكس ولغة C حيث ويوضح مفهوم "كل شيء عبارة عن ملف" الذي يعني أن كل شيء في نظام يونكس يعد ملفًا بما في ذلك الأجهزة والبرامج ودور هذا المفهوم في تسهيل إدارة النظام وتطوير البرامج، ويشرح كذلك مفهوم التجريد abstraction في لغة C وكيفية استخدام القوالب templates لإنشاء تجريد عام لأنواع البيانات أو الوظائف وكتابة كود أكثر مرونة، ويقدم لمفهوم المكتبات ومفهوم واصفات الملفات file descriptors واستخدام واصف الملف لفتح ملف ما وقراءة البيانات منه، وأخيرًا يشرح بالأمثلة العملية طريقة استخدام صدفة يونكس.

ينتقل الفصل الثاني لشرح طريقة تمثيل البيانات في الحاسوب من خلال نظام العد الثنائي ويوضح مفاهيم البتات والبايتات والتكافؤ وأنظمة البت المختلفة والعمليات البولينية مثل NOT و AND و OR و XOR. كما يتناول النظام الست عشري ويشرح طريقة التحويل بين الأنظمة العددية واستخدامها في الشيفرات البرمجية، كما يناقش طريقة تمثيل الأعداد مثل الأعداد العشرية والسالبة بهذه الأنظمة.

أما الفصل الفصل الثالث فيتناول معمارية الحاسوب الداخلية، ويوضح وظيفة وحدة المعالجة المركزية CPU والعمليات الأساسية التي تقوم بها وأنواع معماريات وحدة المعالجة المركزية مثل CISC و RISC و EPIC، ثم يشرح آلية عمل ذاكرة الحاسوب وتسلسل الذواكر الهرمي والذاكرة المخبئية وطريقة عنونتها، وأخيرًا يشرح مفاهيم منوعة ترتبط بالأجهزة الطرفية وأنظمة المعالجات.

وينتقل الفصل الرابع لشرح آلية عمل نظام التشغيل ودوره في الحاسوب وتنظيمه الذي يشمل نواة نظام التشغيل ومجالات المستخدم والوحدات والافتراضية، كما يشرح استدعاءات النظام وهي واجهات يوفرها نظام التشغيل للبرامج لتفاعلها مع العتاد، ويشرح طريقة إدارة الصلاحيات في نظام التشغيل باستخدام الأمثلة حيث يوضح على سبيل المثال كيف يستطيع نظام التشغيل منع برنامج ما من الوصول إلى بيانات برنامج آخر.

ويتوسع الفصل الخامس في شرح مفهوم العمليات ودورها في تمكين نظام التشغيل من تشغيل عدة برامج في نفس الوقت ويوضح عناصر العملية وتسلسل العمليات الهرمي وكيفية ارتباط العمليات ببعضها البعض، ويناقش بعد ذلك استدعاءات النظام `fork` و `exec` المستخدمة لإنشاء عمليات جديدة وتنفيذ ملفات جديدة كما يوضح مفهوم الجدولة Scheduling التي تمكن نظام التشغيل من تحديد ما هي العملية التي ستنفذ في وقت معين.

ويتناول الفصل السادس طريقة عمل الذاكرة الوهمية التي هي تقنية تسمح للبرامج بالوصول إلى مساحة من الذاكرة أوسع من المساحة المتاحة فعليًا عن طريق تقسيم الذاكرة الفعلية إلى صفحات بحجم ثابت وتخزين هذه الصفحات في الذاكرة الفعلية أو على القرص الصلب، ويناقش بعض المفاهيم الأخرى المتعلقة بالذاكرة الوهمية مثل فضاءات العناوين والحماية والتبديل ومشاركة الذاكرة والذاكرة المخبئية للقرص الصلب ودعم العتاد للذاكرة الوهمية.

ويتطرق الفصل السابع لشرح مفهوم سلسلة الأدوات Toolchain، وهي مجموعة من البرامج التي تعمل معًا لتحويل شيفرة المصدر إلى برنامج قابل للتنفيذ ويعرفك على نوعين رئيسيين من البرامج في سلسلة

الأدوات هما البرامج المُصَرَّفة compiled programs والبرامج المُفَسَّرة interpreted programs، كما ستتعرف على مفهوم الآلات الافتراضية virtual machines وهي نوع من البرامج يمكنها تشغيل تطبيقات مكتوبة بلغات مختلفة وتتعرف على الخطوات الأساسية لبناء ملف قابل للتنفيذ وهي التصريف، والتجميع، والربط وآلية تحويل شيفرة مصدرية بلغة C إلى برنامج تنفيذي.

ويتوسع الفصل الثامن في شرح طريقة تمثيل الملفات القابلة للتنفيذ والصيغ المختلفة لهذه الملفات وأبرزها ملفات ELF ويعرفك على مفهوم واجهات ABI وأنواعها، كما يناقش مفهوم المكتبات وأنواعها ويوضح الفرق بين المكتبات الساكنة والمكتبات المشتركة.

وأخيرًا يشرح الفصل التاسع مفهوم الربط الديناميكي وهو تقنية تسمح بتحميل المكتبات المشتركة ديناميكيًا عند تشغيل البرامج ويبين فوائده ومميزاته كما يناقش أيضًا بعض المفاهيم المتقدمة المتعلقة بالربط الديناميكي مثل الانتقالات وجدول الإزاحة العام وجدول البحث عن الإجراءات ودورها في تسهيل مشاركة الشيفرة وكتابة برامج أكثر فعالية وكفاءة.

عند انتهائك من فصول هذا الكتاب ستكون قادرًا على فهم كيفية عمل الحاسوب من المستوى المبتدئ إلى المستوى المتقدم وتفهم بتفصيل أكبر كيفية عمل نظام التشغيل و إدارة الذاكرة وطريقة إنشاء البرامج، وكيفية بدء العمليات وستكون قادرًا على فهم معمارية الحاسوب والتعامل معه بكفاءة أكبر.

أضفنا المصطلحات الأجنبية بجانب المصطلحات العربية لسببين، أولهما التعرف على المصطلحات العربية المقابلة للمصطلحات الأجنبية الأكثر شيوعًا وعدم الخلط بين أي منها، وثانيًا تأهيلك للاطلاع على المراجع فتصبح محيطًا بعد قراءة الكتاب بالمصطلحات الأجنبية التي تخص أنظمة التشغيل ومعمارية الحاسوب وبذلك يمكنك قراءتها وفهمها وربطها بسهولة مع المصطلحات العربية المقابلة والبحث عنها والتوسع فيها إن شئت وأيضًا يسهل عليك قراءة الشيفرات وفهمها. عمومًا، نذكر المصطلح الأجنبي بجانب العربي في أول ذكر له ثم نكمل بالمصطلح العربي، فإذا انتقلت إلى قراءة فصول محددة من الكتاب دون تسلسل، فتذكر إن مررت على أي مصطلح عربي أننا ذكرنا المصطلح الأجنبي المقابل له في موضع سابق.

استخدام الشيفرة

شيفرة أمثلة هذا الكتاب متاحة في المستودع https://github.com/ianw/bottomupcs حيث أن Git هو نظام تحكم بالإصدارات يسمح لك بتتبع الملفات التي يتكون منها المشروع، وتسمى مجموعة الملفات التي يتحكم بها Git بالمستودع repository، وGitHub هي خدمة استضافة توفر تخزينًا لمستودعات Git وواجهة ويب ملائمة انظر فيديو "أساسيات Git" وقسم Git في أكاديمية حسوب لمزيد من التفاصيل.

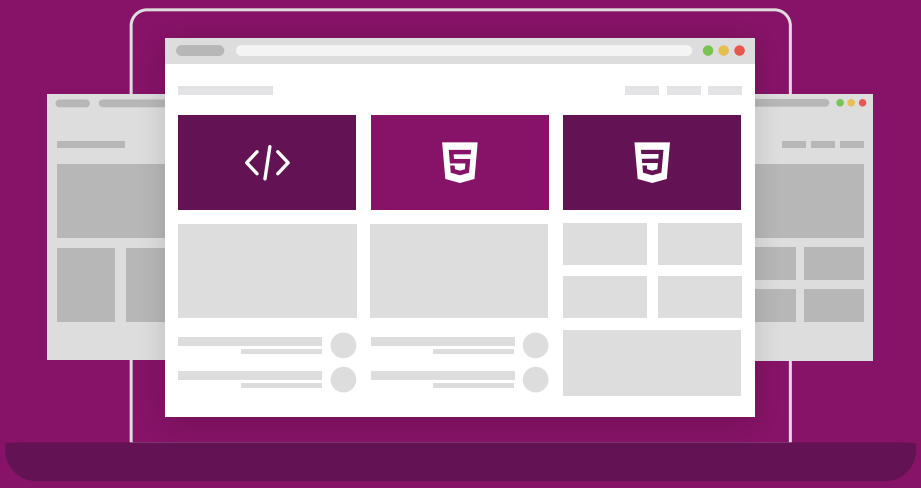
توفّر صفحة GitHub الرئيسية لمستودع شيفرات الكتاب عدة طرق للعمل معها هي:

- يمكنك إنشاء نسخة من المستودع على Fork بالضغط على زر Fork. إذا لم يكن لديك حساب GitHub، فستحتاج إلى إنشاء حساب، ثم سيصبح لديك مستودعك الخاص على GitHub بعد ضغطك على زر Fork والذي يمكنك استخدامه لتتبع التعليمات البرمجية التي تكتبها أثناء العمل على هذا الكتاب، ثم يمكنك استنساخ clone المستودع repository أو اختصارًا repo، مما يعني أنك تنسخ الملفات إلى جهاز الحاسوب الخاص بك.
- أو يمكنك استنساخ المستودع فلا تحتاج إلى حساب GitHub للقيام بذلك، لكنك لن تتمكن من كتابة تغييراتك مرة أخرى على GitHub.
- إذا كنت لا تريد استخدام Git على الإطلاق، فيمكنك تنزيل الملفات في ملف مضغوط zip باستخدام الزر الموجود في الزاوية اليمنى السفلية من صفحة GitHub.

المساهمة

يرجى إرسال بريد إلكتروني إلى academy@hsoub.com إذا كان لديك اقتراح أو تصحيح على النسخة العربية من الكتاب أو أي ملاحظة حول أي مصطلح من المصطلحات المستعملة. إذا ضمَّنتَ جزءًا من الجملة التي يظهر الخطأ فيها على الأقل، فهذا يسهِّل علينا البحث، وتُعد إضافة أرقام الصفحات والأقسام جيدة أيضًا.

دورة تطوير واجهات المستخدم



ابدأ مسارك المهني كمطور واجهات المواقع والمتاجر الإلكترونية
فور انتهائك من الدورة

التحق بالدورة الآن

1. نظرة متقدمة على يونكس ولغة سي

التجريد abstraction هو المفهوم الرئيسي الذي تقوم على أساسه جميع أنظمة تشغيل الحواسيب الحديثة ومفهوم الملف هو تجريد مناسب إما كحوض للبيانات أو مصدر لها، وبالتالي هـو تجريـد ممتـاز لجميـع الأجهـزة التي قد يوصلها المرء بالحاسوب. هذا الإدراك هو سر القوة العظيمة لنظام التشـغيل يـونكس Unix ويتجلى في مجمَل تصميم كامل المنصة. ويُعَدّ توفير تجريد الأجهزة هذا للمبرمج من الأدوار الرئيسية لنظام التشغيل.

1.1 كل شيء عبارة عن ملف

تُعَدّ مقولة كل شيء عبارة عن ملف مبدأً يُستمَد غالبًـا من أنظمـة يـونكس Unix ومـا يشـابهها مثـل نظـام لينكس Linux وبي إس دي BSD.

لنتخيل ملفًا في إطار مألوف مثل معالج النصوص، إذ تكون العمليتان الأساسيتان اللتان نستطيع تنفيذهما على ملف في معالج النصوص التخيلي هذا كما يلي:

1. قراءته، أي قراءة معالج النصوص البيانات الحالية المحفوظة.
2. الكتابة ضمنه، أي كتابة المستخدِم بيانات جديدةً.

لنستعرض بعض الطرفيات الشائعة الموصولة بالحاسوب، وما ارتباطها بالعمليات الأساسية على الملفات:

1. الشاشة.
2. لوحة المفاتيح.
3. الطابعة.
4. القرص المدمَج CD-ROM.

تشبه كل من الشاشة والطابعة ملفًا للكتابة فقط، إذ تُعرَض المعلومات بشكل نقاط على الشاشة أو خطوط على الصفحة بدلًا من تخزينها على هيئة بِتّات على القرص؛ أما لوحة المفاتيح، فتُعَدّ مثل ملف للقراءة فقط، إذ ترد البيانات من ضغطات المستخدِم على المفاتيح، وكذلك الأمر بالنسبة للقرص المضغوط CD-ROM مثلًا، لكن تخزَّن البيانات مباشرةً على القرص بدلًا من أن يدخلها المستخدِم عشوائيًا.

وبالتالي فإن مفهوم الملف هو تجريد abstraction مناسب إما لحوض البيانات أو مصدرها، لذا فهو تجريد ممتاز لجميع الأجهزة التي قد يوصلها المرء بالحاسوب، ويُعَدّ هذا الإدراك هو سر القوة العظيمة لنظام التشغيل يونكس ويتجلى في مجمَل تصميم كامل المنصة، كما يُعَدّ توفير تجريد الأجهزة هذا للمبرمج من الأدوار الرئيسية لنظام التشغيل.

ربما لا نبالغ عندما نقول أنّ التجريد هو المفهوم الأساسي الذي يدعم جميع أشكال الحوسبة الحديثة، حيث لا يمكن لشخص واحد فهم كل الأمور من تصميم واجهة مستخدِم حديثةً إلى العمليات الداخلية لوحدة المعالجة المركزية CPU الحديثة، ناهيك عن بنائها بكاملها بأنفسهم؛ أما بالنسبة للمبرمجين، فالتجريد هو اللغة المشتركة التي تتيح لنا التعاون والابتكار.

يمنحنا تعلّم التنقل بين التجريدات رؤيةً أعمق لطريقة استخدام التجريدات بأفضل الأساليب وأكثرها ابتكارًا، وسندرس في هذا الكتاب التجريدات في الطبقات الدنيا وبين التطبيقات ونظام التشغيل وبين نظام التشغيل والعتاد الصلب، كما توجد العديد من الطبقات الأعلى منها وكل منها تستحق التفرُّد بكتاب خاص بها، ونأمل منك اكتساب بعض الرؤى عن التجريدات التي يقدِّمها نظام التشغيل الحديث مع دراسة كل فصل من فصول هذا الكتاب.

شكل 1: صورة توضح مفهوم التجريد

1.2 تطبيق التجريد

يُطبَّق التجريد عمومًا بما يسمى واجهة برمجة التطبيق API، ويُعَدّ API مصطلحًا مبهمًا نوعًا ما، إذ يشير إلى أمور مختلفة حسب سياقات الأعمال البرمجية المتنوعة، يصمم المبرمج في الأساس مجموعة دوال functions، ويُوثِّق واجهتها ووظيفتها حسب مبدأ أن التنفيذ الفعلي الذي يزوده بواجهة API يكون مخفيًا.

تقدِّم العديد من تطبيقات الويب على سبيل المثال واجهة API يمكن الوصول إليها عن طريق بروتوكول HTTP، ويطلق الوصول إلى البيانات بهذه الطريقة عدة سلاسل معقدة من استدعاءات الإجراءات البعيدة remote procedure calls واستعلامات قاعدة البيانات database queries وعمليات نقل البيانات data transfers، وتكون جميعها غير مرئية بالنسبة للمستخدِم النهائي الذي يتلقى البيانات المقتضبة ببساطة.

سيألف الذين هم على دراية باللغات البرمجية كائنية التوجه object-oriented مثل جافا Java أو بايثون Python أو C++ مفهوم التجريد في الأصناف classes، إذ تزوِّد التوابع methods الصنف بالواجهة لكنها تجرِّد التنفيذ.

1.2.1 تطبيق التجريد بلغة البرمجة C

تُعَدّ مؤشرات الدالة function pointers منهجيةً شائعةً تُستخدَم في نواة نظام تشغيل لينكس وغيرها من الشيفرات البرمجية الأساسية المكتوبة بلغة C والتي لا يكون مفهوم كائنية التوجه مدمجًا فيها، كما يُعَدّ فهم هذا المصطلح أمرًا رئيسيًا لقراءة معظم الشيفرات البرمجية الأساسية المكتوبة بلغة C، إذ يمكّنك فهم طريقة قراءة التجريدات الموجودة ضمن الشيفرة البرمجية من تكوين فكرة عن تصاميم واجهات API الداخلية.

```c
#include <stdio.h>

/* الواجهة البرمجية التي سننفذها */
struct greet_api
{
  int (*say_hello)(char *name);
  int (*say_goodbye)(void);
};

/* hello تطبيق دالة */
int say_hello_fn(char *name)
{
  printf("Hello %s\n", name);
  return 0;
}

/* goodbye تطبيق دالة */
int say_goodbye_fn(void)
{
  printf("Goodbye\n");
```

```
  return 0;
}

/* بنية لتنفيذ الواجهة البرمجية*/
struct greet_api greet_api =
{
  .say_hello = say_hello_fn,
  .say_goodbye = say_goodbye_fn
};

لا تحتاج الدالة main() معرفة أيّ شيء عن آلية عمل*/
/*say_hello/goodbye، فهي لا تعلم إلا أنها تعمل *
int main(int argc, char *argv[])
{
  greet_api.say_hello(argv[1]);
  greet_api.say_goodbye();

  printf("%p, %p, %p\n", greet_api.say_hello, say_hello_fn,
&say_hello_fn);

  exit(0);
}
```

تُعَدّ هذه الشيفرة البرمجية أبسط نموذج عن البنى التي يتكرر استخدامها في جميع أجزاء نواة لينكس والبرامج الأخرى المبنية على اللغة C، ولنلقِ نظرةً على بعض العناصر المحددة.

نبدأ بالبنية التي تحدِّد الواجهة البرمجية `struct greet_api`، فالدوال التي أحيطت أسمائها بأقواس مع محدد المؤشر pointer marker تصف مؤشر الدالة، إذ يصف مؤشر الدالة النموذج الأولي للدالة التي يجب أن يشير إليها، كما سيؤدي توجيهه إلى دالة دون إضافة النوع المُعاد return type الصحيح أو المعاملات الصحيحة على الأقل إلى توليد تحذير من المصرِّف، وإذا تركته في الشيفرة البرمجية، فيحتمل أن يؤدي إلى تنفيذ عملية خاطئة أو أعطال، لذلك إذ ستجد غالبًا أنّ أسماء المعامِلات parameters قد حُذفت ولم يُحدَّد إلا نوع المعامِل، ويتيح هذا للمنفذ تحديد أسماء المعامِلات لتجنب ورود تحذيرات من المصرِّف.

سنتناول الآن تنفيذ الواجهة البرمجية، إذ ستجد عادةً في الدوال الأعقد مصطلحًا يدل على أنّ دوال تنفيذ الواجهة البرمجية هي عبارة عن غلاف حول الدوال الأخرى التي تكون عادةً مسبوقةً بشرطة سفلية أو اثنتين، إذ ستستدعي الدالة `say_hello_fn()` دالةً أخرى `_say_hello_function()` على سبيل المثال، ولهذه عدة استخدامات، إذ نستخدمها عمومًا لنحظى بأجزاء أبسط وأصغر من الواجهة API -في تنظيم الوسائط

arguments أو التحقق منها مثلًا- منفصلةً عن عملية التنفيذ الأعقد، ويسهّل هذا غالبًا المسار إلى تحقيق تغييرات ملموسة في العمليات الداخلية مع ضمان بقاء الواجهة ثابتة، إلا أنّ عملية التنفيذ هنا بسيطة جدًا ولا تحتاج حتى إلى دوال داعمة خاصة بها، كما يختلف مدلول بادئات الدالة التي تكون شرطة سفلية واحدة _ أو مزدوجة __ أو حتى ثلاثية ___ باختلاف المشاريع، لكنها عمومًا تُعَدّ تحذيرًا مرئيًا بأنه لا يُفترَض استدعاء الدالة مباشرةً من خارج الواجهة.

> قد يُشار إلى دالة الشرطة السفلية المزدوجة foo__ في المحادثات بالتابع السحري dunder foo وتكون فو foo مثل س أو ص في الجبر.

نملأ مؤشرات الدالة في المرحلة ما قبل الأخيرة في `greet_api struct greet_api greet_api`، إذ يُعَدّ اسم الدالة مؤشرًا، لذا لا حاجة لأخذ عنوان الدالة مثل `&say_hello_fn`، وأخيرًا يمكننا استدعاء دوال واجهة API ضمن بنية `main`.

ستلاحظ هذا المصطلح باستمرار عند تصفحك الشيفرة المصدرية source code، ويمكن أن نوضح ذلك في هذا المثال البسيط الذي اجتزأناه من الملف include/linux/virtio.h في الشيفرة المصدرية لنواة نظام لينكس:

```
 /**
 * virtio_driver - operations for a virtio I/O driver
 * @driver: underlying device driver (populate name and owner).
 * @id_table: the ids serviced by this driver.
 * @feature_table: an array of feature numbers supported by this
driver.
 * @feature_table_size: number of entries in the feature table array.
 * @probe: the function to call when a device is found.  Returns 0 or
-errno.
 * @remove: the function to call when a device is removed.
 * @config_changed: optional function to call when the device
configuration
 *    changes; may be called in interrupt context.
 */
struct virtio_driver {
        struct device_driver driver;
        const struct virtio_device_id *id_table;
        const unsigned int *feature_table;
        unsigned int feature_table_size;
        int (*probe)(struct virtio_device *dev);
```

```
        void (*scan)(struct virtio_device *dev);
        void (*remove)(struct virtio_device *dev);
        void (*config_changed)(struct virtio_device *dev);
        #ifdef CONFIG_PM
        int (*freeze)(struct virtio_device *dev);
        int (*restore)(struct virtio_device *dev);
        #endif
};
```

كل المطلوب هو أن نفهم فهمًا سطحيًا أنّ هذه البنية هي وصف لجهاز الإدخال والإخراج I/O الافتراضي، ونلاحظ أن المتوقَّع من مستخدِم واجهة API هذه -أي كاتب تعريف الجهاز driver device- هو تقديم عدد من الدوال التي ستُستدعَى في شروط مختلفة أثناء تشغيل النظام، أي عند تقصّي عتاد جديد hardware أو عند إزالة عتاد ما ...إلخ. على سبيل المثال، كما يحتوي على مجموعة بيانات، وهي البُنى التي يجب تعبئتها بالبيانات المرتبطة بها، كما يُعَدّ البدء بعناصر توصيف مثل هذه أسهل طريقة لبدء فهم الطبقات المختلفة لشيفرة النواة البرمجية.

1.2.2 المكتبات

تؤدي المكتبات دورَين يوضحان التجريد، هما:

- تتيح للمبرمجين إعادة استخدام الشيفرة البرمجية المتاح الوصول إليها عمومًا.
- تؤدي دور الصندوق الأسود في تنفيذ الخصائص الوظيفية عن المبرمج.

تختص المكتبة التي تنفذ الوصول إلى البيانات غير المعالَجة في الملفات على سبيل المثال بلاحقة JPEG بميزة تتيح للعديد من البرامج التي ترغب في الوصول إلى ملفات الصور استخدام المكتبة نفسها، كما لا يضطر المبرمجون الذين يبرمجون هذه البرامج إلى الانشغال بالتفاصيل الدقيقة لصيغة الملف JPEG، وإنما يركزون جهودهم على دور الصورة أو موضعها في البرنامج.

يشار إلى المكتبة القياسية في منصة يونكس باسم `libc` عمومًا، ومهمتها توفير الواجهة الأساسية للنظام، والاستدعاءات الأساسية مثل `read()` و `write()` و `printf()`، كما توصَف واجهة API هذه بمجملها بتوصيف يسمى بوسيكس POSIX، وهي متاحة مجانًا على الإنترنت وتصف العديد من الاستدعاءات التي تؤلف واجهة API القياسية في نظام يونكس.

تتبع معظم منصات يونكس عمومًا معايير بوسيكس، مع وجود بعض الفروقات الطفيفة التي تكون مهمةً أحيانًا (وهذا ما يفسر تعقيد أنظمة بناء غنو autotools Gnu المختلفة، التي تحاول دومًا إخفاء هذه الفروقات

عنك). يحتوي نظام لينوكس على العديد من الواجهات التي لا تتبع معايير بوسيكس، لذا فإن بناء تطبيقات تستخدم هذه الواجهات دون غيرها لن يجعل تطبيقك محمولًا portable بما يكفي.

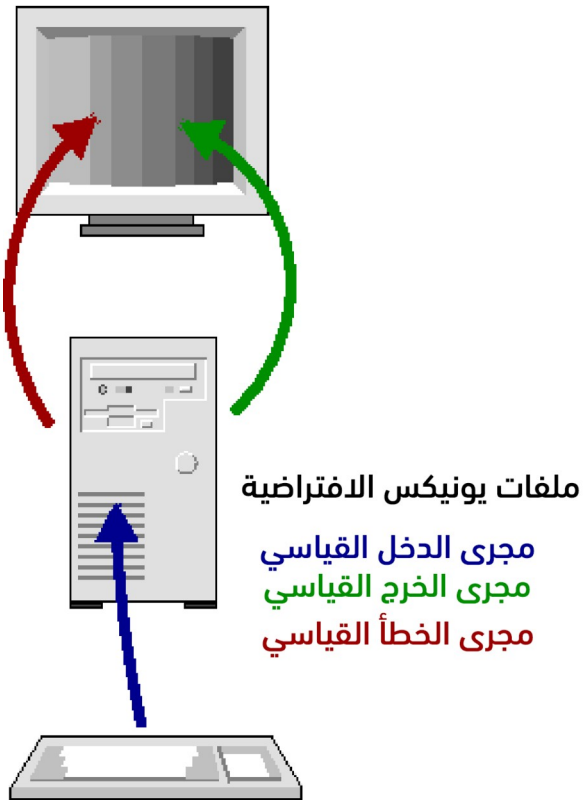
تُعدّ المكتبات تجريدًا أساسيًا يضم الكثير من التفاصيل، وسنتناول في فصول لاحقة آلية عمل المكتبات بالتفصيل.

1.3 مفهوم واصفات الملفات File Descriptors

إحدى أولى المفاهيم التي يتعلمها مبرمج أنظمة يونكس هي أنّ عمل كل برنامج يبدأ بثلاث ملفات تكون مفتوحةً مسبقًا:

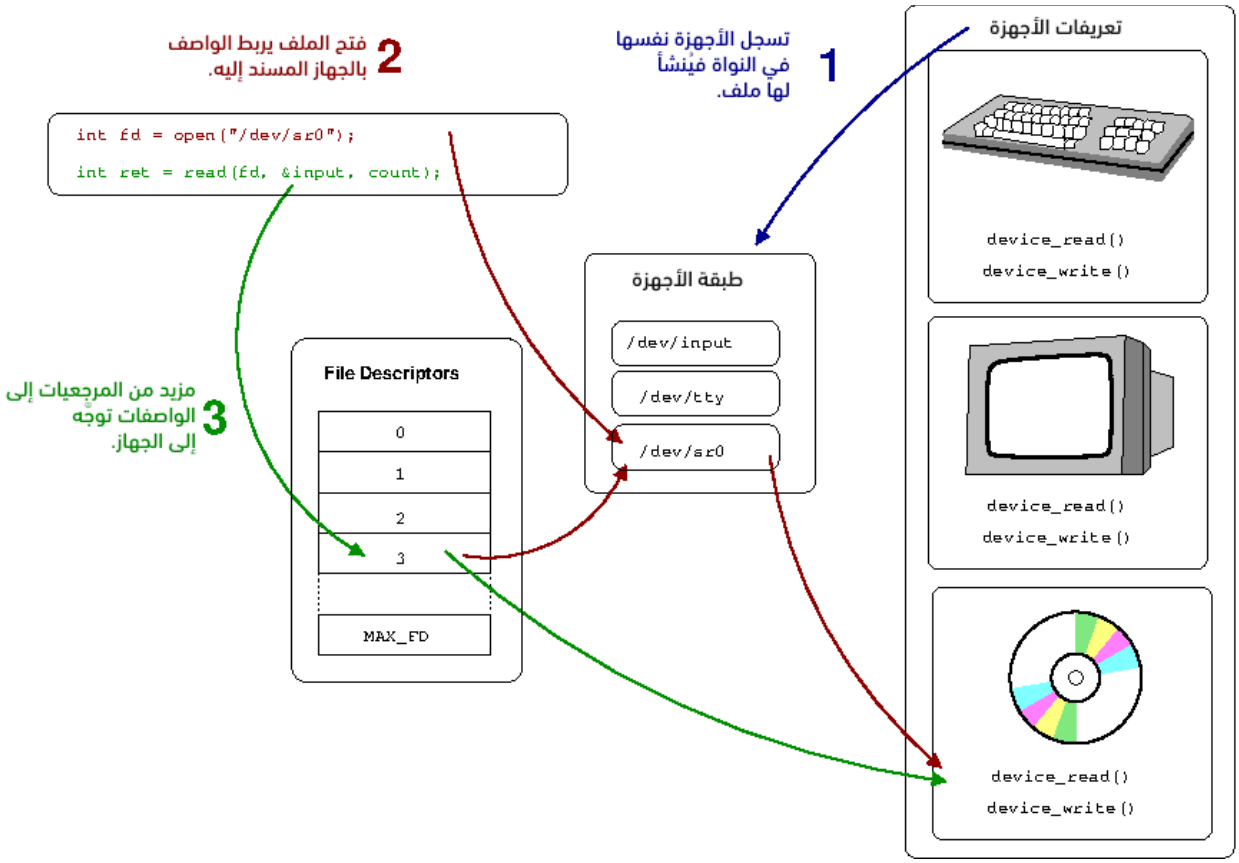
الاسم الوصفي	الاسم المختصر	رقم الملف	الشرح
مجرى الدخل القياسي	stdin	0	الدخل من لوحة المفاتيح
مجرى الخرج القياسي	stdout	1	الخرج الظاهر على الطرفية
مجرى الخطأ القياسي	stderr	2	خرج رسائل الخطأ على الطرفية

جدول 1: الملفات الافتراضية التي يوفرها نظام يونكس



شكل 2: ملفات يونيكس الافتراضية

يستحضر هذا إلى أذهاننا السؤال عمّا يمثله الملف المفتوح وكيفية فتحه، إذ تسمى القيمة التي يعيدها استدعاء open لفتح الملف اصطلاحًا بواصف الملف file descriptor، وهي أساسًا فهرس لمصفوفة من الملفات المفتوحة المخزَّنة في النواة.



شكل 3: فائدة واصفات الملفات في عملية التجريد

تُعَدّ واصفات الملفات فهرسًا لجدول واصفات الملفات تخزنه النواة، بحيث تنشئ النواة واصف ملف استجابةً لاستدعاء open وتربطه ببعض التجريد لكائن يشبه الملف سواءً كان جهازًا فعليًا أو نظام ملفات أو شيء بعيد عن هذا كل البعد، وبالتالي توجه النواة استدعاءات عمليات القراءة read والكتابة write التي تشير إلى واصف الملف ذاك إلى الموضع الصحيح لتنفذ مهمة مفيدة في النهاية.

تعرض الصورة نظرةً عامةً على تجريد العتاد، وباختصار يُعَدّ واصف الملف البوابة إلى تجريدات النواة للعتاد والأجهزة الأساسية.

لنبدأ من المستوى الأدنى، إذ يتطلب نظام التشغيل وجود مبرمج ينشئ تعريفًا للجهاز device driver أو برنامج تعريف حتى يتمكن من التواصل مع أحد أجهزة العتاد، ويُكتَب تعريف الجهاز هذا إلى واجهة API التي توفرها النواة بالطريقة نفسها والتي وردت في المثال السابق، إذ سيوفر تعريف الجهاز مجموعة دوال تستدعيها النواة استجابةً للمتطلبات المختلفة، ويمكننا في المثال المبسَّط في الصورة السابقة رؤية أن تعريف الجهاز يوفِّر دالة القراءة read والكتابة write اللتين ستُستدعيان استجابةً للعمليات المماثلة التي تنفذ على واصف الملف، كما يعلم تعريف الجهاز كيف يحوِّل هذه الطلبات العامة إلى طلبات أو أوامر محددة لجهاز محدد.

تقدم النواة واجهة-ملف file-interface لتوفير التجريد لمساحة المستخدِم عبر ما يسمى بطبقة الجهاز device layer عمومًا، إذ تمثَّل الأجهزة المادية على المضيف بملف له نظام ملفات خاص مثل /dev، ففي أنظمة يونكس وما يشابهها تحتوي عقد الجهاز device-nodes على ما اصطلح تسميته بالعدد الرئيسي major number والعدد الثانوي minor number، مما يتيح للنواة ربط عقد محددة بما يقابلها ببرنامج التعريف الموفر، كما يمكنك الاطلاع عليها من خلال الأمر ls كما هو موضح في المثال التالي:

```
$ ls -l /dev/null /dev/zero /dev/tty
crw-rw-rw- 1 root root 1, 3 Aug 26 13:12 /dev/null
crw-rw-rw- 1 root root 5, 0 Sep  2 15:06 /dev/tty
crw-rw-rw- 1 root root 1, 5 Aug 26 13:12 /dev/zero
```

ينقلنا هذا إلى واصف الملف، وهو الأداة التي تستخدِمها مساحة المستخدِم للتواصل مع الجهاز الأساسي، وبصورة عامة ما يحدث عند فتح الملف هو أنّ النواة تستخدِم معلومات المسار لربط map واصف الملف بشيء يوفِّر واجهتّي API قراءة وكتابة وغيرهما مناسبة، فعندما تكون عملية فتح الملف open للجهاز مثل /dev/sr0 في مثالنا السابق، فسيوفر العدد الرئيسي والثانوي لعقدة الجهاز المفتوح المعلومات التي تحتاجها النواة للعثور على تعريف الجهاز الصحيح وإتمام عملية الربط mapping، كما ستعلم النواة بعد ذلك كيف توجه الاستدعاءات اللاحقة مثل القراءة read إلى الدوال الأساسية التي يوفرها تعريف الجهاز.

يعمل الملف غير المرتبط بجهاز non-device file بآلية مشابهة، على الرغم من وجود طبقات أكثر خلال العملية، فالتجريد هنا هو نقطة الوصل أو الربط mount point، وكما تملك عملية توصيل نظام الملفات file system mounting غايةً مزدوجةً تتمثل في إعداد عملية الربط mapping، بحيث يتعرف نظام الملفات على الجهاز الأساسي الذي يوفِّر التخزين وتعلم النواة أنّ الملفات المفتوحة في نقطة التوصيل تلك يجب أن توجَّه إلى تعريف نظام الملفات، كما تُكتَب أنظمة الملفات على واجهة API محددة لنظام الملفات العام التي توفرها النواة على غرار تعريفات الأجهزة.

بالطبع الصورة الكاملة معقدة أكثر في الواقع، إذ تضم عدة طبقات أخرى، فتبذل النواة على سبيل المثال جهدًا كبيرًا لتخزين cache أكبر قدر ممكن من البيانات الواردة من الأقراص في الذاكرة الخالية، ويقدِّم هذا العديد من الميزات التي تحسِّن السرعة، كما تحاول النواة تنظيم الوصول إلى الجهاز بأكثر طريقة فعالة وممكنة مثل محاولة طلب الوصول إلى القرص للتأكد من أن البيانات المخزَّنة فيزيائيًا بالقرب من بعضها ستستعاد معًا حتى لو لم ترد الطلبات بترتيب تسلسلي، بالإضافة إلى انتماء العديد من الأجهزة إلى فئة أعم مثل أجهزة USB أو SCSI التي توفِّر طبقات التجريد الخاصة بها للكتابة عليها، وبالتالي ستمر أنظمة الملفات في هذه الطبقات المتعددة بدلًا من الكتابة مباشرةً على الأجهزة، أي يكون فهم النواة هو فهم كيفية ترابط واجهات API المتعددة تلك وتواجدها مع بعضها.

1.3.1 الصدفة Shell

تُعَدّ الصدفة بوابة التفاعل مع نظام التشغيل سواءً كانت باش bash أو zsh أو csh أو أيّ نوع من أنواع الأصداف الأخرى العديدة، إذ تشترك جميعها أساسًا في مهمة رئيسية واحدة فقط، وهي أنها تتيح لك تنفيذ البرامج، كما ستبدأ بفهم آلية تنفيذ الصدفة لهذه المهمة فعليًا عندما سنتحدث لاحقًا عن بعض العناصر الداخلية لنظام التشغيل.

لكن الأصداف قادرة على تنفيذ مهام أكبر بكثير من مجرد إتاحة تنفيذ برنامج، إذ تتميز بقدرات قوية لإعادة توجيه الملفات، وتتيح لك تنفيذ عدة برامج في الوقت نفسه وكتابة نصوص برمجية تبني برامج متكاملة، وهذا كله يعيدنا إلى مقولة كل شيء هو عبارة عن ملف.

ا. إعادة التوجيه Redirection

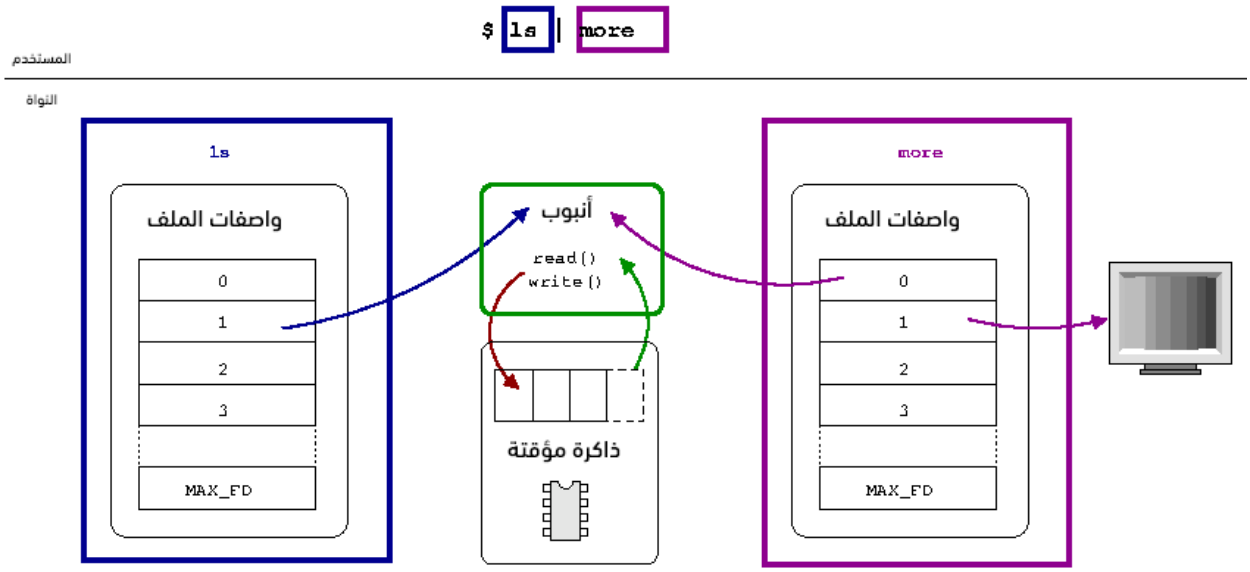
لا نريد في معظم الأحيان أن تشير واصفات الملفات القياسية التي تحدثنا عنها في فقرة سابقة إلى مواضع محددة افتراضيًا، فقد ترغب مثلًا في تسجيل كامل خرج البرنامج على ملف تحدده على القرص أو في جعله يتلقى أوامره من ملف أعددته مسبقًا، وقد ترغب في تمرير خرج برنامج ليكون دخل برنامج آخر، إذ تيسّر الصدفة ذلك وأكثر بالعمل مع نظام التشغيل.

الاسم	الأمر	الوصف	مثال
إعادة التوجيه إلى ملف	`> filename`	أخذ كامل الخرج الناتج عن Standard Out وتسجيله في الملف `filename` (استبدل `filename` باسم الملف). ملاحظة: استخدم `>>` لتُلحق الخرج بنهاية محتوى الملف بدلًا من استبدال محتواه	`ls > filename`
القراءة من ملف	`< filename`	نسخ كافة البيانات من الملف إلى دخل البرنامج القياسي standard input	`cat < filename`
التمرير Pipe	`Program1 | program2`	أخذ كامل خرج standard out البرنامج الأول `program1` وتمريره إلى دخل standard input البرنامج الثاني `program2`	`ls | more`

جدول 2: إعادة التوجيه في الصدفة

ب. تنفيذ عملية التمرير pipe

يُعَدّ تنفيذ الأمر `ls | more` مثالًا آخر على قدرة التجريد، فما يحدث هنا بصورة أساسية هو أنه بدلًا من ربط واصف الملف لمجرى الخرج القياسي بإحدى الأجهزة الأساسية مثل الطرفية لعرض الخرج عليها، يوجَّه الواصف إلى مخزن مؤقت buffer في الذاكرة توفِّره النواة ويطلق عليه عادةً الأنبوب pipe، والمميز هنا هو إمكانية عملية أخرى أن تربط دخلها القياسي standard input بالجانب الآخر من المخزن المؤقت ذاته buffer وتستحوذ على خرج العملية الأخرى بفعالية كما هو موضَّح في الصورة التالية:



شكل 4: الأنبوب

الأنبوب هو مخزن مؤقت في الذاكرة يربط عمليتين معًا، وشير واصفات الملف إلى كائن الأنبوب الذي يخزن البيانات المرسلة إليه من خلال عملية الكتابة ليصرِّفها من خلال عملية القراءة.

تخزِّن النواة عمليات الكتابة في الأنبوب حتى تصرّف عملية قراءة مقابلة من الجانب الآخر للمخزن المؤقت، وهذا مفهوم قوي جدًا وهو أحد الأشكال الأساسية للتواصل بين العمليات inter-process communication -أو IPC اختصارًا- في أنظمة يونكس وما يشابهها، كما لا تقتصر عملية التمرير على نقل البيانات، إذ يمكن أن تؤدي دور قناة إشارات signaling channel، فإذا قرأت إحدى العمليات أنبوبًا فارغًا، فستعطله أو تجمده block افتراضيًا أو تضعه في حالة سبات hibernation إلى حين توفر بعض البيانات، وسنتعمق في هذا أكثر في الفصل الخامس من الكتاب.

وبالتالي قد تستخدِم عمليتان أنبوبًا للإبلاغ عن اتخاذ إجراء ما عن طريق كتابة بايت واحد من البيانات، فبدلًا من أن تكون البيانات الفعلية مهمةً، فإن مجرد وجود أية بيانات في الأنبوب يمكن أن تشير إلى رسالة، فلنفترض مثلًا أنّ إحدى العمليات تطلب طباعة عملية أخرى لملف وهو أمر سيستغرق بعض الوقت، لذا قد تُعِدّ العمليتان أنبوبًا بينهما بحيث تقرأ العملية التي أرسلت الطلب الأنبوب الفارغ،

وبما أنه فارغ، فسيعطّل هذا الاستدعاء وتبطِل العملية، لكن بمجرد الانتهاء من الطباعة، ستكتب العملية الأخرى رسالةً في الأنبوب ويؤدي ذلك إلى إيقاظ العملية التي أرسلت الطلب بصورة فعالة وإرسال إشارة تدل على انتهاء العمل.

ينبثق عن السماح للعمليات بتمرير البيانات بين بعضها بهذه الطريقة مصطلح شائع آخر في يونكس للأدوات الصغيرة التي تنفذ أمرًا معينًا، ويضفي تسلسل هذه الأدوات الصغيرة مرونةً لا تستطيع أداة موحَّدة إضفاءها في معظم الأحيان.

دورة تطوير التطبيقات
باستخدام لغة بايثون

احترف البرمجة وتطوير التطبيقات مع أكاديمية حسوب
والتحق بسوق العمل فور انتهائك من الدورة

التحق بالدورة الآن

2. تمثيل الأعداد والنظام الثنائي

2.1 تعرف على نظام العد الثنائي Binary أساس الحوسبة

النظام الثنائي Binary هو نظام عددي يكون أساس العدد فيه 2، ويمثل المعلومـات بحـالتين متنـافيتين لا ثالث لهما، ويتكون العدد الثنائي من عناصر تسمى بِتات **bits** بحيث يمكن أن يكـون كـل بت بإحـدى الحـالتين المحتمَلتَين، واللتين نمثلهما عمومًا بالرقمين 1 و 0.

2.1.1 نظرية النظام الثنائي

النظام الثنائي هو نظام يكون فيه الأساس هو العدد 2 ليمثل المعلومات بحالتين متنافيتين، ويتكون العــدد الثنائي من عناصر تسمى **بِتّات bits**، إذ يمكن أن يكون كل بِتّ بإحدى الحـالتين المحتمَلـتين واللـتين نمثِّلهمـا عمومًا بالرقمين 1 و 0، ويمكننا القول أنهما تمثِّلان القيمتَين الصحيحة والخاطئة؛ أما من الناحية الكهربائية، فقد تمثَّل الحالتين بجهد كهربائي مرتفع ومنخفض أو مثل زر التشغيل والإيقاف.

نبني الأعداد الثنائية بالطريقة نفسها التي نبني بها الأعداد في نظامنـا التقليـدي العشـري الـذي يكـون فيـه الأساس هو العدد 10، لكن بدلًا من منزلة الآحـاد ومنزلـة العشـرات ومنزلـة المئـات، ...إلخ. لـدينا منزلـة الواحـد ومنزلة الاثنين ومنزلة الأربعة ومنزلة الثمانية، ...إلخ. أي كما هو موضح في الجدول التالي:

$2^{...}$	2^6	2^5	2^4	2^3	2^2	2^1	2^0
...	64	32	16	8	4	2	1

جدول 3: النظام الثنائي

لنمثِّل العدد 203 على سبيل المثال في الأساس العشري، إذ نعلم أننا نضع الرقم 3 في منزلة الآحاد، والرقم 0 في منزلة العشرات والرقم 2 في منزلة المئات، ويمثَّل هذا من خلال الأُسس exponents كما في الجدول التالي:

10^2	10^1	10^0
2	0	3

جدول 4: تمثيل العدد 203 بالأساس العشري

أو نمثلها بطريقة أخرى:

$$2 \times 10^2 + 3 \times 10^0 = 200 + 3 = 203$$

لنمثِّل العدد نفسه بالنظام الثنائي، إذ سيكون لدينا الجدول التالي:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	1	1

جدول 5: تمثيل العدد 203 بالنظام الثنائي

ويكافئ هذا:

$$2^7 + 2^6 + 2^3 + 2^1 + 2^0 = 128 + 64 + 8 + 2 + 1 = 203$$

ا. أسس الحوسبة

قد تتساءل كيف لعدد بسيط أن يكون الأساس الذي بنيَت عليه كل الأمور المذهلة التي يستطيع الحاسوب تنفيذها، ورغم صعوبة تصديق ذلك إلا أنها الحقيقة، إذ يحتوي المعالج الموجود في حاسوبك على مجموعة معقدة -لكنها محدودة في النهاية- من التعليمات instructions التي يمكن تنفيذها على قيم مثل الجمع والضرب، ...إلخ. إذ يُسنَد عدد إلى كل تعليمة من هذه التعليمات بصورة أساسية، حتى يمثَّل برنامج كامل بسلسلة من الأعداد فقط، أي أضف هذا إلى ذاك، اضربه بذاك، قسّم عليه، وهكذا، فإذا كان المعالج مثلًا يعلم أنّ العملية 2 هي الجمع، فإنّ العدد 252 قد يعني "اجمع 5 و 2 وخزِّن الناتج في مكان ما"، وتُعَدّ العمليات في الواقع أعقد بكثير طبعًا، إذ سنتناولها في فصل معمارية الحاسوب لاحقًا، لكن باختصار هذا هو الحاسوب.

كان بمقدور المرء في عهد البطاقات المثقَّبة punch-cards أن يرى بعينه الواحدات والأصفار التي تكوِّن مسار البرنامج من خلال النظر إلى الثقوب الموجودة على البطاقة، طبعًا تحوّل ذلك اليوم إلى آلية التخزين السريع والدقيق بواسطة قطبية الجزيئات الممغنطة الصغيرة مثل الأشرطة tapes أو الأقراص disks، والذي أتاح لنا حمل كميات هائلة تفوق التصور من البيانات في جيوبنا.

إنّ ترجمة هذه الأعـداد إلى خـدمات تنفـع البشـرية هي مـا يجعـل الحاسـوب نافعًـا لهـذه الدرجـة، وتتكـوّن الشاشات مثلًا من ملايين البكسلات pixels المنفصلة، وكل منهـا صـغير لدرجـة لا تميّـزه عين الإنسـان، لكنهـا تكوّن صورةً مكتملةً عندما تكون مجتمعةً، إذ يحتوي كل بكسل عمومًا على عناصـر محـدَّدة من الأحمـر والأخضـر والأزرق التي تكوّن اللون الذي يعرضه، وبالتأكيـد يمكن تمثيـل هـذه القيم بالأعـداد الـتي بـالطبع يمكن تمثيلهـا بالنظام الثنائي، وبالتالي يمكن تقسيم أيّ صورة إلى ملايين النقاط الفردية، وتُمثَّل كل نقطة بمجموعــة من ثلاث قيم تمثِّل قيم الأحمر والأخضر والأزرق للبكسل، وبالتالي عندما يكون لدينا سلسلة طويلة من هذه الأعداد وتكون مُصاغةً بصورة صحيحة، فستتمكن أجهزة الفيديو في حاسـوبك من تحويـل هـذه الأعـداد إلى إشـارات كهربائيـة لتشغيل وإيقاف البكسلات الفردية لعرض صورة.

سنبني بيئة الحوسبة الحديثة بأكملها في الكتاب بدءًا من اللبنة الأساسية هذه، أي من القاعدة إلى القمــة أو من الألف إلى الياء إذا صح التعبير.

ب. البتات Bits والبايتات Bytes

يمكننا بصورة أساسية تمثيل أيّ شيء بعدد كمـا تحـدثنا في الفقـرات السـابقة، ويمكن تحويلـه إلى النظـام الثنائي وإجراء عمليات عليه بواسطة الحاسوب، إذ سـنحتاج على الأقـل لتمثيـل جميـع أحـرف الأبجديـة مثلًا إلى توليفات مختلفة وكافية لتمثيل جميع المحارف الصغيرة lower case والمحارف الكبيرة upper case والأعداد وعلامات الترقيم إلى جانب بعض الأمور الإضافية، ويعني هذا أننا ربما سنحتاج إلى حوالي 80 توليفة مختلفة.

إذا كان لدينا بِتّان، فيمكننا تمثيل 4 توليفات فريدة محتملة وهي 00 01 10 11، أمـا إذا كـان لـدينا ثلاث بِتّـات، فيمكننـا تمثيـل 8 توليفـات مختلفـة، وبصـورة عامـة، إذا كـان لـدينا عـدد n من البِتّـات يمكننـا تمثيـل 2n توليفة فريدة.

تمنحنا 8 بِتّات 28 = 256 تمثيلًا فريدًا، وهذا عدد أكثر من كاف للتوليفات الأبجدية التي نحتاجها، كما أننـا ندعو كل 8 بِتّات ببايت، كما أنّ حجم المتغير من نوع char هو بايت واحد في لغة C.

أسكي ASCII

يستطيع أيّ شخص اختلاق رابط بين الأحرف والأعداد عشوائيًا بما أنّ البـايت يمكنـه تمثيـل أيّ قيمـة بين 0 و 255، فقد تقرّر الشركة المصنعة لبطاقات الفيديو مثلًا أنّ رقم 1 يمثل المحرف 'A'، لذا عندما ترسَـل القيمـة 1 إلى بطاقة الفيديو، ستعرض المحرف 'A' بحالتـه الكبـيرة على الشاشـة، وقـد تقـرّر الشـركة المصـنعة للطابعـة لسبب ما أن الرقم 1 يمثل 'z' بالحالة الصغيرة، وبالتالي سيتطلب عرض وطباعة الشيء نفسه تحويلات معقـدة، ولتجنب حدوث ذلك ابتُكِرت الشيفرة المعيارية الأميركية لتبادل المعلومات American Standard Code for Information Interchange -أو ASCII اختصارًا-، وهذه الشيفرة مبنية على 7 بتات bit code-7، أي توجد 2^7 أو 128 شيفرةً متاحةً.

ينقسم مجال الشيفرات إلى جزأين رئيسيين هما الشيفرات الغير قابلة للطباعة والشيفرات القابلة للطباعة، إذ تكون المحارف القابلة للطباعة مثل الأحرف الكبيرة والصغيرة والأعداد وعلامات الترقيم، في حين تكون المحارف الغير قابلة للطباعة مخصصةً للتحكم وتنفيذ عمليات مثل محارف الرجوع إلى بداية السطر carriage-return، أي العودة إلى بداية السطر الحالي دون النزول إلى السطر التالي، أو رن جرس الطرفية عند ورود المحرف Bell أو شيفرة القيمة الفارغة `NULL` الخاصة التي لا تمثل شيئًا على الإطلاق.

تكفي المحارف 127 الفريدة للغة الإنجليزية الأميركية، لكنها تصبح محدودةً جدًا عندما يريد المرء تمثيل المحارف السائدة في اللغات الأخرى وخاصةً اللغات الآسيوية التي قد تحتوي على عدة آلاف من المحارف الفريدة، وللحد من ذلك، تنتقل الأنظمة الحديثة من شيفرة أسكي إلى يونيكود Unicode التي تستخدِم ما يصل إلى 4 بايتات لتمثل محرفًا، وهذا يفسح مجالًا أكبر بكثير.

التكافؤ Parity

يبقى بت واحد من البايت فائضًا بما أنّ شيفرة الأسكي مبنية على 7 بتات فقط، ويمكن الاستفادة منه في تحقيق التكافؤ parity، إذ يُعَدّ شكلًا بسيطًا من أشكال التحقق من الأخطاء، فتخيّل حاسوبًا يستخدِم بطاقات مثقبة في عملية الإدخال، بحيث يمثِّل وجود الثقب البِتّ 1 وغيابه يمثل البِتّ 0، وستؤدي أية تغطية غير مقصودة لثقب ما إلى قراءة قيمة غير صحيحة وستتسبب في سلوك غير معرَّف.

يتيح التكافؤ إجراء فحص بسيط للبِتّات المؤلِّفة للبايت للتأكد من أنها قُرِئت بصورة صحيحة، ويمكننا تنفيذ التكافؤ الفردي أو الزوجي باستخدام البِتّ الفائض الذي نَعدّه بِتّ التكافؤ، فإذا كان عدد الواحدات في المعلومات المخزَّنة على البِتّات السبعة فرديًا، فسيضبط بت التكافؤ ويكون حينها التكافؤ فرديًا Odd parity، وإذا كان عددها زوجيًا، فلا يضبط بِتّ التكافؤ؛ أما التكافؤ الزوجي Even parity، فهو عكس ذلك، فإذا كان عدد الواحدات زوجي، فسيُضبط بِتّ التكافؤ على الرقم 1، وبهذه الطريقة سينتج عن تغيّر بِتّ واحد خطأ تكافؤ يمكن اكتشافه.

الحواسيب ذات أنظمة 16 و 32 و 64 بت

لا تتسع جميع الأعداد في بايت أو مجموعة محددة من البايتات، فبفرض أن كان رصيدك المصرفي كبيرًا مثلًا فهو يحتاج إلى مجال أوسع مما يمكن أن يتسع في بايت واحد لتمثيله، وتتألف المعماريات الحديثة في الحواسيب حاليًا من أنظمة 32 بت على الأقل، وهذا يعني أنها تعمل مع 4 بايتات في وقت واحد عند المعالجة والقراءة أو الكتابة على الذاكرة، ونشير آنذاك إلى كل 4 بايتات **بالكلمة word**، وهذا مشابه للغة حيث تكوِّن الأحرف -أو البتات- الكلمات في جملة ما، والفارق في الحاسوب عن اللغة أنه تكون كل الكلمات بالحجم نفسه، وهو حجم المتغير من نوع `int` في اللغة C الذي يساوي 32 بِتّ، أما معماريات 64 بت الحديثة، يضاعف حجم عمل المعالج إلى 8 بايت بدلًا من 4 في معماريات 32 بت.

ج. كيلوبايت وميغابايت وغيغابايت

تتعامل الحواسيب مع عدد كبير من البايتات وهـذا مـا يجعلهـا شـديدة القـوة، وبالتـالي نحتـاج إلى وسـيلة للتحدث عن أعداد ضخمة من البايتات، والوسيلة البديهية لـذلك هي اسـتخدام بادئـات نظـام الوحـدات الـدولي International System of Units- أو SI اختصارًا- كما هو متّبع في معظم المجالات العلمية الأخرى، إذ يشير الكيلو مثلًا إلى 10^3 أو 1000 وحدة، بحيث يكون الكيلوغرام الواحد هو 1000 غرام.

يُعَدّ 1000 عددًا تقريبيًا round جيدًا في الأساس العشري، لكنه يمثَّل في النظام الثنـائي بـ 1111101000 وهو ليس عددًا تقريبيًا، لكن 1024 أو 2^{10} هو عدد تقريـبي والـذي يمثَّـل في النظـام الثنـائي بـ 10000000000، وهو قريب جدًا من الكيلو في النظام العشري، أي العـدد 1000 قـريب من العـدد 1024، وبالتـالي أصـبح 1024 بايت بطبيعة الحال يُعرَف بالكيلوبايت.

أما الوحدة التالية في نظام الوحدات الدولي، فهي ميغا mega المقابلـة لقيمـة 10^6، كمـا تسـتمر البادئـات بالازدياد بمقدار 10^3 المقابلة للتجميع المعتاد المكون من ثلاثة أرقام عند كتابة أعداد كبيرة، كما يصـادف مجـددًا أن تكـون 2^{20} قريبـةً من تحديـد نظـام الواحـدات الـدولي للميغـا في النظـام العشـري، أي 1048576 بـدلًا من 1000000، فعند زيادة واحدات النظام الثنائي بالقوى من مضـاعفات 10 تبقى قريبـةً وظيفيًـا من قيمـة النظـام العشري في نظام الواحدات الدولي، مع أنه يحيد قليلًا كل عامل متزايد عن دلالة أساس نظام الواحدات الـدولي، وبالتالي فإنّ وحدات النظام العشري في نظام الواحدات الدولي قريبـة بمـا يكفي على قيم النظـام الثنـائي، وقـد شاع استخدامها لتلك القيم.

الاسم	معامل النظام الثنائي	بايت	معامل النظام العشري القريب	بايت في النظام العشري
1 كيلوبايت	2^{10}	1024	10^3	1000
1 ميغابايت	2^{20}	1.048.576	10^6	1.000.000
1 غيغابايت	2^{30}	1.073.741.824	10^9	1.000.000.000
1 تيرابايت	2^{40}	1.099.511.627.776	10^{12}	1.000.000.000.000
1 بيتابايت	2^{50}	1.125.899.906.842.624	10^{15}	1.000.000.000.000.000
1 إكسابايت	2^{60}	1.152.921.504.606.846.976	10^{18}	1.000.000.000.000.000.000

جدول 6: معاملات النظام الثنائي والعشري المتعلقة بالبايتات

قد يفيدك ترسيخ معامِلات النظام الثنائي في ذاكرتك كثيرًا في الربط السريع للعلاقة بين عـدد البِتّـات والأحجـام التي يفهمها الإنسان، إذ يمكننا بسرعة مثلًا حسـاب إمكانيـة حاسـوب بنظـام 32 بِتّ أن يعـالج مـا يصـل إلى 4 غيغابايت من الذاكرة من خلال ملاحظة إعادة التركيب $(4)\ 2^2\ +\ 2^{30}$، وبالمثل يمكن أن تعالِج قيمة 64 بِتّ مـا

يصل إلى 16 إكسابايت، أي $2^{60} + 2^4$، كما يمكنك حساب ضخامة هذا العدد، ولتأخذ فكرةً عن مدى ضـخامته، فيمكنك حساب المدة التي ستستغرقها في العد إلى 2^{64} إذا عددت رقمًا واحدًا كل ثانية.

كيلوبت وميغابت وغيغابت

سيشار إلى السعات غالبًا بالبِتّات بدلاً من البايتات إلى جانب الارتباك الـذي يحـدث نتيجـة العبء المفـرط لتحويل واحدات نظام الواحدات الدولي SI بين النظامَين الثنائي والعشري، ويحدث هذا عمومًا عند التحـدث في مجال الشبكات أو أجهزة التخزين، فربما لاحظت أنّ اتصال ADSL لديك يشار إليه بقيمة مثل 1500 كيلوبِت في الثانية، إن العملية الحسابية بسيطة، إذ نضرب بالعدد 1000 للكيلو ثم نقسّم على 8 لنحوّله إلى بايت ثم نقسّـمه على العدد 1024 لنحوله إلى كيلوبايت، وبالتالي تكون 1500 كيلوبِت في ثانية = 183 كيلوبايت في الثانية.

أقرَّت هيئة نظام الواحدات الدولي هذه الاستخدامات المزدوجة وحددت بادئات فريدةً للاستخدام الثنائي، إذ تقابل 1024 بايت بموجب المعيار كيبي بايت kibibyte، وهو اختصار للكيلوبايت الثنـائي kilo binary byte وتُختصر بـ KiB؛ أما البادئات الأخرى، فلها بادئة مماثلـة مثـل ميـبي بـايتس Mebibyte وتختصـر MiB، ويمنـع العرف المتَّبع إلى حد كبير استخدام هذه المصطلحات، لكنك قد تراها في بعض المؤلَّفات.

التحويل

يُعّدّ استخدام الحاسوب الطريقة الأسهل للتحويل بين الأنظمة، فبعد كل شيء هذا ما يبر ع فيه. ومـع ذلـك، فمن المفيد معرفة كيفية إجراء التحويلات يدويًا.

تُعَدّ القسمة المتكررة الطريقة الأسهل للتحويل بين الأنظمة، بحيث نقسّم ناتج القسمة بصـورة متكـررة على الأساس إلى أن يصبح ناتج القسمة صفرًا مع تدوين الباقي في كل خطوة، ثم ندوّن الباقي بالعكس، أي نبـدأ من الأسفل ونلحق العدد بالجهة اليمين في كل مرة، وسنذكر مثالًا للتوضيح، كما سيكون الأساس 2 نظرًا لأننا نحـوّل إلى النظام الثنائي.

عملية القسمة	النتيجة	الباقي	اتجاه قراءة الباقي
$203_{10} \div 2$	101	1	
$101_{10} \div 2$	50	1	↑
$50_{10} \div 2$	25	0	↑
$25_{10} \div 2$	12	1	↑
$12_{10} \div 2$	6	0	↑
$6_{10} \div 2$	3	0	↑
$3_{10} \div 2$	1	1	↑
$110 \div 2$	0	1	↑

جدول 7: تحويل العدد 203 للنظام الثنائي

ابدأ بقراءة الباقي من الأسفل وأضف كل عدد منه إلى اليمين لتحصل على النتيجة 11001011، وقد وجـدنا فعلًا أنّ هذه القيمة في النظام الثنائي هي 203 في النظام العشري.

د. العمليات البوليانية Boolean Operations

اكتشف جورج بول عالم الرياضيات مجالًا كاملًا في الرياضيات يسمى جبر بُول Boolean Algebra، وعلى الرغم من أنّ اكتشافاته كانت في منتصف القرن التاسع عشر، إلا أنها أصبحت لاحقًا أساسيات علوم الحاسـوب، ويُعَدّ جبر بول هو موضوع واسع النطاق، لذا سنتناول في هذا الكتاب بعض مبادئه الأساسية فقط حتى تستطيع بدء رحلة التعلم.

تأخذ العمليات البوليانية ببساطة دخلًا معينًا وتنتج خرجًا معينًا حسب قاعدة معينة، وأبسط عمليـة بوليانيـة مثلًا هي not، وهي تعكس قيمة معامِل operand الدخل؛ أمـا المعـامِلات الأخـرى، فتأخـذ عـادةً دخلين وتنتج خرجًا واحدًا.

يسهل تذكر العمليات البوليانية الأساسية المستخدَمة في علوم الحاسوب وقـد أدرجناهـا في هـذا الفصـل، ومثلناها بجداول الحقيقة truth tables الـتي تـبيِّن بمظهـر بسـيط جميـع المـدخلات والمخرجـات المحتمَلـة، ويقابل مصطلح حقيقي true القيمة 1 في النظام الثنائي.

Not

تمثَّل عادةً بالرمز ! ، وهي تعكس قيمة الدخل فتحول 0 إلى 1 و 1 إلى 0.

الدخل	الخرج
1	0
0	1

جدول 8: جدول الحقيقة لـ NOT

And

تذكَّر العبارة التالية: "تكون النتيجة حقيقيةً إذا كان الدخل الأول حقيقيًا و الدخل الثاني حقيقيًـا" لكي يسـهل عليك تذكُّر آلية عمل معامل and.

الدخل الأول	الدخل الثاني	الخرج
0	0	0
1	0	0
0	1	0
1	1	1

جدول 9: جدول الحقيقة لـ AND

Or

تذكَّر العبارة التالية: "تكون النتيجة حقيقيةً إذا كان الدخل الأول حقيقيًا أو الدخل الثاني حقيقيًا" لكي يسـهل عليك تذكُّر آلية عمل معامل or .

الدخل الأول	الدخل الثاني	الخرج
0	0	0
1	0	1
0	1	1
1	1	1

جدول 10: جدول الحقيقة لـ OR

معامل أو الحصرية Exclusive Or

تختصَر عبارة معامـل أو الحصـرية Exclusive Or بـ xor وهي حالـة خاصـة من معامِـل or، بحيث يكـون الخرج حقيقيًا عندما يكون أحد المدخَلين فقط حقيقيًا، وستدهشك الحيل المميزة الـتي يسـتطيع هـذا المعامِـل تنفيذها، لكنها ليست مستخدَمةً كثيرًا في النواة.

الدخل الأول	الدخل الثاني	الخرج
0	0	0
1	0	1
0	1	1
1	1	0

جدول 11: جدول الحقيقة لـ Xor

٥. استخدام العمليات البوليانية في الحواسيب

قد يصعب عليك تصديق أنّ أساس كل ما ينفِّذه حاسوبك هو تلك المعامِلات التي تحـدثنا عنهـا، فالجـامع النصفي half adder مثلًا هو أحد أنواع الدارات التي تتكون من العمليات البوليانيـة الـتي تجمـع البِتّـات، وقـد

سُمّي الجامع النصفي لأنه لا يعالج البتات الفائضة، وستبدأ في بناء كيان يجمع أعداد ثنائية طويلة من خلال وضع أكثر من جامع نصفي معًا، ثم أضف إليه بعض الذواكر الخارجية وستكون قد بنيت حاسوبًا.

تنفَّذ العمليات البوليانية من الناحية الإلكترونية في بوابات gates مصنوعة من الترانزستورات transistors، لذا لا بد أنك سمعت عن عدد الترانزستورات transistor counts وقانون مور وغيرها، وكلما زاد عدد الترانزستورات زاد عدد البوابات وزاد عدد الأشياء التي يمكنك جمعها، كما ستحتاج لبناء الحاسوب الحديث إلى عدد هائل من البوابات وعدد هائل من الترانزستورات، إذ تحتوي بعض معالِجات إيتانيوم Itanium على حوالي 460 مليون ترانزستور.

و. العمل بالنظام الثنائي في اللغة C

توجد واجهة مباشرة لجميع المعامِلات التي ذكرناها في اللغة C، ويشرح الجدول التالي هذه المعامِلات:

المعامل	اصطلاحه في اللغة C
not	!
and	&
or	\
xor	^

جدول 12: العمليات المنطقية في C

نطبّق هذه المعامِلات على المتغيرات لتعديل البِتّات ضمن المتغير، ولكن يجب علينا أولًا أن نتناول شرحًا للترميز الست العشري قبل أن نستعرض أمثلةً عن ذلك.

2.1.2 النظام الست عشري Hexadecimal

يشير النظام الست عشري إلى نظام أساسه العدد 16، والسبب الوحيد لاستخدامنا هذا النظام في علوم الحاسوب هو أنه يسهِّل على الإنسان التفكير في الأرقام الثنائية، إذ يسهِّل عدم تعامل الحواسيب إلا مع النظامَين الثنائي والست عشري على الإنسان محاولته التعامل مع الحاسوب.

لكن لماذا اختير الأساس 16؟ إن الخيار الطبيعي هو الأساس 10 لأننا معتادون على التفكير في الأساس 10 حسب نظامنا العددي اليومي، لكن الأساس 10 لا يتوافق كثيرًا مع النظام الثنائي حيث أننا نحتاج إلى أربع بتات لتمثيل 10 عناصر مختلفة في النظام الثنائي، لكن تلك الأربع بتات توفر لنا ست عشرة توليفة محتمَلة، لذا نحن أمام احتمالين؛ إما أن نختار الطريقة شديدة التعقيد المتمثلة في محاولة التحويل بين النظام العشري والنظام الثنائي، أو أن نختار الطريقة السهلة وننشئ نظامًا عدديًا أساسه العدد 16 وهو النظام الست عشري.

يستخدم النظام الست عشري الأعداد القياسية في النظام العشري مع إضافة الأحرف A B C D E F التي تشير إلى الأعداد 10 11 12 13 14 15، مع الانتباه إلى بدء العدّ من الصفر، فمتى ما رأيت عددًا مسبوقًا بـ 0x،

فاعلم أنه يدل على عدد ست عشري، وكما ذكرنا أنه سنحتاج إلى أربع بتات بالضبط لتمثيل 16 نمط مختلف في النظام الثنائي، لذا يمثِّل كل عدد ست عشري أربع بتات بالضبط، ويجب أن تعـدّه تمرينًـا لتتعلم الجـدول التـالي عن ظهر قلب.

النظام الست عشري	النظام الثنائي	النظام العشري
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

جدول 13: النظام الست عشري والثنائي والعشري

بالطبع لا يوجد سبب للتوقف عن متابعة النمط (مثل تحديـد G للقيمـة 16)، ولكن القيم السـتة عشـرة هي موازنة ممتازة بين تقلبات الذاكرة البشرية وعدد البتات التي يستخدمها الحاسوب، كما سـتجد أيضًـا الأسـاس 8 مستخدَمًا أحيانًا في سماحيات الملفات في أنظمة يونكس مثلًا، ونمثِّل ببساطة أعـدادًا أكـبر من البتـات بأعـداد أكثر، إذ يمكن مثلًا تمثيل متغير يتألف من ستة عشر بت بالقيمة 0xAB12، وما عليك سوى تحويل كل رقم على حدى وفقًا للجدول السابق ثم جمع القيم معًا لتجد مقابلها في النظام الثنائي، أي لتكون القيمة المقابلـة للقيمـة 0xAB12 هي العدد الذي يتألف من 16 بت في النظام الثنائي 1010101100010010، كمـا نسـتطيع التحويـل من النظام الثنائي إلى النظام الست عشري بعكس تلك العملية، كما نستطيع الاستعانة بنهج القسـمة المتكـررة ذاته لتغيير أساس أي عدد، فلإيجاد قيمة العدد 203 بالنظام الست عشري مثلًا:

عملية القسمة	النتيجة	الباقي	اتجاه قراءة الباقي
$203_{10} \div 16$	12	11 (0xB)	
$12_{10} \div 16$	0	12 (0xC)	↑

جدول 14: تحويل العدد 203 إلى النظام الست عشري

لذا تكون قيمة 203 في النظام الست عشري هي 0xCB

2.1.3 الاستخدام العملي للأنظمة العددية

سنطلع فيما يلي على الاستخدام العملي للأنظمة العددية وما النتائج العملية التي ممكن أن نحصل عليها.

ا. استخدام النظام الثنائي في الشيفرات البرمجية

تُعَدّ برمجة حاسوب بلغات عالية المستوى high level دون معرفة أيّ شيء عنه هو أمر عملي بحت على الرغم من أنّ النظام الثنائي هو اللغة الأساسية لكل حاسوب، وعلى أية حال نهتم ببعض مبادئ النظام الثنائي الأساسية والمستخدَمة بصورة متكررة بالنسبة شيفرة البرمجية منخفضة المستوى low level code التي سنتناولها.

ب. التقنع والرايات

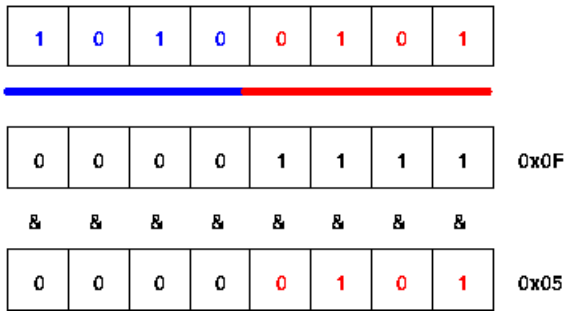
سنشرح مفهوم عمليتي التقنع والرايات وكيفية تطبيقهما عمليًا على الأنظمة العددية.

التقنع Masking

من المهم غالبًا جعل البنى والمتغيرات تحجز مساحةً بأكثر طريقة فعالة ممكنة في الشيفرة البرمجية منخفضة المستوى، وقد يتضمن هذا في بعض الحالات تعبئة packing متغيرين -يكونان مرتبطين ببعضهما عمومًا- بمتغير واحد بطريقة فعالة.

تذكَّر أنّ كل بِتّ يمثل حالتين، فإذا علمنا مثلًا أنّ للمتغير 16 حالة محتمَلة فقط، فيمكن تمثيله بـ 4 بِتّات، أي $2^4 = 16$ قيمةً فريدةً، لكن أصغر نوع يمكننا التصريح عنه في اللغة C هو 8 بتات وهو نوع char أي محرف، فإما نهدر أربع بتات، أو نجد طريقةً نستخدِم فيها تلك البتات الفائضة، ويمكننا تحقيق ذلك بسهولة من خلال عملية التقنُّع التي تتبع قواعد العمليات المنطقية لاستخراج القيم وهي موضَّحة في الصورة التالية.

نحتفظ بقيمتَين منفصلتين تتألفان من 4 بِتّات داخل محرف واحد يتألف من 8 بِتّات، إذ نُعِدّ البِتّات الأربعة الأولى (الزرقاء) قيمةً واحدةً والبِتّات الأربعة الأخيرة (الحمراء) قيمةً أخرى، وقد ضبطنا القناع على تعيين قيمة البتات الأربعة الأخيرة 1 (0x0F) لاستخراج البِتّات الأربعة السفلية، وبما أنّ المعامِل and المنطقي سيضبط البت إلى 1 فقط إذا كانت قيمة كلا البتّين 1، فستخفي البِتّات التي ضبطنا قيمتها على 0 في القناع وهي البتات التي لا تهمنا بصورة فعالة.



شكل 5: التقنُّع

نقلب القناع للحصول على البِتّات الأربعة الأولى (الزرقـاء)، أي نضـبط البِتّـات الأربعـة الأولى على القيمـة 1 والبتات الأربعة الأخيرة على القيمة 0، وسوف تلاحظ أنّ نتيجة هـذا سـتكون 0000 1010 أو 0xA0 في النظـام الست عشري، على حين أننا نريد فعلًا أن نعتبر هذه القيمة الفريدة المؤلفـة من 4 بتـات 1010 أي 0x0A، ومن أجل وضع هذه البتات في الموضع الصحيح نستخدِم المعامِل right shift أربع مرات، وهذا سوف يمنحنـا القيمة النهائية 1010 0000.

```c
#include <stdio.h>

#define LOWER_MASK 0x0F
#define UPPER_MASK 0xF0

int main(int argc, char* argv[])
{
        /* قيمتان بحجم 4 بتات مخزنتان في متغير بحجم 8 بتات */
        char value = 0xA5;
        char lower = value & LOWER_MASK;
        char upper = (value & UPPER_MASK) >> 4;

        printf("Lower: %x\n", lower);
        printf("Upper: %x\n", upper);
}
```

يتطلب ضبط البِتّات المعامِل or المنطقي، لكن سنستخدم الأصفار 0 بدلًا من اسـتخدام الواحـدات 1 على أسـاس قنـاع، كمـا ننصـحك برسـم مخطـط مشـابه للصـورة السـابقة والعمـل على ضـبط البِتّـات بواسـطة المعامِل or المنطقي.

الرايات flags

يتضمن البرنامج غالبًا عددًا كبيرًا من المتغيرات التي توجد فقط بصيغة رايات flags في شروط معينة، فآلة الحالات state machine مثلًا هي خوارزمية تتنقل عبر عدد من الحالات المختلفة، لكنها لا تتواجد إلا في حالة واحدة فقط في المرة الواحدة، ولنقل أنه لديها 8 حالات مختلفة، إذ نستطيع بسهولة التصريح عن 8 متغيرات مختلفة، بحيث يكون هناك متغير واحد لكل حالة، لكن في كثير من الحالات يفضَّل التصريح عن متغير واحد مؤلف من 8 بتات وتعيين راية لكل بِتّ للإشارة إلى حالة معينة.

تُعَدّ الرايات حالةً خاصةً من التقنُّع، لكن يمثِّل كل بِتّ حالةً بوليانيةً معينةً، أي تشغيل أو إيقاف، كما يمكن لمتغير مؤلَّف من عدد n من البتات أن يحمل العدد n من الرايات المختلفة، ويُعَدّ نموذج الشيفرة البرمجية التالي هو مثال نموذجي على استخدام الرايات، وستلحظ اختلافات في هذه الشيفرة البرمجية الأساسية في معظم الأحيان.

```c
#include <stdio.h>

/*
*  تعريف كافة الرايات الثمانية المحتمَلة لمتغير بحجم 8 بتات
*        الاسم      النظام الست عشري      النظام الثنائي
*/
#define FLAG1 0x01 /* 00000001 */
#define FLAG2 0x02 /* 00000010 */
#define FLAG3 0x04 /* 00000100 */
#define FLAG4 0x08 /* 00001000 */
/* ... وهكذا */
#define FLAG8 0x80 /* 10000000 */

int main(int argc, char *argv[])
{
        char flags = 0; /* متغير بحجم 8 بتات */
        /* ضبط الرايات بمعامل or المنطقي */
        flags = flags | FLAG1; /* ضبط الراية الأولى */
        flags = flags | FLAG3; /* ضبط الراية الثالثة

        /* تحقق من الرايات بالمعامل and المنطقي. إذا كانت الراية مضبوطة بالقيمة 1
         * سيرجع المعامل and قيمة 1
         * if مما سيحقق الشرط الوارد في  */
        if (flags & FLAG1)
```

```
            printf("FLAG1 set!\n");

        /* سيكون هذا بالطبع غير صحيح */
        if (flags & FLAG8)
            printf("FLAG8 set!\n");

        /* تحقق من عدة رايات بواسطة or المنطقي
         * سيمرر هذا لأن الراية الأولى مضبوطة */
        if (flags & (FLAG1|FLAG4))
            printf("FLAG1 or FLAG4 set!\n");

        return 0;
}
```

2.2 تمثيل الأنواع والأعداد في الأنظمة الحاسوبية

يجب التصريح عن نوع كل متغير في اللغة التي يحدَّد فيها نوع المتغير typed language مثل اللغة C، إذ يُعلِم النوع المصرِّف مالذي يتوقع تخزينه في المتغير، وبالتالي يستطيع المصرّف تخصيص مساحة كافية لهذا الاستخدام، والتحقق من أن المبرمج لا ينتهك قيود النوع المحدَّد

2.2.1 معايير اللغة C

من الضروري الاطلاع قليلًا على تاريخ اللغة البرمجية C على الرغم من الاختلاف الطفيف بينها وبين بقية اللغات، إذ تُعَدّ C بأنها اللغة السائدة في عالم برمجة الأنظمة، فكل نظام تشغيل ومكتباته المرتبطة به التي يشيع استخدامها مكتوبة باللغة C، كما يوفِّر كل نظام مصرِّفًا compiler للغة C، وقد وضِع معيار صارم لهذه اللغة للحد من اختلافها بين هذه الأنظمة والتي من المؤكد أنّ كل منها سيجري العديد من التغييرات التي لن تتوافق مع بعضها.

يُعرَف هذا المعيار رسميًا باسم ISO/IEC 9899:1999(E)، لكن يشار إليه عادةً بالاختصار C99، إذ تشرف عليه منظمة المعايير الدولية ISO، كما أتيح شراء المعيار كاملًا على الإنترنت، ولم تَعُد الإصدارات القديمة من هذا المعيار مثل الإصدار C89 -الذي سبق C99 وأصدِر في عام 1989- و ANSI C شائعة الاستخدام، وأصبحت جزءًا من أحدث معيار، كما أنّ توثيق المعيار تقني بحت ويذكر بالتفصيل تقريبًا جميع نواحي اللغة، إذ يشرح مثلًا بنيتها بصيغة باكوس نور Backus Naur وقيم `#define` المعيارية والآلية التي يجب أن تعمل وفقها العمليات.

من الضروري أيضًا ملاحظة ما الذي لا تحدده معايير اللغة C، والأهم من ذلك أنه يجب أن يكون المعيار ملائمًا لكل معمارية حاسوبية حالية ومستقبلية، وبالتالي يحرص على عدم تحديد المجالات التي تعتمد على المعمارية، كما يُعَدّ الرابط بين معيار اللغة C والمعمارية الأساسية هو واجهة التطبيق الثنائية Application Binary Interface -أو ABI اختصارًا- التي سنتحدث عنها لاحقًا، كما سيذكر المعيار في عدة مواضع أن أية عملية أو بنية معيّنة سيكون لها نتيجة غير محددة أو نتيجة تعتمد على التنفيذ، ومن البديهي أن المبرمج لا يمكنه الاعتماد على هذه النتائج إذا كان يريد كتابة شيفرة برمجية محمولة portable.

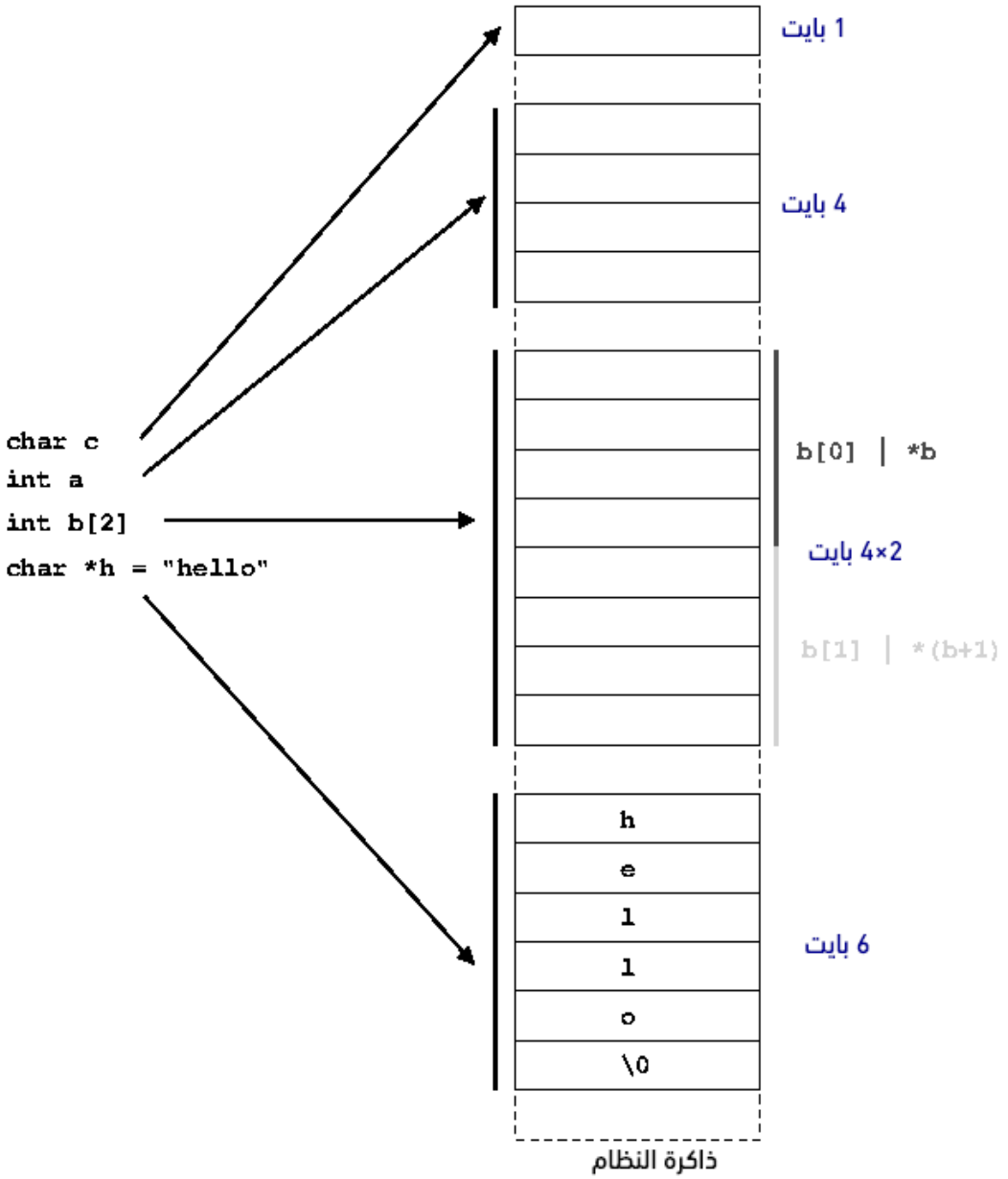
2.2.2 جنوسي GNU C

ينفِّذ مصرِّف GNU C -والذي يشار إليه عادةً بالاختصار gcc- معيار C99 بالكامل تقريبًا، ويطبِّق أيضًا مجموعة إضافات للمعيار سيستخدمها المبرمجون غالبًا للحصول على خصائص وظيفية إضافية على حساب قابلية النقل إلى مصرِّف آخر، إذ ترتبط هذه الإضافات عادةً بالشيفرة البرمجية ذات المستوى المنخفض low level code وهي أكثر شيوعًا في مجال برمجة النظم؛ أما أكثر إضافة يشيع استخدامها في هذا المجال، فهي شيفرة التجميع المُضمّن inline assembly، كما يجب على المبرمجين قراءة توثيق مصرِّف GNU C وفهم متى قد يستخدِمون الخصائص الإضافية على المعيار.

يمكن توجيه مصرِّف GNU C للالتزام بدقة بالمعيار مثل راية `-std = c99` والتحذير أو توليد خطأ عند تنفيذ أمور معينة لا تتوافق مع المعيار. وهذا طبعًا يناسبك عندما تكون بحاجة إلى ضمان إمكانية نقل شيفرتك البرمجية بسهولة إلى مصرِّف آخر.

2.2.3 الأنواع

نحن المبرمجون معتادون على استخدام المتغيرات لتمثيل مساحة من الذاكرة لتحمل قيمةً، إذ يجب التصريح عن نوع كل متغير في اللغة التي يُحدَّد فيها نوع المتغير typed language مثل اللغة C، كما يخبر النوع المصرِّف مالذي يتوقع تخزينه في المتغير، وبالتالي سيستطيع المصرِّف تخصيص مساحة كافية لهذا الاستخدام والتحقق من أنّ المبرمج لا ينتهك قيود النوع المحدَّد، وسنجد في الصورة التالية مثالًا على المساحة المخصصة لبعض الأنواع الشائعة من المتغيرات.



شكل 6: الأنواع

يذكر معيار C99 أصغر حجم ممكن لكل نوع من أنواع المتغيرات المعرَّفة في اللغة C فقط، وذلك لأنّ الحجم الأمثل للأنواع يختلف اختلافًا كبيرًا بين مختلف معماريات المعالجات وأنظمة التشغيل، ولكي تكون العملية صحيحةً تمامًا يجب ألا يفترض المبرمجون أبدًا حجم أيّ من متغيراتهم، لكن يحتاج نظام التشغيل الفعال بطبيعة الحال إلى اتفاقات حول الأحجام التي ستحجزها أنواع المتغيرات في النظام، كما تتقيد كل معمارية ونظام تشغيل بواجهة التطبيق الثنائية Application Binary Interface -أو ABI اختصارًا-، إذ تملأ واجهة التطبيق الثنائية لنظام ما التفاصيل التي تربط بين معيار اللغة C ومتطلبات العتاد الصلب الأساسي ونظام التشغيل، كما تُكتَب واجهة التطبيق الثنائية لمجموعة محدَّدة من المعالج ونظام التشغيل.

النوع	الحجم الأدنى وفق معيار C99 بواحدة البِتّ	الحجم الشائع أي معمارية 32 بِتّ
char	8	8
short	16	16
int	16	32
long	32	32
long long	64	64
المؤشرات Pointers	حسب التنفيذ	32

جدول 15: أنواع وأحجام الأعداد الصحيحة القياسية

نلاحظ في مثالنا السابق أنّ الاختلاف الوحيد عن المعيار C99 هو أن حجم المتغير من نوع `int` هــو 32 بِت عادةً، وهو ضعف الحد الأدنى الصارم لحجم 16 بت الذي يتطلبه المعيار C99، كما أنّ المؤشـرات Pointers هي فعليًا عنوان فقط، أي أنّ قيمتها تكون عنوانًا وبالتالي "تشير" إلى موقع آخر في الذاكرة، لذا يجب تخصيص حجم كافٍ للمؤشر حتى يتمكن من عنونة أيّ موقع في ذاكرة النظام.

ا. 64 بت

إحدى النواحي المربِكة هي إدراج حوسبة 64 بت، إذ يعني هذا أنّ المعالج يمكنه معالجة العناوين التي تخزَّن على 64 بت وتحديدًا تكون سعة السجلات 64 بِتّ، وهو موضوع سنتناوله في فصل معمارية الحاسوب لاحقًا.

يعني هذا أولًا أنّ جميع المؤشرات يجب أن تكون بحجم 64 بِتّ حتى تتمكن من تمثيـل أيّ عنـوان محتمـل في النظام، لكن عندها يجب على منفّذِي النظام system implementers تحديد حجم الأنواع الأخرى، في حين ينتشر استخدام نموذجَين شائعَين على نطاق واسع كما هو موضح في الجدول التالي:

النوع	الحجم الأدنى وفق معيار C99 بواحدة البِتّ	الحجم الشائع LP64	الحجم الشائع في نظام التشغيل ويندوز
char	8	8	8
short	16	16	16
int	16	32	32
long	32	64	32
long long	64	64	64
المؤشرات Pointers	حسب التنفيذ	64	64

جدول 16: أنواع وأحجام أنواع البيانات القياسية التي تتضمن قيمة مفردة

يمكنك ملاحظة أنه في نموذج LP64 -أو 64 اختصارًا- يحدَّد حجم long pointer 64 أي المؤشر الطويل 64 بِتّ، وهذا يختلف عن نموذج 32 بِتّ الذي عرضناه سابقًا، إذ يستخدَم نموذج LP64 في أنظمة يونيكس UNIX على نطاق واسع؛ أما في النموذج الآخر، فيبقى حجم المتغير من نوع `long` بقيمة 32 بت، وهذا يحافظ على أقصى قدر ممكن من التوافق مع الشيفرة البرمجية بنظام 32، إذ يُستخدَم هذا النموذج في نظام ويندوز الذي يدعم 64 بِتّ.

تكمن أسباب وجيهة خلف عدم زيادة حجم المتغير من نوع `int` إلى 64 بِتّ في أيّ من النموذجين، فإذا زاد حجم هذا المتغير إلى 64 بِتّ، فلن تترك للمبرمجين أيّ طريقة للحصول على متغير بحجم 32 بِتّ، وستكون الطريقة الوحيدة هي إعادة تعريف المتغيرات من نوع `short` لتكون من نوع 32 بِتّ الأكبر.

يُعَدّ المتغير بحجم 64 بِتّ كبيرًا جدًا لدرجة أنه ليس مطلوبًا عمومًا لتمثيل العديد من المتغيرات، فنادرًا ما تتكرر الحلقات loops مثلًا عدد مرات أكبر من أن يتسع في متغير حجمه 32 بِتّ الذي يتسع لـ 4294967296 مرة، وعادةً ما تمثَّل الصور بثمانية بِتّات لكل من قيم الأحمر والأخضر والأزرق وثمانية بِتّات إضافية مخصصة للمعلومات الإضافية (قناة ألفا) ما مجموعه 32 بِتّ، وبالتالي سيؤدي استخدام متغير بحجم 64 بِتّ في كثير من الحالات إلى إهدار أول 32 بِتّ على الأقل إذا لم يُهدَر أكثر من ذلك، وليس هذا فحسب، وإنما حجم مصفوفة عدد صحيح integer يتضاعف بذلك أيضًا.

يعني هذا أنّ البرامج ستستهلك حجمًا أكبر من ذاكرة النظام دون أيّ تحسن يذكر في أدائه، وبالتالي حجمًا أكبر من ذاكرة التخزين المؤقت cache التي سنتحدث عنها بالتفصيل في فصل معمارية الحاسوب، ولهذا السبب اختار نظام ويندوز الاحتفاظ بتخزين قيم المتغيرات من نوع long في 32 بت، فبما أنّ الكثير من واجهات API على نظام ويندوز قد كُتبَت في الأصل لاستخدام متغيرات من نوع long مخزَّنة على نظام 32 بِتّ، لذا لا تحتاج إلى بِتّات إضافية، مما سيوفر ذلك مساحةً مهدورةً كبيرةً في النظام دون الحاجة إلى إعادة كتابة كامل واجهة API.

إذا جربنا البديل المقترَح المتمثل في إعادة تعريف المتغير من نوع `short` ليكون متغيرًا يخزَّن على 32 بِت، فسيستطيع المبرمجون الذين يعملون على نظام 64 بِت تحديد هذا النوع للمتغيرات التي يعلمون أنها مرتبطة بقيم أصغر، ولكن عند العودة إلى نظام 32 بِتّ، فسيكون متغير `short` نفسه الذي حدوده الآن بحجم 16 بِتّ فقط، وهي قيمة تجاوزوها بمراحل كبيرة عمليًا، أي 2^{10} = 65536.

سيحقق جعل المبرمج يطلب متغيرات أكبر حجمًا عندما يعلم أنه سيحتاج إليها توازنًا فيما يتعلق بمخاوف قابلية النقل وإهدار المساحة في الأنظمة الثنائية.

ب. مؤهلات الأنواع

يتحدث معيار اللغة C أيضًا عن بعض المؤهلات qualifiers لأنواع المتغيرات، إذ يشير المؤهل `const` مثلًا إلى أنّ المتغير لن تُعدَّل قيمته الأصلية أبدًا، والمؤهل `volatile` يقترح على المصرِّف بأنّ قيمة المتغير

قد تتغير بعيدًا عن تدفق تنفيذ البرنامج، لذا يجب أن يحرص المصرِّف على عدم إعادة ترتيب الوصول إليه بأيّ شكل من الأشكال، كما يُعَدّ كل من مؤهل المؤشَّر `signed` ومؤهل غير المؤشَّر `unsigned` أنهما المؤهلَين الأهم على الأرجح، فهما يحدِّدان فيما إذا كان يُسمَح للمتغير بأن يأخذ قيمةً سالبةً أم لا، وسنتناول هذا بالتفصيل لاحقًا.

الغرض من جميع المؤهلات هو تمرير معلومات إضافية للمصرِّف حول كيفية استخدامِه للمتغير، ويعني هذا أمرَين وهما أنّ المصرِّف قادر على التحقق مما إذا انتهكت القواعد التي وضعتها بنفسك مثل الكتابة في متغير قيمته ثابتة `const`، وقادر على إجراء تحسينات بناءً على المعلومات الإضافية، وسندرس هذا في فصول لاحقة.

ج. الأنواع المعيارية

يدرك واضعو معيار C99 أنّ كل هذه القواعد والأحجام ومخاوف توفر قابلية للنقل قد تصبح مربكةً جدًا، ولتسهيل الأمر فقد قدموا في المعيار سلسلةً من الأنواع الخاصة التي تحدِّد الخصائص المضبوطة للمتغير، وتُحدَّد في الترويسة `<stdint.h>` وصيغتها `qtypes_t`، إذ يرمز المحرف `q` إلى المؤهل ويرمز `type` إلى النوع الأساسي، في حين يرمز المحرف `s` إلى الحجم بواحدة البِتّ و `_t` هو امتداد يشير إلى أنك تستخدم الأنواع المعرَّفة في معيار C99.

تشير الصيغة `uint8_t` مثلًا إلى عدد صحيح غير مؤشَّر يخزَّن على 8 بِتّات بالضبط، وقد عُرِّفَت العديد من الأنواع الأخرى، إذ يمكنك الاطلاع على القائمة الكاملة المفصَّلة في مقطع المكتبة المعيارية 17.8 لمعيار C99 أو في ملف الترويسة الموجود بصورة مشفَّرة، كما إنّ توفير هذه الأنواع هي مهمة النظام الذي يطبق معيار C99 بأن يحدِّد لها الأنواع ذات الحجم الملائم على النظام المستهدَف، فمثلًا توفِّر مكتبات النظام هذه الترويسات في نظام التشغيل لينكس.

لاحظ أنّ معيار C99 فيه عوامل مساعدة لتحقيق قابلية النقل لـ `printf`، إذ يمكن استخدام وحدات ماكرو PRI macros في `<inttypes.h>` على أساس عوامل محددة للأنواع التي حُدِّدت أحجامها، وكما ذكرنا يمكنك الاطلاع على المعلومات كاملةً في المعيار أو باستخراج الترويسات.

د. التطبيق العملي للأنواع

نرى في النموذج التالي الذي يمثِّل التحذيرات التي ترد عندما لا تتطابق الأنواع مثالًا على فرض الأنواع قيودًا تحدِّد أيّ العمليات متاح تنفيذها على المتغير وكيف يستعين المصرِّف بهذه المعلومات لعرض تحذير عند استخدام المتغيرات بطريقة تخالف تلك القيود، إذ تبدأ الشيفرة البرمجية بإسناد قيمة عدد صحيح `integer` للمتغير `char`، وبما أنّ حجم المتغير `char` أصغر، فسنفقد القيمة الصحيحة للعدد الصحيح `integer`.

نحاول بعدها تعيين مؤشر pointer للمتغير `char` يشير إلى الذاكرة التي حددنا بأنها عدد صحيح `integer`، ويمكن تنفيذ هذه العملية، لكنها ليست آمنةً، لذا نُفِّذ المثال الأول على جهاز معالجه بينتيوم

Pentium ذو 32 بِتّ، وأعيدَت القيمة الصحيحة، لكن يبلغ حجم المؤشر 64 بتّ -أي 8 بايت- في نظام معالجه إيتانيوم Itanium ذو 64 بِتّ كما هو موضح في المثال الثاني، ولكن حجم العدد الصحيح `integer` يبلغ 4 بايت فقط، وبالطبع لن تتسع 8 بايت في 4 بايت.

يمكننا محاولة خداع المصرف بتحويل القيمة قبل إسنادها، ولاحظ أننا في هذه الحالة فاقمنا المشكلة عندما نفَّذنا هذا التحويل وتجاهلنا تحذير المصرف، لأنّ المتغير الأصغر لا يمكنه الاحتفاظ بجميع المعلومات الواردة من المؤشر، فنتلقى في النهاية عنوانًا غير صالح.

```
/*
* types.c
*/
#include <stdio.h>
#include <stdint.h>
int main(void)
{
        char a;
        char *p = "hello";
        int i;

        // نقل متغير كبير إلى متغير أصغر منه
        i = 0x12341234;
        a = i;
        i = a;
        printf("i is %d\n", i);

        // integer نقل المؤشر ليشير إلى متغير من نوع
        printf("p is %p\n", p);
        i = p;
        // الخداع بإجراء التحويلات
        i = (int)p;
        p = (char*)i;
        printf("p is %p\n", p);

        return 0;
}
$ uname -m
i686
```

```
$ gcc -Wall -o types types.c
types.c: In function 'main':
types.c:19: warning: assignment makes integer from pointer without
a cast
$ ./types
i is 52
p is 0x80484e8
p is 0x80484e8
$ uname -m
ia64
$ gcc -Wall  -o types types.c
types.c: In function 'main':
types.c:19: warning: assignment makes integer from pointer without a
cast
types.c:21: warning: cast from pointer to integer of different size
types.c:22: warning: cast to pointer from integer of different size
$ ./types
i is 52
p is 0x40000000000009e0
p is 0x9e0
```

2.2.4 تمثيل الأعداد

سنشرح كيفية تمثيل الأعداد بمختلف مجالاتها مثل الأعداد السالبة والأعداد العشرية وغيرهما.

ا. القيم السلبية

نميِّز العدد السالب في نظامنا العشري الحديث بوضع علامة الطرح - قبله؛ أما عندما نستخدِم النظام الثنائي، فعلينا اتباع أسلوب مختلف عند الإشارة إلى الأرقام السالبة، إذ يوجد نظام وحيد شائع استخدامه في العتاد الصلب الحديث، لكن معيار C99 يحدِّد ثلاثة أساليب مقبولة لتمثيل القيمة السلبية.

ب. بت الإشارة Sign Bit

أبسط طريقة هي تخصيص بت واحد من العدد يشير إلى قيمة سالبة أو موجبة حسب هل هو محدَّد أم لا، وهذا مشابه للنهج الرياضي الذي يبين قيمة العدد بإشارتي + و -، إذ يُعَدّ هذا منطقيًا نوعًا ما، وقد مثّلت بعض أجهزة الحاسوب الأولية أعدادًا سالبةً بهذه الطريقة، لكن يتيح استخدام الأعداد الثنائية بعض الاحتمالات الأخرى التي تسهِّل عمل مصممي العتاد الصلب.

لاحظ أنّ القيمة 0 قد أصبح لها الآن قيمتان مكافئتان، واحدة حُدِّد فيها بِتّ إشارة وواحدة دون تحديده، وقد يُشار أحيانًا إلى هذه القيم بـ +0 و -0 على التوالي.

ج. المتمم الأحادي One's complement

يطبِّق نهج المتمِّم الأحادي العملية not على العدد الموجب من أجل تمثيل العدد السالب، لذا تمثَّل القيمة -90 (-0x5A) مثلًا بـ ~01011010 = 10100101.

لاحظ أن العامِل ~ هو عامِل في اللغة C الذي يطبق عامِل NOT على القيمة، كما يدعى أحيانًا بعامِـل المتمم الأحادي لأسباب صارت معروفة لدينا الآن.

الميزة الأكبر في هذا النظام هي أنه لا يشترط تطبيق منطق خاص عند إضافة عدد سالب إلى عـدد مـوجب، باستثناء أنه يجب إضافة أيّ حمل carry إضافي متبقي إلى القيمة النهائية، لذا تأمل الجدول التالي:

النظام العشري	النظام الثنائي	العملية
-90	10100101	+
100	01100100	
---	--------	
10	[1] 00001001	9
	00001010	10

جدول 17: إضافة بت الحِمْل

إذا أضفت البتات الواحد تلو الآخر، فستجد أنه سينتج لديك في النهايـة بِتّ حمـل carry bit الموضَّـح في الجدول، وستنتج لدينا القيمة الصحيحة 10 بإضافته مجددًا إلى العدد الأصلي.

مجددًا لا تزال لدينا مشكلة تمثيل الصِفرين، ولا يوجد حاسوب حديث يسـتخدِم المتمم الأحـادي، والسـبب الرئيسي في ذلك وجود نظام أفضل.

المتمم الثنائي Two's Complement

يتشابه المتمم الثنائي تمامًا مع المتمم الأحادي، باستثناء أنّ التمثيل السالب يضاف إليه واحد ونتجاهل أيّ بِتّات حمل متبقية، فإذا طبقناه على المثال السابق، فسنمثِّل العدد -90 وفق ما يلي:

```
~01011010+1=10100101+1 = 10100110
```

يعني هذا أنّ هناك تماثلًا غريبًا بعض الشيء في الأعداد التي يمكن تمثيلها؛ ففي العـدد الصـحيح integer مثلًا الذي يخزَّن على 8 بِت لدينا 8^2 = 256 قيمة ممكنة، كما يمكننا تمثيل 127- في نهج تمثيـل بِت الإشـارة بواسطة 127، لكن يمكننا تمثيل 127- في نظام المتمم الثنائي بواسطة 128 لأننا أزلنا مشـكلة وجـود صـفرين،

وضَع في الحسبان أنّ الصفر السالب هو `(1 + (~00000000) = (11111111 + 1) = 00000000`، ولاحظ تجاهل بِتّ الحمل.

النظام العشري	النظام الثنائي	العملية OP
-90	10100110	+
100	01100100	
---	--------	
10	00001010	

جدول 18: إضافة المتمم الثنائي

لا بدّ أنك لاحظت أنّ تطبيق المتمم الثنائي لن يُحيج مصممي العتاد الصلب إلا إلى توفير عمليات منطقية لدارات الإضافة، إذ يمكن إجراء عملية الطرح عن طريق متمم ثنائي ينفي القيمة المراد طرحها ثم يضيف القيمة الجديدة، وبالمثل يمكنك تنفيذ عملية الضرب بالجمع المتكرر وعملية القسمة بالطرح المتكرر. وبالتالي يختزل المتمم الثنائي جميع العمليات الحسابية البسيطة بعملية الجمع، ومن الجدير بالذكر أن جميع الحواسيب الحديثة تستخدِم تمثيل المتمم الثنائي.

امتداد الإشارة Sign-extension

بناءً على صيغة المتمم الثنائي، عند زيادة حجم القيمة المؤشَّرة signed value، من المهم أن تمدَّد إشارة sign-extended البتات الإضافية، أي المنسوخة من البِتّ الأولي للقيمة الحالية، إذ تمثِّل قيمة العدد الصحيح `-10` من نوع `int` المخزَّن على 32 بت في المتمم الثنائي في النظام الثنائي عبى سبيل المثال بالعدد `11111111111111111111111111110110`، فإذا أردنا تحويله إلى عدد صحيح من نوع `long long int` مخزَّن على 64 بِتّ، فعلينا أن نحرص على تعيين الرقم `1` للـ 32 بِتّ الإضافية للاحتفاظ بالإشارة نفسها للعدد الأصلي.

بفضل المتمم الثنائي، يكفي أخذ البِتّ الأولي من قيمة الخرج exiting value واستبدال جميع البتات المضافة بهذه القيمة، ويشار إلى هذه العمليات باسم امتداد الإشارة، وعادةً يتعامل معها المصرِّف في الحالات المحدَّدة في معيار اللغة، مع توفير المعالج عمومًا تعليمات خاصة لأخذ قيمة وتمديد إشارتها إلى قيمة أكبر.

د. الأعداد العشرية Floating Point

تحدثنا حتى الآن عن الأعداد الصحيحة integer أو الأعداد الكاملة فقط، وتسمى فئة الأعداد التي يمكن أن تمثِّل القيم العشرية بالأعداد العشرية.

نحتاج لإنشاء عدد عشري إلى طريقة لتمثيل مفهوم الجزء العشري في النظام الثنائي، ويُعرف النظام الأشـهر الذي يحقق ذلك بمعيار الأعداد العشرية IEEE-754 لأن من نشره كان معهد مهندسي الكهربـاء والإلكـترون، كمـا يُعَدّ النظام بسيط للغاية من ناحية المفهوم، وهو مشابه إلى حد ما للصيغة العلمية scientific notation.

قد تمثَّل القيمة 123.45 عمومًا في الصـيغة العلميـة بالصـيغة $1.2345*10^2$، إذ نسـمي 1.2345 الجـزء المعنـوي significant (أو الجـزء الأهم الأساسـي الـذي لـه أهميـة)؛ أمـا 10 فهـو الأسـاس radix و 2 هـو الأُس exponent.

نفكك البتات المتاحة في نموذج العدد العشري IEEE لنمثِّل الإشارة والجزء العشري وأس العدد العشري، إذ يمثَّل العدد العشري بالصيغة: "الإشارة × الجزء المعنـوي × $2^{\text{الأس}}$"، ويعـادل بِتّ الإشـارة إمـا 1 أو 1-، وبمـا أننـا نعمل في النظام الثنائي، فسيكون لدينا دائمًـا الأسـاس الضـمني 2، كمـا تتنـوع أحجـام قيمـة العـدد العشـري، وسندرس في الفقرة التالية القيمة التي تخزَّن على 32 بت فقط، وكلما زاد عدد البتات حظينا بدقة أكبر.

الإشارة	الأس	الجزء المعنوي/الجزء العشري
S	EEEEEEEE	MMMMMMMMMMMMMMMMMMMMMMM

جدول 19:نموذج العدد العشري IEEE

العامل المهم الآخر هو انحياز bias الأس، إذ يجب أن يمثِّل الأس القيم الموجبـة والسـالبة، وبالتـالي تُطـرَح القيمة الضمنية للعدد 127 من الأس، إذ يحتوي الأس 0 مثلًا على حقل أس يسـاوي 127، في حين يمثِّـل 128 العدد 1 ويمثل 126 العدد 1-.

يضيف كل بِتّ من الجزء المعنوي مزيدًا من الدقة إلى القيم التي يمكننا تمثيلها، وضَع في الحسبان تمثيـل الصيغة العلمية للقيمة 198765، إذ يمكننا كتابة هذا بالصيغة 1.98765x106، الذي يقابل التمثيل التالي:

10^0	.	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
1	.	9	8	7	6	5

جدول 20: الصيغة العلمية للقيمة $1.98765x10^6$

يتيح كل رقم إضافي مجالًا أكبر من القيم العشرية التي يمكننا تمثيلها، إذ يزيد كل رقم بعد الفاصلة العشرية من دقة العدد بمقدار 10 مرات في النظام العشري، فيمكننا مثلًا تمثيل 0.0 بـ 0.9 -أي 10 قيم- برقم واحد بعد الفاصلة عشرية، و0.00 بـ 0.99 -أي 100 قيمة- برقمين، وهكذا؛ أما في النظام الثنائي، فبدلًا من أن يمنحنا كل رقم إضافي دقة أكبر بعشر أضعاف، لا نحظى إلا بضعفَي الدقة كما هو موضَّح في الجدول التالي، ويعني هـذا أنّ التمثيل الثنائي الخاص لا يوجّهنا دائمًا بطريقة مباشرة إلى التمثيل العشري.

20	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
1	.	1/2	1/4	1/8	1/16	1/32

1	.	0.5	0.25	0.125	0.0625	0.03125

جدول 21: التمثيل الثنائي للدقة

لا تكون دقة كسورنا كبيرةً جدًا باستخدام بِتّ واحد فقط للدقة، فلا يسعنا إلا أن نقول أنّ الكسر إما 0 أو 0.5، فإذا أضفنا بِتًّا آخرًا للدقة، فيمكننا الآن القول أن القيمة العشرية هي إما 0 أو 0.25 أو 0.5 أو 0.75. ومع إضافة بِتّ آخر للدقة يمكننا الآن تمثيل القيم 0، 0.125، 0.25، 0.375، 0.5، 0.625، 0.75، 0.875.

وبالتالي فكلما زدنا عدد البِتّات حظينا بدقة أكبر، لكن بما أنّ مجال الأعداد المحتملة غير محدود، فلن تكفي البِتّات أبدًا لتمثيل أية قيمة محتمَلة، فإذا كان لدينا بِتّين فقط للدقة على سبيل المثال، وأردنا تمثيل القيمة 0.3، فلا يمكننا القول إلا أنها أقرب إلى 0.25، وطبعًا هذا غير كاف في معظم التطبيقات، لكن عندما يكون لدينا 22 بِتّ للجزء المعنوي، فسنحظى بدقة أفضل بكثير، لكن لا يزال ذلك غير كاف في معظم التطبيقات.

تزيد قيمة متغير من نوع double عدد بتات الجزء المعنوي إلى 52 بت، كما أنها تزيد مجال قيم الأس أيضًا، كما تخصص بعض الأجهزة 84 بِتّ للعدد العشري، و64 بِتّ للجزء المعنوي، إذ تتيح 64 بِتّ تلك دقةً هائلةً لا بدّ أن تكون مناسبةً لجميع التطبيقات باستثناء التطبيقات شديدة التعقيد والتي تحتاج حجمًا أكبر (هل هذا كافٍ لتمثيل طول أقل من حجم الذرة؟).

```
$ cat float.c
#include <stdio.h>

int main(void)
{
            float a = 0.45;
            float b = 8.0;

            double ad = 0.45;
            double bd = 8.0;

            printf("float+float, 6dp    : %f\n", a+b);
            printf("double+double, 6dp  : %f\n", ad+bd);
            printf("float+float, 20dp   : %10.20f\n", a+b);
            printf("dobule+double, 20dp : %10.20f\n", ad+bd);

            return 0;
}
$ gcc -o float float.c
```

```
$ ./float
float+float, 6dp    : 8.450000
double+double, 6dp  : 8.450000
float+float, 20dp   : 8.44999998807907104492
dobule+double, 20dp : 8.44999999999999928946
$ python
Python 2.4.4 (#2, Oct 20 2006, 00:23:25)
[GCC 4.1.2 20061015 (prerelease) (Debian 4.1.1-16.1)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> 8.0 + 0.45
8.4499999999999993
```

يُعَدّ النموذج السابق مثالًا عمليًا لما تحدثنا عنه، ولاحظ تطابق الإجابتين بالنسبة للأجزاء العشرية الستة الافتراضية لتحقيق الدقة التي حددناها في `printf`، وذلك لأن عملية تقريبهما نفِّذت تنفيذًا صحيحًا، لكن عندما يُطلب منك إعطاء نتائج بدقة أكبر ولتكن في هذه الحالة 20 منزلة عشرية، فسنجد أنها تبدأ في الاختلاف.

منحتنا الشيفرة البرمجية التي تستخدِم النوع `double` نتيجةً أدق، لكنها لا تزال غير صحيحة كليًا، كما أنّ المبرمجين الذين لا يتعاملون بوضوح مع القيم من نوع `float` لا يزالون يواجهون مشاكل في دقة المتغيرات.

القيم الموحدة Normalised Values

يمكننا تمثيل قيمة بعدة أساليب مختلفة في الصيغة العلمية مثل = $10023x10^{0}$ = $1002.3x10^{1}$ = $100.23x10^{2}$، وبالتالي نعرّف صيغة التوحيد بأنه الصيغة التي يكون فيها `1/radix <= significand <` 1، إذ تعني radix الأساس وتعني significand الجزء المعنوي، والعدد الموحَّد normalized number هو العدد المكتوب بالصيغة العلمية scientific notation مع رقم عشري واحد غير صفري على الأقل بعد الفاصلة.

يضمن هذا في النظام الثنائي أن يكون البِتّ الذي يقع أقصى اليسار leftmost bit من الجزء المعنوي دائمًا 1، فعند معرفتنا لذلك، يمكننا الحصول على بِتّ إضافي للدقة حسب ما ورد في المعيار أنه عندما يكون البت الأيسر 1 يكون ضمنيًا.

العملية الحسابية	الأس	2^{-5}	2^{-4}	2^{-3}	2^{-2}	2^{-1}	.	2^{0}
$0.375 = 1* (0.25+0.125)2^{0}$	0	0	0	1	1	0	.	0
$0.375= 5. * (0.5+0.25)2^{-1}$	1-	0	0	0	1	1	.	0
$0.375= 0.25 * (1+0.5)2^{-2}$	2-	0	0	0	0	1	.	1

جدول 22: مثال على توحيد القيمة 0.375

كما ترى في المثال السابق، يمكننا جعل القيمة قيمة موحَّدة من خلال تحريك البِتّات للأمام طالما أننا نعوّض عن ذلك بزيادة الأس.

مهارات التوحيد Normalisation

من المشكلات الشائعة التي يواجهها المبرمجون هي العثور على أول بِتّ ضبط في مجموعة البِتّات bitfield، ولنأخذ مجموعة البتات 0100 مثالًا، فعند البدء من اليمين، يكون بِتّ الضبط الأول هو البِتّ 2، إذ نبدأ من الصفر كما هو معتاد.

الطريقة المعيارية للعثور على هذه القيمة هي الإزاحة إلى اليمين والتحقق مما إذا كان البِتّ الأول هو 1 أي بت الضبط، ثم إنهاء العملية أو تكرارها، وتُعَدّ هذه عمليةً بطيئةً، فإذا كان طول مجموعة البِتّات 64 بِتّ وكان بِتّ الضبط هو الأخير فقط، فيجب أن تمر بجميع البتات الثلاثة والستين التي تسبقها.

لكن إذا كانت قيمة مجموعة البتات هذه هي الجزء المعنوي لعدد عشري وكان علينا توحيدها، فسنعرف من قيمة الأس عدد مرات إزاحتها، كما تضمَّن عملية التوحيد عمومًا في وحدة عتاد العدد العشري على المعالج، لـذا تؤدَّى بسرعة كبيرة، وعادةً أسرع بكثير من عمليات الإزاحة والاختبار المتكررة.

يوضِّح البرنامج التالي طريقتين للعثور على أول بِتّ ضبط متَّبعتَين على معالج إتانيوم. إذ يدعم المعالج إتانيوم -مثل حال معظم معالجات الخوادم- نوع العدد العشري الموسَّع الذي يخزَّن على 80 بِتّ، والجزء المعنوي الذي يخزن على 64 بِتّ، ويعني هذا أنّ نوع unsigned long يتوافق بدقة في الجزء المعنوي لنوع long double، فعندما تحمَّل القيمة توحَّد، وبالتالي من خلال قراءة قيمة الأس مطروحًا منها انحياز 16 بِتّ يمكننا رؤية مدى انزياحها.

```
#include <stdio.h>

int main(void)
{
        // 1000 0000 0000 0000 = في التمثيل الثنائي
        // 5432 1098 7654 3210  عدد البتات
        int i = 0x8000;
        int count = 0;
        while ( !(i & 0x1) ) {
        count ++;
              i = i >> 1;
        }
        printf("First non-zero (slow) is %d\n", count);
```

```
        // توحَّد هذه القيمة عندما تُحمَّل
        long double d = 0x8000UL;
        long exp;

        // تعليمات "الحصول على أس العدد العشري" في معالج إتانيوم
        asm ("getf.exp %0=%1" : "=r"(exp) : "f"(d));

        // الأس متضمنًا الانزياح
        printf("The first non-zero (fast) is %d\n", exp - 65535);

}
```

خلاصة الأفكار السابقة

نستخرج مكونات العدد العشري ونطبع القيمة التي يمثلها في نموذج الشيفرة البرمجيـة التاليـة، إذ سـنحرز نتيجـةً فقـط عنـدما تكـون القيمـة عـددًا عشـريًا بحجم 32 بِتّ بصـيغة المعيـار IEEE، وهـذا شـائع في معظم المعماريات من نوع `float` أي عدد عشري.

```
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>

    /* 2^n إرجاع */
    int two_to_pos(int n)
    {
        if (n == 0)
            return 1;
return 2 * two_to_pos(n - 1);
    }

    double two_to_neg(int n)
    {
        if (n == 0)
            return 1;
        return 1.0 / (two_to_pos(abs(n)));
    }
```

```
	double two_to(int n)
	{
		if (n >= 0)
			return two_to_pos(n);
		if (n < 0)
			return two_to_neg(n);
		return 0;
	}

	/* مراجعة بعض أجزاء الذاكرة للمتغير "m" الذي هو الجزء المعنوي
	 للعدد العشري بحجم 24 بت، نبدأ بالمقلوب من البتات في أقصى اليمين
	 دون أي سبب معين */
	double calc_float(int m, int bit)
	{
		/*  23 بت؛ هذا ينهي العودية */
		if (bit > 23)
			return 0;

		/* بت^2/1 إذا كان البت مضبوطًا، فهو يمثل القيمة  */
		if ((m >> bit) & 1)
			return 1.0L/two_to(23 - bit) + calc_float(m, bit + 1);

		/* وإلا انتقل إلى البت التالي */
		return calc_float(m, bit + 1);
	}

	int main(int argc, char *argv[])
	{
		float f;
		int m,i,sign,exponent,significand;

		if (argc != 2)
		{
			printf("usage: float 123.456\n");
			exit(1);
}
```

```
if (sscanf(argv[1], "%f", &f) != 1)
{
      printf("invalid input\n");
      exit(1);
}

/* سنحتاج إلى خداع المصرف، كأننا بدأنا استخدام التحويلات
   فمثلًا (f)(int) ستجري تحويلًا فعليًا لنا
   نريد الوصول إلى البتات الأولية، لذا ننسخها إلى متغير
   بنفس الحجم. */
memcpy(&m, &f, 4);

/* بت الإشارة هو أول بت */
sign = (m >> 31) & 0x1;

/* الأس هو البتات الثمانية التي تلي بت الإشارة */
exponent = ((m >> 23) & 0xFF) - 127;

/* الجزء المعنوي يملأ المنازل العشرية، ويكون أول بت ضمنيًا 1
   بت. وبالتالي هو قيمة المعامل OR 24 */
significand = (m & 0x7FFFFF) | 0x800000;

/* اطبع قيمةً تمثل الأس */
printf("%f = %d * (", f, sign ? -1 : 1);
for(i = 23 ; i >= 0 ; i--)
{
      if ((significand >> i) & 1)
            printf("%s1/2^%d", (i == 23) ? "" : " + ",
               23-i);
}
printf(") * 2^%d\n", exponent);

/* اطبع تمثيلًا كسريًا */
printf("%f = %d * (", f, sign ? -1 : 1);
for(i = 23 ; i >= 0 ; i--)
```

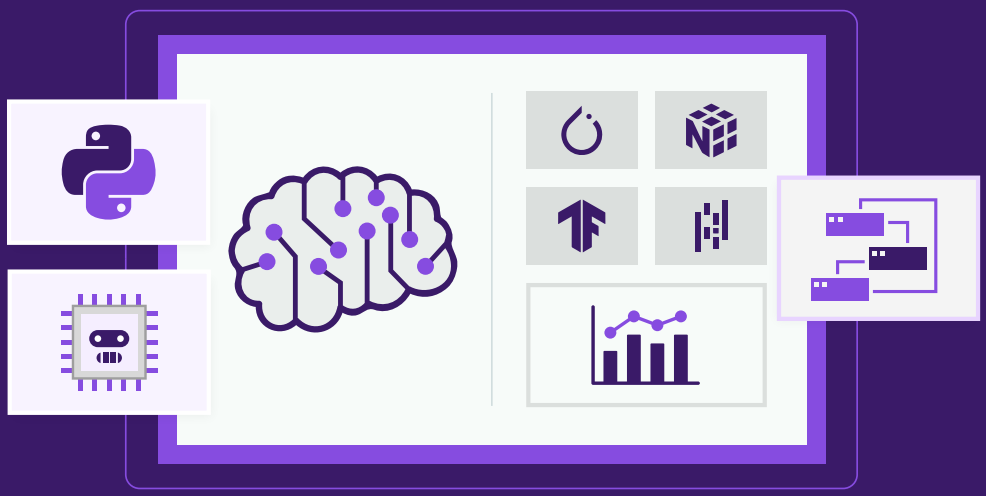
```
        {
              if ((significand >> i) & 1)
                    printf("%s1/%d", (i == 23) ? "" : " + ",
                       (int)two_to(23-i));
        }
        printf(") * 2^%d\n", exponent);

        /* حول هذا إلى قيمة عشرية واطبعه */
        printf("%f = %d * %.12g * %f\n",
               f,
               (sign ? -1 : 1),
               calc_float(significand, 0),
               two_to(exponent));

        /* اجرِ العملية الحسابية الآن */
        printf("%f = %.12g\n",
               f,
               (sign ? -1 : 1) *
               calc_float(significand, 0) *
               two_to(exponent)
              );

        return 0;
}
```

وفيما يلي نموذج خرج القيمة 8.45 الذي درسناه سابقًا:

```
$ ./float 8.45
8.450000 = 1 * (1/2^0 + 1/2^5 + 1/2^6 + 1/2^7 + 1/2^10 + 1/2^11 +
1/2^14 + 1/2^15 + 1/2^18 + 1/2^19 + 1/2^22 + 1/2^23) * 2^3
8.450000 = 1 * (1/1 + 1/32 + 1/64 + 1/128 + 1/1024 + 1/2048 + 1/16384
+ 1/32768 + 1/262144 + 1/524288 + 1/4194304 + 1/8388608) * 2^3
8.450000 = 1 * 1.05624997616 * 8.000000
8.450000 = 8.44999980927
```

نستخلص من هذا المثال فكرةً عن تسلل عدم الدقة إلى أعدادنا العشرية.

دورة الذكاء الاصطناعي



مميزات الدورة

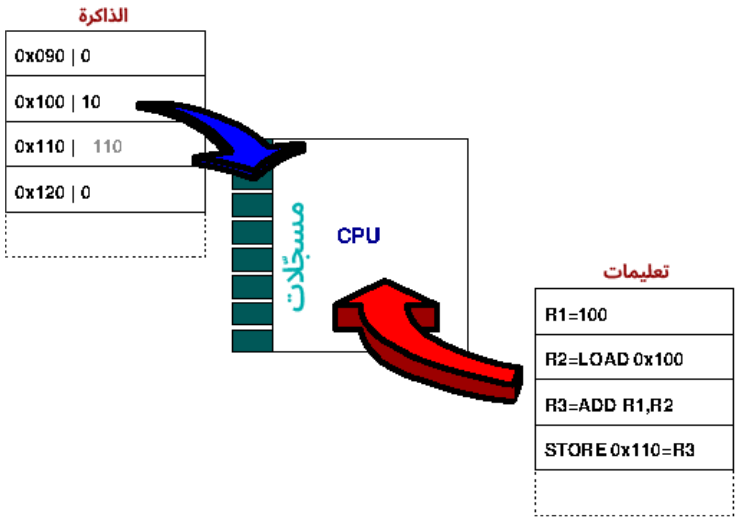
- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

3. معمارية الحاسوب

3.1 تعرف على وحدة المعالجة المركزية وعملياتها

تنفّذ وحدة المعالجة المركزية التعليمات على القيم الموجودة في المسجّلات Registers، إذ يوضّـح المثـال الآتي أولًا ضبط R1 على القيمة 100 وتحميل القيمة من موقع الذاكرة 0x100 إلى R2 وجمع القيمتين، ثم وضع النتيجة في R3، وأخيرًا تخزين القيمة الجديدة 110 في R4.



شكل 7: وحدة المعالجة المركزية CPU

يتكون الحاسوب من وحدة معالجة مركزية Central Processing Unit -أو CPU اختصارًا- متصلة بالذاكرة، إذ توضّح الصورة السابقة المبدأ العام لجميع عمليات الحاسوب، كما تنفّـذ وحـدة المعالجـة المركزيـة التعليمـات المقروءة من الذاكرة، وهناك نوعان من هذه التعليمات هما:

1. التعليمات التي تحمّل القيم من الذاكرة إلى المسجلات وتخزّن القيم من المسجلات إلى الذاكرة.
2. التعليمات التي تُشغَّل على القيم المخزَّنة في المسجّلات مثل جمع أو طرح أو ضرب أو قسمة قيمتين موجودتين في مسجلين، أو إجراء العمليات الثنائية `and` و `or` و `xor` وغيرها، أو إجراء عمليات حسابية أخرى، مثل الجذر التربيعي و sin و cos و tan وغيرها.

لذا نجمع في مثالنا ببساطة العدد 100 مع قيمة مُخزَّنة في الذاكرة ونخزّن النتيجة الجديدة في الذاكرة.

3.1.1 التفريع Branching

يُعَدّ التفريع عمليةً مهمةً لوحدة المعالجة المركزية، وذلك بغض النظر عن عمليتي التحميل أو التخزين، إذ تحتفظ وحدة المعالجة المركزية داخليًا بسجل للتعليمة التالية التي ستنفَّذ في مؤشر التعليمات Instruction Pointer، بحيث يُزاد هذا المؤشر ليؤشّر إلى التعليمة التالية تسلسليًا، إذ ستتحقق التعليمة الفرعية مما إذا كان لمسجل معيّن القيمة صفر، أو تتحقق من وجود من ضبط راية flag ما. فإذا كان الأمر كذلك، فسيُعدَّل المؤشر ليؤشّر إلى عنوان مختلف، وبالتالي ستكون التعليمة التالية للتنفيذ من جزء مختلف من البرنامج، وهذه هي الطريقة التي تعمل بها الحلقات وتعليمات القرار.

يمكن مثلًا تنفيذ التعليمة `if (x==0)` من خلال إيجاد ناتج تطبيق عملية `or` على اثنين من المسجلات، أحدهما يحمل القيمة `x` والآخر يحمل القيمة صفر، فإذا كانت النتيجة صفرًا، فستكون المقارنة صحيحة، أي أنّ جميع بتات `x` أصفار ويجب تنفيذ جسم التعليمة، وإلّا فستتجاوز التعليمة الفرعية هذه الشيفرة.

3.1.2 الدورات

جميعنا على دراية بسرعة الحاسوب المعطاة بالميجاهرتز Megahertz أو الجيجاهرتز Gigahertz التي تقابل ملايين أو آلاف الملايين من الدورات في الثانية، ويسمى ذلك بسرعة الساعة Clock Speed لأنها السرعة التي تنبض بها ساعة الحاسوب الداخلية، إذ تُستخدَم النبضات ضمن المعالج لإبقائه متزامنًا داخليًا، ويمكن البدء بعملية أخرى في كل لحظة أو نبضة.

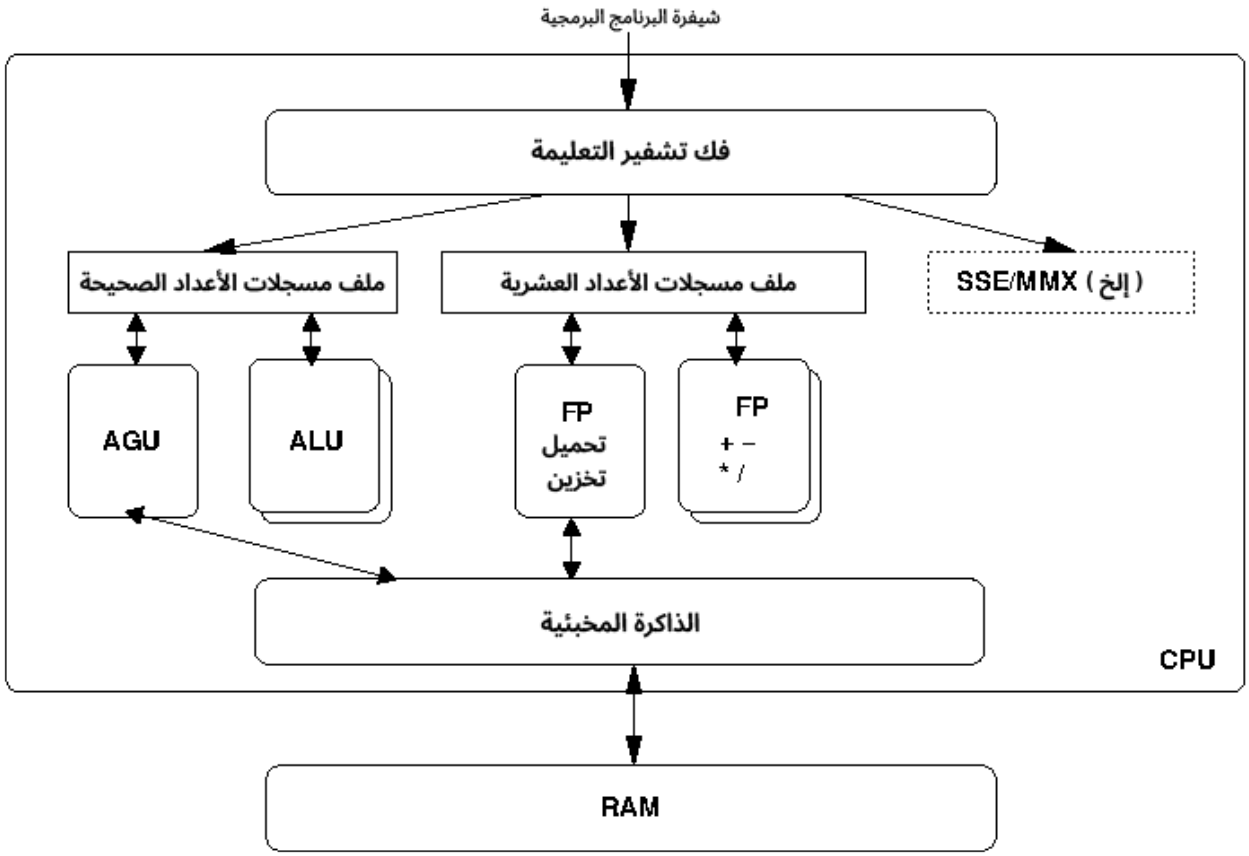
3.1.3 جلب التعليمة وفك تشفيرها وتنفيذها وتخزين نتيجتها

يتكون تنفيذ تعليمة واحدة من دورة معينة من الأحداث، وهي الجلب وفك التشفير والتنفيذ والتخزين، إذ يجب على وحدة المعالجة المركزية تطبيق الخطوات التالية لتنفيذ تعليمة `add` السابقة مثلًا:

1. الجلب Fetch: الحصول على التعليمات من الذاكرة إلى المعالج.
2. فك التشفير Decode: فك تشفير ما يجب أن تفعله داخليًا، أي الجمع في هذه الحالة.
3. التنفيذ Execute: أخذ القيم من المسجلات وجمعها.
4. التخزين Store: تخزين النتيجة في مسجل آخر، كما يمكن رؤية مصطلح انتهاء Retiring التعليمة.

ا. نظرة داخلية إلى وحدة المعالجة المركزية

تحتوي وحدة المعالجة المركزية داخليًا على العديد من المكونات الفرعية المختلفة التي تطبّق كلًا من الخطوات المذكورة سابقًا، كما يمكن أن تحدث جميعها بصورة مستقلة عن بعضها البعض، وهي مشابهة لخط الإنتاج في المصانع، حيث توجد العديد من المحطات ولكل خطوة مَهمة معينة لأدائها، ثم يمكنه تمرير النتائج إلى المحطة التالية وأخذ مدخلات جديدة للعمل عليها.



شكل 8: داخل CPU

تتكون وحدة المعالجة المركزية من العديد من المكونات الفرعية المختلفة، وتطبّق كل منها مهمةً مُخصَّصةً، وتوضِّح الصورة السابقة مخططًا بسيطًا لبعض الأجزاء الرئيسية لوحدة المعالجة المركزية الحديثة، حيث يمكنك رؤية التعليمات تأتي ثم يفك المعالج تشفيرها؛ كما تحتوي وحدة المعالجة المركزية على نوعين رئيسيين من المسجّلات، هما مسجلات العمليات الحسابية الخاصة بالأعداد الصحيحة ومسجلات العمليات الحسابية الخاصة بالأعداد العشرية.

تُعَدّ الأعداد العشرية Floating Point طريقةً لتمثيل الأعداد ذات المنزلة العشرية بصيغة ثنائية، ويجري التعامل معها بطريقة مختلفة ضمن وحدة المعالجة المركزية، كما تُعَدّ المسجلات MMX (توسع الوسائط المتعددة Multimedia Extension) و SSE (مجرى بيانات متعددة لتعليمة مفردة Single Streaming Instruction Multiple Data) أو Altivec مسجلات مماثلة للمسجلات الخاصة الأعداد العشرية.

يُعَدّ ملف المسجلات Register File اسمًا يجمع جميع المسجلات الموجودة ضمن وحدة المعالجة المركزية، وتوجد ضمنه أجزاء وحدة المعالجة المركزية التي تنفّذ كل العمل، إذ تحمّل المعالجات أو تخزّن قيمةً في مسجل أو من مسجل إلى الذاكرة، أو تنفّذ بعض العمليات على القيم الموجودة في المسجلات كما ذكرنا سابقًا.

تُعَدّ وحدة الحساب والمنطق Arithmetic Logic Unit -أو ALU اختصارًا- قلب عمليات وحدة المعالجة المركزية، إذ تأخذ القيم من المسجلات وتنفّذ أيًا من العمليات المتعددة التي تستطيع وحدة المعالجة المركزية تنفيذها، كما تحتوي جميع المعالجات الحديثة على عدد من وحدات ALU، بحيث يمكن لكل منها العمل بصورة مستقلة، وتحتوي المعالجات مثل المعالج بنتيوم Pentium على وحدات ALU سريعة ووحدات ALU بطيئة، إذ تكون الوحدات السريعة أصغر حجمًا، لذا يمكنك استخدام المزيد منها على وحدة المعالجة المركزية، ولكن يمكنك تنفيذ العمليات الأكثر شيوعًا فقط؛ أما وحدات ALU البطيئة، فيمكنها تنفيذ جميع العمليات ولكنها تكون أكبر حجمًا.

تعالِج وحدة إنشاء العناوين Address Generation Unit -أو AGU اختصارًا- التواصل مع الذاكرة المخبئية Cache Memory والذاكرة الرئيسية لجلب القيم إلى المسجلات لكي تعمل وحدة ALU، ثم استعادة القيم من المسجلات إلى الذاكرة الرئيسية، كما تحتوي مسجلات الأعداد العشرية على المفاهيم نفسها، ولكنها تستخدِم مصطلحات مختلفةً قليلًا لمكوناتها.

ب. استخدام خط الأنابيب Pipelining

تُعَدّ عملية وحدة ALU التي تجمع قيم المسجلات منفصلةً تمامًا عن عملية وحدة AGU التي تكتب القيم في الذاكرة، إذ لا يوجد سبب يمنع وحدة المعالجة المركزية من تطبيق هاتين العمليتين معًا في وقت واحد، كما توجد عدة وحدات ALU في النظام والتي يمكن أن تعمل كل منها على تعليمات منفصلة.

يمكن لوحدة المعالجة المركزية تنفيذ بعض عمليات الأعداد العشرية باستخدام منطق الأعداد العشرية أثناء تشغيل تعليمات الأعداد الصحيحة أيضًا، إذ تسمى هذه العملية باستخدام خط الأنابيب Pipelining، ويشار إلى المعالج الذي يمكنه تطبيق هذه العملية بأن له معمارية عددية فائقة Superscalar Architecture، إذ تُعَدّ جميع المعالجات الحديثة معالجات عدديةً فائقةً، ويحتوي أيّ معالج حديث على أكثر من أربع مراحل يمكنه استخدامها ضمن خط أنابيب، وكلما زاد عدد المراحل التي يمكن تنفيذها في الوقت نفسه، زاد عمق خط الأنابيب.

يمكن تشبيه خط الأنابيب بأنبوب مملوء بكرات زجاجية، باستثناء أن هذه الكرات هي تعليمات وحدة المعالجة المركزية، إذ ستضع الكرات الزجاجية في نهاية واحدة، بحيث تضعها واحدةً تلو الأخرى -أي كرة لكل نبضة ساعة- حتى تملأ الأنبوب، وستنتقل كل كرة زجاجية -أو تعليمة- تدفعها للداخل إلى الموضع التالي بمجرد أن يمتلئ الأنبوب مع سقوط كرة في النهاية التي تمثّل النتيجة.

تؤدي التعليمات الفرعية إلى إحداث فوضى في هذا النموذج، إذ يمكن أن تتسبب أو لا تتسبب في بدء التنفيذ من مكان مختلف، فإذا أردت استخدام خط الأنابيب، فسيتعين عليك تخمين الاتجاه الذي ستتجه فيه التعليمة الفرعية حتى تعرف التعليمات التي يجب إحضارها إلى خط الأنابيب، فإذا خمّنت وحدة المعالجة المركزية ذلك بصورة صحيحة، فسيسير كل شيء على ما يرام، إذ تستخدِم المعالجات مثل معالج بنتيوم ذاكرة تخزين مؤقت Trace Cache لتعقب مسار التعليمات الفرعية، حيث يمكن في كثير من الأحيان أن تخمّن الطريق الذي ستذهب إليه التعليمة الفرعية من خلال تذكر نتائجها السابقة، فإذا تذّكرتَ نتيجة التعليمة الفرعية الأخيرة في حلقة تتكرر 100 مرة مثلًا، فستكون على صواب 99 مرة، لأن المرة الأخيرة فقط ستستمر في البرنامج فعليًا؛ بينما إذا جرى تخمين المعالج بطريقة غير صحيحة، فهذا يعني أنّ المعالج قد أهدر كثيرًا من الوقت ويجب عليه مسح خط الأنابيب والبدء من جديد.

يشار إلى هذه العملية عادةً باسم تفريغ خط الأنابيب Pipeline Flush وهي مماثلة للحاجة إلى التوقف وإفراغ كل الكرات من الأنبوب، كما تتكوّن عملية تخمين التعليمة الفرعية Branch Prediction من تفريغ خط الأنابيب وأخذ التخمين أو عدم الأخذ به وفتحات تأخير التعليمة الفرعية Branch delay slots.

ج. إعادة الترتيب

إذا كانت وحدة المعالجة المركزية هي الأنبوب، فسنكون لك الحرية في إعادة ترتيب الكرات ضمنه طالما أنها تخرج من نهايته بالترتيب نفسه الذي وضعتَها فيه، إذ نسمي ذلك بترتيب البرنامج Program Order لأنه ترتيب التعليمات المُعطَى في البرنامج الحاسوبي، كما يمكنك الاطلاع على المثال التالي الذي يمثل إعادة ترتيب المخزن المؤقت Buffer:

```
r3 = r1 * r2
r4 = r2 + r3
r7 = r5 * r6
r8 = r1 + r7
```

افترض مجرى التعليمات الموضح سابقًا، إذ يجب على التعليمة 2 انتظار اكتمال التعليمة 1 قبل أن تبدأ، وهذا يعني أنّ خط الأنابيب يجب عليه التوقف أثناء انتظار القيمة المراد حسابها، كما تعتمد التعليمتان 3 و 4 على قيمة r7، ولكن التعليمتان 2 و 3 لا تعتمدان على بعضهما البعض أبدًا، وهذا يعني أنهما يعملان في مسجلات منفصلة تمامًا، فإذا بدّلنا بين التعليمتين 2 و 3، فسنحصل على ترتيب أفضل لخط الأنابيب، إذ يمكن أن ينفّذ المعالج عملًا مفيدًا بدلًا من انتظار اكتمال خط الأنابيب للحصول على نتيجة التعليمة السابقة.

يمكن أن تتطلب التعليمات بعض الأمان حول كيفية ترتيب العمليات عند كتابة شيفرة منخفضة المستوى، إذ نطلق على هذا المتطلب دلالات الذاكرة Memory Semantics، فإذا أردت اكتساب الدلالات Acquire Semantics، فهذا يعني أنه يجب عليك التأكد من إكمال نتائج جميع التعليمات السابقة للتعليمة

الحالية، وإذا أردت تحرير الدلالات Release Semantics، فهذا يعني أنّ جميع التعليمات بعد هذه التعليمة يجب أن ترى النتيجة الحالية.

توجد دلالات أخرى أكثر صرامة وهي حاجز الذاكرة Memory Barrier أو سور الذاكرة Memory Fence الذي يتطلب أن تكون العمليات مرتبطةً بالذاكرة قبل المتابعة، كما يضمن المعالج هذه الدلالات في بعض المعماريات، بينما يجب أن تحددها بصورة صريحة في المعماريات الأخرى، ولا يحتاج معظم المبرمجين إلى القلق بشأنها على الرغم من أنك قد تصادفها.

3.1.4 معمارية CISC ومعمارية RISC

يمكن تقسيم معماريات الحاسوب إلى معمارية حاسوب مجموعة التعليمات المعقدة Complex Instruction Set Computer -أو CISC اختصارًا- ومعمارية حاسوب مجموعة التعليمات المُخفَّضة Reduced Instruction Set Computer أو RISC اختصارًا.

لاحظ أننا في المثال الأول من مقالنا حمّلنا القيم صراحةً في المسجلات وأجرينا عملية الجمع، ثم خزّنا القيمة الناتجة المحفوظة في مسجل آخر في الذاكرة، إذ يُعَدّ ذلك مثالًا عن نهج RISC للحوسبة الذي يشمل تنفيذ العمليات على القيم الموجودة في المسجلات وتحميل القيم وتخزينها بصورة صريحة من الذاكرة وإليها، كما يمكن أن يكون نهج CISC مجرد تعليمات مفردة تأخذ قيمًا من الذاكرة وتنفذ عملية الجمع داخليًا ثم تكتب النتيجة، وهذا يعني أنّ التعليمات يمكن أن تستغرق عدة دورات، ولكن كلا النهجين يحققان في النهاية الهدف نفسه.

تُعَدّ جميع المعماريات الحديثة معماريات RISC حتى معمارية إنتل بنتيوم Intel Pentium الأكثر شيوعًا والتي تهدم التعليمات داخليًا إلى تعليمات فرعية بأسلوب RISC داخل الشريحة قبل التنفيذ، بالرغم من وجود مجموعة تعليمات مصنَّفة على أنها CISC، وهناك عدة أسباب لذلك وهي:

- تجعل معمارية RISC البرمجة بلغة التجميع Assembly أكثر تعقيدًا، نظرًا لأن جميع المبرمجين تقريبًا يستخدِمون لغات عالية المستوى ويتركون العمل الشاق لإنتاج شيفرة التجميع للمصرّف Compiler، وبالتالي ستتفوق المزايا الأخرى على هذا العيب.
- بما أنّ التعليمات الموجودة في معالج RISC أبسط، فهناك مساحة أكبر ضمن شريحة المسجلات، إذ تُعَدّ المسجلات أسر ع أنواع الذواكر كما نعلم من تسلسل الذواكر الهرمي، ويجب في النهاية تنفيذ جميع التعليمات على القيم المحفوظة في المسجلات، لذا ستؤدي زيادة عدد المسجلات إلى أداء أعلى عند تكافؤ جميع الأشياء الأخرى.
- بما أنّ جميع التعليمات تُنفَّذ في الوقت نفسه، فسيكون استخدام خطوط الأنابيب ممكنًا، وكما نعلم أنّ استخدام خط الأنابيب يتطلب تدفقات من التعليمات باستمرار إلى المعالج، لذلك إذا استغرقت بعض

التعليمات وقتًا طويلًا جدًا دون أن تتطلب التعليمات الأخرى ذلك، فسيصبح خط الأنابيب معقدًا | ليكون فعّالًا.

ا. معمارية EPIC

يُعَدّ معالج إيتانيوم Itanium مثالًا على معمارية معدَّلة تسمى الحوسبة الصريحة للتعليمات الفرعية Explicitly Parallel Instruction Computing.

ناقشنا سابقًا كيف أنّ المعالجات الفائقة لها خطوط أنابيب بها العديد من التعليمات في الوقت نفسه ضمن أجزاء مختلفة من المعالج، إذ يمكن تحقيق ذلك من خلال إعطاء التعليمات للمعالج بالترتيب الذي يمكن أن يحقق أفضل استفادة من العناصر المتاحة في وحدة المعالجة المركزية، وقد كان تنظيم مجرى التعليمات الواردة تقليديًا مهمة العتاد، إذ يصدر البرنامج التعليمات بطريقة تسلسلية، ويجب أن ينظر المعالج إلى الأمام ويحاول اتخاذ قرارات حول كيفية تنظيم التعليمات الواردة.

الفكرة وراء معمارية EPIC هي أنّ هناك مزيد من المعلومات المتاحة على مستويات أعلى والتي يمكن أن تجعل هذه القرارات أفضل مما يفعله المعالج، ويؤدي تحليل مجرًى من تعليمات لغة التجميع -كما تفعل المعالجات الحالية- إلى فقدان الكثير من المعلومات التي قدّمها المبرمج في الشيفرة البرمجية الأصلية.

فكر في الأمر على أنه الفرق بين دراسة مسرحية لشكسبير وقراءة نسخة ملاحظات الجرف Cliff's Notes من المسرحية نفسها، فكلاهما يمنحك النتيجة نفسها، ولكن النسخة الأصلية تحتوي على جميع أنواع المعلومات الإضافية التي تحدد المشهد وتعطيك فهمًا جيدًا للشخصيات، وبالتالي يمكن نقل منطق ترتيب التعليمات من المعالج إلى المصرّف، وهذا يعني أنّ مطوّري المصرّفات يجب أن يكونوا أذكى في محاولة العثور على أفضل ترتيب للشيفرة البرمجية للمعالج، كما يجب تبسيط المعالج كثيرًا، إذ نُقِل الكثير من عمله إلى المصرِّف.

يوجد مصطلح آخر غالبًا ما يُستخدَم مع معمارية EPIC وهو عالم التعليمات الطويلة جدًا Very Long Instruction World -أو VLIW اختصارًا-، إذ تُوسَّع كل تعليمة للمعالج لإخباره بالمكان الذي يجب أن ينفّذ فيه التعليمة في وحداته الداخلية، وتكمن مشكلة هذا الأسلوب في أنّ الشيفرة البرمجية تعتمد كليًا على طراز المعالج الذي صُرِّفت الشيفرة البرمجية من أجله، كما تُجري الشركات دائمًا مراجعات على العتاد، وتجعل العملاء يعيدون تصريف تطبيقاتهم في كل مرة، مما جعل صيانة مجموعة من الشيفرات البرمجية الثنائية المختلفة أمرًا غير عملي.

تحل معمارية EPIC هذه المشكلة بطريقة علوم الحاسوب المعتادة من خلال إضافة طبقة من التجريد، كما تنشئ معمارية EPIC عرضًا مبسطًا مع بعض الوحدات الأساسية مثل الذاكرة ومسجّلات الأعداد الصحيحة والعشرية بدلًا من التحديد الصريح للجزء الدقيق من المعالج الذي يجب أن تنفّذ التعليمات عليه.

3.2 تعرف على ذاكرة الحاسوب

يمكن لوحدة المعالجة المركزية جلب التعليمات والبيانات مباشرةً من الـذاكرة المخبئيـة Cache Memory الموجودة على شريحة المعالج فقط، لـذا يجب تحميـل الـذاكرة المخبئيـة من ذاكـرة النظـام الرئيسـية، أي ذاكـرة الوصول العشوائي Random Access Memory -أو RAM اختصارًا-، ولكن تحتفظ الـذاكرة RAM بمحتوياتهـا فقط عند الوصل بمصدر طاقة، لذلك يجب تخزينها على مساحة تخزين دائمة وغير متطايرة.

3.2.1 تسلسل الذواكر الهرمي

نطلق على طبقات الذواكر التالية اسم تسلسل الذواكر الهرمي Memory Hierarchy:

السرعة	الذاكرة	الوصف
الأسرع	الذاكرة المخبئية Cache	الذاكرة المخبئية هي ذاكرة مضمَّنة في وحدة المعالجة المركزية، وهي ذاكرة سريعة جدًا وتستغرق دورة واحدة فقط للوصول إليها، ولكن هناك حد لحجمها لأنها مُدمَجة مباشرةً في وحدة المعالجة المركزية، كما توجد هناك عدة مستويات فرعية من الذاكرة المخبئية تسمى L1 و L2 و L3 بسرعات متزايدة قليلًا عن بعضها البعض.
الذاكرة RAM		يجب أن تأتي جميع التعليمات وعناوين التخزين الخاصة بالمعالج من الذاكرة RAM، وتستغرق وحدة المعالجة المركزية بعض الوقت للوصول إلى الذاكرة RAM يسمى زمن التأخير Latency بالرغم من أنها ذاكرة سريعة جدًا، كما تُخزَّن الذاكرة RAM في شرائح منفصلة ومخصصة متصلة باللوحة الأم، مما يعني أنها أكبر بكثير من الذاكرة المخبئية.
الأبطأ	القرص الصلب Disk	جميعنا على دراية بالبرامج التي تصلنا على قرص مرن floppy disk أو قرص مضغوط، ونعلم كيفية حفظ ملفاتنا على القرص الصلب، ونعلم الوقت الطويل الذي يمكن أن يستغرقه البرنامج للتحميل من القرص الصلب، إذ يعني وجود آليات فيزيائية مثل الأقراص الدوارة والرؤوس المتحركة أن الأقراص الصلبة هي أبطأ وسيلة من وسائل التخزين، ولكنها أكبرها حجمًا.

جدول 23: تسلسل الذواكر الهرمي

> تجدر الإشارة لأننا نستخدم مصطلح القرص الصلب HDD في سياق هذا الكتاب مع العلم بأن قرص الحالة الصلبة SSD يستخدم اليوم في معظم الحواسيب وهو أحدث وأسرع بكثير.

النقطة المهمة التي يجب معرفتها حول تسلسل الذواكر الهرمي هي المقايضـات بين السـرعة والحجم على حساب بعضهما البعض، فكلما كانت الذاكرة أسرع، كان حجمها أصغر.

سـبب فعاليـة الـذواكر المخبئيـة هـو أنّ شـيفرة الحاسـوب البرمجيـة تعـرض شـكلَين من أشـكال المحليـة Locality هما:

1. تشير المحلية المكانية Spatial Locality إلى احتمالية الوصول إلى البيانات الموجودة ضمن الكتل مع بعضها بعضًا.
2. تشير المحلية الزمانية Temporal Locality إلى أن البيانات المستخدَمة مؤخرًا يُحتمَل أن تُستخدَم مرة أخرى قريبًا.

يعني ذلك أنه يمكن الاسـتفادة من تنفيـذ أكـبر قـدر ممكن من عمليـات الوصـول السـريعة إلى الـذاكرة أي المحلية الزمانية وتخزين كتل صغيرة من المعلومات ذات الصلة أي المحلية المكانية.

3.2.2 نظرة معمقة على الذاكرة المخبئية

تُعَدّ الذاكرة المخبئية أحد أهم عناصر معمارية وحدة المعالجـة المركزيـة، إذ يجب على المطـورين فهم كيفيـة عمل الذاكرة المخبئية في أنظمتهم لكتابة شيفرة برمجية فعالـة، كمـا تُعَـدّ نسـخةً سـريعةً جـدًا من ذاكـرة النظـام الرئيسية الأبطأ، وهي أصغر بكثير من الذواكر الرئيسية لأنها مضـمنة داخـل شـريحة المعـالج جنبًـا إلى جنب مـع المسجلات ومنطق المعالج، وهناك حدود اقتصادية ومادية لأقصى حجم لها.

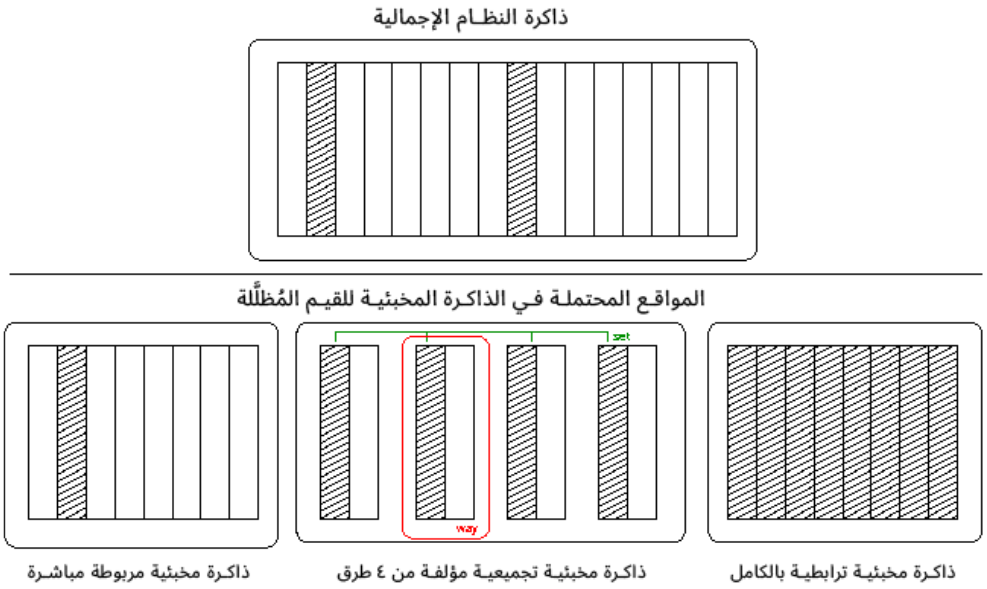
تجد الشركات المصنعة مزيدًا من الطرق لحشر مزيد من الترانزستورات على الشريحة، مما يـؤدي إلى زيـادة أحجام الذواكر المخبئية بصورة كبيرة، ولكن يُقدَّر حجم حتى أكبر الذواكر المخبئيـة بعشـرات الميجابايتـات بعكس حجم الذاكرة الرئيسية المقدَّر بالجيجابايتات أو حجم القرص الصلب المقدَّر بالتيرابايتات.

تتكون الذاكرة المخبئية من قطع صغيرة تعكس محتوى أجزاء من الذاكرة الرئيسية، إذ يُطلَـق على حجم هـذه القطع بحجم الخط Line Size، ويسـاوي تقريبًـا 32 أو 64 بـايت، ومن الشـائع التحـدث عن حجم الخـط أو خـط الذاكرة المخبئية عند الحديث عن الـذاكرة المخبئيـة، والـذي يشـير إلى قطعـة واحـدة تعكس محتـوى قطعـة من الذاكرة الرئيسية، كما يمكن للـذاكرة المخبئيـة فقـط تحميـل وتخـزين الـذاكرة بأحجـام مضـاعفة من خـط الـذاكرة المخبئية.

تحتوي الذواكر المخبئية على تسلسلها الهـرمي الخـاص، ويطلـق عليـه عـادةً L1 و L2 و L3، إذ تُعَـدّ الـذاكرة المخبئية L1 هي الأسر ع والأصغر و L2 أكبر وأبطأ منها و L3 هي الأكبر والأبطأ، كمـا تُقسَـم الـذاكرة المخبئيـة L1 إلى ذواكر مخبئية خاصة بالتعليمات وأخرى بالبيانات، وتُعرف باسم معمارية هارفـارد Harvard Architecture بعد أن قدمها حاسوب Harvard Mark-1 القائم على المُرحّلات Relay.

تساعد الذواكر المخبئية المقسـمة على تقليـل الاختناقـات في خطـوط الأنـابيب، حيث تشـير مراحـل خـط الأنابيب السابقة إلى تعليمات الذاكرة المخبئية وتشير المراحل اللاحقة إلى بيانات الذاكرة المخبئيـة، كمـا يسـمح توفير ذاكرة مخبئية منفصلة للتعليمات بإجراء تطبيقات بديلة تستفيد من طبيعة مجرى التعليمات بغض النظـر

عن فائدة تقليل التنازع على مورد مشترك، إذ تكون الذاكرة المخبئية الخاصة بالتعليمات للقراءة فقط، أي لا تحتاج إلى ميزات باهظة الثمن على الشريحة مثل تعدد المنافذ، ولا تحتاج إلى التعامل مع عمليات قراءة الكتل الفرعية لأن مجرى التعليمات يستخدِم عمومًا عمليات وصول ذات أحجام أكثر انتظامًا.



شكل 9: ترابط الذاكرة المخبئية.

يوضح الشكل السابق ترابط الذاكرة المخبيئة حيث يمكن أن يجد خط ذاكرة مخبئية معيّن مكانًا صالحًا في أحد الإدخالات المظللة.

يطلب المعالج باستمرار من الذاكرة المخبئية أثناء التشغيل العادي التحقق من تخزين عنوان معيّن في الذاكرة المخبئية، لذلك تحتاج الذاكرة المخبئية لطريقة ما لمعرفة ما إذا كان لديها خط صالح أم لا، فإذا أمكن تخزين عنوان معيّن في أيّ مكان ضمن الذاكرة المخبئية، فيجب البحث في كل خط من الذاكرة المخبئية في كل مرة يُنشَأ فيها مرجع لتحديد وصول صحيح أو خاطئ، كما يمكن الاستمرار في البحث السريع من خلال إجرائه على التوازي في عتاد الذاكرة المخبئية، ولكن يكون البحث في كل إدخال مكلفًا للغاية بحيث يتعذر تطبيقه في ذاكرة مخبئية ذات حجم معقول، لذا يمكن جعل الذاكرة المخبئية أبسط من خلال فرض قيود على مكان وجود عنوان معيّن.

يُعَدّ ذلك مقايضةً، فالذاكرة المخبئية أصغر بكثير من ذاكرة النظام، لذا يجب أن تحمل بعض العناوين أسماء بديلة Alias للعناوين الأخرى، فإذا جرى تحديث عنوانَين يحملان أسماء بديلةً لبعضهما البعض باستمرار، فسيقال أنهما يتنازعان على خط الذاكرة المخبئية، كما يمكننا تصنيف الذواكر المخبئية إلى ثلاثة أنواع عامة كما هو موضح في الشكل السابق وهي:

- **الذواكر المخبئية المربوطة مباشرةً Direct mapped Caches** التي تسمح لخط الذاكرة المخبئية بالتواجد فقط في إدخال واحد في الذاكرة المخبئية، ويُعَدّ ذلك أبسط تطبيق في العتاد، ولكن -كما هو

موضح في الشكل السابق- لا توجد إمكانية لتجنب استخدام الأسماء البديلة لأن العنوانَين المظلَّلين يجب عليهما التشارك في خط الذاكرة المخبئية نفسه.

- **الذواكر المخبئية الترابطية بالكامل Fully Associative Caches** التي تسمح بوجود خط الذاكرة المخبئية في أيّ إدخال منها، مما يؤدي إلى تجنّب مشكلة الأسماء البديلة، لأن أيّ إدخال يكون متاحًا للاستخدام، لكن يُعَدّ تطبيق ذلك في العتاد مكلفًا للغاية لأنه يجب البحث عن كل موقع محتمَل في الوقت نفسه لتحديد ما إذا كانت القيمة موجودةً في الذاكرة المخبئية.
- **الذواكر المخبئية التجميعية Set Associative Caches** التي تُعَدّ عبارةً عن مزيج من الذواكر المخبئية المربوطة مباشرةً والذواكر المخبئية الترابطية بالكامل، وتسمح بوجود قيمة معينة للذاكرة المخبئية في بعض المجموعات الفرعية من الخطوط الموجودة ضمن هذه الذاكرة المخبئية، كما تُقسَم الذاكرة المخبئية إلى مناطق تسمَّى طرقًا Ways، ويمكن وجود عنوان معيّن في أيّ طريق، وبالتالي ستسمح الذاكرة المخبئية التجميعية المؤلفة من مجموعة من الطرق عددها n لخط الذاكرة المخبئية بالتواجد ضمن مجموعة الإدخالات التي عددها يساوي باقي قسمة مجموعة الكتل الإجمالية ذات الحجم المحدد على n، ويظهِر الشكل السابق عينةً من ذاكرة تجميعية مؤلفة من 8 عناصر و 4 طرق، إذ يكون للعنوانَين أربعة مواقع محتملة، مما يعني أنه يجب البحث عن نصف الذاكرة المخبئية فقط في كل عملية بحث، وكلما زاد عدد الطرق، زادت المواقع الممكنة ونقصت الأسماء البديلة، مما يؤدي إلى أداء أفضل.

يجب أن يتخلص المعالِج من الخط بمجرد امتلاء الذاكرة المخبئية لإفساح المجال لخط جديد، وهناك العديد من الخوارزميات التي يمكن للمعالج من خلالها اختيار الخط الذي سيتخلص منه مثل خوارزمية الأقل استخدامًا مؤخرًا Least Recently Used -أو LRU اختصارًا- والتي تُعَدّ خوارزميةً يجري فيها التخلص من أقدم خط غير مستخدَم لإفساح المجال للخط الجديد.

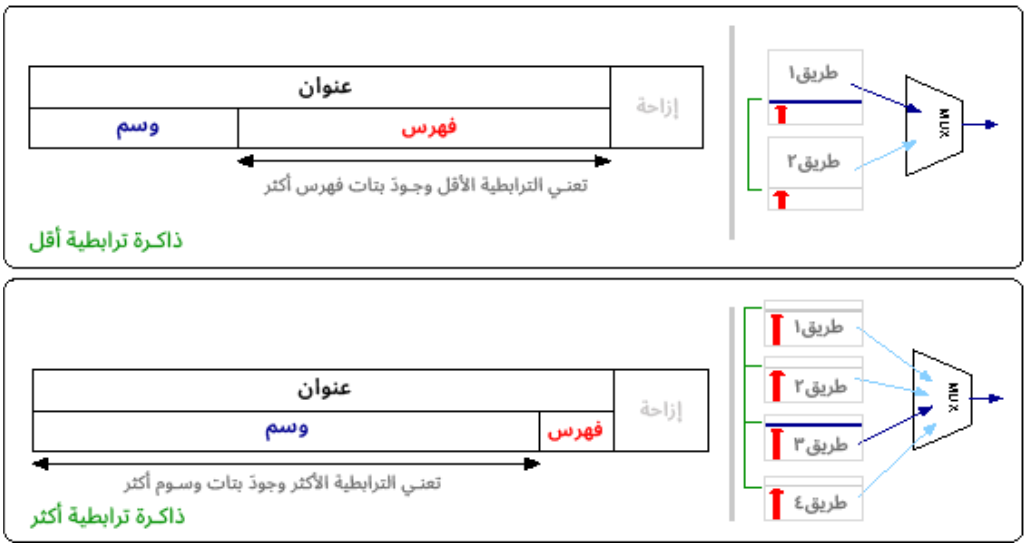
ليس هناك داع لضمان التوافق مع الذاكرة الرئيسية عندما تكون البيانات للقراءة فقط من الذاكرة المخبئية، لكن يحتاج المعالج لاتخاذ بعض القرارات حول كيفية تحديث الذاكرة الرئيسية الأساسية عندما يبدأ في الكتابة في خطوط الذاكرة المخبئية، إذ ستكتب طريقة التخزين الخاصة بالذاكرة المخبئية التي تُسمَّى Write-through Cache التغييرات مباشرةً في ذاكرة النظام الرئيسية عندما يحدّث المعالج الذاكرة المخبئية، ويُعَدّ ذلك أبطأ لأن عملية الكتابة في الذاكرة الرئيسية أبطأ، في حين تؤخر طريقة التخزين الخاصة بالذاكرة المخبئية التي تُسمَّى Write-back Cache كتابةَ التغييرات على الذاكرة RAM حتى الضرورة القصوى، والميزة الواضحة لذلك هي أنّ الوصول إلى الذاكرة الرئيسية مطلوب عند كتابة إدخالات الذاكرة المخبئية.

يُشار إلى خطوط الذاكرة المخبئية المكتوبة دون وضعها في الذاكرة على أنها متسخة Dirty، فعيبها هو أنه يمكن أن يتطلب الأمر وصولَين إلى الذاكرة أحدهما لكتابة بيانات الذاكرة الرئيسية المتسخة والآخر لتحميل البيانات الجديدة عند التخلص من إدخال معيّن من الذاكرة المخبئية.

إذا كان الإدخال موجودًا في كل من الـذاكرة المخبئيـة ذات المسـتوى الأعلى والمسـتوى الأدنى في الـوقت نفسه، فإننا نسمّي الذاكرة المخبئية ذات المستوى الأعلى بالشاملة Inclusive. بينما إذا أزالت الذاكرة المخبئيـة ذات المستوى الأعلى التي تحتوي على خط معيّن إمكانيةَ احتواء ذاكرة مخبئية ذات مستوى أقل على هذا الخط، فإننا نقول أنها حصرية Exclusive وسنناقش ذلك لاحقًا.

ا. عنونة الذاكرة المخبئية

لم نناقش حتى الآن كيف تقرر الذاكرة المخبئية ما إذا كان عنوان معيّن موجودًا في الذاكرة المخبئية أم لا، إذ يجب أن تحتفظ الذواكر المخبئية بمجلد للبيانات الموجودة حاليًا في خطوط الذاكرة المخبئية، ويمكن وضع مجلد وبيانات الذاكرة المخبئية على المعالج معًا، ولكن يمكن أن يكونا منفصلَين أيضًا كما في حالة المعالج POWER5 الذي يحتوي على مجلد ذاكرة L3 على المعالج، ولكن يتطلب الوصول إلى البيانات اجتيـاز ناقـل L3 للوصـول إلى ذاكرة خارجية ليست على المعالج، ويمكن أن يسهّل هذا الترتيب معالجة عمليات الوصول الصحيحة أو الخاطئـة بصورة أسر ع دون التكاليف الأخرى للاحتفاظ بالذاكرة المخبئية بالكامل على المعالج.



شكل 10: وسوم الذاكرة المخبئية Cache Tags

وسوم الذاكرة المخبئية Cache Tags: يجب التحقق من الوسوم على التوازي للحفاظ على وقت الاسـتجابة منخفضًا، إذ يتطلب المزيدُ من بتات الوسوم (أي ارتباطات مجموعات أقـل) عتـادًا أكـثر تعقيـدًا لتحقيـق ذلـك. بينما تعني ارتباطاتُ المجموعات الأكثر وسومًا أقل، ولكن يحتاج المعالج الآن إلى عتـاد لمضـاعفة خـرج العديـد من المجموعات التي يمكن أن تضيف زمن تأخير أيضًا.

يمكن تحديد ما إذا كان العنوان موجـودًا في الـذاكرة المخبئيـة بسـرعة من خلال فصـله إلى ثلاثـة أجـزاء هي الوسم Tag والفهرس Index والإزاحة Offset.

تعتمد بتات الإزاحة على حجم خط الذاكرة المخبئية، إذ يمكن استخدام خط بحجم 32 بايت مثلًا آخر 5 بتـات أي 2^5 من العنوان بوصفه إزاحةً في الخط، ويُعَدّ الفهرس خط ذاكرة مخبئية معيّن يمكن أن يتواجد فيـه الإدخـال،

فلنفترض أنه لدينا ذاكرة مخبئية تحتوي على 256 إدخالًا مثلًا، فإذا كانت هذه الذاكرة هي ذاكـرة مخبئيـة مربوطـة مباشرةً، فيمكن أن تكون البيانات موجودة في خط واحد محتمَل فقط، لذا تصف 8 بتات التالية (2^8) بعد الإزاحة الخط المراد التحقق منه بين 0 و 255.

لنفترض الآن أنّ الـذاكرة المخبئيـة المكونـة من 256 عنصـرًا مقسـمة إلى طـريقين، وهـذا يعـني أنّ هنـاك مجموعتين مؤلفتين من 128 خط، ويمكن أن يقع العنوان المحدد في أيّ من هاتين المجموعتين، وبالتالي فـإن المطلوب هو 7 بتات فقط على أسـاس فهـرس للإزاحـة في الطـرق المؤلفـة من 128 إدخـالًا، كمـا نخفّض عـدد البتات المطلوبة على أساس فهرس لأن كل طريق يصبح أصغر عندما نزيد عـدد الطـرق بالنسـبة إلى حجم ذاكـرة مخبئية معيّن.

لا يزال مجلد الذاكرة المخبئية بحاجـة إلى التحقـق ممـا إذا كـان العنـوان المخـزن في الـذاكرة المخبئيـة هـو العنوان الذي يريده، وبالتالي فإن البتـات المتبقيـة من العنـوان هي بتـات الوسـوم الـتي يتحقـق مجلـد الـذاكرة المخبئية منها مقابل بتات وسم العنوان الواردة لتحديـد مـا إذا كـان هنـاك عمليـة وصـول صـحيحة أم لا، وهـذه العلاقة موضحة في الصورة السابقة.

إذا كان هناك طرق متعددة، فيجب إجراء هذا التحقق على التوازي في كل طريق، ثم تُمرَر النتيجة بعد ذلــك إلى معـدد إرسـال Multiplexor ينتج عنـه نتيجـة وصـول صـحيحة hit أو خاطئـة miss، وكلمـا كـانت الـذاكرة المخبئية أكثر ارتباطًا، قل عدد البتات المطلوبة للفهرس وزاد عـدد البتـات المطلوبـة للوسـم، حـتى الوصـول إلى أقصى حد للذاكرة المخبئية الترابطية بالكامل حيث لا تُستخدَم بتـات كبتـات للفهـرس، كمـا تُعَـدّ المطابقـة على التوازي لبتات الوسوم مكونًا باهظًا لتصميم الـذاكرة المخبئيـة وهي عمومًـا العامـل المحـدّد لعـدد الخطـوط -أي حجمها- التي يمكن أن تنمو إليها الذاكرة المخبئية.

3.3 الأجهزة الطرفية Peripherals ونواقلها Buses

الأجهـزة الطرفيـة peripherals هي مجموعـة الأجهـزة الخارجيـة الـتي تتصـل بحاسـوبك، ويجب أن يكـون للمعالج طريقة ما للتواصل مع هذه الأجهزة الطرفية لجعلها مفيدة، وتسمى قناة الاتصـال بين المعـالج والأجهـزة الطرفية بالناقل Bus.

3.3.1 المفاهيم الخاصة بنواقل الأجهزة الطرفية

يتطلب الجهاز عمليات إدخال وإخـراج ليكـون مفيـدًا، ويوجـد هنـاك عـدد من المفـاهيم الشـائعة المطلوبـة للتواصل المفيد مع الأجهزة الطرفية التي سنستعرضها فيما يلي.

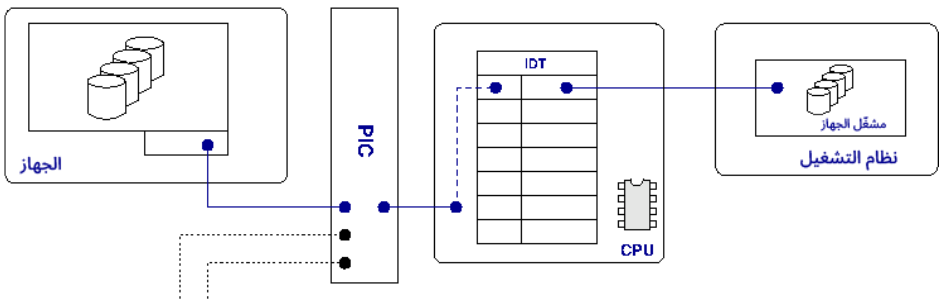
ا. المقاطعات Interrupts

تسمح المقاطعة للجهاز بمقاطعة المعالج حرفيًا بما تعنيه الكلمة للإشارة إلى بعض المعلومات، فمثلًا تُنشَأ مقاطعة لتسليم حدث الضغط على مفتاح إلى نظام التشغيل عند الضغط عليه، إذ تسنِد تركيبة من نظام التشغيل وبيوس BIOS مقاطعةً لكل جهاز.

ترتبط الأجهزة عمومًا بمتحكم المقاطعة القابل للبرمجة Programmable Interrupt Controller أو PIC اختصارًا، وهو شريحة منفصلة تُعَدّ جزءًا من اللوحة الأم التي تخزّن معلومات المقاطعة مؤقتًا وتنقلها إلى المعالج الرئيسي، كما يحتوي كل جهاز على خط مقاطعة فيزيائي بينه وبين أحد خطوط PIC التي يوفرها النظام، فإذا أراد الجهاز مقاطعة المعالج، فسيعدّل الجهد على هذا الخط.

هناك وصف واسع جدًا لدور متحكم PIC وهو أنه يتلقى هذه المقاطعة ويحولها إلى رسالة ليستخدمها المعالج الرئيسي، كما يختلف هذا الإجراء حسب المعمارية، ولكن المبدأ العام هو أن يضبط نظام التشغيل جدول واصف المقاطعات Interrupt Descriptor Table الذي تربط فيه كل مقاطعة محتمَلة بعنوان شيفرة برمجية للانتقال إليها عند تلقي المقاطعة كما هو موضح في الشكل الآتي.

كتابة معالج المقاطعة Interrupt Handler هو عمل مطور برنامج تشغيل الجهاز بالتزامن مع نظام التشغيل.



شكل 11: نظرة عامة على معالجة المقاطعة

توضح الصورة السابقة نظرة عامة على معالجة المقاطعة إذ يرفع الجهاز المقاطعة إلى متحكم المقاطعة، وتمرِّر هذه المقاطعة المعلومات إلى المعالج. ينظر المعالج إلى جدول واصف مقاطعاته الذي يملؤه نظام التشغيل للعثور على الشيفرة البرمجية التي تعالج الخطأ.

تقسّم معظم المشغّلات معالجة المقاطعات إلى نصفين سفلي وعلوي، إذ يتعرف النصف السفلي على المقاطعة ويضع الإجراءات في رتل للمعالجة ويعيد المعالج إلى ما كان يفعله سابقًا بسرعة، في حين سيُشغَّل النصف العلوي لاحقًا عندما تكون وحدة المعالجة المركزية متاحةً، وسينفّذ المعالجة الإضافية، كما يؤدي ذلك إلى وقف المقاطعة التي تعطل وحدة المعالجة المركزية بأكملها.

حفظ الحالة

بما أنّ المقاطعة يمكن أن تحدث في أيّ وقت، فيجب أن تتمكن من العودة إلى العملية الجارية عند الانتهاء من معالجة المقاطعة، كما أنّ مهمة نظام التشغيل هي التأكد من أنه يحفظ أيّ حالة State عند الدخول إلى معالج المقاطعة، أي يسجلها ويستعيدها عند العودة من معالج المقاطعة، وتكون بذلك المقاطعة واضحةً تمامًا في كل ما يحدث في ذلك الوقت بغض النظر عن الوقت الضائع.

المقاطعات Interrupts والمصائد Traps والاستثناءات Exceptions

ترتبط المقاطعة عمومًا بحدث خارجي من جهاز فيزيائي، ولكن تُعَدّ الآلية نفسها مفيدةً للتعامل مع عمليات النظام الداخلية، فإذا اكتشف المعالج مثلًا حالات مثل الوصول إلى ذاكرة غير صالحة أو محاولة القسمة على صفر أو تعليمات غير صالحة، فيمكنه داخليًا رفع استثناء ليعالجه نظام التشغيل، كما تُستخدَم هذه الآلية ليلتقط نظام التشغيل استدعاءات النظام ولتطبيق الذاكرة الوهمية virtual memory، في حين تبقى مبادئ مقاطعة الشيفرة البرمجية المُشغَّلة بطريقة غير متزامنة كما هي بالرغم من إنشائها داخليًا وليس من مصدر خارجي.

أنواع المقاطعات

هناك طريقتان رئيسيتان لإصدار إشارات إلى المقاطعات على الخط هما المستوى level والحافة edge المُنبَّهة، إذ تحدّد المقاطعات ذات المستوى المُنبَّه جهد خط المقاطعة الذي يُحتفَظ به مرتفعًا للإشارة إلى وجود مقاطعة معلَّقة، في يحن تكتشف المقاطعات ذات الحافة المُنبَّهة الانتقالات في الناقل عندما ينتقل جهد الخط من منخفض إلى مرتفع، ويكتشف متحكم المقاطعة PIC نبضة الموجة المربعة باستخدام المقاطعة ذات الحافة المنبَّهة عند إصدار الإشارة ورفع المقاطعة.

يظهر الفرق عندما تشترك الأجهزة في خط مقاطعة، إذ سيكون خط المقاطعة مرتفعًا في نظام المقاطعة ذي المستوى المنبَّه حتى معالجة جميع الأجهزة التي رفعت المقاطعة وإلغاء تأكيد مقاطعتها، كما تشير النبضة الموجودة على الخط إلى متحكم المقاطعة PIC الذي تنشئه المقاطعة في نظام المقاطعة ذي الحافة المنبَّهة، وستصدر هذه النبضة إشارةً إلى نظام التشغيل لمعالجة المقاطعة في حالة ظهور نبضات أخرى على الخط المؤكَّد مسبقًا من جهاز آخر.

تكمن مشكلة المقاطعات ذات المستوى المنبَّه في أنها يمكن أن تتطلب قدرًا كبيرًا من الوقت لمعالجة مقاطعة أحد الأجهزة، إذ يظل خط المقاطعة مرتفعًا أثناء هذا الوقت ولا يمكن تحديد ما إذا تسبّب أيّ جهاز آخر في حدوث مقاطعة على الخط، وهذا يعني أنه يمكن أن يكون هناك زمن تأخير كبير وغير متوقع في خدمة المقاطعات.

يمكن ملاحظة المقاطعة طويلة الأمد ووضعها في رتل انتظار في المقاطعات ذات الحافة المُنبَّهة، ولكن لا يزال بإمكان الأجهزة الأخرى التي تشترك في الخط الانتقال -وبالتالي رفع المقاطعات- أثناء حدوث ذلك، ويؤدي

ذلك إلى حدوث مشاكل جديدة، إذ يمكن تفويت أحد المقاطعات في حالة مقاطعة جهازين في الـوقت نفسـه أو يمكن أن يؤدي التشويش البيئي أو غيره إلى حدوث مقاطعة زائفة يجب تجاهلها.

المقاطعات غير القابلة للتقنع أو الإخفاء Non-maskable Interrupts

يجب أن يكون النظام قادرًا على إخفاء المقاطعـات أو منعهـا في أوقـات معينـة، ويمكن وضـع المقاطعـات لتكون قيد الانتظار، لكن هناك صنف معيّن من المقاطعات يسـمى المقاطعـات غـير القابلـة للتقنّـع أو الإخفـاء Non-maskable Interrupts أو NMI اختصارًا، إذ تُعَدّ هذه المقاطعات استثناءً من هذه القاعدة مثل مقاطعة إعادة الضبط reset.

يمكن أن تكون مقاطعات NMI مفيدةً لتطبيق أشياء مثل مراقبة النظـام، حيث تُرفَـع مقاطعـة NMI دوريًـا وتضبِط بعض الرايات التي يجب أن يقرّ بها نظام التشغيل، فإذا لم يظهر هذا الإقرار قبل مقاطعـة NMI الدوريـة التالية، فيمكن عَدّ النظام أنه لا يحرز أيّ تقدم، كما يمكن استخدام مقاطعات NMI لتشخيص Profiling النظام، إذ يمكن رفع مقاطعات NMI الدورية واستخدامها لتقييم الشيفرة البرمجية التي يعمل بهـا المعـالج حاليًـا، ممـا يؤدي بمرور الوقت إلى إنشاء ملف تعريف للشيفرة البرمجية التي تعمل والحصول على رؤية مفيدة للغايـة حـول أداء النظام.

ب. فضاء الإدخال والإخراج IO

يجب أن يتصل المعالج بالجهاز الطرفي عبر عمليات الإدخال والإخراج IO، ويُطلَق على الشكل الأكثر شـيوعًا من عمليات IO عمليات الإدخال والإخراج المرتبطة بالذاكرة Memory Mapped IO، إذ ترتبط مسجلات الجهاز مع الذاكرة، وما عليك سوى القراءة أو الكتابة في عنوان محدد من الذاكرة للتواصل مع الجهاز.

3.3.2 الوصول المباشر للذاكرة DMA

بما أن سرعة الأجهزة أقل بكثير من سرعة المعالجات، فيجب أن يكون هناك طريقة ما لتجنب انتظار وحـدة المعالجة المركزية للبيانات من الأجهزة.

يُعَدّ الوصول المباشر للذاكرة Direct Memory Access -أو DMA اختصارًا- طريقةً لنقل البيانات مباشـرةً بين الجهاز الطرفي وذاكرة RAM الخاصة بالنظام، ويمكن لمشغّل الجهـاز إعـداده لإجـراء نقـل باسـتخدام طريقـة الوصول DMA من خلال إعطائه منطقةً من ذاكرة RAM لوضع بياناته فيها، ثم يمكنـه بـدء نقـل DMA والسـماح لوحدة المعالجة المركزية بمواصلة تنفيذ المهام الأخرى.

سيرفع الجهاز المقاطعة بعد الانتهـاء ويرسـل لمشـغّل الجهـاز إشـارةً باكتمـال النقـل، ثم سـتكون البيانـات القادمـة من الجهـاز مثـل ملـف من قـرص صـلب أو إطـارات من بطاقـة التقـاط الفيـديو موجـودةً في الـذاكرة وجاهزةً للاستخدام.

3.3.3 نواقل أخرى

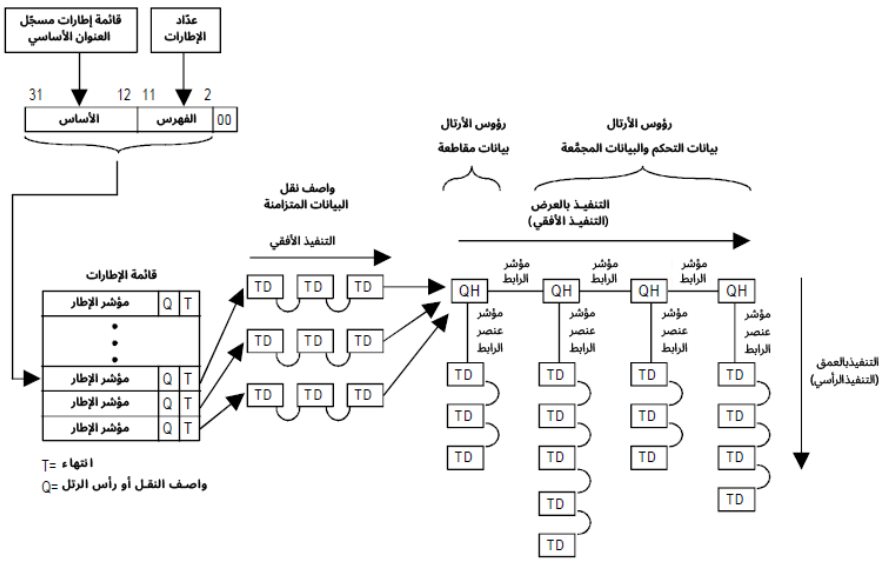
تصل نواقل أخرى بين ناقل PCI والأجهزة الخارجية مثل ناقل USB الذي سنتعرف عليه فيما يلي.

I. USB

يُعَدّ جهاز USB من وجهة نظر نظام التشغيل أنه مجموعة من نقاط النهاية المجمَّعة معًا في واجهة ما، إذ يمكن أن تكون نقطة النهاية إما نقطة إدخال أو إخراج، بحيث تنقل نقطة النهاية البيانات باتجاه واحد فقط، كما يمكن أن تحتوي نقاط النهاية على عدد من الأنواع المختلفة هي:

- نقاط نهاية خاصة بعمليات التحكم Control End-points: مخصصة لإعداد الجهاز وغير ذلك.
- نقاط نهاية خاصة بالمقاطعات Interrupt End-points: تُستخدَم لنقل كميات صغيرة من البيانات، ولديها أولوية عليا.
- نقاط النهاية المجمَّعة Bulk End-points: تنقل كميات كبيرة من البيانات ولكنها لا تحصل على قيود زمنية مضمونة.
- عمليات النقل المتزامنة Isochronous Transfers: هي عمليات نقل ذات أولوية عالية في الوقت الحقيقي، ولكن إذا جرى تفويتها، فلن يعاد تجربتها، وتُستخدَم لبيانات البث مثل الفيديو أو الصوت حيث لا توجد فائدة من إرسال البيانات مرةً أخرى.

يمكن أن يكون هناك العديد من الواجهات المكونة من نقاط نهاية متعددة، وتُجمَّع الواجهات ضمن إعدادات Configurations، ولكن معظم الأجهزة لها إعداد واحد فقط.



شكل 12: نظرة عامة على متحكم UCHI (مأخوذة من توثيق إنتل Intel)

يوضح الشكل السابق نظرة عامة على واجهة متحكم المضيف العامة Universal Host Controller Interface والبرمجيات، كما تضبط البرمجيات قالب بيانات بتنسيق محدد لمتحكم المضيف لقراءته وإرساله عبر ناقل USB. أو UHCI اختصارًا، كما يوفر نظرةً عامةً حول الكيفية التي تنقل بها بيانات USB خارج النظام عن طريق مجموعة من العتاد.

يحتوي المتحكم بدءًا من أعلى يسار الشكل السابق على مسجل إطارات مع عدّاد يُزاد دوريًا في كل ميلي ثانية، إذ تُستخدَم هذه القيمة للفهرسة ضمن قائمة إطارات تنشئها البرمجيات، ويؤشّر كل إدخال في هذا الجدول إلى رتل واصفات النقل Transfer Descriptors، كما تضبط البرمجيات هذه البيانات في الذاكرة ويقرؤها المتحكم المضيف الذي يُعَدّ شريحةً منفصلةً تشغّل ناقل USB، ويجب أن تجدول البرمجيات أرتال العمل بحيث يُمنَح 90% من وقت الإطار للبيانات المتزامنة ويُمنَح 10% المتبقية لبيانات المقاطعة والتحكم والبيانات المُجمَّعة.

تعني الطريقة التي تُربَط بها البيانات أنّ واصفات النقل للبيانات المتزامنة ترتبط بمؤشر إطار معيّن واحد فقط -أي فترة زمنية معينة واحدة فقط- ثم ستُهمَل، لكن تُوضَع جميع بيانات المقاطعة والتحكم والبيانات المُجمَّعة ضمن رتل انتظار بعد البيانات المتزامنة، وبالتالي إذا لم تُرسَل في إطار واحد -أو فترة زمنية واحدة- فسيجري ذلك في المرة التالية.

تتواصل طبقات USB عبر كتل طلبات USB أو URB اختصارًا، إذ تحتوي كتل URB على معلومات حول نقطة النهاية التي يرتبط بها هذا الطلب والبيانات وأي معلومات أو سمات ذات صلة ودالة رد نداء call-back function تُستدعَى عند اكتمال كتلة URB، كما ترسِل مشغّلات USB كتل URB بتنسيق ثابت إلى مركز USB الذي يديرها بالتنسيق مع متحكم مضيف USB على النحو الوارد أعلاه، وتُرسَل بياناتك إلى جهاز USB عبر مركز USB، ثم تُشغَّل دالة رد النداء.

3.4 أنظمة المعالجات في معمارية الحاسوب

نَمت قوة الحوسبة بوتيرة سريعة دون ظهور أيّ علامات على التباطؤ كما توقّع قانون مور Moore، فليس مألوفًا أن تحتوي أيّ خوادم عالية الجودة على وحدة معالجة مركزية واحدة فقط مع إمكانية تحقيق ذلك باستخدام عدد من الأساليب المختلفة.

3.4.1 المعالجة المتعددة المتماثلة Symmetric Multi-Processing

تُعَدّ المعالجة المتعددة المتماثلة Symmetric Multi-Processing -أو SMP اختصارًا- الإعداد الأكثر شيوعًا حاليًا لتضمين وحدات المعالجة المركزية CPU المتعددة في نظام واحد، ويشير المصطلح متماثل Symmetric إلى حقيقة أنّ جميع وحدات المعالجة المركزية في النظام هي نفسها من حيث المعمارية وسرعة الساعة مثلًا، كما توجد في نظام SMP معالجات متعددة تشترك في جميع موارد النظام الأخرى مثل الذاكرة والقرص الصلب وغير ذلك.

ا. ترابط الذواكر المخبئية Cache Coherency

تعمل وحدات المعالجة المركزية في النظام بصورة مستقلة عن بعضها بعضًا، فلكل منها مجموعته الخاصة من المسجلات وعدّاد البرنامج وغير ذلك، ولكن يوجد مكوِّن واحد يتطلب تزامنًا صارمًا بالرغم من تشغيل وحدات المعالجة المركزية بصورة منفصلة عن بعضها بعضًا، وهذا المكوِّن هو الذاكرة المخبئية Cache الخاصة بوحدة المعالجة المركزية.

تذكّر أنّ الذاكرة المخبئية هي مساحة صغيرة من الذاكرة يمكن الوصول إليها بسرعة وتعكس القيم المخزّنة في ذاكرة النظام الرئيسية، فإذا عدّلت إحدى وحدات المعالجة المركزية البيانات في الذاكرة الرئيسية وكان لدى وحدة معالجة مركزية أخرى نسخة قديمة من تلك الذاكرة في ذاكرتها المخبئية، فلن يكون النظام في حالة متناسقة، ولاحظ أنّ هذه المشكلة تحدث عندما تكتب المعالجات في الذاكرة فقط، إذ ستكون البيانات متناسقةً إذا كانت القيمة للقراءة فقط.

يستخدِم نظام SMP عملية التنصت Snooping لتنسيق الحفاظ على ترابط الذواكر المخبئية على جميع المعالجات، إذ يُعَدّ التنصت العمليةَ التي يستمع فيها المعالج إلى ناقل تتصل به جميع المعالجات لمعرفة أحداث الذاكرة المخبئية، ثم يحدّث الذاكرة المخبئية وفقًا لذلك.

يمكن تحقيق ذلك باستخدام بروتوكول واحد هو بروتوكول MOESI الذي يرمز إلى الكلمات مُعدَّل Modified ومالك Owner وحصري Exclusive ومشارَك Shared وغير صالح Invalid التي تمثّل الحالة التي يمكن أن يكون فيها خط الذاكرة المخبئية على معالج في النظام، كما توجد بروتوكولات أخرى لذلك، ولكن تشترك جميعها في مفاهيم متشابهة، وسنوضح فيما يلي بروتوكول MOESI.

إذا طلب المعالج قراءة خط ذاكرة مخبئية من الذاكرة الرئيسية، فيجب عليه أولًا التنصت على جميع المعالجات الأخرى في النظام لمعرفة ما إذا كانت تعرف حاليًا أيّ شيء عن تلك المنطقة من الذاكرة مثل تخزينها في الذاكرة المخبئية، فإذا لم تكن موجودةً في أيّ عملية أخرى، فيمكن للمعالج تحميل الذاكرة في الذاكرة المخبئية وتمييزها على أنها حصرية Exclusive، ويغير الحالة إلى معدَّلة Modified عند الكتابة في الذاكرة المخبئية.

تلعب هنا تفاصيل الذاكرة المخبئية دورًا أساسيًا، إذ ستعيد بعض الذواكر المخبئية مباشرةً كتابة الذاكرة المخبئية المعدَّلة إلى ذاكرة النظام المعروفة باسم الذاكرة المخبئية من النوع Write-through، لأن عمليات الكتابة تنتقل إلى الذاكرة الرئيسية، في حين لن تفعل ذلك الذواكر المخبئية الأخرى، بل ستترك القيمة المُعدَّلة في الذاكرة المخبئية فقط حتى التخلص منها عندما تمتلئ الذاكرة المخبئية مثلًا.

الحالة الأخرى هي المكان الذي يطبّق فيه المعالج عملية التنصت ويكتشف أن القيمة موجودة في ذاكرة مخبئية خاصة بمعالجات أخرى، فإذا كانت هذه القيمة مميَّزةً بوصفها مُعدَّلةً Modified، فسينسخ المعالج البيانات في ذاكرته المخبئية ويميّزها على أنها مشتركة Shared، كما سيرسل رسالةً إلى المعالج الآخر الذي

حصلنا على البيانات منه لتمييز خط ذاكرته المخبئية بوصفه المالك Owner، لنفترض الآن أن معالجًا ثالثًا في النظام يريد استخدام تلك الذاكرة أيضًا، فسيتنصت ويبحث عن نسخة مشتركة ونسخة مالكة، وبالتالي سيأخذ قيمته من قيمة المالك.

تقرأ جميع المعالجات الأخرى القيمة فقط، ولكن يبقى خط الذاكرة المخبئية مشتركًا في النظام، فإذا احتاج معالج ما تحديث القيمة، فإنه يرسل رسالة إلغاء صلاحية Invalidate عبر النظام، كما يجب على أيّ معالج له هذا الخط الخاص بالذاكرة المخبئية تمييزه بوصفه غير صالح Invalid لأنه لم يَعُد يعكس القيمة الحقيقية، ويميز المعالج خط الذاكرة المخبئية بوصفه معدَّلًا في ذاكرته المخبئية عندما يرسل رسالة إلغاء الصلاحية وستميّزه المعالجات الأخرى على أنه غير صالح.

لاحظ أنه إذا كان خط الذاكرة المخبئية حصريًا، فسيعلم المعالج أنه لا يوجد معالج آخر يعتمد عليه، لذا يمكنه تجنّب إرسال رسالة إلغاء صلاحية، وتبدأ بعدها العملية من جديد، وبالتالي يتحمل أيُّ معالج له القيمة المعدّلة مسؤوليةَ كتابة القيمة الحقيقية مرةً أخرى إلى الذاكرة RAM عند التخلص منها من الذاكرة المخبئية، ولاحظ أنّ هذا البروتوكول يضمن تناسق خط الذاكرة المخبئية بين المعالجات.

هناك العديد من المشاكل في هذا النظام عند زيادة عدد المعالجات، إذ يمكن التحكم في تكلفة التحقق من وجود معالج آخر يحتوي على خط ذاكرة مخبئية (التنصت على عملية القراءة) أو إلغاء صلاحية البيانات في كل معالج آخر (إلغاء عملية التنصت) عند استخدام عدد قليل من المعالجات، ولكن تزداد حركة النواقل مع زيادة عدد المعالجات وهذا هو السبب في أن أنظمة SMP تصل إلى حوالي 8 معالجات فقط.

يعطي وجود جميع المعالجات في الناقل نفسه مشاكل فيزيائية أيضًا، إذ تسمح خصائص الأسلاك الفيزيائية بوضعها على مسافات معينة من بعضها بعضًا وتسمح بأن يكون لها أطوال معينة فقط، وتبدأ سرعة الضوء في أن تصبح أحد الجوانب التي يجب مراعاتها في المدة التي تستغرقها الرسائل للتنقل في النظام مع المعالجات التي تعمل بسرعة مقدَّرةً بالجيجاهيرتز.

لاحظ أنّ برمجيات النظام ليس لها أيّ جزء من هذه العملية، بالرغم من أنّ المبرمجين يجب أن يكونوا على دراية بما يطبّقه العتاد استجابةً للبرمجيات التي يصمّمونها لزيادة الأداء إلى الحد الأقصى.

حصرية الذاكرة المخبئية في أنظمة SMP

شرحنا في فقرة سابقة الذواكر المخبئية الشاملة Inclusive والحصرية Exclusive، إذ تكون الذواكر المخبئية L1 شاملةً، أي أن جميع البيانات الموجودة في الذاكرة المخبئية L1 موجودة في الذاكرة المخبئية L2، وتعني الذاكرة المخبئية L1 الشاملة أنّ الذاكرة المخبئية L2 يجب أن تتنصت حركة مرور الذاكرة للحفاظ على ترابطها في نظام متعدد المعالجات، إذ ستضمن L1 عكس أيّ تغييرات في الذاكرة L2، مما يقلل من تعقيد ذاكرة L1 ويفصله عن عملية التنصت، وبالتالي سيسمح لها بأن تكون أسرع.

تحتوي معظم المعالجات الحديثة المتطورة مثل المعالجات التي ليست مدمَجة على سياسة كتابة الذاكرة المخبئية L1 من النوع Write-through وسياسة الكتابة من النوع Write-back في الذواكر المخبئية ذات المستوى الأدنى، وهناك عدة أسباب لذلك، فبما أنّ ذواكر L2 المخبئية في هذا الصنف من المعالجات تكون حصريةً تقريبًا على الشريحة وسريعةً جدًا عمومًا، فليست العقوبات المفروضة على كتابة الذاكرة المخبئية L1 من النوع Write-through الأمر الرئيسي، كما يمكن أن تتسبّب مجمّعات البيانات المكتوبة التي لا يُحتمَل قراءتها في المستقبل في تلوث مورد L1 المحدود لأن أحجام L1 صغيرة.

ليس هناك داع للقلق بشأن الكتابة في L1 من النوع Write-through إذا احتوت على بيانات متسخة معلَّقة، وبالتالي يمكن أن تمرّر منطق الترابط الإضافي إلى ذاكرة L2 التي لديها دور أكبر تلعبه في ترابط الذاكرة المخبئية.

ب. تقنية خيوط المعالجة الفائقة Hyperthreading

يمكن أن يقضي المعالج الحديث كثيرًا من وقته في انتظار أجهزة أبطأ بكثير في تسلسل الذواكر الهرمي لتقديم البيانات للمعالجة، وبالتالي فإن إستراتيجيات الحفاظ على خط أنابيب المعالج ممتلئًا لها أهمية قصوى، وتتمثل إحدى الإستراتيجيات في تضمين عدد كاف من المسجلات ومنطق الحالة بحيث يمكن معالجة مجريين من التعليمات في الوقت نفسه، مما يجعل وحدة معالجة مركزية واحدة تبحث عن جميع النوايا والأهداف المطلوبة كأنها وحدتان CPU.

تحتوي كل وحدة معالجة مركزية على مسجلاتها الخاصة، ولكن يجب عليها مشاركة منطق المعالج الأساسي والذاكرة المخبئية وحيز النطاق التراسلي للإدخال والإخراج من وحدة المعالجة المركزية إلى الذاكرة، لذا يمكن أن يحافظ مجريان من التعليمات على المنطق الأساسي للمعالج أكثر انشغالًا، ولكن لن تكون زيادة الأداء كبيرةً بسبب وجود وحدتَي CPU منفصلتين فيزيائيًا، ويكون تحسين الأداء أقل من 20%، ولكن يمكن أن يكون أفضل أو أسوأ كثيرًا اعتمادًا على الحِمل.

ج. الأنوية المتعددة Multi Core

أصبح وضع معالِجَين أو أكثر في الحزمة الفيزيائية نفسها ممكنًا مع زيادة القدرة على احتواء مزيد من الترانزستورات على شريحة واحدة، ولكن المعالجات الأكثر شيوعًا هي المعالجات ثنائية النواة، إذ توجد نواتان للمعالج على الشريحة نفسها، وتُعَدّ هذه الأنوية -على عكس تقنية خيوط المعالجة الفائقة Hyperthreading- معالجات كاملةً، وبالتالي تبدو على أنها معالجات منفصلة فيزيائيًا مثل نظام SMP.

تحتوي المعالجات على ذاكرة L1 المخبئية الخاصة بها، ولكن يجب عليها مشاركة الناقل المتصل بالذاكرة الرئيسية وأجهزة أخرى، وبالتالي لن يكون الأداء جيدًا مثل نظام SMP الكامل، ولكنه أفضل بكثير من نظام خيوط المعالجة الفائقة، ويمكن لكل نواة تطبيق تقنية خيوط المعالجة الفائقة لتحسين إضافي.

تتمتع المعالجات متعددة الأنوية ببعض المزايا التي لا تتعلق بالأداء، كما أنّ للناقلات الفيزيائية الخارجية بين المعالجات حدود فيزيائية، ولكن يمكن حل بعض هذه المشاكل من خلال احتواء المعالجات على قطعة السيليكون نفسها بحيث تكون قريبةً جدًا من بعضها بعضًا.

تُعَدّ متطلبات الطاقة للمعالجات متعددة الأنوية أقل بكثير من المعالجات المنفصلة عن بعضها بعضًا، وهذا يعني أن هناك حاجة أقل لتبريد الحرارة والتي يمكن أن تكون ميزةً كبيرةً في تطبيقات مراكز البيانات حيث تُجمَّع الحواسيب مع وجود حاجة كبيرة للتبريد، كما يجعل وجود الأنوية في الحزمة الفيزيائية نفسها المعالجةَ المتعددة عمليةً في التطبيقات التي لن تكون فيها كذلك مثل الحواسيب المحمولة، كما يُعَدّ إنتاج شريحة واحدة بدلًا من شريحتين أرخص بكثير.

3.4.2 العناقيد Clusters

تتطلب العديد من التطبيقات أنظمةً أكبر بكثير من عدد المعالجات التي يمكن لنظام SMP التوسع إليها، وتُعَدّ العناقيد Clusters إحدى الطرق لتوسيع النظام أكثر، وهي عدد من الحواسيب التي لديها بعض القدرة على التواصل مع بعضها بعضًا، كما لا تعرف الأنظمة بعضها بعضًا على مستوى العتاد، إذ تُترَك مهمة ربط هذه الحواسيب للبرمجيات.

تسمح البرمجيات مثل MPI للمبرمجين بكتابة برامجهم ثم وضع أجزاء منها على حواسيب أخرى في النظام مثل تمثيل حلقة تُنفَّذ عدة آلاف من المرات وتطبّق إجراءً مستقلًا، أي لا يوجد تكرار للحلقة يؤثر على أيّ تكرار آخر، ويمكن للبرمجيات جعل كل حاسوب يشغّل 250 حلقة لكل منها مع وجود أربعة حواسيب في العنقود.

يختلف الترابط بين الحواسيب، إذ يمكن أن يكون بطيئًا مثل روابط شبكة الإنترنت أو سريعًا مثل الناقلات المخصَّصة والخاصة مثل روابط إنفيني باند Infiniband، ومهما كان هذا الترابط، فسيبقى في المستوى الأخفض من تسلسل الذواكر الهرمي وسيكون أبطأ بكثير من الذاكرة RAM، وبالتالي لن يقدّم العنقود أداءً جيدًا في الموقف الذي تتطلب فيه كل وحدة معالجة مركزية الوصول إلى البيانات المُخزَّنة في الذاكرة RAM الخاصة بحاسوب آخر، إذ ستحتاج البرمجيات في كل مرة أن تطلب نسخةً من البيانات من الحاسوب الآخر، وتنسخها عبر الرابط البطيء إلى الذاكرة RAM المحلية قبل أن يتمكن المعالج من إنجاز أيّ عمل.

لا تتطلب العديد من التطبيقات هذا النسخ المستمر بين الحواسيب، وأحد الأمثلة الشائعة عن ذلك هو SETI@Home، إذ تُحلَّل البيانات التي جرى جمعها من هوائي راديو بحثًا عن علامات على وجود كائن فضائي، ويمكن توزيع كل حاسوب لبضع دقائق للحصول على البيانات لتحليلها ويعطي تقريرًا ملخصًا لما وجده، إذ يُعَدّ SETI@Home عنقودًا مخصَّصًا وكبيرًا جدًا.

يوجد تطبيق آخر هو تطبيق تصيير الصور Rendering of Images الذي يُستخدَم خاصةً للتأثيرات الخاصة في الأفلام، إذ يُسلَّم كل حاسوب إطارًا واحدًا من الفيلم يحتوي على نماذج إطارات شبكية وخامات Textures ومصادر إضاءة يجب دمجها أو تصييرها في التأثيرات الخاصة المذهلة التي نحصل عليها، كما يُعَدّ كل إطار

ساكنًا، لذلك لا يحتاج الحاسوب بمجرد حصوله على الدخل الأولي لمزيد من الاتصال حتى يصبح الإطـار النهـائي جاهزًا لإرساله ودمجه في الحركة، فقد كان لفيلم سيد الخواتم مثلًا تأثيرات خاصة مصيَّرة على عنقود ضخم يعمل بنظام لينكس.

3.4.3 الوصول غير الموحد للذاكرة Non-Uniform Memory Access

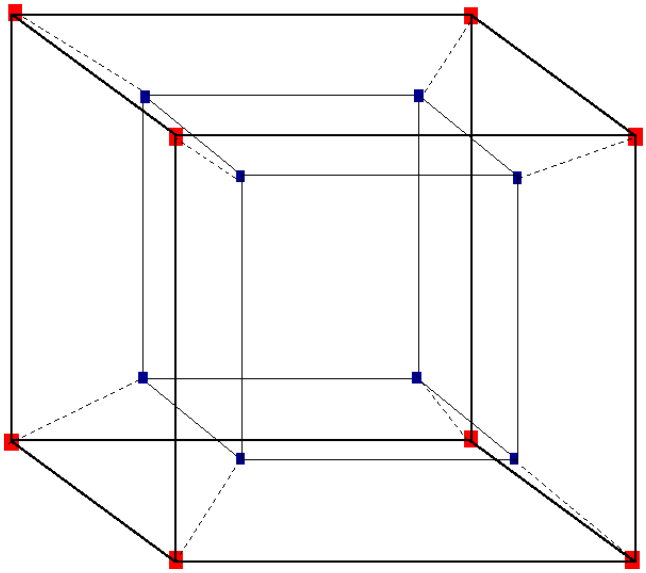
يُعَدّ الوصول غير الموحـد للـذاكرة Non-Uniform Memory Access -أو NUMA اختصـارًا- عكس نظـام العناقيد السابق تقريبًا، ولكنه -كما هو الحال في نظام العنقود- يتكون من عقد فردية مرتبطة ببعضـها بعضًـا، إلا أنّ الارتباط بين العقد شديد التخصص ومكلف، ولا يمتلك العتاد أيّ معرفة بالربط بين العقد في نظـام العنقـود، في حين لا تمتلك البرمجيات في نظام NUMA معرفةً جيدةً أو تمتلك معرفةً أقل حول تخطيط النظام، إذ يطبّـق العتاد كل العمل لربط العقد مع بعضها بعضًا.

يأتي مصطلح الوصول غير الموحّد إلى الذاكرة من حقيقـة أن الـذاكرة RAM ليسـت محليـة بالنسـبة لوحـدة المعالجة المركزية، وبالتالي يمكن أن هناك حاجة لأن تصل عقدة على بعد مسـافة مـا إلى البيانـات، إذ يسـتغرق ذلك وقتًا أطـول على النقيض من معـالج واحـد أو نظـام SMP حيث يمكن الوصـول إلى الـذاكرة RAM مباشـرةً، ويستغرق ذلك دائمًا وقتًا ثابتًا أو موحّدًا.

ا. تخطيط نظام NUMA

يُعَدّ تقليل المسافة بين العقد أمرًا بالغ الأهمية مع وجـود العديـد من العقـد الـتي تتواصـل مـع بعضـها في النظام، إذ يُفضَّل أن يكون لكل عقدة رابط مباشر بكل عقدة أخرى لأنـه يقلّـل المسـافة الـتي تحتاجهـا أيّـة عقـدة للعثور على البيانات، لكن لا يُعَدّ ذلك موقفًا عمليًا عندما ينمو عدد العقد إلى المئات والآلاف كما هـو الحـال مـع الحواسيب العملاقة الكبيرة، فالأساس في هذا النمو هـو مجموعـة مؤلفـة من عقـدتين تتواصـلان مـع بعضـهما بعضًا ثم ستنمو إلى $\frac{n!}{2\times(n-2)!}$.

تُستخدَم التخطيطات البديلة لمقايضة المسافة بين العقد مع الوصـلات المطلوبـة بهـدف التقليـل من هـذا النمو الأسي، فأحد هذه التخطيطات الشائعة في معماريات NUMA الحديثة هـو المكعب الفـائق Hypercube الذي يحتوي على تعريف رياضي صارم، ويكون المكعب الفائق هو نظير ربـاعي الأبعـاد للمكعب الـذي هـو نظـير ثلاثي الأبعاد للمربع.



شكل 13: المكعب الفائق Hypercube

يوضح الشكل السابق مثالًا عن المكعب الفائق Hypercube الذي يـوفر مقايضـةً جيـدةً بين المسـافة بين العقد وعدد الوصلات المطلوب. يمكننا أن نرى في هذا الشكل أن المكعب الخارجي يحتوي على 8 عقـد، والحـد الأقصى لعدد المسارات المطلوبة لأي عقدة للتواصل مع عقدة أخرى هو 3، فـإذا وضـعنا مكعبًـا آخـر داخـل هـذا المكعب، فسيكون لدينا ضعف عدد المعالجات ولكن زادت التكلفة القصوى للمسار إلى 4، مما يعني نمو تكلفة المسار القصوى خطيًا فقط عند نمو عدد المعالجات بمقدار 2n.

ب. ترابط الذاكرة المخبئية Cache Coherency

لا يزال الحفاظ على ترابط الذاكرة المخبئية في نظام NUMA ممكنًا، إذ يشـار إلى ذلـك باسـم نظـام NUMA مع ترابط الذاكرة المخبئية Cache Coherent NUMA System أو ccNUMA اختصارًا، ولا يتوسّـع المخطـط القائم على البث الإذاعي المُستخدَم للحفاظ على ترابط ذاكرة المعالج المخبئية في نظام SMP إلى مئات أو حـتى آلاف المعالجات في نظام NUMA كبير.

يشار إلى أحد المخططات الشائعة لترابط الـذاكرة المخبئيـة في نظـام NUMA باسـم النمـوذج المسـتند إلى الـدليل Directory Based Model الـذي تتصـل فيـه المعالجـات الموجـودة في النظـام بعتـاد دليـل الـذاكرة المخبئية، إذ يحافظ عتاد الدليل على صورة متناسقة لكـل معـالج، كمـا يخفي هـذا التجريـد عمـل نظـام NUMA عن المعالج.

يحتفظ المخطط المستند إلى الدليل لصـاحبيه Censier و Feautrier بـدليل مركـزي، إذ تحتـوي كـل كتلـة ذاكرة على بِت راية يُعرَف بالبِت الصالح Valid Bit لكل معـالج وبِت واحـد يُسـمَّى بـالبِت المتسـخ Dirty Bit، ويضبط الدليل البِت الصالح للمعالج الذي يقرأ الذاكرة إلى ذاكرته المخبئية.

إذا أراد المعالج الكتابة إلى خط الذاكرة المخبئية، فيجب أن يضبِط الدليل البِتَّ المتسخ لكتلة الذاكرة من خلال إرسال رسالة إلغاء صلاحية إلى تلك المعالجات التي تستخدِم خط الذاكرة المخبئية والمعالجات التي جرى ضبط رايتها فقط بهدف تجنب حركة مرور البث broadcast traffic.

يجب بعد ذلك أن يحاول أيّ معالج آخر قراءة كتلة الذاكرة، وسيجد الدليل ضبط البِت المتسخ، كما يجب أن يحصل الدليل على خط الذاكرة المخبئية المُحدَّث من المعالج مع البِت الصالح المضبوط حاليًا، ويعيد كتابة البيانات المتسخة إلى الذاكرة الرئيسية ثم إعادة هذه البيانات إلى المعالج المطلوب، مما يؤدي إلى ضبط البِت الصالح للمعالج الطالب في هذه العملية، ولاحظ أنّ هذا الأمر واضح للمعالج الطالب ويمكن أن يحتاج الدليل الحصول على تلك البيانات من مكان قريب جدًا أو من مكان بعيد جدًا.

لا يمكن أن يتوسع المخطط المؤلَّف من آلاف المعالجات التي تتصل بدليل واحد بصورة جيدة، إذ تتضمن توسّعات المخطط وجود تسلسل هرمي من الدلائل التي تتواصل فيما بينها باستخدام بروتوكول منفصل، كما يمكن أن تستخدِم الدلائل شبكة اتصالات ذات أغراض أعم للتواصل فيما بينها بدلًا من ناقل وحدة المعالجة المركزية، مما يسمح بالتوسع إلى أنظمة أكبر بكثير.

ج. تطبيقات NUMA

تُعَدّ أنظمة NUMA الأنسب لأنواع المشاكل التي تتطلب قدرًا كبيرًا من التفاعل بين المعالج والذاكرة، فمن المصطلحات الشائعة في محاكاة الطقس مثلًا هو تقسيم البيئة إلى صناديق صغيرة تستجيب بطرق مختلفة، بحيث تعكس المحيطات والأرض أو تخزن كميات مختلفة من الحرارة مثلًا، ويجب تغذية الاختلافات الصغيرة لمعرفة النتيجة الإجمالية أثناء تشغيل عمليات المحاكاة.

يؤثر كل صندوق على الصناديق المحيطة، إذ يعني وجود الشمس أكثر قليلًا مثلًا أنّ صندوقًا معينًا ينشر مزيدًا من الحرارة مما يؤثر على الصناديق المجاورة له، ولكن سيكون هناك الكثير من الاتصالات على عكس إطارات الصور الفردية في عملية التصيير Rendering التي لا تؤثر على بعضها، كما يمكن أن تحدث عمليةً مماثلةً إذا أردت تصميم نموذج لحادث سيارة، حيث سيُطوى كل صندوق صغير من السيارة التي تحاكيها بطريقة ما وسيمتص قدرًا من الطاقة.

ليس للبرمجيات معرفة مباشرة بأن النظام الأساسي هو نظام NUMA، ولكن يجب أن يتوخّى المبرمجون الحذر عند البرمجة لهذا النظام للحصول على أفضل أداء، وسيؤدي الاحتفاظ بالذاكرة بالقرب من المعالج الذي سيستخدِمها إلى أفضل أداء، ولكن يجب أن يستخدِم المبرمجون تقنيات مثل التشخيص Profiling لتحليل مسارات الشيفرة البرمجية المتّبَعة والعواقب التي تسببها الشيفرة البرمجية للنظام لاستخراج أفضل أداء.

3.4.4 ترتيب الذاكرة وقفلها

تجلب الذاكرة المخبئية متعددة المستويات والمعمارية متعددة المعالجات الفائقة بعض المشاكل المتعلقة بكيفية رؤية المبرمج لشيفرة المعالج البرمجية التي تكون قيد التشغيل.

لنفترض أنّ شيفرة البرنامج البرمجية تعمل على معالجَين في الوقت نفسه، وأنّ كلا المعالجين يشتركان بفعالية في منطقة واحدة كبيرة من الذاكرة، فإذا أصدر أحد المعالجَين تعليمات تخزين لوضع قيمة مسجّل في الذاكرة، فلا بد أنك تتساءل عن الوقت الذي يمكن فيه التأكد من أن المعالج الآخر يحمّل تلك الذاكرة التي سيرى قيمتها الصحيحة.

يمكن للنظام في أبسط الحالات أن يضمن أنه في حالة تنفيذ أحد البرامج لتعليمات التخزين، وبالتالي سترى أيّ تعليمات تحميل لاحقة هذه القيمة، إذ يُشار إلى ذلك باسم ترتيب الذاكرة الصارم Strict Memory Ordering، لأن القواعد لا تسمح بأيّ مجال للحركة، كما يجب أن تدرك أنّ هذا النوع من الأشياء يُعَدّ عائقًا خطيرًا أمام أداء النظام.

لا يُطلَب من ترتيب الذاكرة أن يكون صارمًا جدًا في كثير من الأحيان، إذ يمكن للمبرمج تحديد النقاط التي يحتاجها للتأكد من رؤية جميع العمليات المُعلَّقة بطريقة عامة، ولكن يمكن أن يكون هناك العديد من التعليمات من بين هذه النقاط حيث لا تكون الدلالات Semantics مهمة، ولنفترض الموقف التالي مثلًا الذي يمثل ترتيب الذاكرة:

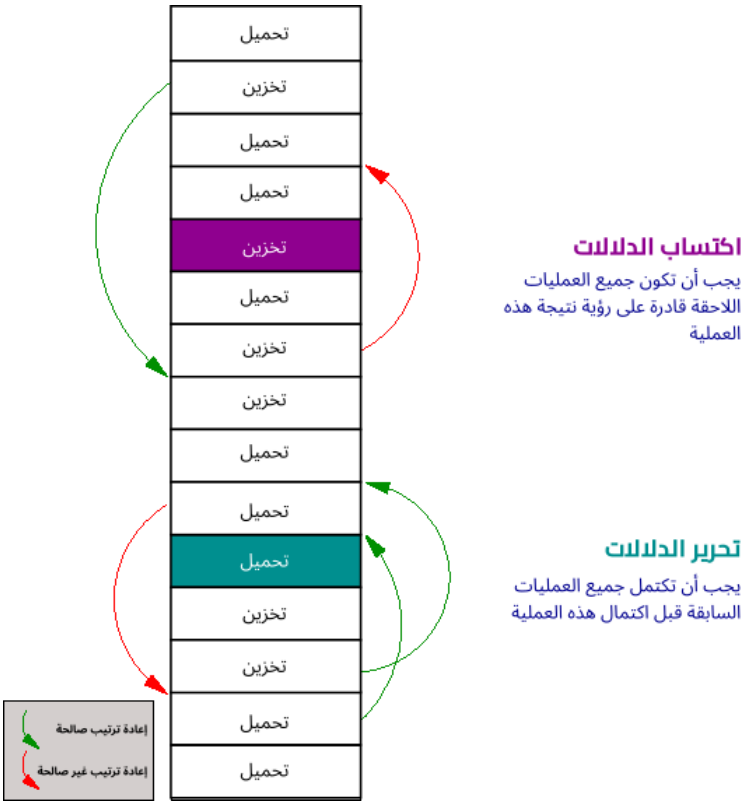
```
typedef struct {
int a;
int b;
} a_struct;

/*
 * مرّر مؤشرًا لتخصيصه بوصفه بنيةً جديدةً
 */
void get_struct(a_struct *new_struct)
{
  void *p = malloc(sizeof(a_struct));

  /* لا نهتم بترتيب التعليمتين التاليتين
  * اللتين ستُنفَّذان في النهاية */
  p->a = 100;
  p->b = 150;

  /* لكن يجب أن تُنفَّذا قبل التعليمة التالية.
  * وإلّا فسيتمكن معالج آخر ينظر إلى قيمة p
  * من أن يجدها تؤشّر إلى بنية قيمها غير مملوءة.
  */
```

```
    new_struct = p;
}
```

لدينا في هذا المثال عمليتَي تخزين يمكن تطبيقهما بأيّ ترتيب معيّن بما يناسب المعالج، ولكن يجب في الحالة الأخيرة تحديث المؤشر فقط بمجرد التأكد من اكتمال عمليتَي التخزين السابقتين، وإلّا فيمكن أن ينظر معالج آخر إلى قيمة p ويتبع المؤشر إلى الذاكرة ويحمّلها ويحصل على قيمة غير صحيحة تمامًا، لذا يجب أن تحتوي عمليات التحميل والتخزين على دلالات تصف سلوكها.

توصَف دلالات الذاكرة من حيث الأسوار Fences التي تحدّد كيفية إعادة ترتيب عمليات التحميل والتخزين، كما يمكن افتراضيًا إعادة طلب عملية التحميل أو التخزين في أيّ مكان، ويشبه اكتساب الدلالات Acquire Semantics السورَ الذي يسمح فقط لعمليات التحميل والتخزين بالتحرك للأسفل عبره، أي يمكنك ضمان أنّ أيّ عملية تحميل أو تخزين لاحقة سترى القيمة -لأنه لا يمكن نقلها فوقها- عند اكتمال هذا التحميل أو التخزين.

يُعَدّ تحرير الدلالات Release Semantics عكس ذلك، أي يسمح السور بأي عملية تحميل أو تخزين أن تكتمل قبله -أي التحرك للأعلى-، ولكن لا يوجد شيء قبلها للتحرك للأسفل. وبالتالي يمكنك تخزين أيّ عملية تحميل أو تخزين سابقة مكتملة عند معالجة التحميل أو التخزين باستخدام تحرير الدلالات.



شكل 14: اكتساب وإطلاق الدلالات

يمثّل الرسم التوضيحي عمليات إعادة الترتيب الصالحة للعمليات باستخدام اكتساب الدلالات وتحريرها.

سور الذاكرة الكامل full memory fence هو مزيج من اكتساب الدلالات وتحريرها، حيث لا يمكن إعادة ترتيب عمليات التحميل أو التخزين في أيّ اتجاه حول عملية التحميل أو التخزين الحالية، كما يستخدِم نموذج الذاكرة الأكثر صرامة سور ذاكرة كامل لكل عملية، في حين سيترك النموذج الأضعف كل عملية تحميل وتخزين على أساس تعليمات عادية قابلة لإعادة الترتيب.

ا. المعالجات ونماذج الذاكرة

تطبّق المعالجات المختلفة نماذج ذاكرة مختلفة، إذ يحتوي معالج x86 ومعالج AMD64 على نموذج ذاكرة صارم تمامًا، حيث تحتوي جميع عمليات التخزين على تحرير دلالات، أي يجب أن ترى أيّة عملية تحميل أو تخزين لاحقة نتيجةَ عملية التخزين، ولكن جميع عمليات التحميل لها دلالات عادية، كما تعطي بادئة القفل سورًا للذاكرة، في حين يسمح المعالج إيتانيوم Itanium لجميع عمليات التحميل والتخزين بأن تكون عاديةً ما لم يُجرَى إخباره صراحةً بغير ذلك.

ب. القفل

ليست معرفة متطلبات ترتيب الذاكرة لكل معمارية عمليةً ومناسبةً لجميع المبرمجين وسيجعل ذلك نقل البرامج وتنقيحها عبر أنواع المعالجات المختلفة أمرًا صعبًا، إذ يستخدِم المبرمجون مستوًى أعلى من التجريد يسمى القفل Locking للسماح بالتشغيل المتزامن للبرامج عندما يكون هناك وحدات معالجة مركزية متعددة، كما لا يمكن لأيّ معالج آخر الحصول على القفل حتى يُحرَّر عندما يحصل برنامج ما عليه لجزء من شيفرة برمجية، كما يجب أن يحاول المعالج أخذ القفل قبل أيّ أجزاء مهمة من الشيفرة البرمجية، فإذا لم يستطع الحصول عليه، فلن يستمر في عمله.

يمكنك رؤية كيف أنّ ذلك مقيَّد بتسمية دلالات ترتيب الذاكرة الموضَّحة سابقًا، كما نريد التأكد من أنه لن يُعاد طلب أيّ عمليات يجب أن يحميها القفل قبل الحصول عليه، وهذه هي الطريقة التي تعمل بها عملية اكتساب الدلالات، في حين يجب التأكد من أنّ كل عملية طبّقناها أثناء احتفاظنا بالقفل مكتملة عندما نحرره مثل مثال تحديث المؤشر الموضَّح سابقًا، وهذا ما يسمى بتحرير الدلالات.

هناك العديد من المكتبات البرمجية المتاحة التي تسمح للمبرمجين بعدم القلق بشأن تفاصيل دلالات الذاكرة واستخدام المستوى الأعلى من تجريد القفل `lock()` وإلغاء القفل `unlock()`.

صعوبات الأقفال

تجعل أنظمة القفل البرمجة أكثر تعقيدًا، إذ يمكنها أن تؤدي إلى تعطيل البرامج، ولنفترض أنّ معالجًا ما يحتفظ بقفل على بعض البيانات، وينتظر قفلًا على بيانات أخرى حاليًا، فإذا انتظر معالج آخر البيانات التي

يحتفظ بها المعالج الأول وكان قبل ذلك وقبل دخوله في حالة قفل يحتفظ ببيانات يريدها المعالج الأول ذاك لفك قفله، فسنواجه حالة تعطل تام، بحيث ينتظر كل معالج المعالج الآخر ولا يمكن لأيّ منهما الاستمرار بدون قفل المعالج الآخر.

ينشأ هذا الموقف بسبب حالة التسابق Race Condition في أغلب الأحيان التي تُعَدّ إحدى أصعب الأخطاء التي يمكن تعقّبها، فإذا كان هناك معالِجان يعتمدان على عمليات تحدث بترتيب معيّن في الوقت، فهناك دائمًا احتمال حدوث حالة تسابق، كما يمكن أن تصطدم أشعة جاما المنبعثة من نجم متفجر في مجرة أخرى بأحد المعالجات، مما يؤدي إلى الخروج عن ترتيب العمليات، ثم ستحدث حالة تعطل تام كما رأينا سابقًا، لذا يجب ضمان ترتيب البرامج باستخدام الدلالات وليس عبر الاعتماد على سلوكيات محددة لمرة واحدة.

يوجد وضع مماثل يسمى المنع Livelock وهو عكس التعطل Deadlock، إذ يمكن أن تكون إحدى الاستراتيجيات لتجنب التعطل أن يكون لديك قفل مؤدب Polite يرفض إعطاء القفل لكل مَن يطلبه، وقد يتسبب هذا القفل المؤدّب في جعل خيطين Threads يمنحان بعضهما القفل باستمرار دون الحاجة إلى أخذ القفل لفترة كافية لإنجاز العمل المهم والانتهاء من القفل، إذ يمكن أن يكون هناك وضع مشابه في الحياة الواقعية لشخصين يلتقيان عند الباب في الوقت نفسه، ويقول كلاهما: "لا، أنت أولًا، أنا أصر على ذلك" دون المرور عبر الباب نهائيًا.

استراتيجيات القفل

هناك العديد من الاستراتيجيات المختلفة لتطبيق سلوك الأقفال، إذ يُشار إلى القفل البسيط الذي يحتوي ببساطة على حالتين -مقفل Locked أو غير مقفل Unlocked- على أنه كائن مزامنة Mutex، وهو اختصار للاستبعاد المتبادل Mutual Exclusion الذي يعني أنه إذا كان لدى شخص ما قفلًا، فلا يمكن لشخص آخر الحصول عليه، وهناك عدد من الطرق لتطبيق قفل كائن المزامنة، إذ لدينا في أبسط الحالات ما يسمى بالقفل الدوار Spinlock إذ يبقى المعالج ضمن حلقة في انتظار أخذ القفل مثل طفل صغير يطلب من والديه شيئًا ويقول "هل يمكنني الحصول عليه الآن؟" باستمرار.

تكمن مشكلة هذه الاستراتيجية في أنها تضيع الوقت، إذ لا ينفّذ المعالج أيّ عملٍ مفيد بينما يكون متوقفًا ويطلب القفل باستمرار، وقد يكون ذلك مناسبًا للأقفال التي يُحتمَل أن تُقفَل لفترة قصيرة جدًا من الوقت فقط، ولكن يمكن أن يكون مقدار الوقت الذي يستغرقه القفل أطول بكثير في كثير من الحالات.

الاستراتيجية الأخرى هي السكون Sleep، حيث إذا لم يتمكن المعالج من الحصول على القفل، فسينفّذ بعض الأعمال الأخرى في انتظار إشعار بأن القفل متاح للاستخدام، وسنرى في المقالات القادمة كيف يمكن لنظام التشغيل تبديل العمليات وإعطاء المعالج مزيدًا من العمل لتنفيذه.

يُعَدّ كائن المزامنة حالةً خاصةً من متغير تقييد الوصول Semaphore الذي اخترعه عالم الحاسوب الهولندي ديكسترا Dijkstra، إذ يمكن ضبط متغير تقييد الوصول Semaphore لحساب عدد مرات الوصول

إلى الموارد في حالة توفر العديد منها، في حين يكون لديك كائن المزامنة Mutex في الحالة التي يكون فيها عدد الموارد يساوي واحدًا فقط.

لكن لا تزال أنظمة القفل هذه تواجه بعض المشاكل، إذ يرغب معظم الأشخاص في قراءة البيانات التي تُحدَّث في حالات نادرة فقط. يمكن أن يؤدي وجود جميع المعالجات التي ترغب في قراءة البيانات فقط التي تتطلب قفلًا إلى تنازع القفل حيث يُنجَز القليل من العمل لأن الجميع ينتظر الحصول على القفل نفسه لبعض البيانات.

دورة إدارة تطوير المنتجات

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

4. نظام التشغيل

4.1 دور نظام التشغيل وتنظيمه في معمارية الحاسوب

يدعم نظام التشغيل العملية الكاملة للحواسيب الحديثة، فهو عنصر أساسي في معمارية الحواسيب، لذا سنتعرّف في هذا المقال على دوره وكيفية تنظيمه.

4.1.1 تجريد العتاد

تتمثل العملية الأساسية لنظام التشغيل Operating System -أو OS اختصارًا- في تجريد Abstraction العتاد للمبرمج والمستخدِم، إذ يوفّر نظام التشغيل واجهات عامة للخدمات التي يقدمها العتاد الأساسي، كما يجب على المبرمجين معرفة تفاصيل العتاد الأساسي الأكثر خصوصيةً لتشغيل أيّ شيء في عالم خال من أنظمة التشغيل، إذ لن تعمل برامجهم على عتاد آخر حتى عند وجود اختلافات طفيفة في هذا العتاد.

4.1.2 تعدد المهام Multitasking

نتوقع من الحواسيب الحديثة تنفيذ العديد من الأشياء المختلفة في وقت واحد، لذا يجب التحكيم بين جميع البرامج المختلفة التي تعمل على النظام، ويُعَدّ ذلك وظيفة أنظمة التشغيل التي تسمح بحدوث ذلك بسلاسة، فنظام التشغيل مسؤول عن إدارة الموارد داخل النظام، إذ تتنافس المهام المتعددة على موارده أثناء تشغيله بما في ذلك وقت المعالج والذاكرة والقرص الصلب ودخل المستخدِم، كما تتمثل وظيفته في التحكيم في وصول المهام المتعددة لهذه الموارد بطريقة منظمة.

لا بد أنك مررت بفشل حاسوبك وتعطله مثل ظهور شاشة الموت الزرقاء Blue Screen of Death الشهيرة بسبب التنافس على هذه الموارد.

4.1.3 الواجهات الموحدة Standardised Interfaces

يرغب المبرمجون في كتابة برامج تعمل على أكبر عدد ممكن من المنصات العتادية، ويمكن ذلك من خلال دعم نظام التشغيل للواجهات الموحَّدة المعيارية، فإذا كانت دالة فتح ملف مثلًا على أحد الأنظمة `open()` وكانت `open_file()` و `openf()` على نظام آخر، فسيواجه المبرمجون مشكلةً مزدوجةً تتمثل في الاضطرار إلى تذكّر ما يفعله كل نظام مع عدم عمل البرامج على أنظمة متعددة.

تُعَدّ واجهة نظام التشغيل المتنقلة Portable Operating System Interface -أو بوسيكس POSIX اختصارًا- معيارًا مهمًا للغاية تطبّقه أنظمة تشغيل من نوع يونيكس UNIX، كما يملك نظام مايكروسوفت ويندوز معايير مشابهة، ويأتي حرف X في POSIX من نظام يونيكس Unix الذي نشأ منه المعيار، وهو اليوم الإصدار رقم 3 من مواصفات يونيكس الواحدة Single UNIX Specification Version 3 أو ISO/IEC 9945:2002 نفسه، كما أنه معيار مجاني ومتاح على الإنترنت.

كانت مواصفات يونيكس الواحدة ومعايير POSIX كيانات منفصلةً سابقًا، وقد أصدر اتحاد يسمّى المجموعة المفتوحة Open Group مواصفات يونيكس الواحدة، وكان متاحًا مجانًا وفقًا لمتطلبات هذا الاتحاد، وأحدث إصدار هو الإصدار الثالث من مواصفات يونيكس الواحدة، كما أُصدِرت معايير POSIX IEEE بوصفها معايير بالشكل [رقم المراجعة، رقم الإصدار] 1003 IEEE Std، ولم تكن متاحةً مجانًا، ومن إصداراتها IEEE 1003.1-2001 وهو مكافئ للإصدار الثالث من مواصفات يونيكس الواحدة.

دُمِج هذان المعياران المنفصلان فيما يُعرف باسم الإصدار الثالث من مواصفات يونيكس الواحدة، ووحّدته منظمة ISO بالاسم ISO/IEC 9945:2002 في بداية عام 2002، لذا عندما يتحدث الناس عن معيار POSIX أو SUS3 أو ISO/IEC 9945:2002، فإنهم يعنون الشيء نفسه.

4.1.4 الأمن

يُعَدّ الأمن مهمًا جدًا في الأنظمة متعددة المستخدِمين، إذ يكون نظام التشغيل -بصفته المتحكّم في الوصول إلى النظام- مسؤولًا عن ضمان أنّ الأشخاص الذين لديهم الأذونات الصحيحة فقط يمكنهم الوصول إلى الموارد، فإذا امتلك مستخدِم أحد الملفات مثلًا، فلا ينبغي السماح لمستخدِم آخر بفتحه وقراءته، ولكن هناك حاجة لوجود آليات لمشاركة هذا الملف بأمان بين المستخدِمين إذا أرادوا ذلك.

أنظمة التشغيل هي برامج كبيرة ومعقدة تحتوي على مشكلات أمنية في أغلب الأحيان، إذ يستفيد الفيروس أو الدودة الفيروسية من هذه الأخطاء غالبًا للوصول إلى الموارد التي لا ينبغي السماح لها بالوصول إليها مثل الملفات أو اتصال الشبكة، لذا يجب عليك تثبيت حزم التصحيح أو التحديثات التي يوفرها مصنّع نظام التشغيل لمحاربتها.

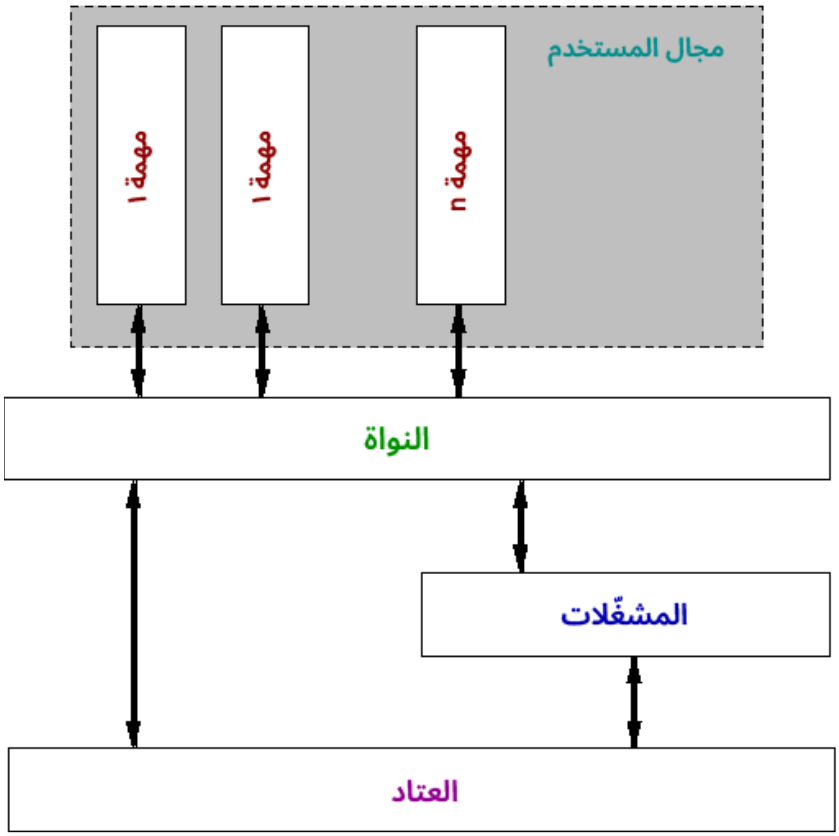
4.1.5 الأداء

يوفِّر نظام التشغيل العديد من الخدمات للحاسوب، لذلك يُعَدّ أداؤه أمرًا بالغ الأهمية، إذ تعمل أجـزاء كثـيرة من نظام التشغيل بصـورة متكـررة، كمـا يمكن أن تـؤدي زيـادة عـدد دورات المعـالج إلى انخفـاض كبـير في أداء النظام، ويحتاج نظام التشغيل لاسـتغلال مـيزات العتـاد الأساسـي للتأكـد من الحصـول على أفضـل أداء ممكن لإجراء العمليـات، وبالتـالي يجب على مـبرمجي الأنظمـة فهم التفاصـيل الدقيقـة للمعماريـة الـتي يبنـون نظـام التشغيل من أجلها.

تكون مهمة مبرمجي الأنظمة في كثير من الحـالات هي تحديـد سياسـات النظـام، إذ تـؤدي الآثـار الجانبيـة لجعل جزء من نظام التشغيل يعمل بصورة أسر ع إلى جعل جزء آخر يعمل بصـورة أبطـأ أو أقـل كفـاءة، لـذا يجب على مبرمجي الأنظمة فهم كل هذه المقايضات عند بناء نظام التشغيل.

4.2 تنظيم نظام التشغيل

يُعَدّ نظام التشغيل منظمًا تقريبًا كما في الصورة التالية:



شكل 15: نظام التشغيل

تنظيم النواة Kernel: تُشغَّل عمليات النـواة مباشـرةً في مجـال المسـتخدِم Userspace، وتتواصـل النـواة مباشرةً مع العتاد Hardware وعبر المشغّلات Drivers.

4.2.1 النواة Kernel

تُعَدّ النواة نظام تشغيل، وتجرّد المشغّلات Drivers العتاد للنواة كما تجرّد النواة العتاد لبرامج المستخدِم، حيث يوجد العديد من أنواع بطاقات الرسوم المختلفة على سبيل المثال، ولكل منها ميزات مختلفة قليلًا عن بعضها البعض، ولكن طالما أن النواة تصدّر واجهة برمجة تطبيقات API، فيمكن للأشخاص الذين لديهم إذن الوصول إلى مواصفات العتاد كتابةُ برامج للمشغّلات لتطبيق هذه الواجهة، ويمكن للنواة باستخدام هذه الطريقة الوصول إلى أنواع مختلفة من العتاد.

تُوصَف النواة بأن لها صلاحيات Privileged، فللعتاد أدوار مهمة يؤديها لتشغيل مهام متعددة والحفاظ على أمان النظام، ولكن لا تُطبَّق هذه القواعد على النواة، كما يجب أن تتعامل النواة مع البرامج التي تتعطل، فوظيفة أنظمة التشغيل هي فقط تنظيم العمل والتحكيم بين العديد من البرامج التي تعمل على النظام نفسه، وليس هناك ما يضمن أنها ستتصرف لحل المشاكل، ولكن سيصبح النظام بأكمله عديم الفائدة في حالة تعطل أي جزء داخلي من نظام التشغيل، كما يمكن أن تستغل عمليات المستخدِم مشاكل الأمان لترفع مستواها إلى مستوى صلاحيات النواة، وبالتالي يمكنها الوصول إلى أيّ جزء من النظام.

4.2.2 النواة الأحادية Monolithic والنواة الدقيقة Microkernel

أحد الأمور الجدلية التي تُطرَح غالبًا حول أنظمة التشغيل هو ما إذا كانت النواة أحادية Monolithic أو نواة دقيقة Microkernel.

تُعَدّ النواة الأحادية الأكثر شيوعًا كما هو الحال في معظم أنظمة يونيكس الشائعة مثل لينكس، إذ تكون النواة في هذا النموذج ذات صلاحيات كبيرة، وتحتوي على مشغّلات العتاد ومتحكمات الوصول إلى نظام الملفات وفحص الأذونات والخدمات مثل نظام ملفات الشبكة Network File System -أو NFS اختصارًا.

تتمتع النواة دائمًا بصلاحيات، لذلك إذا تعطل أيّ جزء منها، فيُحتمَل أن يتوقف النظام بأكمله، وإذا كان لدى مشغّل خطأ برمجي ما bug، فيمكنه الكتابة في أيّ ذاكرة في النظام دون أيّ مشاكل، مما يؤدي في النهاية إلى تعطل النظام.

تحاول معمارية النواة الدقيقة تقليل هذا الاحتمال من خلال جعل الجزء الذي يمتلك الصلاحيات من النواة صغيرًا قدر الإمكان. هذا يعني أن معظم النظام يعمل كبرامج دون صلاحيات، مما يحد من الضرر الذي يمكن أن يسبّبه أيّ مكونٍ معطَّل، فمثلًا يمكن تشغيل مشغّلات العتاد في عمليات منفصلة، وبالتالي إذا تعطّل أحد هذه المشغّلات، فلن يتمكّن من الكتابة في أيّ ذاكرة غير تلك المخصصة له.

تبدو معمارية النواة الدقيقة جيدةً، ولكنها ستؤدي إلى المشكلتين التاليتين:

1. انخفاض الأداء، إذ يمكن أن يؤدي التواصل بين العديد من المكونات المختلفة إلى تقليل الأداء.
2. يُعَدّ تطبيقها أصعب قليلًا على المبرمجين.

تأتي هذه المشاكل بسبب تطبيق معظم الأنوية الدقيقة باستخدام نظام قائم على تمرير الرسائل Message Passing بهدف الحفاظ على الفصل بين المكونات، ويشار إلى هذا النظام عـادةً باسـم التواصـل بين العمليـات Inter-process Communication أو IPC اختصارًا.

يحدث التواصل بين المكونات باستخدام رسائل منفصلة يجب تجميعها ضـمن حـزم وإرسـالها إلى المكـوِّن الآخر وتفكيكها وتشغيلها وإعادة تجميعها وإعادة إرسـالها ثم تفكيكهـا مـرةً أخـرى للحصـول على النتيجـة، وهـذه خطوات كثيرة لطلبٍ بسيط إلى حد ما من مكون خارجي، ويمكن أن تجعـل أحـد الطلبـات المكـون الآخـر يُجـري طلبات أكثر لمكونات أكثر، وستتفاقم المشكلة.

كانت تطبيقات تمرير الرسـائل البطيئـة مسـؤولة إلى حـد كبـير عن الأداء الضـعيف لأنظمـة النـواة الدقيقـة القديمة، وكانت مفاهيم تمرير الرسـائل أصـعب قليلًا على المـبرمجين، ولم تكن الحمايـة المُحسَّـنة من تشـغيل المكونات بصورة منفصلة كافيةً للتغلب على هذه العقبات في أنظمة النواة الدقيقة القديمة، لذا أصبحت قديمة الطراز، في حين تكون الاستدعاءات بين المكونات استدعاءات وظيفيةً بسيطةً في النواة الأحادية كما هو معتـاد لدى جميع المبرمجين.

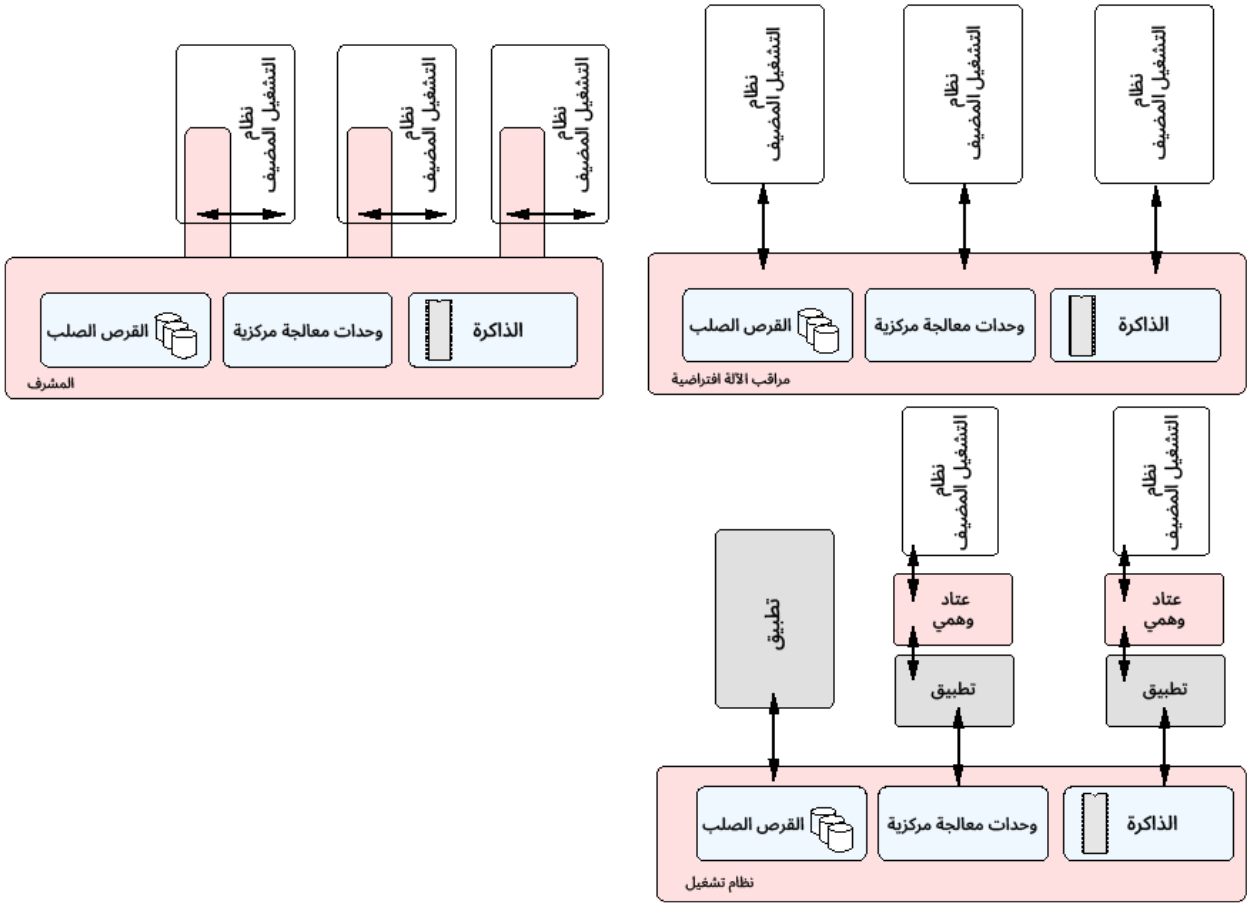
لا توجد إجابة محددة حول أفضل تنظيم، وقد بـدأت العديـد من المناقشـات في الأوسـاط الأكاديميـة وغـير الأكاديمية حول ذلك، لذا نأمل أن تكون قادرًا على اتخاذ قرار بنفسك عندما تتعلم المزيد عن أنظمة التشغيل.

ا. الوحدات Modules

تطبّق نواة لينكس نظـام الوحـدات، حيث يمكن تحميـل المشـغّلات في النـواة المشـغَّلة مباشـرةً كمـا هـو مطلوب، وهذا أمر جيد لأنّ المشغّلات التي تشكّل جزءًا كبيرًا من شـيفرة نظـام التشـغيل لا تُحمَّـل للأجهـزة غـير الموجودة في النظام، كما يمكن لأيّ شخص يريـد أن يصـنع أكـثر نـواة عامـة ممكنـة -أي تعمـل على العديـد من الأجهزة المختلفة مثل RedHat أو Debian- تضمينَ معظم المشغّلات بوصفها وحدات تُحمَّل فقـط إذا احتـوى النظام الذي يعمل عليه على العتاد المتاح، لكن تُحمَّل الوحدات مباشرةً في النواة ذات الصـلاحيات وتعمـل على مستوى الصلاحيات نفسه لبقية أجزاء النواة، لذلك لا يزال يُعَدّ النظام نواةً أحاديةَ.

4.2.3 الافتراضية Virtualisation

يرتبط مفهوم العتاد الوهمي أو الافتراضي ارتباطًا وثيقًا بـالنواة، إذ تُعَـدّ الحواسـيب الحديثـة قويـة جـدًا، ولا يُفضَّل استخدامها على أساس نظام واحد كامـل، وإنمـا تقسـيم الحاسـوب الحقيقي الواحـد إلى آلات افتراضـية منفصلة virtual machines، إذ تبحث كلٌّ من هذه الآلات الافتراضية عن جميع الأهداف والأغراض بوصفها آلة منفصلة تمامًا بالرغم من أنها فيزيائيًا موجودة في المكان نفسه.



شكل 16: بعض طرق تطبيق الافتراضية المختلفة

يمكن تنظيم الافتراضية بعدة طـرق مختلفـة، إذ يمكن تشـغيل مـراقب آلـة افتراضـية Virtual Machine Monitor صغير مباشرةً على العتاد وتوفير واجهة لأنظمة تشغيل المضيف التي تعمـل في الأعلى، ويُطلَـق على مراقب الآلة الافتراضية VMM اسم المشرف Hypervisor من الكلمة Supervisor.

يشترك المشرف في كثير الأمور مع النواة الدقيقة، كما يسعيان ليكوّنا طبقات صغيرةً لتقديم العتاد بطريقة آمنة عن الطبقات التي تعلوها، ويمكن ألّا يكون لدى نظام التشغيل الموجود في الطبقة العليا أيّ فكرة عن وجـود المشرف Hypervisor على الإطلاق، إذ يقدم هذا المشرف ما يبدو أنه نظام كامل، ويعترض العمليات بين نظـام التشغيل المضيف والعتاد ويقدّم مجموعةً فرعيةً من موارد النظام لكل منها.

يُستخدَم المشرف غالبًا على الأجهزة الكبيرة التي تحتوي على العديد من وحدات المعالجـة المركزيـة والكثـير من ذواكر RAM لتطبيق عملية التجزيء Partitioning، وهذا يعني أنه يمكن تقسيم الجهاز إلى أجهزة افتراضــية أصغر، كما يمكنك تخصيص المزيد من الموارد لتشغيل الأنظمة حسب المتطلبات، ويُعَدّ المشرفون الموجودون على العديد من أجهزة IBM الكبيرة معقدةً للغاية مع ملايين الأسطر من الشيفرة البرمجية مـع توفـير العديـد من خدمات إدارة النظام.

الخيار الآخر هو جعل نظام التشغيل على دراية بالمشرف الأساسي وطلب مـوارد النظـام عـبره، إذ يشـار إلى ذلك في بعض الأحيان باسـم شـبه الوهميـة Paravirtualisation نظـرًا لطبيعتـه غـير المكتملـة، وهـو مشـابه

للطريقة التي تعمل بها الإصدارات الأولى من نظام Xen الذي يُعَدّ حلًا وسطًا، إذ تـوفّر هـذه الطريقـة أداءً أفضـل لأن نظام التشغيل يطلب صراحةً موارد النظام من المشرف عند الحاجة بدلًا من أن يطبّق المشرف الأمور آليًا.

أخيرًا، يمكن أن تصادف موقفًا حيث يقدّم التطبيق الذي يعمـل على نظـام التشـغيل الحـالي نظامًـا وهميًـا يتضمن وحدة معالجة مركزية وذاكرةً ونظام BIOS وقرص صلب وغير ذلك، ويمكن تشغيل نظـام تشـغيل عـادي عليه، إذ يحوّل التطبيق الطلبات إلى العتاد ثم إلى العتـاد الأساسـي عـبر نظـام التشـغيل الحـالي، وهـذا مشـابه لكيفية عمل برنامج VMWare.

تتطلب هذه الطريقة تكلفةً أكبر، إذ يتعين على عملية التطبيق محاكاة نظـام بأكملـه وتحويـل كـل شـيء إلى طلبات من نظام التشغيل الأساسي، ولكنها تتيح محاكاةً معماريةً مختلفةً تمامًا، إذ يمكنك ترجمة التعليمات آليًا من نوع معالج إلى آخر كما يفعل نظام روزيتا Rosetta مع برمجيات Apple التي انتقلت من معـالج PowerPC إلى المعالجات القائمة على إنتل Intel.

يُعَدّ الأداء مصدر قلق كبير عند استخدام أيّ من تقنيات الوهمية، إذ يجب أن تمر العمليات -التي كانت تُعَدّ سابقًا عمليات سريعةً ومباشرةً على العتاد- عبر طبقات التجريد.

ناقشت شركة إنتل Intel دعم العتاد للوهمية لتكون موجودةً في أحدث معالجاتها، إذ تعمل هذه التوسعات من خلال رفع استثناء خـاص للعمليـات الـتي يمكن أن تتطلب تـدخّل مـراقب الآلـة الافتراضـية، وبالتـالي فـإن المعالج يشبه المعالج غير الافتراضي الخـاص بـالتطبيق الـذي يعمـل عليـه، ولكن يمكن اسـتدعاء مـراقب الآلـة الافتراضية عندما يقدّم هذا التطبيق طلبات للحصول على موارد يمكن مشاركتها بين أنظمـة تشـغيل المضـيف الأخرى.

يوفّر ذلك أداءً فائقًا لأنّ مراقب الآلة الافتراضية لا يحتاج إلى مراقبة كل عمليـة لمعرفـة مـا إذا كـانت آمنـةً، ولكن يمكنه الانتظار حتى يُعلِم المعالج بحدوث شيء غير آمن.

ا. القنوات السرية Covert Channels

إذا لم يكن تقسيم النظام ساكنًا وإنما آليًا، فهناك مشكلة أمنية محتملة متضمنة في النظام ويُعَـدّ هـذا عيبًـا أمنيًا يتعلق بالآلات الافتراضية. تُخصَّص الموارد لأنظمة التشغيل التي تعمل في الطبقـة العليـا حسـب الحاجـة في النظام الآلي، وبالتالي إذا كان أحد هذه الأنظمة ينفّذ عمليات مكثفةً لوحـدة المعالجـة المركزيـة بينمـا ينتظـر النظام الآخر وصـول البيانـات من الأقـراص الصـلبة، فسـتُمنَح المهمـة الأولى مزيـدًا من طاقـة وحـدة المعالجـة المركزية، في حين سيحصل كل منهما على 50% من طاقة وحدة المعالجة المركزية في النظام الساكن، وسيُهدَر الجزء غير المستخدَم.

يفتح التخصيص الآلي قناة اتصال بين نظامَي التشغيل التي تكون كافيةً للتواصـل في نظـام ثنـائي في أيّ مكـان يمكن الإشـارة فيـه إلى تلـك الحـالتين، لكن تخيـل أنّ كلا النظـامين آمنـان جـدًا، ولا ينبغي أن تكـون أيّ

معلومـات قـادرةً على المـرور بينهمـا على الإطلاق، كمـا يمكن أن يتـآمر شخصـان لـديها إذن وصـول لتمريـر المعلومات فيما بينهما من خلال كتابة برنامجين يحاولان أخذ كميات كبيرة من الموارد في الوقت نفسه.

إذا أخذ أحدهما مساحةً كبيرةً من الذاكرة، فسيكون هناك قدر أقل من المساحة المتاحة للآخر؛ أمـا إذا تعقّبـا الحد الأقصى من التخصيصات، فيمكن نقل القليل من المعلومات فقط، ولنفترض أنهما اتفقا على التحقق في كل ثانية مما إذا كان بإمكانهما تخصيص هذا القدر الكبير من الذاكرة، فإذا كان الطـرف الهـدف قـادرًا على ذلـك، فستُعَدّ هذه الحالة 0 ثنائيًا، وإذا لم يستطع ذلك -أيّ أن الجهاز الآخر يحتوي على كل الذاكرة-، فستُعَدّ هذه الحالـة 1 ثنائيًـا، كمـا أنـه ليس معـدل البيانـات المُقـدَّر ببت واحـد في الثانيـة مـذهلًا، ولكن هـذا يـدل على وجـود تدفق للمعلومات.

يسمى ذلك بالقناة السرية Covert Channel، وهـذا يظهِـر أنّ الأمـور ليسـت بهـذا البسـاطة على مـبرمج الأنظمة بالرغم من وجود أمثلة عن انتهاكات أمنية في مثل هذه الآليات.

4.2.4 مجال المستخدم

نسمي المكان الذي يشغِّل فيه المستخدِم البرامج باسم مجال المستخدِم Userspace، إذ يعمل كل برنامج في مجال مستخدِم، ويتواصل مع النواة عبر استدعاءات النظام التي سنوضحّها في المقال القادم، كمـا لا يتمتـع مجال المستخدِم بصلاحيات Unprivileged، إذ يمكن لبرامج المسـتخدِم تطـبيق مجموعـة محـدودة فقـط من الأشياء، ويجب ألّا تكون قادرةً على تعطيل البرامج الأخرى حتى إذا تعطلت هي نفسها.

4.3 استدعاءات النظام System Calls

استدعاءات النظام system calls هي كيفية تفاعل برامج مجال المستخدِم Userspace مـع نـواة النظـام Kernel، إذ سنشرح فيما يلي المبدأ العام لكيفية عمل هذه الاستدعاءات، وسنتعرّف على الصلاحيات في نظـام التشغيل للوصول إلى الموارد.

4.3.1 أرقام استدعاءات النظام

لكل استدعاء نظام رقم يعرفه مجـال المسـتخدِم والنـواة، إذ يعـرِف كلاهمـا أنّ رقم اسـتدعاء النظـام 10 هـو الاستدعاء `open()` ورقم استدعاء النظام 11 هو الاستدعاء `read()` على سبيل المثال.

تُعَدّ واجهة التطبيق الثنائية Application Binary Interface -أو ABI اختصارًا- مشابهةً جدًا لواجهة برمجة التطبيقـات API، ولكنهـا مُخصَّصـة للعتـاد بـدلًا من أن تكـون خاصـةً بالبرمجيـات، إذ سـتحدّد واجهـة برمجـة التطبيقات API المسجّل Register الذي يجب إدخال رقم استدعاء النظام فيـه لتتمكّن النـواة من العثـور عليـه عندما يُطلب منها إجراء استدعاء النظام.

4.3.2 الوسائط Arguments

لا تكون استدعاءات النظام جيدةً بدون الوسائط، فالاستدعاء `open()` مثلًا يحتاج إلى إعلام النواة بالضبط بالملف الذي يجب فتحه، وستحدّد واجهة ABI أيًا من وسائط المسجّلات التي يجب وضعها لاستدعاء النظام.

4.3.3 المصيدة Trap

يجب أن تكون هناك طريقة ما للاتصال بالنواة التي نريد إجراء استدعاء نظام إليها، إذ تعرِّف جميع المعماريات الحاسوبية تعليمةً تسمى عادةً `break` أو شيئًا آخر مشابه يشير إلى العتاد الذي نريد إجراء استدعاء النظام إليه، وستخبر هذه التعليمة العتاد بتعديل مؤشر التعليمة ليؤشّر إلى معالج استدعاءات النظام الخاص بالنواة، إذ يخبر نظام التشغيل العتاد بمكان وجود معالج استدعاء النظام عندما يضبط نفسه، لذلك يفقد مجال المستخدِم السيطرة على البرنامج وتمريره إلى النواة بمجرد أن يستدعي التعليمة `break`.

يُعَدّ ما تبقى من هذه العملية بسيطًا إلى حد ما، إذ تبحث النواة في المسجل المُعرَّف مسبقًا عن رقم استدعاء النظام وتبحث عنه في جدول لمعرفة الدالة التي يجب أن تستدعيها، وتُستدعَى هذه الدالة وتنفّذ ما يجب تنفيذه، ثم تضع القيمة المُعادة في مسجل آخر تعرّفه الواجهة ABI بوصفه مسجّل إعادة Return.

تتمثل الخطوة الأخيرة في أن تنفّذ النواة تعليمات قفز إلى برنامج مجال المستخدِم لتتمكّن من المتابعة من حيث توقفت، ويحصل برنامج مجال المستخدِم على البيانات التي يحتاجها من مسجِّل الإعادة ثم يكمل عمله، كما يمكن أن تصبح تفاصيل هذه العملية خطيرة للغاية، إلّا أنّ هذا كله يتعلق باستدعاء النظام.

4.3.4 مكتبة libc

يمكنك تنفيذ كل ما سبق يدويًا لكل استدعاء نظام، لكن تنفّذ مكتبات النظام معظم العمل نيابةً عنك عادةً، والمكتبة القياسية التي تتعامل مع استدعاءات النظام على أنظمة يونيكس هي مكتبة `libc`.

4.3.5 تحليل استدعاء النظام

بما أنّ مكتبات النظام تجعل الأنظمة تستدعي نيابة عنك، فيجب تطبيق اختراق منخفض المستوى لتوضيح كيفية عمل استدعاءات النظام، وسنوضح كيفية عمل أبسط استدعاء نظام `getpid()` الذي لا يأخذ أيّ وسيط ويعيد معرّف البرنامج أو العملية التي تكون قيد التشغيل حاليًا.

```
#include <stdio.h>

/* خاصة باستدعاء النظام syscall() */
#include <sys/syscall.h>
#include <unistd.h>
```

```
/* أرقام استدعاءات النظام */
#include <asm/unistd.h>

void function(void)
{
  int pid;

  pid = __syscall(__NR_getpid);
}
```

نبدأ بكتابة برنامج صغير بلغة C لتوضيح آلية عمل استدعاءات النظام، وأول شيء يجب ملاحظته هو وجود الوسيط `syscall` الذي توفّره مكتبات النظام لإجراء استدعاءات النظام مباشرةً، إذ يوفِّر هذا الوسيط طريقةً سهلةً للمبرمجين لإجراء استدعاءات النظام مباشرةً دون الحاجة إلى معرفة إجراءات لغة التجميع الدقيقة لإجراء الاستدعاء على عتادهم.

نستخدِم الدالة `getpid()` لأنّ استخدام اسم دالة رمزي في شيفرتك البرمجية أوضح وتعمل الدالة `getpid()` بطرق مختلفة جدًا على أنظمة مختلفة، إذ يمكن تخزين الاستدعاء `getpid()` في الذاكرة المخبئية في نظام لينكس مثلًا، لذا إذا جرى تشغيله مرتين، فلن تتحمل مكتبة النظام عقوبة الاضطرار إلى إجراء استدعاء نظام بالكامل للعثور على المعلومات نفسها مرةً أخرى.

تُعرَّف أرقام استدعاءات النظام في الملف asm/unistd.h من مصدر النواة في نظام لينكس، وبما أنّ هذا الملف موجود في المجلد الفرعي `asm`، فسيختلف ذلك لكل معمارية يعمل عليها نظام لينكس، كما تُعطَى أرقام استدعاءات النظام اسمًا `#define` يتكون من `__NR_`، وبالتالي يمكنك رؤية أنّ شيفرتك البرمجية ستجري استدعاء النظام `getpid` ويخزّن القيمة في المعرِّف `pid`.

سنلقي نظرةً على كيفية تطبيق العديد من المعماريات لهذه الشيفرة البرمجية وسنطّلع على الشيفرة البرمجية الحقيقية التي يمكن أن تكون خطيرةً ولكن يجب الالتزام بها، فهذه هي بالضبط الطريقة التي يعمل بها نظامك.

ا. معمارية PowerPC

يُعَدّ نظام PowerPC معماريةَ RISC شائعة في حواسيب Apple القديمة، وهو جوهر أجهزة أحدث إصدار من Xbox مثلًا، وفيما يلي مثال عن استدعاء نظام PowerPC:

```
/* يتلِف استدعاءُ النظام المسجّلاتِ نفسها لاستدعاء الدالة في نظام powerpc،
* باستثناء المسجّل LR الذي يحتاجه التسلسل "sc; bnslr"
* والمسجّل CR حيث يُتلَف المسجل CR0.SO فقط الذي يشير إلى
```

```
* حالة إعادة خطأ.
*/

#define __syscall_nr(nr, type, name, args...)
  unsigned long __sc_ret, __sc_err;
  {
    register unsigned long __sc_0  __asm__ ("r0");
    register unsigned long __sc_3  __asm__ ("r3");
    register unsigned long __sc_4  __asm__ ("r4");
    register unsigned long __sc_5  __asm__ ("r5");
    register unsigned long __sc_6  __asm__ ("r6");
    register unsigned long __sc_7  __asm__ ("r7");
    __sc_loadargs_##nr(name, args);
    __asm__ __volatile__
      ("sc           \n\t"
        "mfcr %0      "
      : "=&r" (__sc_0),
        "=&r" (__sc_3),  "=&r" (__sc_4),
        "=&r" (__sc_5),  "=&r" (__sc_6),
        "=&r" (__sc_7)
      : __sc_asm_input_##nr
      : "cr0", "ctr", "memory",
        "r8", "r9", "r10","r11", "r12");
    __sc_ret = __sc_3;
    __sc_err = __sc_0;
  }
  if (__sc_err & 0x10000000)
  {
    errno = __sc_ret;
    __sc_ret = -1;
  }
  return (type) __sc_ret

#define __sc_loadargs_0(name, dummy...)
  __sc_0 = __NR_##name
#define __sc_loadargs_1(name, arg1)
```

```
  __sc_loadargs_0(name);
  __sc_3 = (unsigned long) (arg1)
#define __sc_loadargs_2(name, arg1, arg2)
  __sc_loadargs_1(name, arg1);
  __sc_4 = (unsigned long) (arg2)
#define __sc_loadargs_3(name, arg1, arg2, arg3)
  __sc_loadargs_2(name, arg1, arg2);
  __sc_5 = (unsigned long) (arg3)
#define __sc_loadargs_4(name, arg1, arg2, arg3, arg4)
  __sc_loadargs_3(name, arg1, arg2, arg3);
  __sc_6 = (unsigned long) (arg4)
#define __sc_loadargs_5(name, arg1, arg2, arg3, arg4, arg5)
  __sc_loadargs_4(name, arg1, arg2, arg3, arg4);
  __sc_7 = (unsigned long) (arg5)

#define __sc_asm_input_0 "0" (__sc_0)
#define __sc_asm_input_1 __sc_asm_input_0, "1" (__sc_3)
#define __sc_asm_input_2 __sc_asm_input_1, "2" (__sc_4)
#define __sc_asm_input_3 __sc_asm_input_2, "3" (__sc_5)
#define __sc_asm_input_4 __sc_asm_input_3, "4" (__sc_6)
#define __sc_asm_input_5 __sc_asm_input_4, "5" (__sc_7)

#define _syscall0(type,name)
type name(void)
{
  __syscall_nr(0, type, name);
}

#define _syscall1(type,name,type1,arg1)
type name(type1 arg1)
{
  __syscall_nr(1, type, name, arg1);
}

#define _syscall2(type,name,type1,arg1,type2,arg2)
type name(type1 arg1, type2 arg2)
```

```
{
  __syscall_nr(2, type, name, arg1, arg2);
}

#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)
type name(type1 arg1, type2 arg2, type3 arg3)
{
  __syscall_nr(3, type, name, arg1, arg2, arg3);
}

#define
_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)
type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4)
{
  __syscall_nr(4, type, name, arg1, arg2, arg3, arg4);
}

#define
_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,
arg5)
type name(type1 arg1, type2 arg2, type3 arg3, type4 arg4, type5 arg5)
{
  __syscall_nr(5, type, name, arg1, arg2, arg3, arg4, arg5);
}
```

يوضِّح جزء الشيفرة البرمجية السابق من ملف ترويسة النواة asm/unistd.h كيف يمكننا تطبيق استدعاءات النظام على نظام PowerPC، ويمكن أن يبدو الأمر معقدًا للغاية، ولكن لنشرحه خطوةً خطوة.

انتقل أولًا إلى نهاية المثال إلى تعريف وحدات الماكرو `_syscallN`، إذ يمكنك رؤية أنّ هناك العديد من وحدات الماكرو ويأخذ كل منها وسيطًا آخر تدريجيًا، وسنركّز على أبسط إصدار وهو `_syscall0` للبدء به والذي لا يتطلب سوى وسيطَين هما نوع القيمة المُعادة لاستدعاء النظام مثل `int` أو `char` واسم استدعاء النظام، إذ يكون مع الاستدعاء `getpid` بالصورة `_syscall0(int,getpid)`.

سنبدأ الآن بتفكيك الماكرو `__syscall_nr` الذي لا يختلف عما كان عليه سابقًا، إذ سنأخذ عدد الوسائط على أنه المعامِل الأول ثم النوع والاسم والوسائط الفعلية، فالخطوة الأولى هي التصريح عن بعض الأسماء للمسجّلات، إذ يشير الاسم `__sc_0` إلى المسجّل `r0` أي المسجّل 0، ويستخدِم المصرِّف Compiler

المسجلات بالطريقة التي يريدها، لهذا السبب يجب أن نعطيه قيودًا حتى لا يقرّر استخدام المسجّل الذي نحتاجه بطريقة مخصصة.

نستدعي بعد ذلك `sc_loadargs` مع المعامِل `##` الذي يُعَدّ أمر لصق يُستبدَل بالمتغير `nr`، وسنوسّعه إلى `__sc_loadargs_0(name, args);`، ويمكننا رؤية `__sc_loadargs` الذي يضبط `__sc_0` ليكون رقم استدعاء النظام، ولاحظ معامِل اللصق مرةً أخرى مع البادئة `__NR_` واسم المتغير الذي يشير إلى مسجّل معيّن، لذا تُستخدَم هذه الشيفرة البرمجية السابقة ذات المظهر الصعب لوضع رقم استدعاء النظام في المسجّل 0، كما يمكنك باتباع الشيفرة البرمجية السابقة رؤية أن وحدات الماكرو الأخرى ستضع وسائط استدعاء النظام في المسجّل `r3` عبر المسجّل `r7`، ويمكنك فقط الحصول على 5 وسائط على أساس حد أقصى لاستدعاء النظام.

سنعالج الآن القسم `__asm__`، إذ لدينا هنا ما يسمّى بالتجميع المُضمَّن Inline Assembly لأنها شيفرة تجميع مختلطة مع الشيفرة المصدرية، وهذه الصيغة معقدة بعض الشيء، لذلك سنشير إلى الأجزاء المهمة منها فقط، كما عليك تجاهل البت `__volatile__` حاليًا والذي يخبر المصرّف أنّ هذه الشيفرة البرمجية لا يمكن التنبؤ بها، لذا لا تحاول التعامل معها بذكاء، كما أنّ كل الأشياء الموجودة بعد النقطتين هي طريقة للتواصل مع المصرّف حول ما يفعله التجميع المضمَّن لمسجلات وحدة المعالجة المركزية، ويجب أن يعرِف المصرّف ذلك حتى لا يحاول استخدام أيّ من هذه المسجلات بطرق يمكنها التسبب في حدوث عطل.

لكن الجزء المهم هو وجود تعليمتَي التجميع في الوسيط الأول، إذ ينفِّذ الاستدعاء `sc` كل العمل، وهذا كل ما عليك تطبيقه لإجراء استدعاء نظام، وبالتالي يحدث ما يلي عند إجراء استدعاء النظام، إذ يعرف المعالِج المُقاطَع أنه يجب عليه نقل التحكم إلى جزء معيّن من إعداد الشيفرة البرمجية في وقت بدء تشغيل النظام لمعالجة المقاطعات، كما توجد هناك العديد من المقاطعات، وتُعَدّ استدعاءات النظام إحداها، وتبحث هذه الشيفرة البرمجية بعد ذلك في المسجل 0 للعثور على رقم استدعاء النظام، ثم تبحث في جدول لإيجاد الدالة الصحيحة للانتقال إليها لمعالجة استدعاء النظام، وتتلقى هذه الدالة وسائطها من المسجل 3 إلى المسجل 7.

يعود التحكم إلى التعليمة التالية بعد `sc` وهي في هذه الحالة تعليمات سور الذاكرة Memory Fence بمجرد تشغيل معالج استدعاء النظام واكتماله، كما تتأكد تعليمات سور الذاكرة من أن كل شيء ملتزم بالذاكرة، حيث تضمن هذه التعليمة أنّ كل ما نعتقد أنه مكتوب في الذاكرة قد حدث فعليًا دون المرور عبر خط أنابيب Pipeline في مكان ما.

انتهينا تقريبًا، ولكن الشيء الوحيد المتبقي هو إعادة القيمة من استدعاء النظام، إذ نرى ضبط القيمة `__sc_ret` من المسجل `r3` وضبط القيمة `__sc_err` من المسجل `r0`، فالقيمة الأولى هي القيمة المُعادة، والأخرى هي قيمة الخطأ، إذ يمكن أن تفشل استدعاءات النظام مثل أيّ دالة أخرى، لكن تكمن المشكلة في أنّ استدعاء النظام يمكن أن يعيد أيّ قيمة ممكنة، إذ لا يمكننا أن نقول أن القيمة السالبة تشير إلى الفشل، لأنها

يمكن أن تكون مقبولةً لبعض استدعاءات النظام، لذا تضمن دالة استدعاء النظـام أنّ نتيجتهـا في المسـجل r3 وأنّ أيّ رمز خطأ موجود في المسجل r0 قبل إعادة النتيجة.

يجب التحقق من رمز الخطأ للتأكد من ضبط البِتّ العلوي الذي من شأنه الإشارة إلى عدد سالب، فـإذا كـان الأمـر كـذلك، فسنضـبط قيمـة المتغـير errno العـام على هـذه القيمـة وهي المتغـير القياسـي للحصـول على معلومات الخطأ عند فشل الاستدعاء، كما سنضبط القيمة المُعادة على -1، وسـنعيد النتيجـة مباشـرةً في حالـة تلقّي نتيجة صالحة، لذا يجب على دالة الاسـتدعاء التحقـق من أنّ القيمـة المعـادة ليسـت -1، فـإذا كـان الأمـر كذلك، فيمكنها التحقق من المتغير errno للعثور على سبب فشل الاستدعاء، وهذا هـو اسـتدعاء نظـام كامـل على نظام PowerPC.

ب. استدعاءات نظام x86

إليك الواجهة المطبَّقة لمعالج x86:

```
/* توجد أرقام الأخطاء المرئية للمستخدِم ضمن المجال من -1 إلى -124
*  راجع <asm-i386/errno.h>
*/

#define __syscall_return(type, res)
do {
  if ((unsigned long)(res) >= (unsigned long)(-125)) {
    errno = -(res);
    res = -1;
  }
  return (type) (res);
} while (0)

/* يجب أن تكون _foo، بينما __NR_bar يمكن أن تكون _NR_bar */
#define _syscall0(type,name)
type name(void)
{
long __res;
__asm__ volatile ("int $0x80"
  : "=a" (__res)
  : "0" (__NR_##name));
__syscall_return(type,__res);
}
```

```
#define _syscall1(type,name,type1,arg1)
type name(type1 arg1)
{
long __res;
__asm__ volatile ("int $0x80"
  : "=a" (__res)
  : "0" (__NR_##name),"b" ((long)(arg1)));
__syscall_return(type,__res);
}

#define _syscall2(type,name,type1,arg1,type2,arg2)
type name(type1 arg1,type2 arg2)
{
long __res;
__asm__ volatile ("int $0x80"
  : "=a" (__res)
  : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)));
__syscall_return(type,__res);
}

#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)
type name(type1 arg1,type2 arg2,type3 arg3)
{
long __res;
__asm__ volatile ("int $0x80"
  : "=a" (__res)
  : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),
                  "d" ((long)(arg3)));
__syscall_return(type,__res);
}

#define
_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4)
{
long __res;
```

```
__asm__ volatile ("int $0x80"
  : "=a" (__res)
  : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),
          "d" ((long)(arg3)),"S" ((long)(arg4)));
__syscall_return(type,__res);
}

#define
_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,
          type5,arg5)
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5)
{
long __res;
__asm__ volatile ("int $0x80"
  : "=a" (__res)
  : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),
          "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)));
__syscall_return(type,__res);
}

#define
_syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,
          type5,arg5,type6,arg6)
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5
arg5,type6 arg6)
{
long __res;
__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int
$0x80 ; pop %%ebp"
  : "=a" (__res)
  : "i" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),
          "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)),
          "0" ((long)(arg6)));
__syscall_return(type,__res);
}
```

تختلف معمارية x86 كثيرًا عن PowerPC التي تحدّثنا عنها سابقًا، إذ يُصنَّف x86 على أنه معالِج من النـوع CISC على عكس PowerPC الذي يُعَدذ من النوع RISC، ولديه مسجلات أقل بكثير.

اطّلع على أبسط ماكرو `_syscall0` الذي يستدعي تعليمة من النوع `int` والقيمة `0x80`، إذ تعمل هذه التعليمة على جعل وحدة المعالجة المركزية ترفع المقاطعة 0x80 التي ستنتقل إلى الشيفرة البرمجية التي تعالج استدعاءات النظام في النواة.

لنفحص الآن كيفية تمرير الوسائط باستخدام وحدات الماكرو الأطول، ولاحظ كيف أنّ نظام PowerPC قد طبّق تتال وانسياب cascade من وحدات الماكرو من خلال إضافة وسيط واحد في كل مرة، كما يحتوي هذا التطبيق على شيفرة برمجية منسوخة ولكن اتباعه أسهل قليلًا.

تستند أسماء المسجّلات في معمارية x86 إلى الأحرف عوضًا عن أسماء المسجّلات الرقمية في PowerPC، إذ يمكننا رؤية من الماكرو عديم الوسائط أنّ المسجّل `A` يُحمَّل فقط، وبالتالي يمكننا القول أنّ رقم استدعاء النظام متوقَّع وجوده في المسجّل `EAX`، كما يمكنك رؤية أسماء المسجلات المختصرة في وسائط استدعاء `__asm__` عندما نبدأ بتحميل المسجلات في وحدات الماكرو الأخرى.

لاحظ الماكرو `__syscall6__` الذي يأخذ 6 وسائط، إذ تعمل التعليمتان `push` و `pop` مع المكدس في x86، بحيث تدفع إحداهما قيمةً إلى أعلى المكدس في الذاكرة وتسحب الأخرى القيمة من المكدس في الذاكرة، كما يجب تخزين قيمة المسجل `ebp` في الذاكرة ووضع الوسيط في التعليمة `mov` وإجراء استدعاء للنظام، ثم إعادة القيمة الأصلية إلى المسجل `ebp` في حالة وجود ستة مسجلات، كما يمكنك هنا رؤية عيوب عدم وجود مسجلات كافية، إذ يُعَدّ التخزين في الذاكرة باهظ الثمن، لذا كلما تمكنت من تجنّبها، كان ذلك أفضل.

لاحظ عدم وجود تعليمات سور الذاكرة التي رأيناها سابقًا مع PowerPC لأنّ معمارية x86 تضمن أن يكون تأثير جميع التعليمات مرئيًا عند اكتمالها، مما يسهّل البرمجة على المصرّف والمبرمج ولكنه يقلل من المرونة.

يوجد أيضًا اختلاف في القيمة المُعادة، فقد كان لدينا مسجّلَين مع قيم مُعادة من النواة في معمارية PowerPC، إحداهما هي القيمة والأخرى هي رمز الخطأ، في حين لدينا قيمة مُعادة واحدة في معمارية x86 تُمرَّر إلى الماكرو `__syscall_return` الذي يغيّر نوع القيمة المُعادة إلى النوع `unsigned long` ويوازنها مع مجال من القيم السالبة تعتمد على المعمارية والنواة، حيث تمثّل هذه القيم رموز الخطأ.

لاحظ أنّ قيمة رمز الخطأ `errno` موجبة، مما يؤدي إلى إلغاء النتيجة السالبة من النواة، لكن يعني هذا أنّ استدعاءات النظام لا يمكنها إعادة قيم سالبة صغيرة، إذ لا يمكن تمييزها عن رموز الخطأ، كما تضيف بعض استدعاءات النظام التي لديها هذا المتطلب مثل الاستدعاء `getpriority()` إزاحةً إلى القيمة المُعادة لإجبارها بأن تكون دائمًا موجبة، فالأمر متروك لمجال المستخدِم لإدراك ذلك وطرح هذه القيمة الثابتة للحصول على القيمة الحقيقية.

4.4 الصلاحيات في نظام التشغيل

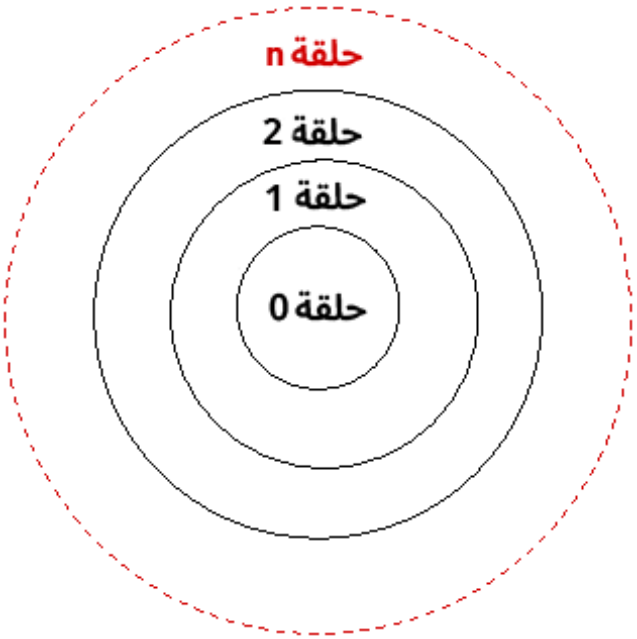
يُعَدّ تطبيق الأمان أحد مهام نظام التشغيل الرئيسية بهدف عدم السماح لتطبيق أو مستخدِم بالتضارب مع أيّ تطبيق آخر يعمل في النظام، وهذا يعني أن التطبيقات يجب ألّا تكون قادرةً على الكتابة في ذاكرة أو ملفات التطبيقات الأخرى، ويجب أن تصل فقط إلى الموارد وفق سياسة النظام.

لكن يكون لأحد التطبيقات عند تشغيلها استخدام حصري للمعالج، إذ سنرى كيف يعمل ذلك عندما نتعرّف على العمليات في المقال التالي، ويمكن التأكد من وصول التطبيق إلى الذاكرة التي يمتلكها فقط باستخدام نظام الذاكرة الوهمية Virtual Memory، إذ يُعَدّ العتاد مسؤولًا عن تطبيق هذه القواعد.

تُعَدّ واجهة استدعاء النظام بوابة التطبيق للوصول إلى موارد النظام، إذ يمكن للنواة فرض قواعد حول نوع الوصول الذي يمكن توفيره من خلال إجبار التطبيق على طلب الموارد من خلال استخدام استدعاء نظام إلى النواة، فإذا أجرى أحد التطبيقات استدعاء النظام `open()` لفتح ملف على القرص الصلب مثلًا، فسيتحقق من أذونات المستخدِم المقابلة لأذونات الملف ثم سيسمح بوصوله أو يرفضه.

4.4.1 مستويات الصلاحيات

تُعَدّ حماية العتاد مجموعةً من الحلقات متحدة المركز حول مجموعة أساسية من العمليات.



شكل 17: مستويات الصلاحيات في معمارية x86

توجد التعليمات ذات الحماية الأكبر في الحلقة الداخلية، وهي التعليمات التي يجب السماح للنواة فقط باستدعائها مثل التعليمة HLT المُستخدَمة لإيقاف المعالج، إذ يجب ألّا يُسمَح بأن يشغّلها تطبيق مستخدِم، لأن ذلك سيوقف الحاسوب بأكمله عن العمل، لكن يجب أن تكون النواة قادرةً على استدعاء هذه التعليمة عند إيقاف تشغيل الحاسوب بطريقة نظامية، إذ يرفع العتاد استثناءً عندما يستدعي تطبيق ما هذه التعليمة،

ويتضمّن هذا الاستثناء القفز إلى معالج محدد في نظام التشغيل مشابه لمعالج استدعاء النظام، كما يُحتمَل أن ينهي نظام التشغيل بعد ذلك البرنامج ويعطي المستخدِم بعض الأخطاء حول كيفية تعطل التطبيق.

يمكن لكل حلقة داخلية الوصول إلى أيّ تعليمات تحميها حلقة خارجية، ولكن لا يمكنها الوصول إلى تعليمة تحميها حلقة داخلية، كما لا تحتوي جميع المعماريات على مستويات متعددة من الحلقات كما في الشكل السابق، ولكن سيوفر معظمها على الأقل مستوى النواة Kernel ومستوى المستخدِم User.

ا. نموذج الحماية 386

يحتوي نموذج الحماية 386 على أربع حلقات بالرغم من أن معظم أنظمة التشغيل مثل لينكس ووينـدوز تستخدِم حلقتَين فقط للحفاظ على التوافق مع المعماريات الأخرى التي تسمح الآن بأكبر عدد من مستويات الحماية المنفصلة، كما يحتفظ النموذج 386 بالصلاحيات من خلال أن يكون لكل جزء من شيفرة التطبيق البرمجية المُشغَّلة في النظام واصف صغير يسمى واصف الشيفرة البرمجية Code Descriptor الذي يصِف مستوى صلاحياتها.

تقفز شيفرة التطبيق عند تشغليها سريعًا إلى الشيفرة البرمجية الموجودة خارج المنطقة التي يصِفها واصفُ شيفرة التطبيق مع التحقق من مستوى صلاحيات الهدف، فإذا كانت الصلاحيات أعلى من صلاحيات الشيفرة المُشغَّلة حاليًا، فلن يسمح العتاد بهذه القفزة وسيتعطل التطبيق.

ب. رفع مستوى الصلاحيات

يمكن أن ترفع التطبيقات مستوى صلاحياتها فقط من خلال استدعاءات محددة تسمح بذلك مثل التعليمات الخاصة بتنفيذ استدعاء النظام، إذ يشار إليها عادةً باسم بوابة الاستدعاءات Call Gate لأنها تعمل مثل بوابة حقيقية تسمح بمدخل صغير عبر جدار غير قابل للاختراق.

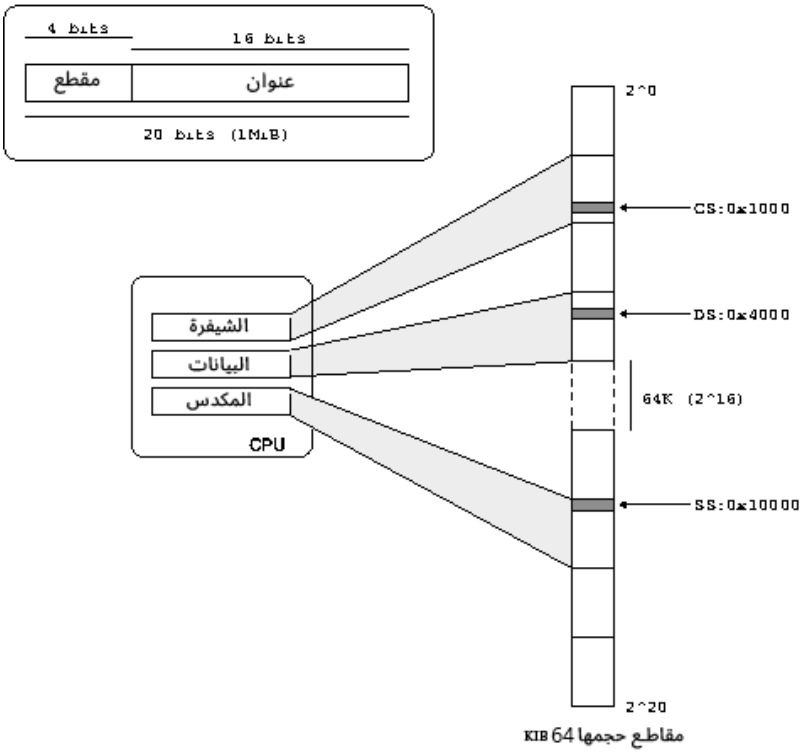
رأينا كيف يوقِف العتاد التطبيق الذي يكون قيد التشغيل ويسلّم التحكم إلى النواة عند استدعاء هذه التعليمات، ويجب أن تعمل النواة بوصفها حارسًا للبوابة للتأكد من عدم دخول أيّ شيء غير مرغوب به من البوابة، إذ يجب التحقق من وسائط استدعاء النظام بعناية للتأكد من أنه لن ينخدع بفعل شيء لا ينبغي أن يفعله، وبالتالي حدوث خطأ أمني.

تعمل النواة في الحلقة الداخلية، لذا فهي تمتلك الأذونات اللازمة لإجراء أيّ عملية تريدها، ثم ستعيد التحكم في النهاية إلى التطبيق الذي سيعمل مرةً أخرى بمستوى صلاحيات أقل.

ج. استدعاءات النظام السريعة

تتمثل إحدى مشاكل المصائد كما هو موضح سابقًا في أنها باهظة الثمن بالنسبة للمعالج لتطبيقها، فهناك الكثير من الحالات التي يجب حفظها قبل تبديل السياق، وقد أدركت المعالجات الحديثة هذا الحِمل وتسعى جاهدة لتقليله.

يتطلب فهم آلية بوابة الاستدعاءات الموضحة سابقًا التدقيق في مخطـط التقطيـع المبتكـر والمعقـد الـذي يستخدمه المعالج، وقد كان السبب الأصلي لتطبيق التقطيع هو القدرة على استخدام أكثر من 16 بِتًا متوفرًا في المسجل لعنوان ما كما هو موضح في الشكل التالي:

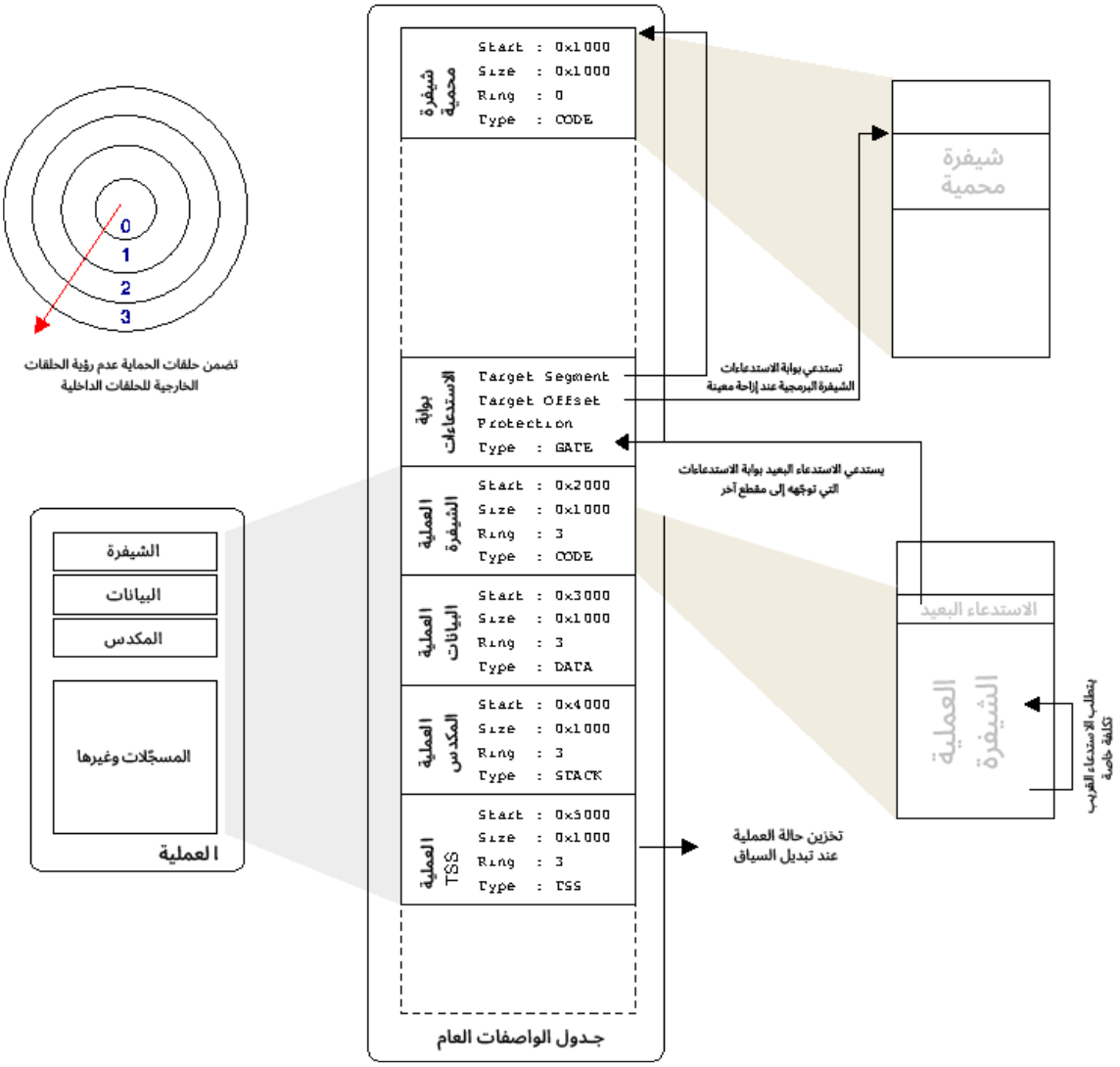


شكل 18: تقطيع العنونة Segmentation Addressing في معمارية x86

يوضـح الشـكل السـابق تقطيـع العنونـة Segmentation Addressing في معماريـة x86 حيث يـؤدي التقطيع إلى توسيع مساحة عناوين المعـالج من خلال تقسـيمه إلى أجـزاء. يحتفـظ المعـالج بمسـجلات مقـاطع خاصة، ويمكن تحديد العناوين من خلال مسجّل المقطع والإزاحة. تُضاف قيمة مسجل المقطع إلى جـزء الإزاحـة للعثور على العنوان النهائي.

بقي مخطط التقطيع كما هو ولكن بتنسيق مختلـف عنـدما انتقلت معماريـة x86 إلى مسـجّلات بحجم 32 بِتًا، إذ يُسمَح للمقاطع بأن تكون بأيّ حجم بدلًا من اسـتخدام أحجـام ثابتـة، ويجب أن يتعقّب المعـالج كـل هـذه المقاطع المختلفة وأحجامها، وهو ما يفعله باستخدام الواصفات Descriptors.

تكون واصفات المقاطع المتاحة للجميع محفوظةً في جدول الواصفات العام Global Descriptor Table أو GDT اختصارًا، كما تحتوي كل عملية على عدد من المسجلات التي توشّر إلى مدخلات في جدول GDT، وهذه المدخلات هي المقاطع التي يمكن للعملية الوصول إليها، كما توجد جداول واصفات محليـة، وتتفاعـل جميعهـا مع مقاطع حالة المهمات، لكنها ليست مهمةً حاليًا.



شكل 19: مقاطع x86

لاحظ كيف يمر الاستدعاء البعيد عبر بوابة الاستدعاءات التي توجّهه إلى مقطع من الشيفرة يعمل على مستوى الحلقة الأدنى. الطريقة الوحيدة لتعديل محدّد مقطع الشيفرة -المستخدَم ضمنيًا لجميع عناوين الشيفرة- هي استخدام آلية الاستدعاء، حيث تضمن آلية بوابة الاستدعاءات اختيار واصف مقطع جديد، مما يؤدي إلى تغيير مستويات الحماية التي يجب عليك الانتقال إليها عبر نقطة دخول معروفة.

يضبط نظام التشغيل مسجلات المقطع بوصفها جزءًا من حالة العملية، لذا يعرِف عتاد المعالِج مقاطع الذاكرة التي يمكن للعملية المُشغَّلة الوصول إليها، كما يمكنه فرض الحماية لضمان عدم وصول العملية لأيّ شيء لا يُفترَض أن تصل إليه، فإذا خرجت العملية خارج حدودها المفروضة، فستتلقّى خطأ تقطيع Segmentation Fault يعرفه معظم المبرمجين.

إذا احتاج تشغيل الشيفرة البرمجية إلى إجراء استدعاءات إلى شيفرة موجودة في مقطع آخر، فستطبّق معمارية x86 ذلك كما في الحلقات Rings، إذ تكون الحلقة 0 هي الحلقة ذات الإذن الأعلى والحلقة 3 هي الأدنى، ويمكن للحلقات الداخلية الوصول إلى الحلقات الخارجية ولكن ليس العكس.

إذا أرادت شيفرة الحلقة 3 القفز إلى شيفرة الحلقة 0، فستعدّل محدّد مقطع الشيفرة الخاص بها ليؤشّـر إلى مقطع مختلف، لذلك يجب أن تستخدِم تعليمة استدعاءات بعيدة خاصة بحيث يتأكد العتاد من مرورها عبر بوابة الاستدعاءات، ولا توجد طريقة أخرى للعملية المُشغَّلة لاختيار واصف مقطع شيفرة جديد، وسـيبدأ المعـالج بعـد ذلك في تنفيذ الشيفرة البرمجية عند الإزاحة المعروفة في مقطع الحلقة 0، وهذا هو سبب الحفاظ على السـلامة مثل عدم قراءة الشيفرة البرمجية العشوائية والضارة وتنفيذها، كما سيبحث المهـاجمون دائمًـا عن طـرق لجعـل شيفرتك البرمجية تفعل شيئًا لا تريده.

يسمح ذلك بتسلسل هرمي كامل للمقاطع والأذونات، ولاحظ أنّ استدعاء المقطع العرضي يشـبه اسـتدعاء النظام، فإذا سبق لـك أن شـاهدت لغـة تجميـع لينكس x86، فالطريقـة القياسـية لإجـراء اسـتدعاء النظـام هي باستخدام `int 0x80` التي ترفع المقاطعة `0x80`، إذ توقِف المقاطعـة المعـالج وتنتقـل إلى بوابـة المقاطعـات التي تعمل بعد ذلك بطريقة بوابة الاسـتدعاءات نفسـها بحيث تغيّـر مسـتوى الصـلاحيات وتعيـدك إلى منطقـة أخرى من الشيفرة البرمجية.

مشكلة هذا المخطط أنه بطيء، إذ يتطلب الأمر الكثير من الجهـد لتطـبيق كـل هـذا الفحص، ويجب حفـظ العديد من المسجلات للوصول إلى الشيفرة الجديدة، كما يجب استعادة كل شيء مرةً أخرى في طريق العودة.

لا يُستخدَم نظام الحلقات ذو المستويات الأربعـة في تقطيـع نظـام x86 الحـديث بفضـل الـذاكرة الوهميـة، والشيء الوحيد الذي يحدث فعليًا مع تبديل التقطيع هو استدعاءات النظام التي تتحوّل من الوضع 3 -أي مجـال المستخدِم- إلى الوضع 0 وتقفز إلى شيفرة معالج استدعاء النظام في النواة.

يوفّر المعالِج تعليمات استدعاء نظام فائقة السرعة تسمى `sysenter` (و `sysexit` للعودة)، إذ تسرّع هذه التعليمات العملية برمتها عبر الاستدعاء `int 0x80` من خلال إزالة الطبيعة العامة للاستدعاء البعيد، أي إمكانية الانتقال إلى أيّ مقطع في أيّ مستوى حلقة، وتقييد الاسـتدعاء للانتقـال فقـط إلى شـيفرة الحلقـة 0 في مقطـع معيّن مع الإزاحة كما هي مخزّنة في المسجلات.

بما أننا استبدلنا هـذه الطبيعـة العامـة بـالكثير من المعلومـات المعروفـة مسـبقًا، فيمكن تسـريع العمليـة، وبالتالي سنحصل على استدعاء النظام السريع الذي ذكرناه سـابقًا، والشـيء الآخـر الـذي يجب ملاحظتـه هـو أنّ الحالة لا تُحفَظ عندما ينتقل التحكم إلى النـواة، إذ يجب أن تكـون النـواة حريصـةً على عـدم تـدمير الحالـة، ولكن يعني هذا أنها حرة في حفظ الحالة الصغيرة كما هو مطلوب لتنفيذ المهمة، لذلك يمكن أن تكون أكثر فاعليـةً، إذ تتعلق هذه الفكرة بمعمارية RISC، وتوضّح كيفية تلاشي الخط بين معالجات RISC و CISC.

هناك طرق أخرى للتواصل مع النواة مثل ioctl وأنظمة الملفات مثل proc و sysfs و debugfs وغير ذلك.

هل تطمح لبيع منتجاتك الرقمية عبر الإنترنت؟

استثمر مهاراتك التقنية وأطلق منتجًا رقميًا
يحقق لك دخلًا عبر بيعه على متجر بيكاليكا

أطلق منتجك الآن

5. العمليات في نظام تشغيل الحاسوب

5.1 ما هي العملية؟

جميعنا على دراية بنظام التشغيل الحديث الذي يدير عدة مهام في وقت واحد أو ما يسمى بتعدد المهام Multitasking، حيث تُعَدّ العملية حزمةً من العناصر التي تحتفظ بها النواة لتعقّب جميع المهام قيد التشغيل.

5.2 عناصر العملية

شكل 20: عناصر العملية

5.2.1 معرف العملية

يضبط نظام التشغيل معرّف العملية Process ID -أو PID اختصارًا- ويكون فريدًا لكل عملية مُشغَّلة.

5.2.2 الذاكرة

سنتعلم كيف تحصل عملية ما على ذاكرتها لاحقًا، وتُعَدّ الذاكرة أحد الأجزاء الأساسية لكيفية عمل نظام التشغيل، ولكن سنكتفي حاليًا بمعرفة أنّ كل عملية لها قسمها الخاص من الذاكرة.

تُخزَّن شيفرة البرنامج في هذه الذاكرة مع المتغيرات وأيّ عمليات تخزين أخرى مخصَّصة، ويمكن مشاركة أجزاء من الذاكرة بين العمليات، إذ تسمَّى بالذاكرة المشتركة Shared Memory، كما يمكن أن تراها بالاسم System Five Shared Memory -أو SysV SHM اختصارًا- بعد التطبيق الأصلي في نظام تشغيل أقدم.

مفهوم مهم آخر يمكن أن تستخدِمه العملية هو مفهوم ربط ملف موجود في القرص الصلب مع الذاكرة أو ما يُسمى mmaping، إذ يبدو الملف كما لو كان أيّ نوع آخر من الذاكرة RAM بدلًا من الاضطرار إلى فتح الملف واستخدام أوامر مثل `read()` و `write()`، كما تمتلك مناطق `mmaped` أذونات يجب تعقّبها مثل القراءة والكتابة والتنفيذ، فمهمّة نظام التشغيل هي الحفاظ على الأمن والاستقرار، لذلك يجب التحقق مما إذا كانت العملية تحاول الكتابة في منطقة للقراءة فقط وإعادة خطأ بذلك.

ا. الشيفرة والبيانات

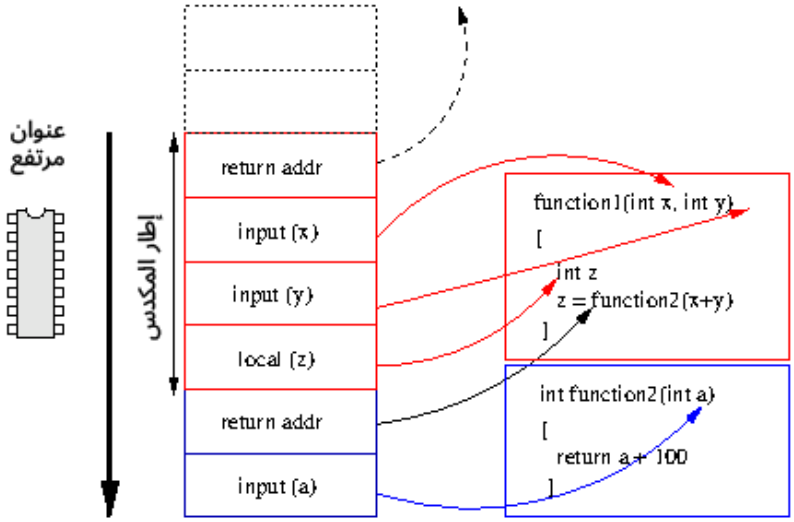
يمكن تقسيم العملية بصورة أكبر إلى قسمين هما الشيفرة Code والبيانات Data، إذ يجب الاحتفاظ بشيفرة البرنامج وبياناته بصورة منفصلة لأنها تتطلب أذونات مختلفة من نظام التشغيل، ويسهّل هذا الفصل بينهما مشاركة الشيفرة كما سنرى لاحقًا، كما يجب أن يعطي نظام التشغيل إذنًا لشيفرة البرنامج للتمكّن من قراءتها وتنفيذها دون الكتابة فيها، في حين تتطلب البيانات (المتغيرات) أذونات القراءة والكتابة ولا ينبغي أن تكون قابلةً للتنفيذ، ولكن لا تدعم جميع المعماريات ذلك، مما أدى إلى مجموعة واسعة من مشاكل الأمان في العديد منها.

ب. المكدس Stack

يُعَدّ المكدس جزءًا مهمًا آخر من العملية وهو منطقة من الذاكرة وجزء من قسم البيانات في العملية، ويشارك في تنفيذ أيّ برنامج، كما يُعَدّ بنية بيانات عامة تعمل مثل مجموعة الأطباق، إذ يمكنك دفع `push` عنصر أو وضع طبق أعلى كومة من الأطباق بحيث يصبح العنصر العلوي، أو يمكنك سحب `pop` عنصر أو سحب طبق وظهور الطبق السابق.

المكدسات أساسية لاستدعاءات الدوال، إذ تحصل على إطار مكدس stack frame جديد في كل مرة تُستدعَى فيها الدالة، وهي منطقة من الذاكرة تحتوي على الأقل على العنوان الذي يجب الرجوع إليه عند الانتهاء ووسائط دخل الدالة وفضاء المتغيرات المحلية.

تنمو المكدسات عادةً إلى الأسفل، إذ يبدأ المكـدس عنـد عنـوان مرتفـع في الـذاكرة وينخفض تـدريجيًا، في حين تحتوي بعض المعماريات مثل PA-RISC من HP على مكدسات تنمو للأعلى، وتوجد في بعض المعماريات الأخرى مثل IA64 مناطق تخزين أخرى (مخزن داعم للمسجّل) تنمو من الأسفل باتجاه المكدس.



شكل 21: المكدس

يعطي وجود مكدس العديد من الميزات للدوال ومنها ما يلي:

أولًا، لكل دالة نسختها الخاصة من وسائط الدخل، إذ يُخصَّص إطار مكدس جديد لكل دالة مع وسـائطها في منطقة جديدة من الذاكرة، وبالتالي لا يمكن أن ترى الدوال الأخرى المتغير المُعـرَّف في دالـة مـا، في حين تُخـزَّن المتغيرات العامة التي يمكن أن تراها أيّ دالة في منطقة منفصلة من ذاكرة البيانات، مما يسـهّل الاسـتدعاءات العودية Recursive Calls، وهذا يعني أنّ الدالة حرة في استدعاء نفسها مرةً أخرى، إذ سيُنشَأ إطار مكدس جديد لجميع متغيراتها المحلية.

ثانيًا، يحتوي كل إطار على عنوان للعودة إليه، وتسمح لغة C فقط بإعادة قيمة واحدة من الدالـة، لـذلك تُعـاد هذه القيمة إلى دالة الاستدعاء في مسجّل محدد بدلًا من المكدس.

ثالثًا، يحتوي كـل إطـار على مرجـع للإطـار الـذي يسـبقه، لـذا يمكن لمنقّح الأخطـاء العـودة للخلـف متتبّعًـا المؤشرات ليصل إلى أعلى المكدس، ويمكن أن ينتج متعقّب مكدسات Stack Trace الـذي يُظهِـر لـك جميـع الدوال التي جرى استدعاؤها حتى الوصول إلى هذه الدالة، ويُعَدّ ذلك مفيـدًا لتنقيح الأخطـاء، كمـا يمكنـك رؤيـة كيف تتناسب الطريقة التي تعمل بها الدوال مع طبيعـة المكـدس، إذ يمكن لأيّ دالـة اسـتدعاءُ أيّ دالـة أخـرى، وبالتالي تصبح الدالة الأعلى بحيث تُوضَع في أعلى المكدس، وستعيد هذه الدالة في النهايـة النتيجـة إلى الدالـة التي استدعتها، أي تخرج من المكدس.

رابعًا، تجعل المكدسات استدعاء الدوال أبطأ، لأنه يجب نقل القيم خارج المسجلات إلى الذاكرة، في حين تسمح بعض المعماريات الحاسوبية بتمرير الوسائط في المسجلات مباشرةً، ولكن يجب تدوير المسجلات للحفاظ على دلالات حصول كل دالة على نسخة فريدة من كل وسيط.

خامسًا، لا بد أنك سمعت بمصطلح طفحان المكدس Stack Overflow الذي يُعَدّ طريقةً شائعةً لاختراق النظام من خلال تمرير قيم وهمية، فإذا كنت مبرمجًا تقبل بالإدخال العشوائي في متغير المكدس مثل القراءة من لوحة المفاتيح أو عبر الشبكة، فيجب عليك تحديد حجم هذه البيانات صراحةً، إذ يؤدي السماح بأيّ كمية من البيانات دون تحديد إلى الكتابة فوق الذاكرة، مما يؤدي إلى حدوث عطل، ولكن أدرك بعض الأشخاص أنهم إذا كتبوا في ذاكرة كافية فقط لوضع قيمة محددة في جزء العنوان المُعاد من إطار المكدس، فيمكنهم إعادتها في البيانات التي أرسلوها للتو عند اكتمال الدالة بدلًا من الإعادة إلى المكان الصحيح الذي استدعاها، فإذا احتوت هذه البيانات على شيفرة ثنائية قابلة للتنفيذ وتخترق النظام مثل تشغيل طرفية للمستخدِم مع صلاحيات الجذر، فهذا يعني تعرّض حاسوبك للاختراق. يحدث ذلك بسبب نمو المكدس للأسفل، ولكن تُقرَأ البيانات للأعلى أي من العنوان الأدنى إلى العناوين الأعلى.

هناك عدة طرق للتغلب على هذه المشكلة، إذ يجب عليك التأكد -بصفتك مبرمجًا- من أنك تتحقق دائمًا من كمية البيانات التي تتلقاها في متغير، ويمكن أن يساعد نظام التشغيل في تجنّب ذلك نيابةً عن المبرمج من خلال التأكد من تمييز المكدس على أنه غير قابل للتنفيذ، وبالتالي لن يشغّل المعالج أيّ شيفرة برمجية، حتى إذا حاول مستخدِم سيئ تمرير شيفرة برمجية إلى برنامجك، وتدعم المعماريات وأنظمة التشغيل الحديثة هذه الوظيفة.

سادسًا، يدير المصرّف Compiler المكدسات، فهو المسؤول عن إنشاء شيفرة البرنامج، ويبدو المكدس بالنسبة لنظام التشغيل مثل أيّ منطقة أخرى من الذاكرة الخاصة بالعملية.

يعرِّف العتاد المسجِّل بوصفه مؤشر المكدس Stack Pointer بهدف تعقّب نمو المكدس الحالي، إذ يستخدِم المصرّف -أو المبرمج عند الكتابة باستخدام لغة التجميع- هذا المسجِّل لتعقّب الجزء العلوي الحالي من المكدس، وإليك مثال عن مؤشر المكدس:

```
$ cat sp.c
void function(void)
{
  int i = 100;
  int j = 200;
  int k = 300;
}

$ gcc -fomit-frame-pointer -S sp.c
```

```
$ cat sp.s
  .file   "sp.c"
  .text
.globl function
  .type   function, @function
function:
  subl    $16, %esp
  movl    $100, 4(%esp)
  movl    $200, 8(%esp)
  movl    $300, 12(%esp)
  addl    $16, %esp
  ret
  .size   function, .-function
  .ident  "GCC: (GNU) 4.0.2 20050806 (prerelease) (Debian 4.0.1-4)"
  .section        .note.GNU-stack,"",@progbits
```

عرضنا في المثال السابق دالةً بسيطةً تخصّص ثلاثة متغيرات على المكدس، إذ توضّح شيفرة فك التجميع السابقة استخدام مؤشر المكدس في معمارية x86.

أولًا، نخصّص مساحةً على المكدس لمتغيراتنا المحلية، كما تنمو المكدسات للأسفل، لذلك يجب أن نطرح من القيمة الموجودة في مؤشر المكدس.

تُعَدّ القيمة 16 قيمةً كبيرةً بما يكفي للاحتفاظ بمتغيراتنا المحلية، ولكن يمكن ألّا تكون بالحجم المطلوب للحفاظ على محاذاة المكدس في الذاكرة ضمن الحدود التي يتطلبها المصرّف، إذ نحتاج مثلًا 12 بايت فقط وليس 16 مع 3 قيم من النوع `int` المكوَّن من 4 بايتات، وننقل بعد ذلك القيم إلى ذاكرة المكدس التي تستخدِمها الدالة الحقيقية.

أخيرًا،يتوجب علينا أن نسحب القيم من المكدس قبل العودة إلى الدالة الأصلية من خلال تحريك مؤشر المكدس إلى حيث كان قبل أن نبدأ.

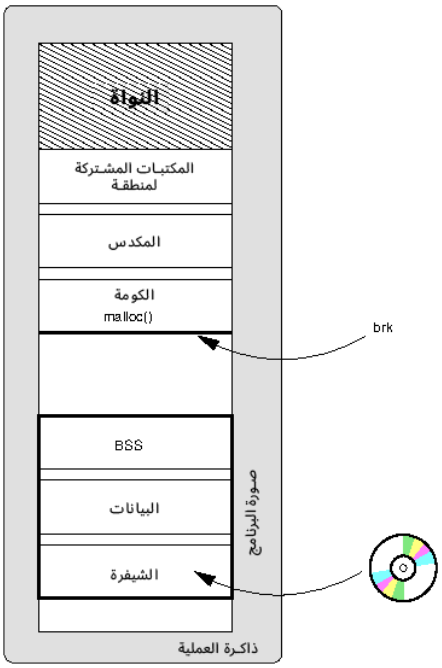
> لاحظ أننا استخدمنا رايةً خاصةً في مصرّف gcc، وهذه الراية هي `-fomit-frame-pointer` التي تحدِّد أنه لا ينبغي استخدام مسجل إضافي للاحتفاظ بمؤشر إلى بداية إطار المكدس، إذ يساعد وجود هذا المؤشر منقّحات الأخطاء للانتقال للأعلى عبر إطارات المكدس، ولكنه يجعل المسجِّل متاحًا بصورة أقل للتطبيقات الأخرى.

ج. الكومة Heap

الكومة Heap هي مساحة من الذاكرة تديرها العملية لتخصيص الذاكرة السريع، وتُستخدَم مع المتغيرات التي لا تكون متطلبات ذاكرتها معروفةً في وقت التصريف، ويُعرَف الجزء السفلي من الكومة بالاسم `brk` وهو استدعاء النظام الذي يعدّل الكومة، كما يمكن للعملية أن تطلب من النواة تخصيص مزيد من الذاكرة لاستخدامها باستخدام الاستدعاء `brk` لتوسيع المنطقة إلى الأسفل.

يدير استدعاء مكتبة `malloc` الكومةَ، وهذا يجعل إدارة الكومة أمرًا سهلًا للمبرمج من خلال السماح له بتخصيص وتحرير كومة ذاكرة باستخدام الاستدعاء `free`، كما يمكن أن يستخدِم الاستدعاء `malloc` أنظمةً مثل مخصّص الأصدقاء Buddy Allocator لإدارة كومة ذاكرة المستخدِم، ويمكن أن يكون الاستدعاء `malloc` ذكيًا فيما يتعلق بعملية التخصيص ويمكنه استخدام عمليات ربط مجهولة Anonymous mmaps لذاكرة عملية إضافية، وهو المكان الذي يربط منطقة من ذاكرة RAM الخاصة بالنظام مباشرةً بدلًا من ربط ملف مع ذاكرة العملية، إذ يمكن أن يكون ذلك أكثر كفاءةً، كما أنه ليس مألوفًا أن يكون لأيّ برنامج حديث سبب لاستدعاء `brk` مباشرة نظرًا لتعقيد إدارة الذاكرة بصورة صحيحة.

د. تخطيط الذاكرة



شكل 22: تخطيط ذاكرة العملية

تمتلك العملية مناطق أصغر من الذاكرة المخصَّصة لها ويكون لكل منها غرض محدد.

ذكرنا في الشكل السابق كيفية وضع العملية في الذاكرة باستخدام النواة، إذ تحتفظ النواة لنفسها ببعض الذاكرة في الجزء العلوي من العملية، ويمكن مشاركة هذه الذاكرة فعليًا بين جميع العمليات باستخدام الذاكرة الوهمية، كما يوجد أسفل ذلك مساحة للملفات والمكتبات المربوطة `mmaped`، ثم يوجد المكدس وتحته الكومة،

في حين توجد في الجزء السفلي صورة البرنامج كما جرى تحميلها من الملف القابل للتنفيذ على القرص الصلب، وسنلقي نظرةً على عملية تحميل هذه البيانات في مقالات لاحقة.

5.2.3 واصفات الملف File Descriptors

تعرّفنا سابقًا على الملفات الافتراضية المعطاة لكل عملية وهي `stdin` و `stdout` و `stderr`، إذ يكون لهذه الملفات دائمًا رقم واصف الملف نفسه (0 و 1 و 2 على التوالي)، وبالتالي تحتفظ النواة بواصفات الملفات بصورة فردية لكل عملية.

تمتلك واصفات الملفات أذونات أيضًا، إذ يمكن أن تتمكّن من القراءة من ملف ولكن لا يمكنك الكتابة فيه مثلًا، إذ يحتفظ نظام التشغيل عند فتح الملف بسجل أذونات العمليات لهذا الملف في واصف الملف ولا يسمح للعملية بفعل أيّ شيء لا ينبغي فعله.

5.2.4 المسجلات Registers

يطبّق المعالج عمليات بسيطة على القيم الموجودة في المسجّلات، وتُقرَأ هذه القيم وتُكتَب في الذاكرة، فلكل عملية منطقة مخصَّصة لها في الذاكرة التي تتعقّبها النواة، لذا يجب تعقّب المسجلات، فإذا حان الوقت لتتخلّى العملية المُشغَّلة عن المعالج لتشغيل عملية أخرى، فيجب حفظ حالتها الحالية، كما يجب أن نكون قادرِين على استعادة هذه الحالة عند منح العملية مزيدًا من الوقت للتشغيل على وحدة المعالجة المركزية، لذا يجب على نظام التشغيل تخزين نسخة من مسجّلات وحدة المعالجة المركزية في الذاكرة.

ينسخ نظام التشغيل قيم المسجّل مرةً أخرى من الذاكرة إلى مسجلات وحدة المعالجة المركزية عندما يحين وقت تشغيل العملية مرةً أخرى وستعود العملية للعمل من حيث توقفت.

5.2.5 حالة النواة

يجب أن تتعقّب النواة عددًا من العناصر لكل عملية والتي سنوضّحها فيما يلي.

ا. حالة العملية

يجب على نظام التشغيل تعقّب حالة العملية، فإذا كانت العملية قيد التشغيل حاليًا، فيجب أن تكون بحالة تشغيل Running، لكن إذا طلبت العملية قراءة ملف من القرص الصلب، فسنعلَم من تسلسل الذواكر الهرمي أنّ ذلك يمكن أن يستغرق وقتًا طويلًا، إذ يجب على العملية التخلي عن تنفيذها الحالي للسماح بتشغيل عملية أخرى، ولكن لا يجب أن تسمح النواة بتشغيل العملية مرةً أخرى حتى تصبح البيانات من القرص الصلب متاحةً في الذاكرة، وبالتالي يمكن تحديد العملية على أنها في حالة انتظار القرص الصلب Disk Wait حتى تصبح البيانات جاهزةً.

ب. الأولوية Priority

تُعَدّ بعض العمليات أكثر أهميةً من غيرها وتحظى بأولوية أعلى.

ج. الإحصائيات

يمكن للنواة الاحتفاظ بإحصائيات حول سلوك كل عملية والتي يمكن أن تساعدها في اتخاذ قرارات حول كيفية تصرف العملية مثل معرفة ما إذا كانت العملية تقرأ من القرص الصلب في أغلب الأحيان أم أنها تنفّذ عمليات مكثفةً في وحدة المعالجة المركزية بمعدّل أعلى.

5.3 تسلسل العمليات الهرمي

يمكن لنظام التشغيل تشغيل العديد من العمليات في الوقت نفسه، إلّا أنه يبدأ بتشغيل عملية واحدة مباشرةً تُدعَى بالعملية الأولية init -اختصارًا للكلمة Initial- التي لا تُعَدّ عمليةً خاصةً باستثناء أنَّ معرِّف العملية PID الخاص بها هو 0 دائمًا وستبقى مُشغَّلةً دائمًا.

تُعَدّ جميع العمليات الأخرى أبناءً Children لهذه العملية الأولية، فللعمليات شجرة عائلة مثل أيّ شجرة أخرى، إذ يكون لكل عملية أبًا Parent ويمكن أن يكون لها العديد من الأشقاء Siblings التي تُعَدّ عمليات أنشأها الأب نفسه.

يُستخدَم المصطلح "تولّد Spawn" عند الحديث عن العمليات الآباء التي تنشئ العمليات الأبناء مثل القول بأن "عملية ولّدت ابنًا"، كما يمكن أن تنشئ العمليات الأبناء مزيدًا من الأبناء وهكذا، وإليك مثال عن تنفيذ الأمر `pstree` الذي يعرض العمليات المُشغَّلة مثل شجرة:

```
init-+-apmd
     |-atd
     |-cron
     ...
     |-dhclient
     |-firefox-bin-+-firefox-bin---2*[firefox-bin]
     |             |-java_vm---java_vm---13*[java_vm]
     |             `-swf_play
```

يمكن إنشاء عمليات جديدة باستخدام واجهتين متعلقتين ببعضهما هما `fork` و `exec`.

5.4 استدعاءات النظام Fork و Exec

5.4.1 استدعاءات Fork

إذا وصلتَ إلى مفترق طرق، فسيكون لديك خياران لتختار من بينهما وسيؤثر هذا القرار على مستقبلك، كما تصل البرامج الحاسوبية إلى مفترق طرق عندما تضغط على استدعاء النظام `fork()`، إذ سيُنشئ نظام التشغيل عمليةً جديدةً مماثلةً للعملية الأب، إذ ستُنسَخ جميع الحالات التي تحدّثنا عنها سابقًا بما في ذلك الملفات المفتوحة وحالة المسجّل وجميع عمليات تخصيص الذاكرة التي تتضمن شيفرة البرنامج.

تُعَدّ القيمة المُعادة من استدعاء النظام الطريقةَ الوحيدة التي يمكن للعملية من خلالها تحديد ما إذا كانت العملية موجودةً مسبقًا أم عملية جديدة، إذ ستكون القيمة المُعادة إلى العملية الأب هي معرّف عملية الابن Process ID أو PID اختصارًا، في حين سيحصل الابن على القيمة المُعادة 0، وعندها نقول أن العملية متفرعة forked مع وجود علاقة أب-ابن.

5.4.2 استدعاءات Exec

يوفّر التفريع Forking طريقةً للعملية الحالية بأن تبدأ عملية جديدة، فإذا لم تكن العملية الجديدة جزءًا من برنامج العملية الأب كما هو الحال في الصدَفة Shell، إذ يجب أن يشغِّل المستخدِم أمرًا في عملية جديدة ليس لها علاقة بالصدَفة، فيجب تشغيل استدعاء النظام `exec` الذي سيبدّل بمحتويات العملية المُشغَّلة حاليًا معلومات من برنامج ثنائي.

العملية التي تتبعها الصدَفة عند إطلاق برنامج جديد هي `fork` أولًا، مما يؤدي إلى إنشاء عملية جديدة، ثم تنفيذ الاستدعاء `exec`، أي تحميل البرنامج الثنائي الذي يُفترَض تشغيله في الذاكرة وتنفيذه.

5.4.3 كيفية تعامل لينكس مع fork و exec

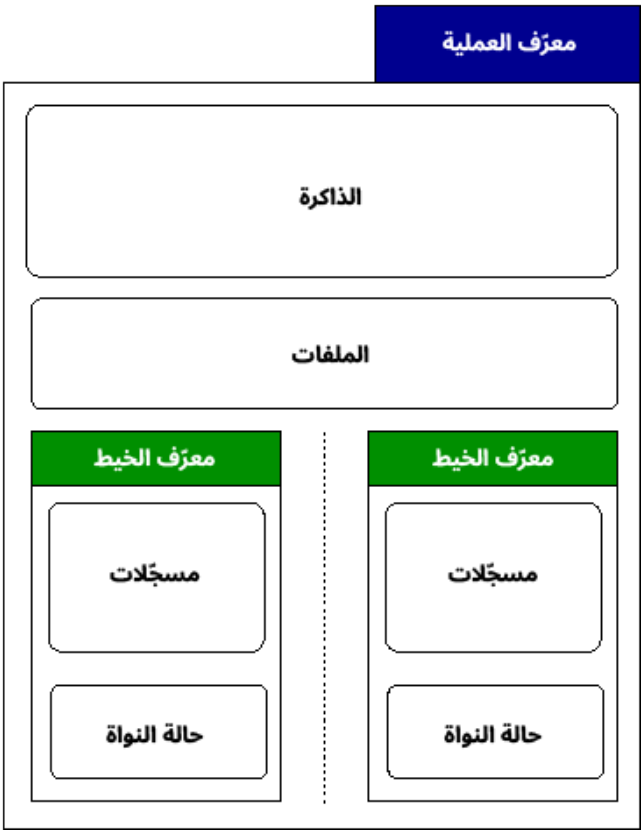
سنشرح كيفية تعامل نظام التشغيل لينكس مع عملية النسخ fork وعملية الاستدعاء exec.

ا. النسخ

يُنفَّذ الاستدعاء `fork` باستخدام استدعاء النظام `clone` في النواة، إذ توفّر واجهات `clone` بفعالية مستوًى من التجريد لكيفية إنشاء نواة لينكس للعمليات، كما يتيح الاستدعاء `clone` تحديد أجزاء العملية الجديدة المنسوخة في العملية الجديدة والأجزاء المشتركة بين العمليتين صراحةً، وقد يبدو هذا غريبًا بعض الشيء في البداية، لكنه يسمح لنا بسهولة بتطبيق الخيوط Threads باستخدام واجهة واحدة بسيطة جدًا.

الخيوط Threads

ينسخ الاستدعاء `fork` جميع السمات التي ذكرناها سابقًا. تخيّل نسخ كل شيء للعملية الجديدة باستثناء الذاكرة، فهذا يعني اشتراك الأب والابن في الذاكرة نفسها التي تتضمن شيفرة البرنامج والبيانات.



شكل 23: الخيوط Threads

يُسمَّى الابن الهجين السابق بالخيط، كما تحتوي الخيوط على عدد من المزايا بالموازنة مع المكان الذي يُستخدَم فيه الاستدعاء `fork` ومنها ما يلي:

1. لا يمكن للعمليات المنفصلة أن ترى ذاكرة بعضها بعضًا، وإنما يمكنها التواصل مع بعضها بعضًا عبر استدعاءات النظام الأخرى فقط، لكن مع ذلك تشترك الخيوط في الذاكرة نفسها، لذا سيكون لديك ميزة العمليات المتعددة مع الاضطرار إلى استخدام استدعاءات النظام للتواصل فيما بينها، وتكمن مشكلة ذلك في إمكانية تداخل الخيوط بسهولة مع بعضها بعضًا، إذ يمكن أن يزيد أحد الخيوط متغيرًا، في حين يمكن أن ينقصه خيط آخر بدون إعلام الخيط الأول، وتسمى هذه الأنواع من المشاكل بمشاكل التزامن وهي كثيرة ومتنوعة، ولكن يمكن حل هذه المشكلة باستخدام مكتبات مجال المستخدِم التي تساعد المبرمجين على العمل مع الخيوط بصورة صحيحة، وتسمى أكثر الخيوط شيوعًا بخيوط POSIX أو كما يشار إليها `pthreads` بصورة شائعة.
2. يُعَدّ التبديل بين العمليات مكلفًا جدًا، ومن أكثر الأمور تكلفةً هو تعقّب الذاكرة التي تستخدِمها كل عملية، ويمكن تجنّب ذلك من خلال مشاركة الذاكرة التي تزيد الأداء بصورة ملحوظة.

هناك العديد من الطرق المختلفة لتطبيق الخيوط، إذ يمكن أن يطبّق مجال المستخدِم الخيوط ضمن عملية دون أن تكون لدى النواة أيّ فكرة عن ذلك، إذ تظهر جميع الخيوط للنواة كأنها تعمل في عملية واحدة، ويُعَدّ ذلك دون المستوى الأمثل لأنّ النواة تحجب معلومات عمّا يجري تشغيله في النظام؛ أما مهمة النواة فهي التأكد من

استخدام موارد النظام بأفضل طريقة ممكنة، وإذا كان ما تعتقده النواة هو وجود عملية واحدة تشغّل خيوط عمليات متعددة، فيمكن أن تتخذ قرارات دون المستوى الأمثل.

الطريقة الأخرى هي أن تكون للنواة معرفة كاملة بالخيط، إذ يمكن إنشاء ذلك في نظام لينكس من خلال جعل جميع العمليات قادرة على مشاركة الموارد عبر استدعاء النظام `clone`، ولا يزال كل خيط يحتوي على موارد مرتبطة بالنواة، لذلك يمكن أن تأخذها النواة في حساباتها عند إجراء عمليات تخصيص الموارد.

تحتوي أنظمة التشغيل الأخرى على طريقة هجينة من الطريقتين السابقتين، إذ يمكن تحديد بعض الخيوط للتشغيل في مجال المستخدِم فقط -أي مخفيّة عن النواة- ويمكن أن يكون البعض الآخر من الخيوط عمليةً خفيفة الوزن، ويُعَدّ ذلك مؤشرًا مشابهًا للنواة على أن العمليات هي جزء من مجموعة خيوط.

النسخ عند الكتابة

يُعَدّ نسخ ذاكرة عملية كاملة إلى عملية أخرى عند استخدام الاستدعاء `fork` عمليةً مكلفةً كما ذكرنا سابقًا، ويمكن تحسين ذلك باستخدام ما يُسمَّى بالنسخ عند الكتابة Copy On Write، إذ يمكن مشاركة الذاكرة فعليًا بدلًا من نسخها بين العمليتين عند استدعاء `fork`، فإذا كانت العمليات ستقرأ الذاكرة فقط، فلن يكون نسخ البيانات ضروريًا، لكن يجب أن تكون النسخة خاصةً وغير مشتركة عندما تكتب عملية ما في ذاكرتها.

يعمل النسخ عند الكتابة -كما يوحي اسمه- على تحسين ذلك من خلال النسخ من الذاكرة فقط عندما تُكتَب النسخة فيها، وللنسخ عند الكتابة فائدة كبيرة للاستدعاء `exec` الذي سيكتب البرنامج الجديد في الذاكرة، لذا سيضيّع نسخ الذاكرة الكثير من الوقت، وبالتالي ستوفّر عملية النسخ عند الكتابة علينا النسخ فعليًا.

5.4.4 العملية الأولية Init Process

ناقشنا سابقًا الهدف العام للعملية الأولية وسنفهم الآن كيفية عملها، إذ تبدأ النواةُ العمليةَ الأولية `init` عند بدء التشغيل وتفرّع وتنفّذ هذه العملية بعد ذلك سكربتات بدء تشغيل الأنظمة التي تفرّع وتنفّذ مزيدًا من البرامج، وينتهي بها الأمر في النهاية إلى تفريع عملية تسجيل الدخول.

الوظيفة الأخرى للعملية `init` هي الحصاد Reaping، إذ سترغب العملية الأب في التحقق من الشيفرة المُعادة للتأكد من إنهاء العملية الابن بصورة صحيحة أم لا عندما تستدعي العملية استدعاء الإنهاء `exit` باستخدام الشيفرة المُعادة، لكن تُعَدّ شيفرة الإنهاء جزءًا من العملية التي استدعت `exit`، لذا يُقال أنّ هذه العملية ميتة Dead مثل أنها لا تعمل، ولكنها لا تزال بحاجة إلى البقاء حتى جمع الشيفرة المُعادة، وتسمى العملية في هذه الحالة شبه ميتة أو زومبي Zombie.

تبقى العملية في حالة زومبي حتى تجمع العملية الأب الشيفرة المُعادة من الاستدعاء `wait`، ولكن إذا انتهت العملية الأب قبل جمع الشيفرة المُعادة، فستبقى عملية الزومبي موجودةً وتنتظر بلا هدف لإعطاء حالتها

لعمليةٍ ما، ثم ستُنسَب العملية الابن التي تكون في حالة زومبي إلى العملية الأولية الـتي تمتلـك معالجًـا خاصًـا يحصد القيمة المُعادة، وستكون العملية حرةً في النهاية ويمكن إزالة واصفها من جدول عمليات النواة.

ا. مثال عملية شبه ميتة Zombie

إليك مثال عن عملية شبه ميتة أو عملية زومبي Zombie كما يقال:

```
$ cat zombie.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  pid_t pid;
  printf("parent : %d\n", getpid());
  pid = fork();

  if (pid == 0) {
    printf("child : %d\n", getpid());
    sleep(2);
    printf("child exit\n");
    exit(1);
  }

  /* في الأب */
  while (1)
  {
    sleep(1);
  }
}

$ ps ax | grep [z]ombie
pts/9    S      0:00 ./zombie
pts/9    Z      0:00 [zombie] <defunct>
```

أنشأنا في المثال السابق عملية زومبي، إذ ستكون العملية الأب في حالة سكون Sleep إلى الأبـد، في حين ستنتهي العملية الابن بعد بضع ثوان، ويمكنك رؤية نتائج تشغيل البرنامج بعد الشيفرة البرمجية.

تكون العملية الأب (16168) في الحالة `S` للإشارة إلى أنها في حالة سكون وتكون العملية الابن في الحالة `Z` للإشارة إلى أنها في حالة زومبي، في حين يخبرنا خرج الأمر `ps` أن العملية أصبحت زومبي أو `defunct` في وصف العملية.

> الأقواس المربعة حول الحرف "z" في الكلمة "zombie" هي خدعة صغيرة لتزيل عمليات الأمر `grep` نفسها من خرج الأمر `ps`، إذ يفسّر الأمر `grep` كل شيء بين الأقواس المربعة على أنه صنف محرفي Character Class، أي يبحث عن تطابق واحد فقط بين المحارف الموجودة بين القوسين والنص، ولكن بما أن اسم العملية سيكون "grep [z]ombie" مع الأقواس، فلن يكون هناك تطابق.

5.5 الجدولة Scheduling

يحتوي النظام المُشغَّل على مئات أو حتى أُلوف العمليات، ويُطلَق على جزء النواة Kernel الذي يتعقّب جميع هذه العمليات اسم المجدوِل Scheduler لأنه يجدول أيّ عملية يجب تشغيلها لاحقًا.

تُعَدّ خوارزميات الجدولة كثيرةً ومتنوعةً، إذ يكون لمعظم المستخدِمين أهداف مختلفة تتعلق بما يريدون تنفيذه من حواسيبهم، وهذا يؤثّر على قرارات الجدولة، فأنت تريد مثلًا التأكد من منح التطبيقات الرسومية في حاسوبك المكتبي متسعًا من الوقت للتشغيل حتى إذا استغرقت عمليات النظام وقتًا أطول قليلًا، مما سيؤدي إلى زيادة الاستجابة التي يشعر بها المستخدِم، وبالتالي سيكون لأفعالهم استجابات فورية، في حين يمكن أن ترغب في إعطاء الأولوية لتطبيق خادم الويب إذا عملتَ على خادم.

ينشئ الناس دائمًا خوارزميات جديدةً، كما يمكنك إنشاء خوارزمياتك الخاصة بسهولة إلى حد ما، ولكن هناك عدد من المكونات المختلفة للجدولة.

5.5.1 الجدولة ذات الأولوية Preemptive والجدولة التعاونية Co-operative

يمكن أن تنقسم استراتيجيات الجدولة إلى فئتين:

1. الجدولة التعاونية Co-operative Scheduling: هي المكان الذي تتخلى فيه العملية المُشغَّلة حاليًا طواعيةً عن التنفيذ للسماح بتشغيل عملية أخرى، والعيب في هذه الاستراتيجية هو أنّ العملية يمكنها اتخاذ قرار بعدم التخلي عن التنفيذ بسبب خطأ تسبَّب في شكل من أشكال الحلقة اللانهائية مثلًا، وبالتالي لا يمكن تشغيل أيّ شيء آخر.
2. الجدولة الاستباقية Pre-emptive Scheduling: هي المكان الذي تُقاطَع فيه العملية لإيقافها للسماح بتشغيل عملية أخرى، إذ تحصل كل عملية على شريحة زمنية Time-slice لتعمل فيها، كما سيُعاد ضبط عدّاد الوقت عند كل عملية تبديل سياق Context Switching وستُشغَّل العملية ثم تُقاطَع عند انتهاء الشريحة الزمنية، فتبديل السياق Context Switching هو العملية التي تطبّقها النواة للتبديل من عملية إلى أخرى، في حين يتعامل العتاد مع المقاطعة على أنها مستقلة عن العملية المُشغَّلة،

وبالتالي سيعود التحكم إلى نظام التشغيل عند حدوث المقاطعة، كما يمكن أن يقـرِّر المجـدوِل العمليـة التالية التي ستُشغَّل، وهذا هو نوع الجدولة الذي تستخدمه جميع أنظمة التشغيل الحديثة.

5.5.2 الوقت الفعلي Realtime

تحتاج بعض العمليات إلى معرفة المدة التي ستستغرقها شريحتها الزمنية والمدة التي المُسـتغرَقة قبـل أن تحصل على شريحة زمنية أخرى لتعمل، ولنفترض أنه لـديك نظامًـا يشـغِّل جهـاز القلب والرئـتين، إذ لا تريـد أن تتأخر النبضة التالية لأنّ شيئًا آخر قرّر العمل في النظام.

تقدّم أنظمة الوقت الفعلي الصارمة Hard Realtime ضمانات حول جدولة القـرارات مثـل الحـد الأقصـى لمقدار الوقت الذي ستُقاطَع فيه العملية قبل تشغيلها مرةً أخرى، إذ تُستخدَم غالبًا في التطبيقات الحرجـة مثـل التطبيقات الطبية والعسكرية وتطبيقات الطائرات، في حين لا تكون الضـمانات في أنظمـة الـوقت الفعلي غـير الصارمة Soft Realtime صارمةً ولكن يمكن التنبؤ بسلوك النظام العام.

يمكن استخدام نظام لينكس على أساس نظام وقت فعلي غير صارم، إذ يُستخدَم في الأنظمة الـتي تتعامـل مع الصوت والفيديو، وإذا أردتَ تسجيل بث صوتي، فلا بد أنك لا تريد مقاطعتك لفترات طويلة من الوقت لأنك ستفقد البيانات الصوتية التي لا يمكن استرجاعها.

5.5.3 القيمة اللطيفة Nice Value

تسنِد أنظمة يونيكس لكل عملية قيمةً لطيفة أو درجة لطف خاصة بها Nice Value، إذ ينظر المجـدوِل إلى هذه القيمة ويمكن أن يعطي الأولوية لتلك العمليات التي تتمتع بأعلى قيمة لطيفة.

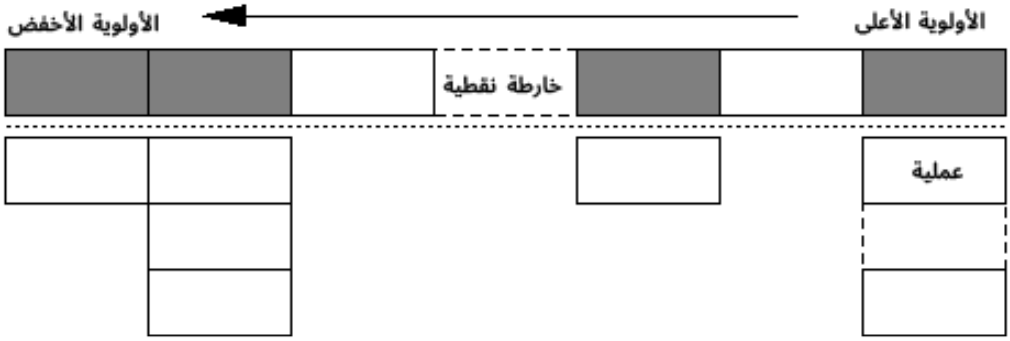
5.5.4 مجدول لينكس

خضع مجدول لينكس ولا يزال يخضع للعديد من التغييرات، إذ يحـاول المطـورون الجـدد تحسـين سـلوكه، ويُعرَف المجدول الحالي باسم المجدول O(1) الذي يشير إلى الخاصية التي تمثل أنّ المجـدول سـيختار العمليـة التالية لتشغيلها في فترة زمنية ثابتة بغض النظر عن عدد العمليات التي يجب عليه الاختيار من بينها.

تُعَدّ صيغة Big-O طريقةً لوصف الوقت الذي تستغرقه الخوارزمية للتشغيل بالنظر إلى الدخل المتزايد، فإذا استغرقت الخوارزمية ضعف الوقت للتشغيل مع ضعف الدخل، فهذا يـؤدي إلى التزايـد خطيًـا، وإذا اسـتغرقت خوارزمية أخرى أربعة أضعاف الوقت للتشغيل مع ضعف الدخل، فهـذا يـؤدي إلى تزايـد أسـي؛ أمـا إذا اسـتغرق الأمر الوقت نفسه مهما كان مقدار الدخل، فستُشغَّل الخوارزمية في وقت ثابت، ويمكنك رؤية أنـه كلمـا كـانت الخوارزمية تنمو بصورة أبطأ مع مزيد من الدخل، كان ذلك أفضل.

استخدمت مجدولات لينكس السـابقة مفهـوم الجـودة Goodness لتحديـد العمليـة التاليـة لتشـغيلها، إذ يُحتفَظ بجميع المهام المُحتمَلة في رتل تشغيل Run Queue، وهو قائمة مترابطـة من العمليـات الـتي تعـرِف النواة أنها في حالة قابلية للتشغيل، أي لا تنتظر نشاطًا من القرص الصلب أو ليست في حالة سكون.

تبرز مشكلة أنه يجب حساب مدى جودة كل عملية قابلة للتشغيل بحيث تفوز العملية التي تتمتع بأعلى جودة لتكون العملية التالية التي يجب تشغيلها، إذ سيستغرق الأمر وقتًا أطول بكثير لمزيد من المهام لتحديد العمليات التالية التي ستشغَّل.



شكل 24: المجدول O(1)

يستخدِم المجدول O(1) بنية رتل التشغيل الموضح في الشكل السابق، كما يحتوي رتل التشغيل على عدد من الحزم Buckets مرتبةً حسب الأولوية وخارطة نقطية Bitmap تشير إلى الحزم التي تحتوي على عمليات متاحة، إذ يُعَدّ البحث عن العملية التالية لتشغيلها بمثابة قراءة الخارطة النقطية للعثور على حزمة العمليات الأولى، ثم اختيار العملية الأولى من رتل الحزم.

يحتفظ المجدول ببنيتَين هما مصفوفة العمليات النشطة Active التي يمكن تشغيلها ومصفوفة العمليات منتهية الصلاحية Expired التي استخدمت شريحتها الزمنية بالكامل، كما يمكن تبديل هاتين البنيتَين ببساطة من خلال تعديل المؤشرات عندما يكون لجميع العمليات بعض الوقت من وحدة المعالجة المركزية.

لكن الجزء المهم هو كيفية تحديد المكان الذي يجب أن تذهب إليه العملية في رتل التشغيل، فمن الأشياء التي يجب أخذها في الحسبان هو المستوى اللطيف Nice Level، وتقارب المعالج Processor Affinity أو الحفاظ على العمليات مرتبطة بالمعالج الذي تُشغَّل عليه لأن نقل العملية إلى وحدة معالجة مركزية أخرى في نظام SMP يمكن أن يكون عمليةً مكلفةً، بالإضافة إلى دعم أفضل لتحديد البرامج التفاعلية مثل تطبيقات واجهة المستخدم الرسومية التي يمكن أن تقضي الكثير من الوقت في حالة سكون في انتظار الدخل من المستخدِم، ولكن يريد المستخدِم استجابةً سريعةً عندما يتفاعل معها.

5.6 الصدفة Shell

تُعَدّ الصدَفة في نظام يونيكس الواجهة المعيارية لمعالجة العمليات على نظامك، ولكن تحتوي أنظمة لينكس الحديثة على واجهة مستخدِم رسومية وتوفّر صدفةً عبر تطبيق طرفية Terminal أو ما شابه ذلك، كما تتمثل مهمة الصدَفة الأساسية في مساعدة المستخدِم على التعامل مع بدء العمليات المُشغَّلة في النظام وإيقافها والتحكم فيها.

إذا كتبتَ أمرًا في موجّه أوامر الصدفة، فسيؤدي ذلك إلى تطبيق الاستدعاء `fork` على نسخة منه وتطبيق الاستدعاء `exec` على الأمر الذي حددته، ثم تنتظِر الصدَفة بعد ذلك افتراضيًا حتى ينتهي تشغيل هذه العملية قبل العودة إلى موجّه الأوامر لبدء العملية بأكملها مرةً أخرى.

كما تسمح لك الصدَفة بتشغيل وظيفة ما في الخلفية Background من خلال وضع `&` بعد اسم الأمر للإشارة إلى وجوب تفر ع الصدَفة وتنفيذ الأمر دون الانتظار حتى يكتمل الأمر قبل أن تُظهِر لك موجّه الأوامر مرةً أخرى، في حين تعمل العملية الجديدة في الخلفية مع جهوزية الصدَفة في انتظار بدء عملية جديدة إذا رغبت في ذلك، لكن يمكنك إخبار الصدَفة بتنفيذ عملية ما في الأمام Foreground، مما يعني أننا نريد بالفعل انتظار انتهاء العملية.

5.7 الإشارات Signals

تتطلب العمليات المُشغَّلة في النظام طريقةً لإخبارنا بالأحداث التي تؤثر عليها، إذ توجد بنية تحتية في نظام يونيكس بين النواة Kernel والعمليات تسمّى الإشارات Signals التي تسمح للعملية بتلقي إشعار بالأحداث المهمة بالنسبة لها.

تستدعي النواة معالِجًا Handler يجب أن تسجّله العملية مع النواة للتعامل مع الإشارة المُرسَلة إلى عملية ما، والمعالج هو دالة مصمّمة في الشيفرة البرمجية التي كُتِبت لمعالجة المقاطعة، كما تُرسَل الإشارة في أغلب الأحيان من النواة نفسها، ولكن يمكن أن ترسِل إحدى العمليات إشارةً إلى عملية أخرى، وهذا يمثِّل أحد أشكال التواصل بين العمليات Interprocess Communication.

يُستدعَى معالج الإشارة بصورة غير متزامنة، إذ يُقاطَع البرنامج المشغَّل حاليًا عمّا يفعله لمعالجة حدث الإشارة، كما تُعَدّ المقاطعة أحد أنواع الإشارات التي تُحدَّد في ترويسات النظام بالاسم `SIGINT`، إذ تُسلَّم إلى العملية عند الضغط على الاختصار `ctrl-c`.

تستخدِم العملية استدعاء نظام `read` لقراءة الدخل من لوحة المفاتيح، إذ ستراقب النواة مجرى الدخل بحثًا عن محارف خاصة، لكن ستنتقل إلى وضع معالجة الإشارة في حالة ظهور الاختصار `ctrl-c`، وستبحث النواة لمعرفة ما إذا سجّلت العملية معالجًا لهذه المقاطعة، فإذا كان الأمر كذلك، فسيُمرَّر التنفيذ إلى تلك الدالة التي ستعالج المقاطعة، وإذا لم تسجّل العملية معالجًا لهذه الإشارة، فستتخذ النواة بعض الإجراءات الافتراضية، ويكون الإجراء الافتراضي هو إنهاء العملية باستخدام `ctrl-c`.

يمكن أن تختار العملية تجاهل بعض الإشارات ولكن لا تسمح بتجاهل الإشارات الأخرى، فالإشارة `SIGKILL` مثلًا هي الإشارة المرسَلة عندما يجب إنهاء العملية، حيث سترى النواة أن العملية أرسلت هذه الإشارة وتنهي تشغيل العملية دون طرح أيّ أسئلة، كما لا يمكن للعملية الطلب من النواة تجاهل هذه الإشارة، إذ تكون النواة حريصةً للغاية بشأن العملية المسموح لها بإرسال هذه الإشارة إلى عملية أخرى، فلا يجوز لك إرسالها إلا إلى العمليات التي تمتلكها إلا إذا كنت المستخدِم الجذر.

لا بد أنك رأيت الأمر `kill -9` الذي يأتي من تطبيق الإشارة SIGKILL، إذ تُعرَّف الإشارة SIGKILL على أنها 0x9، لذا ستتوقف العملية المحددة مباشرةً عند تحديدها على أساس وسيط لبرنامج kill، ونظرًا لأنه لا يمكن للعملية اختيار تجاهل هذه الإشارة أو معالجتها، فسيُنظَر إلى هذه الإشارة على أنها الملاذ الأخير، إذ لن يكون لدى البرنامج فرصةً للتنظيف أو الإنهاء بصورة نظيفة.

يُفضَّل إرسال الإشارة SIGTERM -للإنهاء Terminate- إلى العملية أولًا، فإذا تعطلت أو لم تنتهي، فيمكنك اللجوء إلى الإشارة SIGKILL، كما تثبّت معظم البرامج معالجًا للإشارة SIGHUP، أي تعليق Hangup الطرفيات وأجهزة المودِم التسلسلية، إذ سيعيد هذا المعالج تحميل البرنامج لالتقاط التغييرات في ملف الإعداد أو ما شابه ذلك.

إذا سبق لك وبرمجتَ على نظام يونيكس، فستكون على دراية بأخطاء التقطيع segmentation faults عندما تحاول القراءة أو الكتابة في ذاكرة غير مخصَّصة لك، فإذا لاحظت النواة أنك تحاول الوصول إلى ذاكرة ليست مخصَّصة لك، فسترسل لك إشارة خطأ تقطيع segmentation fault signal، ولن تمتلك العملية معالجًا مثبَّتًا لهذه الإشارة، وبالتالي فإنّ الإجراء الافتراضي هو إنهاء البرنامج وتعطيل برنامجك، كما يمكن أن يثبّت البرنامج معالجًا لأخطاء التقطيع في بعض الحالات المحدودة.

لكن يمكنك التساؤل عمّا يحدث بعد تلقي الإشارة، إذ سيُعاد التحكم إلى العملية التي تستأنف عملها من حيث توقفت بمجرد انتهاء معالج الإشارة من عمله، ويقدّم البرنامج البسيط التالي تشغيل بعض الإشارات:

```
$ cat signal.c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void sigint_handler(int signum)
{
  printf("got SIGINT\n");
}

int main(void)
{
  signal(SIGINT, sigint_handler);
  printf("pid is %d\n", getpid());
  while (1)
    sleep(1);
}
```

```
$ gcc -Wall -o signal signal.c
$ ./signal
pid is 2859
got SIGINT # press ctrl-c
           # press ctrl-z
[1]+  Stopped                 ./signal

$ kill -SIGINT 2859
$ fg
./signal
got SIGINT
Quit # press ctrl-\

$
```

يعرّف البرنامج البسيط السابق معالجًا للإشارة SIGINT التي تُرسَل عندما يضغط المستخدِم على الاختصار ctrl-c، إذ تُعرَّف جميع إشارات النظام في مكتبة signal.h بما في ذلك الدالة signal التي تسمح لنا بتسجيل دالة المعالجة.

يبقى البرنامج ضمن حلقة لا تفعل شيئًا حتى يتوقف، وحاول الضغط على الاختصار ctrl-c عند بدء البرنامج لإنهائه، إذ يُستدعَى المعالج ونحصل على الخرج المتوقَّع بدلًا من اتخاذ الإجراء الافتراضي، ثم نضغط بعد ذلك على الاختصار ctrl-z الذي يرسل الإشارة SIGSTOP التي تضع العملية افتراضيًا في وضع السكون، أي أنها لم تُوضَع في رتل تشغيل المجدول وبالتالي تُعَدّ خاملةً في النظام.

نستخدم برنامج kill لإرسال الإشارة نفسها من نافذة طرفية أخرى، إذ يمكن تطبيق ذلك فعليًا باستخدام استدعاء النظام kill الذي يأخذ إشارة ومعرّف PID لإرسالها، ويُعَدّ اسم هذه الدالة خاطئًا بعض الشيء، إذ لا تقتل جميعُ الإشارات العمليةَ فعليًا، ولكن تُستخدَم الدالة signal لتسجيل المعالج Handler، كما توضَع الإشارة في رتل خاص بهذه العملية عند توقفها، وبالتالي تأخذ النواة ملاحظةً بالإشارة وتسلّمها في الوقت المناسب.

ننبّه العملية بعد ذلك باستخدام الأمر fg الذي يرسل الإشارة SIGCONT إلى العملية، مما يؤدي إلى تنشيط العملية افتراضيًا، كما تدرك النواة وضع العملية في رتل التشغيل وتمنحها وقتًا من وحدة المعالجة المركزية مرةً أخرى، إذ نرى في هذه المرحلة تسليم الإشارة الموجودة في رتل التشغيل.

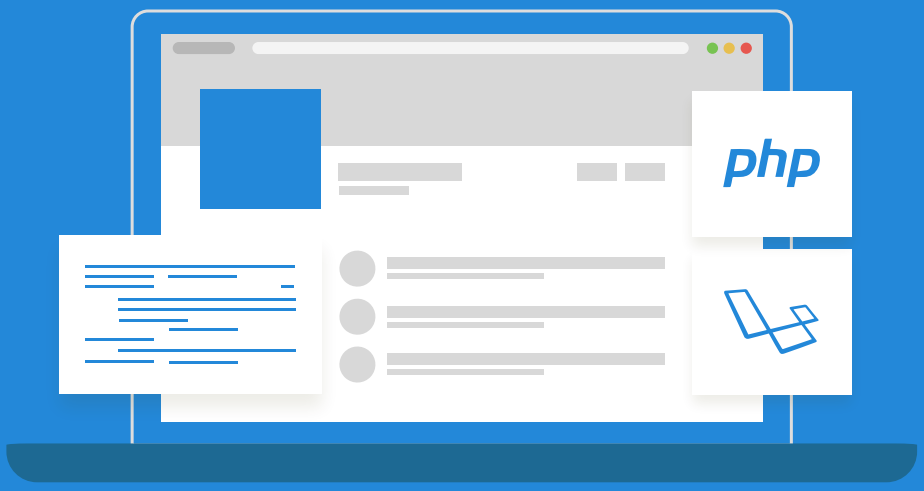
نحاول أخيرًا الضغط على الاختصار ctrl-\ الذي يرسل الإشارة SIGQUIT -أي إلغاء- إلى العملية، ويأتي خرج الإلغاء Quit من استخدام مزيد من الإشارات بالرغم من إلغاء العملية، وإذا كان لدى الأب عملية ابن ميتة

أو منتهيـة، فسيحصـل على الإشـارة SIGCHLD، إذ تُعَـدّ الصـدَفةُ أنهـا العمليـة الأب في هـذه الحالـة، أي أنهـا ستحصل على الإشارة.

تذكّر أنّ العملية الزومبي Zombie التي يجب حصادها باستخدام الاسـتدعاء wait للحصـول على الشـيفرة المُعادة من العملية الابن، ولكن هناك شيء آخر يمنحه الابن للأب وهـو رقم الإشـارة الـتي أدّت إلى مـوت الابن، وهكذا تعرف الصدَفة أنّ العملية الابن قد مـاتت أو انتهت بسـبب الإشـارة SIGABRT وتطبـع معلومـات أخـرى للمستخدِم على أساس خدمة إعلامية، إذ تحدُث العملية نفسها لطباعـة خطـأ التقطيـع Segmentation Fault عندما تموت العملية الابن بسبب الإشارة SIGSEGV.

يمكنك رؤية استخدام حوالي خمس إشـارات مختلفـة للتواصـل بين العمليـات والنـواة والحفـاظ على سـير الأمور حتى في برنامج بسيط، وهناك العديد من الإشارات الأخـرى، لكننـا اسـتخدمنا في هـذا المثـال الإشـارات الأكـثر شـيوعًا، إذ تحتـوي معظمهـا على دوال نظـام تعرّفهـا النـواة، ولكن هنـاك بعض الإشـارات المحجـوزة للمستخدِمين لاستخدامها لأغراضهم الخاصة في برامجهم (SIGUSR).

دورة تطوير تطبيقات الويب
باستخدام لغة PHP



مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

6. الذاكرة الوهمية Virtual Memory

يمكن القول بأن الذاكرة الوهمية Virtual Memory هي طريقة لتوسيع الذاكرة RAM من خلال استخدام القرص الصلب بوصفه ذاكرة نظام إضافية ولكنها أبطأ، أي ينتقل النظام إلى القرص الصلب الذي يُستخدَم بوصفه ذاكرةً وهمية بمجرد نفاد الذاكرة في نظامك.

يُشار إلى الذاكرة الوهمية عادةً في أنظمة التشغيل الحديثة باسم ذاكرة سواب Swap Space، لأن الأجزاء غير المُستخدَمة من الذاكرة تُبعَد إلى القرص الصلب لتحرير الذاكرة الرئيسية، إذ لا يمكن تنفيذ البرامج إلا من الذاكرة الرئيسية. تُعَد القدرة على إبعاد الذاكرة إلى القرص الصلب أمرًا مهمًا، ولكنها ليست الغرض الأساسي للذاكرة الوهمية، بل لها تأثيرٌ آخر مفيد للغاية سنراه لاحقًا.

6.1 ما هي الذاكرة الوهمية Virtual Memory؟

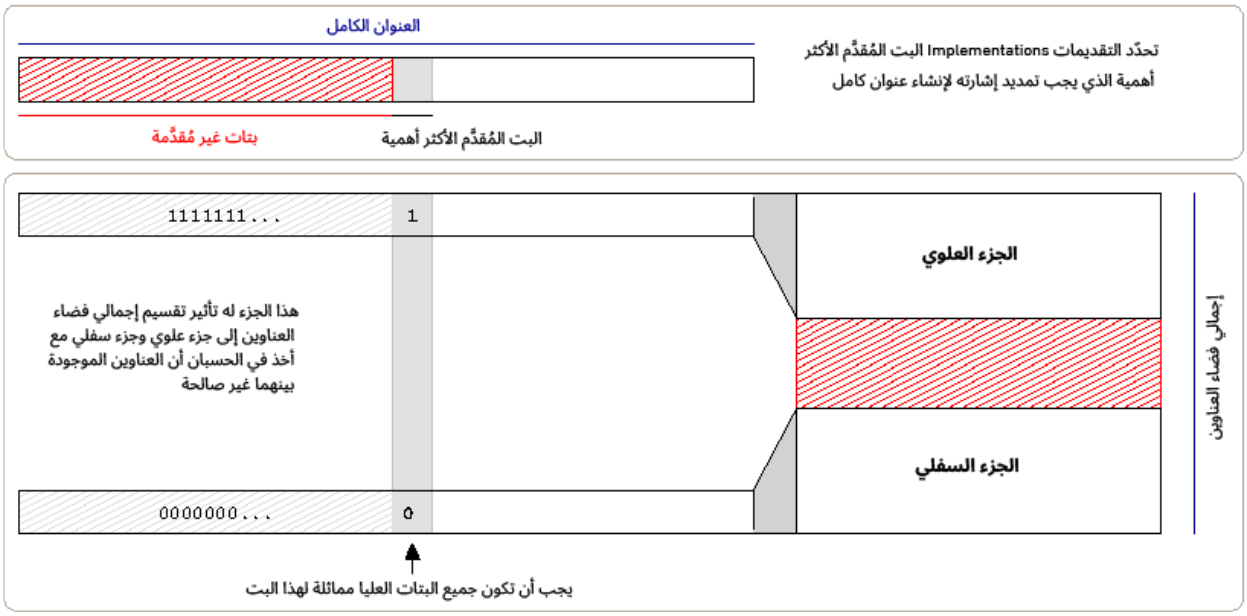
تدور الذاكرة الوهمية حول فكرة الاستفادة من فضاء العناوين Address Space، حيث يشير فضاء العناوين الخاص بالمعالج إلى مجال العناوين المُحتمَلة التي يمكن استخدامها عند التحميل والتخزين في الذاكرة. يُعَد فضاء العناوين محدودًا بعرض المسجّلات Registers، لأننا نحتاج لتحميل عنوانٍ إطلاقَ تعليمة تحميل `load` مع العنوان الذي سيُحمَّل منه العنوان المُخزَّن في المسجّل، إذ يمكن مثلًا أن تحتوي المسجلات التي يبلغ عرضها 32 بت على عناوين في مجال المسجل من `0x00000000` إلى `0xFFFFFFFF`. يساوي 232 ما مقداره 4 جيجابايت، لذلك يمكن للمعالج ذي 32 بت تحميلُ أو تخزين ما يصل إلى 4 جيجابايتات من الذاكرة.

6.1.1 المعالجات ذات 64 بت

جميع المعالجات الجديدة هي معالجات 64 بت التي -كما يوحي اسمها- تحتوي على مسجّلات بعرض 64 بت، حيث يكون فضاء العناوين المتاح لهذه المعالجات كبيرًا. تحتوي المعالجات ذات 64 بت على بعض المقايضات مقابل استخدام معالجات ذات عرض بتات أصغر، حيث يتطلب كل برنامجٍ مُصرَّفٍ Compiled في وضع 64 بت مؤشرات حجمها 8 بايتات، والتي يمكن أن تزيد من حجم الشيفرة البرمجية والبيانات، وبالتالي تؤثر على أداء كل من الذاكرة المخبئية الخاصة بالتعليمة والبيانات، ولكن تميل معالجات 64 بت إلى الحصول على عدد أكبر من المسجلات، مما يعني تقليل الحاجة إلى حفظ المتغيرات المؤقتة في الذاكرة عندما يكون المصرّف Compiler واقعًا تحت الضغط القادم من المسجّلات.

ا. العناوين المعيارية Canonical Addresses

تحتوي معالجات 64 بت على مسجلات بعرض 64 بت، ولكن لا تطبّق الأنظمة جميع هذه 64 بت للعنونة، إذ لا يُعَد تحميل `load` أو تخزين `store` كل 16 إكسابايت من الذاكرة الحقيقية أمرًا ممكنًا. لذا تحدّد معظم المعماريات منطقةً غير قابلة للاستخدام Unimplemented من فضاء العناوين التي يَعُدّها المعالج غير صالحة للاستخدام. تعرّف كلٌّ من المعماريتين x86-64 وإيتانيوم Itanium البت الصالح الأكثر أهمية في العنوان، ويجب بعد ذلك تمديد إشارته لإنشاء عنوان صالح، والنتيجة هي تقسيم إجمالي فضاء العناوين بفعالية إلى جزأين هما: جزء علوي وجزء سفلي مع وجود عناوين غير صالحة بينهما، وهذا موضح في الشكل التالي، حيث تُسمَّى العناوين الصالحة عناوين معيارية Canonical Addresses، بينما تُسمَّى العناوين غير الصالحة عناوين غير معيارية Non-canonical.



شكل 25: توضيح للعناوين المعيارية canonical addresses

يمكن العثور على قيمة البت الأكثر أهمية للمعالج من خلال الاستعلام عن المعالج نفسه باستخدام تعليمة الحصول على المعلومات. ستكون قيمة البت الأكثر أهمية 48 بالرغم من أن القيمة الدقيقة تعتمد على التقديم Implementation، مما يؤدي إلى توفير 256 = 248 تيرابايت TiB من فضاء العناوين القابلة للاستخدام.

يُعَد تقليل فضاء العناوين المُحتمَل أنه يمكن تحقيق توفيرٍ كبير مع جميع أجزاء منطق العنونة في المعالج والمكونات ذات الصلة، لأنها تعلم أنها لن تحتاج للتعامل مع عناوين 64 بت كاملة. يحدّد التقديمُ البتات العليا على أنه يجب تمديد إشارتها، مما يؤدي إلى منع أنظمة التشغيل القابلة للنقل التي تستخدم هذه البتات لتخزين أو تحديد المعلومات الإضافية وضمان التوافق عند الرغبة في تقديم مزيدٍ من فضاء العناوين مستقبلًا.

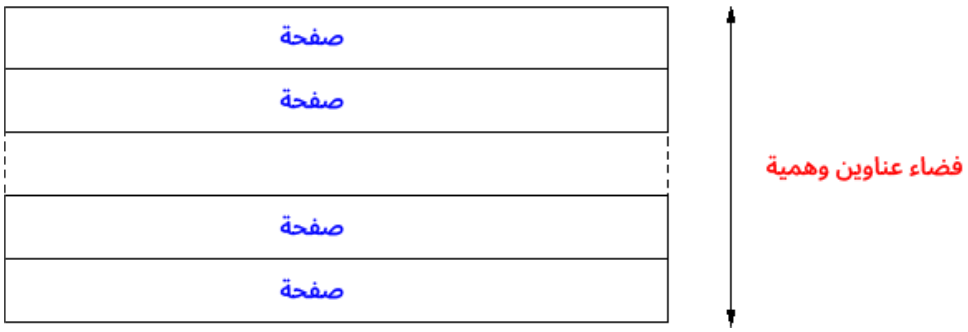
6.1.2 استخدام فضاء العناوين

تعمل الذاكرة الافتراضية -كما هو الحال مع معظم مكونات نظام التشغيل- بوصفها تجريدًا بين فضاء العناوين والذاكرة الحقيقية المتوفرة في النظام، أي إذا استخدم برنامجٌ ما عنوانًا، فلن يشير العنوان إلى البتات الموجودة في الموقع الفعلي الحقيقي في الذاكرة، لذا نقول أن جميع العناوين التي يستخدمها البرنامج هي عناوين وهمية. يتعقّب نظام التشغيل العناوين الوهمية وكيفية تخصيصها للعناوين الحقيقية، فإذا طبّق أحد البرامج عملية تحميل أو تخزين من عنوانٍ ما، فسيعمل المعالج ونظام التشغيل مع بعضهما البعض لتحويل هذا العنوان الوهمي إلى العنوان الحقيقي في شرائح ذاكرة النظام.

6.2 الصفحات Pages

يُقسَم إجمالي فضاء العناوين إلى صفحات Pages. يمكن أن تكون الصفحات بأحجام مختلفة، حيث يمكن أن يبلغ حجمها حوالي 4 كيلوبايت KiB، ولكنها ليست قاعدة صارمة ويمكن أن تكون أكبر بكثير ولكنها ليست أصغر من ذلك. تُعَد الصفحة أصغر وحدة ذاكرة يمكن لنظام التشغيل والعتاد التعامل معها.

تحتوي كل صفحة على عدد من السمات التي يضبطها نظام التشغيل، وتشمل أذونات القراءة والكتابة والتنفيذ للصفحة الحالية، حيث يمكن لنظام التشغيل مثلًا تمييز صفحات الشيفرة البرمجية لعمليةٍ ما باستخدام راية قابلة للتنفيذ ويمكن للمعالج اختيار عدم تنفيذ أيّ شيفرة برمجية من الصفحات بدون ضبط هذه البتات.



شكل 26: صفحات الذاكرة الوهمية

يمكن أن يفكر المبرمجون في هذه المرحلة في أنه يمكنهم بسهولة تخصيص كميات صغيرة من الـذاكرة -أي أصغر بكثير من 4 كيلوبايتات- باستخدام الاستدعاء `malloc` أو اسـتدعاءات مماثلـة. تـدعم عمليـات تخصـيص حجم الصفحة كومةَ Heap الذاكرة، حيث يقسمها تقديم الاستدعاء `malloc` ويديرها بطريقة فعّالة.

6.3 الذاكرة الحقيقية Physical Memory

يقسم نظام التشغيل فضاء العناوين المُحتمَلة إلى صفحات Pages، ويقسم الـذاكرة الحقيقيـة المتاحـة إلى إطـارات Frames، حيث يُعَـد الإطـار الاسـم التقليـدي لقطعـة كبـيرة من الـذاكرة الحقيقيـة لهـا حجم صـفحة النظام نفسها.

يحتفظ نظام التشغيل بجدول الإطارات Frame-table الذي يُعَد قائمةً بجميع الصفحات المُحتمَلـة للـذاكرة الحقيقية ويحدد ما إذا كانت حرةً أو متاحة للتخصيص أم لا. إذا خُصِّصت الذاكرة لعمليـةٍ مـا، فسـتُميَّز على أنهـا مُستخدَمة في جدول الإطارات، وبذلك يتعقّب نظام التشـغيل جميـع عمليـات تخصـيص الـذاكرة. يعـرف نظـام التشغيل الذاكرة المتوفرة من خلال تمرير المعلومات الخاصة بمكان وجود الذاكرة ومقدارها وسـماتها وغـير ذلـك إلى نظام التشغيل باستخدام نظام BIOS أثناء عملية التهيئة Initialisation.

6.4 جداول الصفحات

تتمثـل مهمـة نظـام التشـغيل في تعقّب نقـاط الصـفحة الوهميـة المقابلـة للإطـار الحقيقي، حيث يجـري الاحتفاظ بهذه المعلومات في جدول صفحات. يمكن أن يكون جدول الصفحات في أبسط أشكاله جدولًا يحتـوي كل صف فيه على الإطار المرتبط به، وهذا ما يسمى بجدول الصفحات الخطي Linear Page-table.

إن استخدمتَ هذا النظام البسيط مع فضاء عناوين بحجم 32 بت وصـفحات بحجم 4 كيلوبـايت، فسـيكون هناك 1048576 صفحة يمكن تعقّبها في جدول الصفحات (أي 4096 ÷ 232)، وبالتالي سيكون طول الجدول 1048576 مدخلةً لضمان أنه يمكننا دائمًـا ربـط صـفحة وهميـة مـع صـفحة حقيقيـة. يمكن أن تحتـوي جـداول الصفحات على العديد من البنى المختلفة ويمكن تحسينها بدرجة كبيرة، إذ يمكن أن تستغرق عمليــة البحث عن صفحة في جدول الصفحات وقتًا طويلًا. سنتطرق إلى جدول الصفحات بمزيد من التفصيل لاحقًا.

يخضع جدول صفحات العملية لتحكم نظام التشغيل الحصري، فإذا طلبت إحدى العمليات ذاكـرةً، فسـيجد نظام التشغيل صـفحة خاليـة من الـذاكرة الحقيقيـة ويسـجّل ترجمـة الصـفحة الوهميـة إلى الصـفحة الحقيقيـة Virtual-to-physical في جدول صفحات العمليات. بينما إن تخلت العملية عن الذاكرة، فسيُزال سجل ترجمــة الصفحة الوهمية إلى الصفحة الحقيقية ويصبح الإطار الأساسي حرًا لتخصيصه لعملية أخرى.

6.5 العناوين الوهمية Virtual Address

لا يعرف أو يهتم البرنامج عند وصوله إلى الذاكرة بمكان تخزين الذاكرة الحقيقية التي تدعم العنوان، ولكنه يعرف أن الأمر متروك لنظام التشغيل والعتاد، بحيث يتعاونان للربط مع العنوان الحقيقي الصحيح وبالتالي توفير الوصول إلى البيانات التي يريدها. لذا نطلق على العنوان الذي يستخدمه البرنامج للوصول إلى الذاكرة عنوانًا وهميًا Virtual Address الذي يتكون من جزأين هما: الصفحة Page والإزاحة Offset في هذه الصفحة.

6.5.1 الصفحة

يُقسَم فضاء العناوين المُحتمَل إلى صفحات ذات حجم ثابت، حيث يتواجد كل عنوان ضمن صفحة، ويعمل مكون الصفحة الخاص بالعنوان الوهمي بوصفه فهرسًا إلى جدول الصفحات. تُعَد الصفحة أصغر وحدة لتخصيص الذاكرة في النظام، لذلك هناك مقايضة بين جعل الصفحات صغيرة جدًا مع وجود عدد كبير جدًا منها ليديرها نظام التشغيل وبين جعل الصفحات أكبر مع وجود احتمال في هدر الذاكرة.

6.5.2 الإزاحة Offset

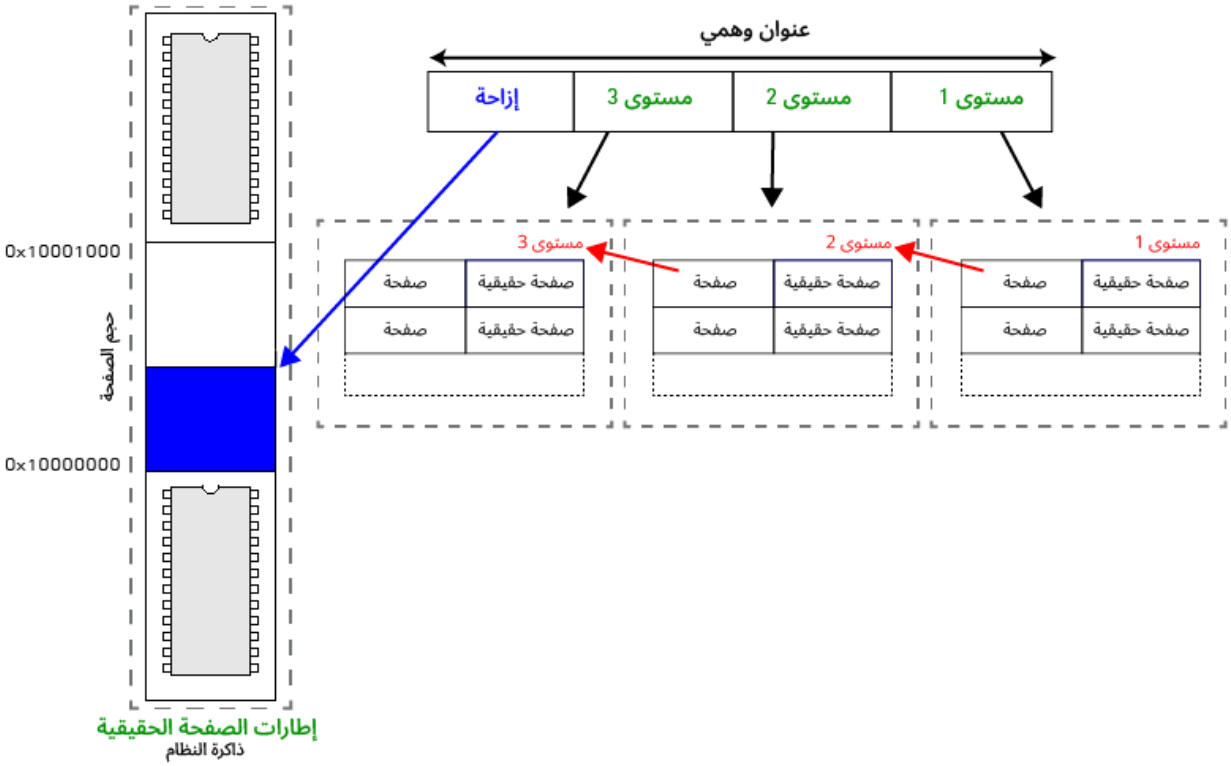
تُسمَّى البتات الأخيرة من العنوان الوهمي بالإزاحة Offset التي تعبّر عن الفرق في الموقع بين عنوان البايت الذي تريده وبداية الصفحة، ويجب وجود بتات كافية في الإزاحة لتتمكن من الوصول إلى أيّ بايت في الصفحة، وتحتاج صفحة بحجم 4 كيلوبايتات إلى 12 بت للإزاحة حيث أن 212 = 4096 = 1024 * 4K.

تذكر أن أقل قدر من الذاكرة يتعامل معه نظام التشغيل أو العتاد يساوي صفحة، لذا يوجد كل بت من 4096 بايت ضمن صفحة واحدة ويجري التعامل معها على أنها كتلة واحدة.

6.5.3 ترجمة العنوان الوهمي Virtual Address

تشير ترجمة أو نقل العناوين الوهمية Virtual address translation إلى عملية اكتشاف الصفحة الحقيقية المربوطة مع الصفحة الوهمية. سنتعامل فقط مع رقم الصفحة عند ترجمة عنوان وهمي إلى عنوان حقيقي، حيث نأخذ رقم الصفحة من العنوان المُعطَى ونبحث عنه في جدول الصفحات للعثور على مؤشر إلى عنوان حقيقي مع إضافة الإزاحة من العنوان الوهمي إليه، مما يؤدي إلى إعطاء الموقع الفعلي في نظام الذاكرة.

تخضع جداول الصفحات لسيطرة نظام التشغيل، فإن لم يكن العنوان الوهمي موجودًا في جدول الصفحات، فسيعرف نظام التشغيل أن العملية تحاول الوصول إلى الذاكرة التي ليست مخصَّصةً لها ولن يُسمَح لها بالوصول.



شكل 27: ترجمة العنوان الوهمي

يوضّح المثال السابق جدول صفحات خطي بسيط، حيث سيتطلب فضاء العناوين ذو 32 بت جدولًا مؤلفًا من 1048576 مدخلةً عند استخدام صفحات بحجم 4 كيلوبايتات، وبالتالي ستكون الخطوة الأولى لربط العنوان 0x80001234 هي إزالة بتات الإزاحة. نعلم في هذه الحالة أن لدينا 12 بت (2^{12} = 4096) من الإزاحة مع صفحات بحجم 4 كيلوبايتات. لذا سنزيح 12 بت من العنوان الوهمي إزاحةً يمنى، وبالتالي يبقى لدينا 0x80001، وستكون القيمة العشرية الموجودة في السطر رقم 524289 من جدول الصفحات الخطي هي الإطار الحقيقي المقابل لهذه الصفحة.

يمكن أن ترى مشكلة في جدول الصفحات الخطي، حيث يجب حساب كل صفحة سواء كانت قيد الاستخدام أم لا، وبالتالي لا يُعَد جدول الصفحات الخطي الحقيقي عمليًا تمامًا مع فضاء عناوين 64 بت. ضع في حساباتك فضاء عناوين 64 بت المقسَّم إلى صفحات مؤلفة من 64 كيلوبايت (كبيرة جدًا)، حيث ينشئ هذا الفضاء $2^{64}/2^{16}=2^{52}$ صفحة لإدارتها. لنفترض أن كل صفحة تتطلب مؤشرًا بحجم 8 بايتات لموقع حقيقي، فسيتطلب ذلك $2^{52}*2^{3}=2^{55}$ أو 512 جيجابايت GiB من الذاكرة المتجاورة لجدول الصفحات فقط.

6.6 مفاهيم متعلقة بالعناوين الوهمية والصفحات وجداول الصفحات

تُعَد العناوين الوهمية والصفحات وجداول الصفحات أساس كل نظام تشغيل حديث، لأنها تشكّل أساس معظم الأشياء التي نستخدم أنظمتنا من أجلها.

6.6.1 فضاءات العناوين المفردة

يمكن لكل عملية التظاهر بأنها تستطيع الوصول إلى فضاء العناوين الكامل المتاح من المعالج من خلال إعطاء كل عملية جدول صفحات خاص بها، إذ يمكن أن تستخدم عمليتان العنوان نفسه، حيث ستربط جداولُ الصفحات المختلفة العمليةَ مع إطار مختلف من الذاكرة الحقيقية، إذ توفّر أنظمة التشغيل الحديثة لكل عمليةٍ فضاءَ عناوين خاص بها.

تصبح الذاكرة الحقيقية مجزأة Fragmented بمرور الوقت، مما يعني أن هناك ثقوب في الفضاء الحر من الذاكرة الحقيقية. سيكون الاضطرار إلى حل مشكلة هذه الثقوب أمرًا مزعجًا في أحسن الأحوال ولكنه سيصبح أمرًا خطيرًا للمبرمجين، فإذا نفّذتَ الاستدعاء `malloc` لتخصيص 8 كيلوبايتات من الذاكرة مثلًا، فسيتطلب ذلك دعم إطارين بحجم 4 كيلوبايتات، وبالتالي لن تكون هذه الإطارات متجاورة، أي بجوار بعضها البعض فعليًا. لا يُعَد استخدام العناوين الوهمية أمرًا مهمًا بقدر ما يتعلق الأمر باحتواء العملية على 8 كيلوبايت من الذاكرة المتجاورة، حتى لو كانت هذه الصفحات مدعومة بإطارات متباعدة جدًا. يمكن للمبرمج ترك مهمة حل مشكلة التجزئة لنظام التشغيل من خلال إسناد فضاء عناوين وهمية لكل عملية.

6.6.2 الحماية

يُدعَى الوضع الوهمي للمعالج 386 بالوضع المحمي Protected Mode، وينشأ هذا الاسم من الحماية التي يمكن أن توفرها الذاكرة الوهمية للعمليات التي تعمل عليها. تتمتع كل عملية في نظام بدون ذاكرة وهمية بوصولٍ كامل إلى ذاكرة النظام بأكملها، وهذا يعني أنه لا يوجد شيء يمنع عمليةً ما من الكتابة فوق ذاكرة عمليات أخرى، مما يؤدي إلى تعطّلها أو إعادة قيم غير صحيحة في أسوأ الأحوال خاصة إذا كان هذا البرنامج يدير حسابك المصرفي مثلًا. لذا يجب توفير هذا المستوى من الحماية لأن نظام التشغيل يُعَد طبقة تجريد بين العملية والوصول إلى الذاكرة، فإذا أعطت العملية عنوانًا وهميًا لا يغطيه جدول الصفحات الخاص بها، فسيعلم نظام التشغيل أن هذه العملية تطبّق شيئًا خاطئًا ويمكنه إبلاغ العملية أنها تعدّت حدودها.

تمتلك كل صفحة سمات إضافية، لذا يمكن ضبط الصفحة للقراءة فقط أو للكتابة فقط أو غيرها من الخاصيات الأخرى. إذا حاولت العملية الوصول إلى الصفحة، فيمكن لنظام التشغيل التحقق مما إذا كان لديها أذونات كافية وإيقافها إن لم تكن كذلك مثل محاولة الكتابة في صفحة للقراءة فقط.

تُعَد الأنظمة التي تستخدم الذاكرة الوهمية أكثر استقرارًا لأنه يمكن للعملية في نظام تشغيل مثالي أن تعطّل نفسها فقط دون تعطيل النظام بأكمله، حيث تُبرمَج أنظمة تشغيل مع تجاهل الأخطاء التي يمكن أن تتسبّب في تعطل الأنظمة بأكملها.

6.6.3 التبديل Swap

يمكننا الآن أن نرى كيفية تقديم تبديل ذاكرة، حيث يمكن تغيير مؤشر الصفحة ليؤشّر إلى موقع على القرص الصلب بدلًا من التأشير إلى منطقة من ذاكرة النظام. يحتاج نظام التشغيل عند الرجوع إلى هذه الصفحة إلى نقلها من القرص الصلب إلى ذاكرة النظام، إذ لا يمكن تنفيذ شيفرة البرنامج إلا من ذاكرة النظام. إذا كانت ذاكرة النظام ممتلئة، فيجب إخراج صفحة أخرى من ذاكرة النظام وتبديلها بالقرص الصلب قبل وضع الصفحة المطلوبة في الذاكرة. إذا كانت هناك عملية أخرى تريد الصفحة التي أُخرِجت للتو، فستتكرر هذه العملية مرة أخرى.

يمكن أن يؤدي ذلك إلى مشكلةٍ كبيرة في تبديل الذاكرة، حيث يُعَد التحميل من القرص الصلب بطيئًا جدًا بالموازنة مع العمليات التي تُنجَز في الذاكرة، وسيكون معظم الناس متآلفين مع فكرة الجلوس أمام الحاسوب أثناء توقف القرص الصلب مرارًا وتكرارًا مع بقاء النظام غير مستجيب.

I. mmap

تُعَد عملية ربط الذاكرة Memory Map أو `mmap` (من اسم استدعاء النظام) عمليةً مختلفة ولكنها ذات صلة، حيث إن لم يؤشّر جدول الصفحات إلى الذاكرة الحقيقية أو لم يؤشّر تبديل جدول الصفحات إلى ملف على القرص الصلب، فسنقول أن الملف مربوط بالذاكرة `mmap`.

تحتاج عادةً إلى فتح `open` ملف على القرص الصلب للحصول على واصف الملف ثم قراءته `read` وكتابته `write` في صيغة تسلسلية. إذا كان الملف مربوطًا بالذاكرة، فيمكن الوصول إليه مثل الذاكرة RAM الخاصة بالنظام.

6.6.4 مشاركة الذاكرة

تحصل كل عملية على جدول صفحات خاص بها، لذلك يُربَط أيّ عنوان تستخدمه مع إطار فريد في الذاكرة الحقيقية، ولكن إن أشّر نظام التشغيل إلى مدخلتَين من جدول الصفحات إلى الإطار نفسه، فهذا يعني التشارك في هذا الإطار، وستكون أيّ تغييرات تجريها إحدى العمليتين مرئية للعملية الأخرى.

يمكنك أن ترى الآن كيفية تقديم الخيوط Threads. يمكن للدالة `clone()` الخاصة بنظام لينكس مشاركة قدر كبير أو صغير من العملية الجديدة مع العملية القديمة وفق ما هو مطلوب. إن استدعت عمليةٌ الدالة `clone()` لإنشاء عملية جديدة، ولكنها تطلب أن تشترك العمليتان في جدول الصفحات نفسه، فسيكون لديك خيط حيث ترى كلتا العمليتين الذاكرة الحقيقية الأساسية نفسها.

كما يمكنك معرفة كيفية إجراء النسخ عند الكتابة، حيث إذا ضبطتَ أذونات إحدى الصفحات لتكون للقراءة فقط، فسيجري إعلام نظام التشغيل عندما تحاول إحدى العملياتُ الكتابةَ في الصفحة. إذا عَلِم نظام التشغيل أن هذه الصفحة هي صفحة نسخ عند الكتابة، فيجب إنشاء نسخة جديدة من الصفحة في ذاكرة النظام ويجب أن

توشّر الصفحة في جدول الصفحات إلى هذه الصفحة الجديدة. يمكن بعد ذلك تحديث سمات الصفحة للحصول على أذونات الكتابة ويكون للعملية نسختها الفريدة من الصفحة.

6.6.5 ذاكرة القرص الصلب المخبئة Cache

توجد في الأنظمة الحديثة ذاكرة متوفرة أكثر مما يستخدمه النظام حاليًا بدلًا من وجود ذاكرة قليلة جدًا والاضطرار إلى تبديل الذاكرة. يخبرنا تسلسل الذواكر الهرمي أن الوصول إلى القرص الصلب أبطأ بكثير من الوصول إلى الذاكرة، لذلك يُفضَّل نقل أكبر قدر ممكن من البيانات من القرص الصلب إلى ذاكرة النظام إن أمكن ذلك.

ينسخ نظام لينكس والعديد من الأنظمة الأخرى البيانات من الملفات الموجودة على القرص الصلب إلى الذاكرة عند استخدامها. يُحتمَل أن يرغب البرنامج في الوصول إلى بقية الملف مع استمراره في المعالجة حتى إن طلب في البداية جزءًا صغيرًا فقط من الملف، ويتحقق نظام التشغيل عند قراءة ملف أو الكتابة فيه أولًا مما إذا كان الملف موجودًا في الذاكرة المخبئة Cache. يجب أن تكون هذه الصفحات هي أولى الصفحات التي ستُزال عند زيادة ضغط الذاكرة في النظام.

ا. ذاكرة الصفحة المخبئة Page Cache

المصطلح الذي يمكن أن تسمعه عند مناقشة النواة Kernel هو ذاكرة الصفحة المخبئية Page Cache التي تشير إلى قائمة الصفحات التي تحتفظ بها النواة والتي تشير إلى الملفات الموجودة على القرص الصلب، حيث تندرج صفحة التبديل والصفحات المربوطة بالذاكرة وصفحات ذاكرة القرص الصلب المخبئية ضمن هذه الفئة. تحتفظ النواة بهذه القائمة لأنها تحتاج إلى أن تكون قادرة على البحث عنها بسرعة استجابةً لطلبات القراءة والكتابة.

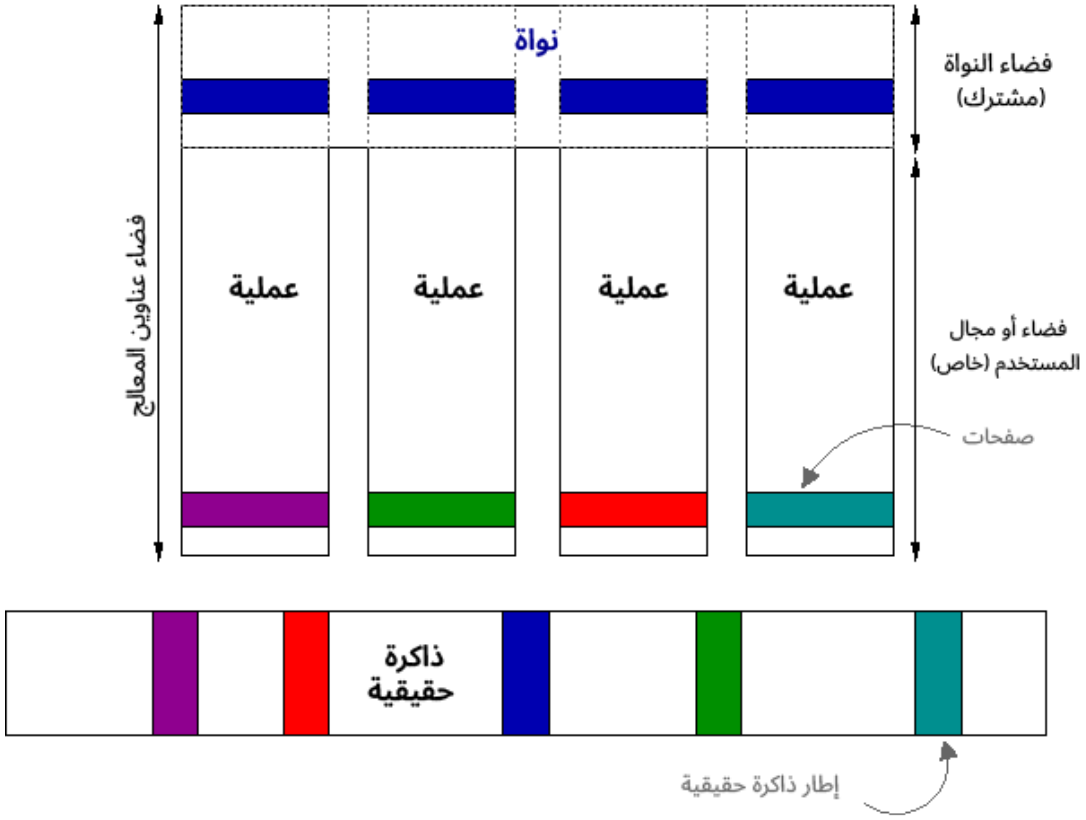
6.7 مواصفات الذاكرة الوهمية في لينكس

تبقى مفاهيم الذاكرة الوهمية الأساسية ثابتة، إلّا أن تفاصيل التقديمات تعتمد بصورة كبيرة على نظام التشغيل والعتاد.

6.7.1 مخطط فضاء العناوين

يقسم لينكس فضاء العناوين المتاح إلى مكون نواة Kernel مشترك وفضاء عناوين خاص بالمستخدم، وهذا يعني أن العناوين الموجودة في منفذ النواة لفضاء العناوين ترتبط مع الذاكرة الحقيقية نفسها لكل عملية، بينما يكون فضاء عناوين المستخدم خاصًا بالعملية، ويوجد في نظام لينكس فضاء النواة المشترك في أعلى فضاء العناوين المتاح. يحدث هذا الانقسام على المعالج x86 الأكثر شيوعًا المكون من 32 بت عند حجم 3 جيجابايتات، وبما أن 32 بت يمكنها ربط 4 جيجابايتات كحد أقصى، مما يؤدي إلى ترك المنطقة العليا بمقدار

1 جيجابايت لتكون منطقة النواة المشتركة. لكن تريد العديد من الأجهزة دعم أكثر من 4 جيجابايتات لكل عملية، حيث يسمح دعم الذاكرة العالي للمعالجات بالوصول إلى 4 جيجابايتات كاملة باستخدام توسّعات خاصة.

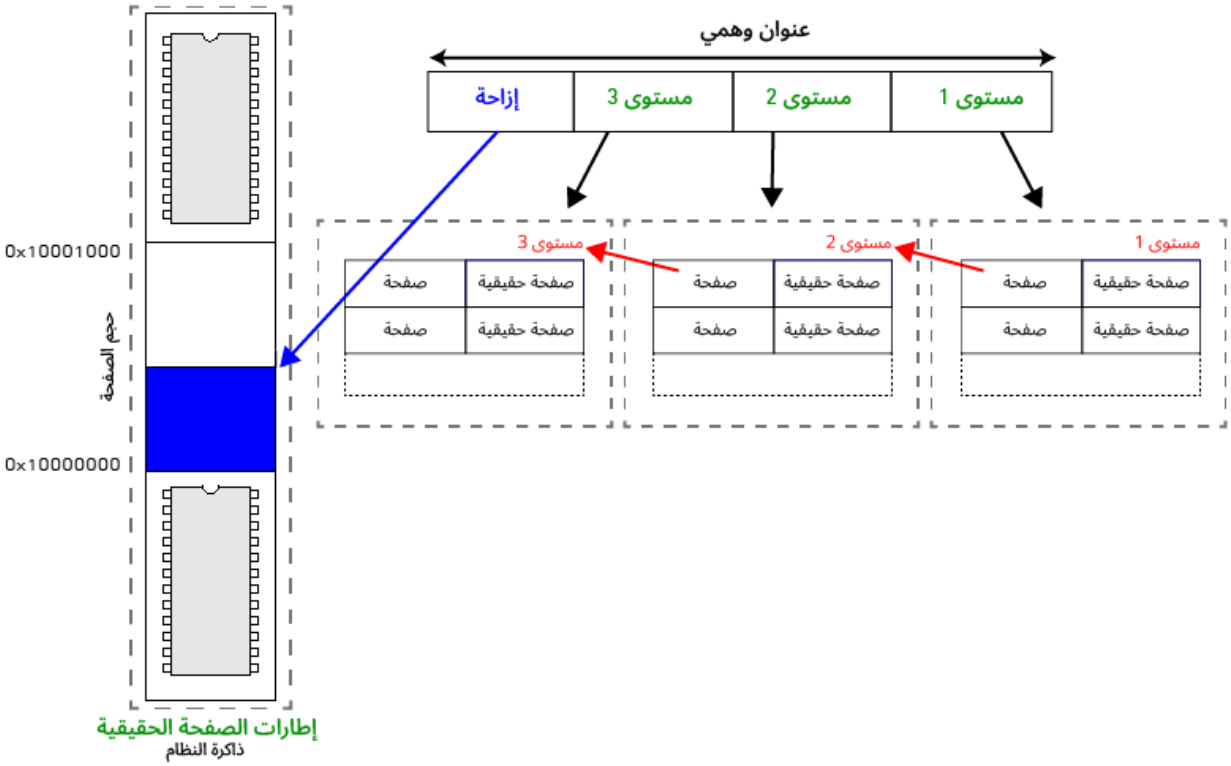


شكل 28: مخطط فضاء العناوين في لينكس

6.7.2 جدول الصفحات المكون من المستويات الثلاثة

هناك العديد من الطرق المختلفة لنظام التشغيل لتنظيم جداول الصفحات، ولكن يختار نظام لينكس استخدام النظام الهرمي. تستخدم جداول الصفحات تسلسلًا هرميًا بعمق ثلاثة مستويات، لذلك يُشار إلى نظام لينكس باسم جدول الصفحات المكوَّن من ثلاثة مستويات. أثبت جدول الصفحات المكون من ثلاثة مستويات أنه اختيار قوي بالرغم من أنه لا يخلو من بعض المساوئ. تختلف تفاصيل تقديم الذاكرة الوهمية بين المعالجات، مما يعني أن جدول الصفحات العام الذي يختاره نظام لينكس يجب أن يكون قابلًا للنقل وعامًا نسبيًا.

لا يُعَد مفهوم مستويات جدول الصفحات الثلاثة أمرًا صعبًا، لأننا نعلم أن العنوان الوهمي يتكون من رقم صفحة وإزاحة في صفحة الذاكرة الحقيقية، إذ يُقسَم العنوان الوهمي إلى مستويات مُرقَّمة في جدول الصفحات المكون من ثلاثة مستويات. يُعَد كل مستوًى جدولَ صفحات بحد ذاته، أي أنه يرتبط مع رقم الصفحة الحقيقية. ترتبط مدخلة المستوى 1 مباشرةً مع الإطار الحقيقي في جدول صفحات مؤلفٍ من مستوًى واحد، بينما يعطي كل مستوًى من المستويات العليا عنوان إطار الذاكرة الحقيقية الذي يحتفظ بجدول صفحات المستويات الدنيا التالي في الإصدار متعدد المستويات من جدول الصفحات.



شكل 29: جدول صفحات لينكس المكون من ثلاثة مستويات

يتضمن المثال السابق الانتقال إلى جدول الصفحات ذي المستوى الأعلى، والعثور على الإطار الحقيقي الذي يحتوي على عنوان المستوى التالي، وقراءة مستويات ذلك الجدول وإيجاد الإطار الحقيقي الذي يوجد فيه جدول صفحات المستويات التالية من جدول الصفحات وما إلى ذلك.

يبدو أن هذا النموذج معقدًا في البداية، ولكن السبب الرئيسي لتنفيذ هذا النموذج هو متطلبات الحجم. تخيل مثلًا عملية ما لها صفحة واحدة مرتبطة بالقرب من نهاية فضاء العناوين الوهمية، حيث قلنا سابقًا أنه يمكن العثور على مدخلة جدول الصفحات بوصفها إزاحةً من مسجل جدول الصفحات الأساسي، لذلك يجب أن يكون جدول الصفحات مصفوفةً متجاورةً في الذاكرة، وبالتالي تتطلب الصفحة القريبة من نهاية فضاء العناوين المصفوفةَ بأكملها والتي يمكن أن تشغَل مساحةً كبيرة، أي العديد والعديد من صفحات الذاكرة الحقيقية.

يكون المستوى الأول في نظام مؤلفٍ من ثلاثة مستويات هو إطار ذاكرة حقيقي واحد فقط، ويرتبط مع المستوى الثاني الذي هو إطار ذاكرة واحد، والذي بدوره يرتبط مع المستوى الثالث، وبالتالي يقلّل نظام المستويات الثلاثة من عدد الصفحات المطلوبة إلى جزء صغير فقط من الصفحات المطلوبة لنظام المستوى الواحد.

هناك عيوب واضحة في هذا النظام، إذ يتطلب البحث عن عنوان واحد مزيدًا من المراجع، ويمكن أن يكون ذلك مكلفًا. يتفهم لينكس أن هذا النظام يمكن ألّا يكون مناسبًا للعديد من أنواع المعالجات المختلفة، لذلك يمكن أن تقلل بعض المعماريات من مستويات جدول الصفحات بسهولة مثل المعمارية x86 الأكثر شيوعًا التي تستخدم نظامًا مؤلفًا من مستويين فقط في التقديم الخاص بها.

6.8 دعم العتاد للذاكرة الوهمية في معمارية الحاسوب

ذكرنا من في السابق ما هي الذاكرة الوهمية والذاكرة الحقيقية في معمارية الحاسوب، ووضحنا بأن العتاد يعمل مع نظام التشغيل لتقديم Implementation الذاكرة الوهمية، وألقينا نظرة على تفاصيل كيفية حدوث ذلك، حيث تعتمد الذاكرة الوهمية بصورة كبيرة على معمارية العتاد، حيث يكون لكل معمارية خواصها الدقيقة، ولكن هناك عدد من العناصر الخاصة بالذاكرة الوهمية في العتاد التي سنستعرضها في الفقرات التالية.

6.8.1 الوضع الحقيقي والوضع الوهمي

تحتوي جميع المعالجات على مفهومٍ ما للعمل في الوضع الحقيقي Physical Mode أو الوضع الوهمي Virtual Mode، إذ يتوقع العتاد في الوضع الحقيقي أن يشير أيّ عنوان إلى عنوانٍ موجود في ذاكرة النظام الفعلية، بينما يعرف العتاد في الوضع الوهمي أنه يجب ترجمة العناوين للعثور على العنوان الحقيقي.

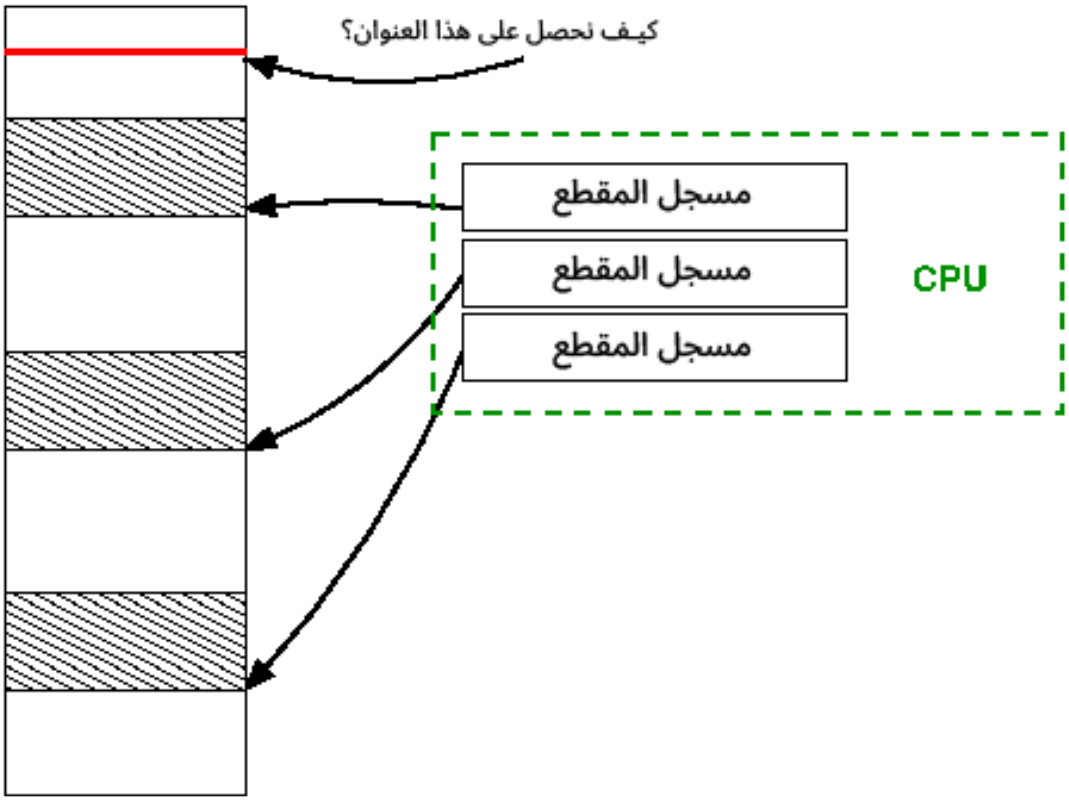
يُشار إلى هذين الوضعين في العديد من المعالجات مثل المعالج إيتانيوم Itanium ببساطة على أنهما الوضع الحقيقي والوضع الوهمي. المعالج x86 الأكثر شيوعًا لديه الكثير من الخاصيات منذ الأيام الماضية التي تسبق الذاكرة الوهمية، حيث يُشار إلى هذين الوضعين على أنهما الوضع الفعلي Real Mode والوضع المحمي Protected Mode. كان أول معالج يطبق الوضع المحمي هو المعالج 386، ولا تزال أحدث المعالجات من عائلة x86 بإمكانها العمل في الوضع الفعلي بالرغم من عدم استخدامه. يطبّق المعالج في الوضع الفعلي شكلًا من أشكال تنظيم الذاكرة يسمى التقطيع Segmentation.

ا. مشاكل التقطيع

كان التقطيع أمرًا مهمًا سابقًا، ولكن قلّلت الذاكرة الوهمية من أهميته، فللتقطيع عيوبه مثل كونه مربكًا للمبرمجين المبتدئين، لذا أُخترِعت أنظمة الذاكرة الوهمية لحل هذه المشاكل إلى حد كبير.

يوجد في التقطيع عدد من المسجّلات التي تحتوي على عنوان يمثل بداية المقطع، والطريقة الوحيدة للوصول إلى عنوان في الذاكرة هي تحديده بوصفه إزاحة من أحد هذه المسجلات. يمكن تحديد حجم المقطع والحد الأقصى للإزاحة الذي يمكنك تحديده من خلال عدد البتات المتاحة للإزاحة من مسجل المقطع الأساسي. الحد الأقصى للإزاحة في المعالج x86 هو 16 بت أو 64 كيلوبايت فقط، مما يؤدي إلى ظهور جميع أشكال الفوضى، حيث إذا أراد شخصٌ ما استخدام عنوان يبعد أكثر من 64 كيلوبايت، فستتحول الأمور من مجرد إزعاج بسيط إلى فشل كامل عندما ينمو حجم الذاكرة إلى عدد من الميجابايتات أو الجيجابايتات.

لنفترض أن أقصى إزاحة هي 32 بت، وبالتالي يمكن الوصول إلى فضاء العناوين بالكامل بوصفه إزاحة من مقطع عند العنوان 0x00000000 وسيكون لديك تخطيط مسطح، ولكن لا تضاهي هذه الحالة جودة الذاكرة الوهمية. السبب الوحيد لكون الإزاحة بمقدار 16 بت هو أن معالجات إنتل Intel الأصلية كانت محدودة بهذا المقدار مع محافظة الشرائح على التوافق مع الإصدارات السابقة.



شكل 30: التقطيع Segmentation

هناك ثلاثة مسجلات مقاطع في الشكل السابق تؤشّر جميعها إلى المقاطع، ويظهر حد الإزاحة الأقصى المُقيَّد بعدد البتات المتاحة في المنطقة المُظلَّلة. إذا أراد البرنامج عنوانًا خارج هذا النطاق، فيجب إعادة ضبط مسجّلات المقاطع الذي سرعان ما يصبح مصدر إزعاج كبير فيما بعد. بينما تسمح ذاكرة البرنامج الوهمية بتحديد العنوان الذي تريده ويطبّق نظام التشغيل والعتاد عملية الترجمة إلى عنوان حقيقي.

6.8.2 مخزن الترجمة المؤقت TLB

يُعَد مخزن الترجمة المؤقت Translation Lookaside Buffer -أو TLB اختصارًا- مكون المعالج الرئيسي المسؤول عن الذاكرة الوهمية، وهو ذاكرةٌ مخبئية لعمليات ترجمة الصفحة الوهمية إلى الإطار الحقيقي في المعالج. يعمل نظام التشغيل والعتاد مع بعضهما البعض لإدارة مخزن TLB أثناء تشغيل النظام.

ا. أخطاء الصفحات

إذا طُلِب عنوان وهمي من العتاد باستخدام تعليمة تحميل `load` مثلًا للحصول على بعض البيانات، فسيبحث المعالج عن ترجمة العنوان الوهمي إلى العنوان الحقيقي في مخزن TLB الخاص به، فإن احتوى على ترجمة صالحة، فيمكن عندئذٍ دمجه مع جزء الإزاحة للانتقال مباشرةً إلى العنوان الحقيقي وإكمال التحميل. لكن إن لم يتمكّن المعالج من العثور على ترجمةٍ في مخزن TLB، فيجب على المعالج إصدار خطأ الصفحة Page Fault الذي يشبه المقاطعة، ويجب أن يعالجه نظام التشغيل، إذ يجب أن يستعرض نظام التشغيل جدول الصفحات للعثور على الترجمة الصحيحة وإدخالها في مخزن TLB عند حدوث خطأ صفحة.

إن لم يتمكّن نظام التشغيل من العثور على ترجمةٍ في جدول الصفحات أو إن تحقّق نظام التشغيل من أذونات الصفحة المطلوبة ولم يُسمَح للعملية بالوصول إليها، فيجب على نظام التشغيل إنهاء العملية. إن رأيتَ مسبقًا خطأ تقطيع أو ما يُسمَّى Segfault، فهذا يعني أن نظام التشغيل ينهي العملية التي تجاوزت حدودها. بينما إن تمكّن نظام التشغيل من العثور على الترجمة وكان مخزن TLB ممتلئًا حاليًا، فيجب إزالة ترجمة قبل إدخال ترجمةٍ أخرى. ليست إزالة الترجمة التي يُحتمَل استخدامها لاحقًا أمرًا منطقيًا، إذ ستتحمل عناء العثور على المدخلة في جداول الصفحات مرة أخرى.

تستخدم مخازن TLB شيئًا يشبه خوارزمية الأقل استخدامًا مؤخرًا Least Recently Used -أو LRU اختصارًا، حيث تُخرَج أقدم ترجمة غير مُستخدمَة لإدخال ترجمة جديدة. يمكن بعد ذلك محاولة الوصول مرة أخرى وسيسير كل شيء على ما يرام، حيث يجب العثور على الترجمة في مخزن TLB وستحدث الترجمة بصورة صحيحة.

العثور على جدول الصفحات

لا بد أنك تساءلت عن كيفية إيجاد نظام التشغيل للذاكرة التي تحتوي على جدول الصفحات page table عندما قلنا أن نظام التشغيل يجد الترجمة في هذا الجدول. يُحتفَظ بقاعدة جدول الصفحات في مسجّل مرتبط بكل عملية، حيث يسمى هذا المسجل بالمسجل الأساسي لجدول الصفحات page-table base-register أو ما شابه ذلك. يمكن تحديد موقع المدخلة الصحيحة بوضع العنوان في هذا المسجل وإضافة رقم الصفحة إليه.

ب. أخطاء أخرى متعلقة بالصفحات

هناك عيبان مهمان آخران يمكن أن يولّدهما مخزن TLB ويساعدان على إدارة الصفحات المتسخة dirty pages أي الصفحات التي جرى الوصول إليها مسبقًا، حيث تحتوي كل صفحة على سمة مُمثَّلة ببتٍ واحد تحدّد إذا جرى الوصول إلى الصفحة أي أنها أصبحت صفحة متسخة.

يمكن تمييز الصفحة على أنها صفحة جرى الوصول إليها مسبقًا عند تحميل ترجمة الصفحة مبدئيًا في مخزن TLB. إذا حمّلتَ الصفحة دون وصول معلَّق، فيمكن تسمية ذلك بالتأمّل Speculation مثل تطبيق شيءٍ ما مع التوقّع بأنه سيؤتي ثماره، حيث إذا قرأت الشيفرة البرمجية من الذاكرة خطيًا مثلًا، فيمكن أن يؤدي وضع ترجمة الصفحة التالية في مخزن TLB إلى توفير الوقت وتحسين الأداء.

يعمل نظام التشغيل على تصفح جميع الصفحات دوريًا ويصفّر بت الوصول ليميز الصفحات التي تكون قيد الاستخدام حاليًا من غيرها. تكون الصفحات التي لم يُعاد ضبط بت الوصول الخاص بها (بعد تصفيره) هي أفضل المرشحين للإزالة لأنها لم تُستخدَم لفترة أطول عندما تمتلئ ذاكرة النظام ويحين الوقت لنظام التشغيل لاختيار الصفحات التي ستُبدَّل إلى القرص الصلب.

الصفحة المتسخة هي الصفحة التي تحتوي على بياناتٍ مكتوبةٍ عليها، وبالتالي لا تتطابق مع أيّ بيانات موجودة فعليًا على القرص الصلب. إذا بُدِّلت الصفحة من ثم كتبت فيها عملية مثلًا، فيجب تحديث نسختها

الموجودة على القرص الصلب قبل تبديلها. لا تُجرَى أيّ تغيـيرات على الصـفحة النظيفـة، لـذلك لا حاجـة لنسـخ الصفحة مرة أخرى إلى القرص الصلب.

يتشابه هذان النوعان من الصفحات من حيث أنهمـا يسـاعدان نظـام التشـغيل في إدارة الصـفحات، حيث تحتوي الصفحة على بتين إضـافيين همـا: البت المتسـخ Dirty Bit وبت الوصـول Accessed Bit، إذ يُضـبَط هذان البتان عند وضع الصفحة في مخزن TLB للإشارة إلى أن وحدة المعالجة المركزية يجب أن تصدر خطأ.

يطبّق العتادُ عملية الترجمة المعتادة عندما تحاول إحدى العمليات الرجـوع إلى الـذاكرة، ولكنـه يجـري أيضًـا فحصًا إضافيًا للتأكد من عدم ضبط راية الوصول. إذا كان الأمر كذلك، فسيؤدي ذلك إلى حـدوث خطـأ في نظـام التشغيل الذي يجب أن يضبط البت ويسمح للعملية بالاستمرار. إذا اكتشف العتاد أنـه يكتب في صـفحةٍ يكـون بتها المتسخ غير مضبوط، فسيؤدي ذلك إلى حدوث خطأ في نظام التشغيل لتمييز الصفحة بوصفها متسخة.

6.8.3 إدارة مخزن TLB

يمكننا القول أن مخزن TLB يستخدمه العتاد ولكن تديره البرمجيات، فالأمر متروك لنظام التشغيل لتحميل المدخلات الصحيحة إلى مخزن TLB ولإزالة المدخلات القديمة منه.

ا. تفريغ مخزن TLB

تُسمَّى عملية إزالة المدخلات من مخزن TLB باسم التفريغ Flushing. يُعَد تحديث مخزن TLB جـزءًا مهمًـا من الحفاظ على فضاءات عناوين العمليات منفصلة، لأن كل عملية يمكن أن تسـتخدم العنـوان الـوهمي نفسـه دون تحديث مخزن TLB، وهذا يعني أن العملية يمكن أن تكتب فوق ذاكـرة العمليـات الأخـرى، بينمـا تريـد في حالة الخيوط Threads مشـاركة فضـاء العنـاوين، وبالتـالي لا يُفـرَّ غ مخـزن TLB عنـد التبـديل بين الخيـوط في العملية نفسها.

يُفرَّ غ مخزن TLB بالكامل في بعض المعالجات في كل مرة يوجد فيها تبديل سياق، ويمكن أن يكـون ذلـك مرهقًا للغاية، لأنه يعني أن العملية الجديدة يجب أن تمر عبر المراحل كاملةً من أخذ خطأ الصفحة ثم العثـور على الصفحة في جداول الصفحات وإدخال الترجمة.

بينما تطبّق المعالجات الأخـرى معـرّف فضـاء عنـاوين إضـافي Address Space ID -أو ASID اختصـارًا- يُضاف إلى كل ترجمة في مخزن TLB لجعلها فريدة، وبالتالي يحصل كل فضاء عناوين -أو كل عمليـة حيث تريـد الخيوط مشاركة فضاء العناوين نفسه- على معرّفها الخاص الذي يُخزَّن مـع الترجمـات في مخـزن TLB، أي ليس هناك داعٍ لتفريغ مخزن TLB عند تبديل السياق، لأن العملية التالية سيكون لها معرّف فضـاء عنـاوين مختلـف، وسيختلف معرّف فضاء العناوين وسـتختلف الترجمـة إلى الصـفحة الحقيقيـة حـتى إن طلبت العمليـة العنـوان الوهمي نفسه. يقلل هذا النظام من عمليـة التفريـغ ويزيـد من أداء النظـام، ولكنـه يتطلب مزيـدًا من عتـاد TLB ليحتفظ ببتات معرّف ASID.

يمكن تنفيذ ذلك من خلال وجود مسجل إضافي بوصفه جزءًا من حالة العملية التي تتضمن معرّف ASID. ينظر مخزن TLB إلى هذا المسجّل عند ترجمة الصفحة الوهمية إلى الصفحة الحقيقية، وسيطابق فقط المدخلات التي لها معرّف ASID الخاص بالعملية التي تكون قيد التشغيل حاليًا. يحدّد حجم هذا المسجل رقم معرّفات ASID المتاحة وبالتالي له تأثير على الأداء.

ب. مخزن TLB المحمل برمجيا وعتاديًا

يُعَد التحكم في مخزن TLB من اختصاص نظام التشغيل، ولكنها ليست القصة كاملة، إذ تشرح العملية الموضَّحة في فقرة "أخطاء الصفحات" خطأ الصفحة الذي يُرفَع إلى نظام التشغيل، ويمر على جدول الصفحات للعثور على ترجمة الصفحة الوهمية إلى الحقيقية وتثبيتها في مخزن TLB. يمكن أن يسمَّى ذلك بمخزن TLB المُحمَّل برمجيًا Software-loaded TLB، وهناك بديل آخر هو مخزن TLB المُحمَّل عتاديًا Hardware-loaded TLB.

تحدد معمارية المعالج تخطيطًا معينًا لمعلومات جدول الصفحات في مخزن TLB المُحمَّل عتاديًا، ويجب اتباعها حتى ترجمة العنوان الوهمي. سيمر المعالج تلقائيًا على جداول الصفحات لتحميل مدخلة الترجمة الصحيحة استجابةً للوصول إلى عنوان وهمي غير موجود في مخزن TLB، وسيرفع المعالج استثناءً ليعالج نظام التشغيل مدخلة الترجمة غير الموجودة في مخزن TLB.

يوفّر التنفيذ المُقدَّم للمرور على جدول الصفحات في العتاد المتخصص مزايا السرعة عند البحث عن الترجمات، ولكنه يزيل المرونة عن مطبّقي أنظمة التشغيل الذين يرغبون في تنفيذ مخططات بديلة لجداول الصفحات.

يمكن تصنيف جميع المعماريات على نطاق واسع ضمن هاتين المنهجيتين السابقتين، وسنطّلع تاليًا على بعض المعماريات الشائعة ودعم الذاكرة الوهمية.

6.9 دعم العتاد للذاكرة الوهمية

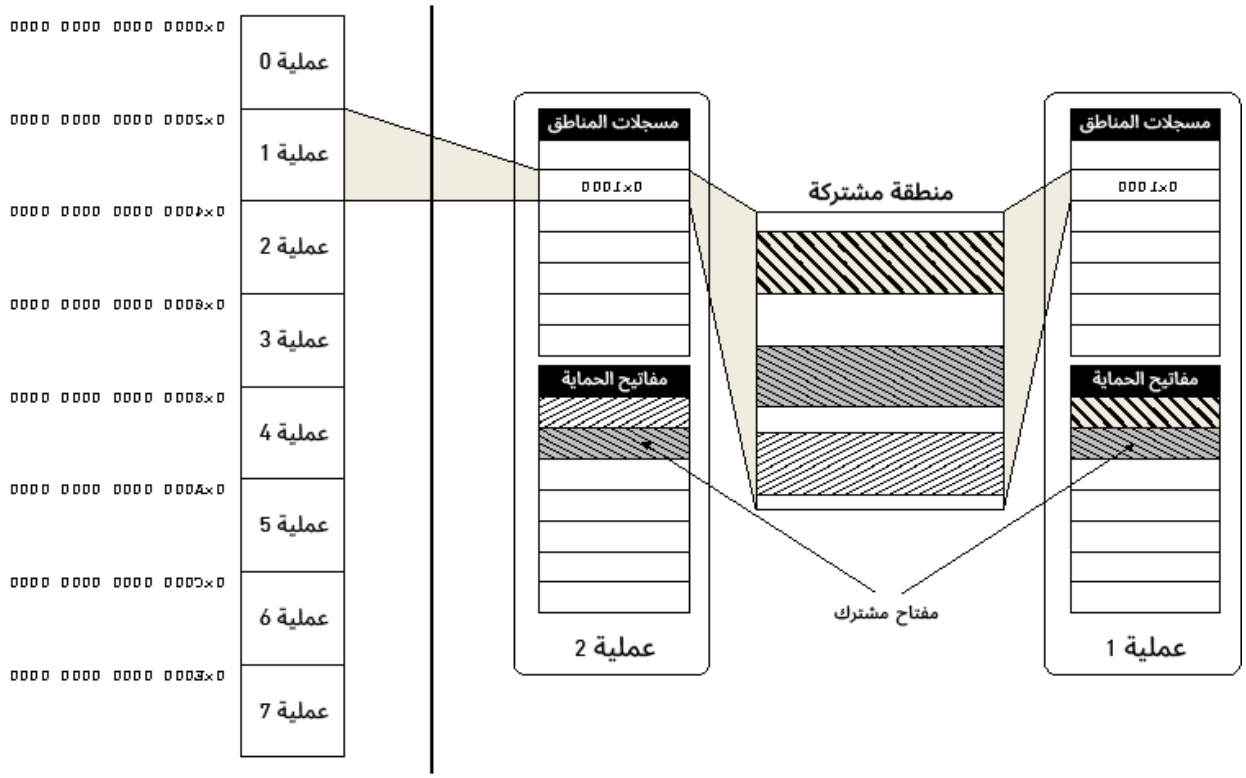
يوفّر عتاد المعالج جدول بحث يربط العناوين الوهمية بالعناوين الحقيقية، حيث تحدد معماريات المعالجات طرقًا مختلفة لإدارة مخزن TLB مع مزايا وعيوب مختلفة. يشار إلى جزء المعالج الذي يتعامل مع الذاكرة الوهمية باسم وحدة إدارة الذاكرة Memory Management Unit أو MMU اختصارًا.

6.9.1 معالج إيتانيوم

توفر وحدة MMU في معالج إيتانيوم ميزات متعددة لنظام التشغيل للتعامل مع الذاكرة الوهمية والتي سنوضحها فيما يلي.

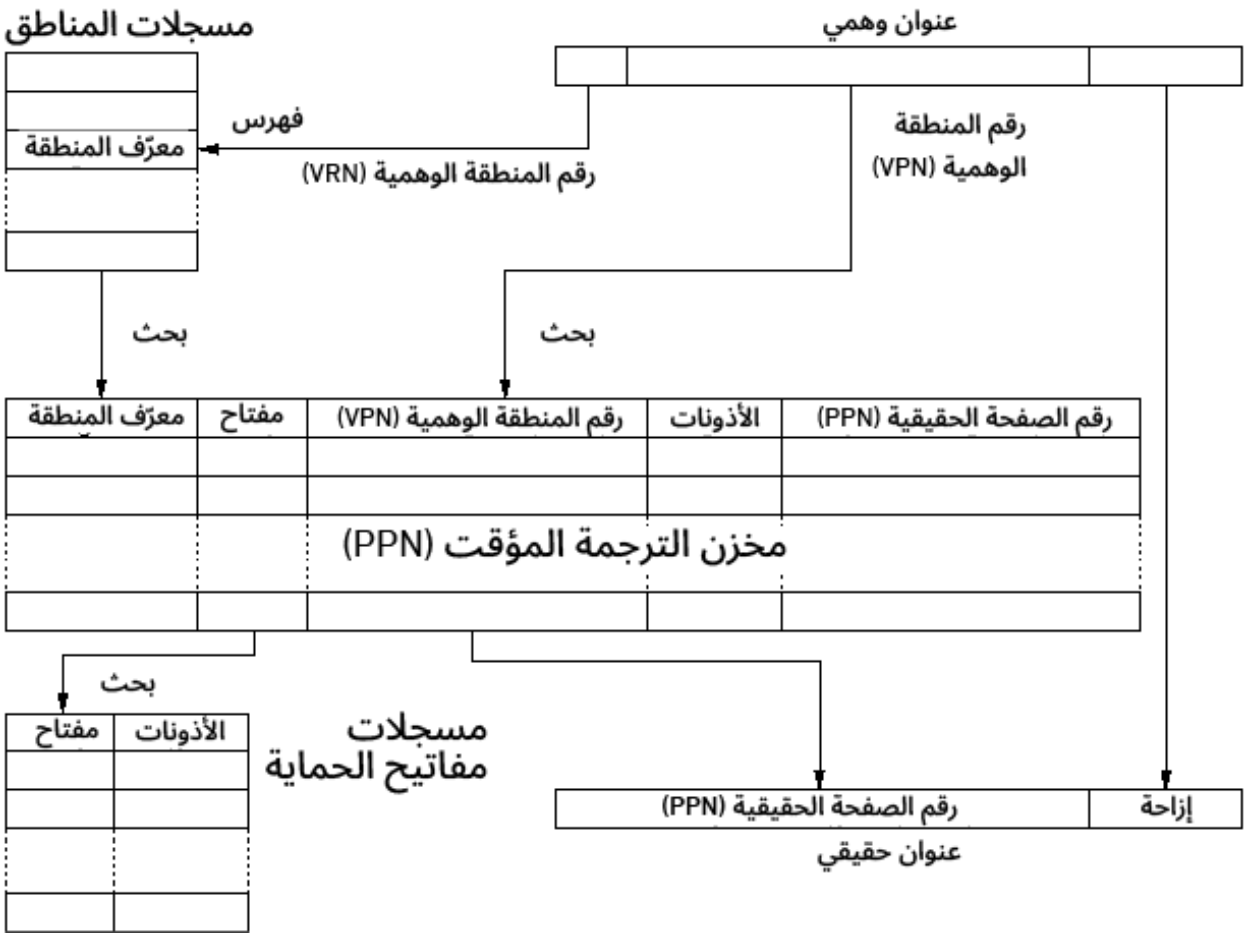
ا. فضاءات العناوين Address spaces

شرحنا سابقًا مفهوم معرّف فضاء العناوين لتقليل تكلفة تفريغ مخزن TLB عند تبديل السياق، ولكن يستخدم المبرمجون في أغلب الأحيان الخيوط Threads للسماح لسياقات التنفيذ بمشاركة فضاء العناوين، حيث تحتوي جميع الخيوط على معرّف ASID نفسه، وبالتالي يتشاركون بمدخلات مخزن TLB، مما يؤدي إلى زيادة الأداء. لكن يمنع معرّف ASID واحد مخزن TLB من فرض الحماية، حيث تصبح المشاركة متمثلة بنهج "الكل أو لا شيء"، إذ يجب أن تتخلى الخيوط عن الحماية من بعضها البعض لمشاركة بعض البايتات.



شكل 31: رسم توضيحي للمناطق ومفاتيح الحماية في معالج إيتانيوم

تدرس وحدة MMU في معالج إيتانيوم المشاكل السابقة وتوفر القدرة على مشاركة فضاء العناوين ومدخلات الترجمة بدقة أقل بكثير مع الحفاظ على الحماية داخل العتاد. يقسم معالج إيتانيوم فضاء العناوين المؤلف من 64 بت إلى 8 مناطق كما هو موضَّح في الشكل السابق. تحتوي كل عملية على ثمانية مسجلات مناطق بحجم 24 بت بوصفها جزءًا من حالتها، ويحتوي كل منها على معرّف منطقة Region ID -أو RID اختصارًا- لكل منطقة من المناطق الثمانية الخاصة بفضاء عناوين العملية. تُوسَم ترجمات مخزن TLB بمعرّف RID، وبالتالي لن تتطابق إلا إذا احتوت العملية على معرّف RID نفسه كما هو موضَّح في الشكل التالي:



شكل 32: رسم توضيحي لترجمة مخزن TLB في معالج إيتانيوم

لا تُحتسَب البتات الثلاثة الأولى (بتات المنطقة) في ترجمة العنوان الوهمي، لذلك إذا كانت هناك عمليتان تشتركان في معرّف RID أي تحتفظان بالقيمة نفسها في أحد مسجلات المنطقة الخاصة بهما، فسيكون لديهما اسم بديل لتلك المنطقة. إذا احتوت العملية A على معرّف RID قيمته 0x100 في مسجل المنطقة 3 واحتوت العملية B معرّف RID نفسه الذي قيمته 0x100 في مسجّل المنطقة 5، فستُسمَّى المنطقة 3 من العملية A باسم بديل هو process-B, region 5. تعني هذه المشاركة المحدودة أن كلتا العمليتين تتلقيان فوائد مدخلات مخزن TLB المشتركة دون الحاجة إلى منح إذن الوصول إلى كامل فضاء العناوين.

مفاتيح الحماية Protection Keys

تُوسَم كل مدخلة من مخزن TLB في معالج إيتانيوم بمفتاح حماية للسماح بمشاركة أكثر دقة. تحتوي كل عملية على عدد إضافي من مسجلات مفاتيح الحماية يحدده نظام التشغيل.

تُوسَم كل صفحة بمفتاح فريد ويمنح نظام التشغيل العمليات المسموح بها للوصول إلى الصفحات التي تستخدم هذا المفتاح عند مشاركة سلسلة من الصفحات مثل شيفرة برمجية لمكتبة نظام مشتركة. يفحص مخزن TLB المفتاح المرتبط بمدخلة الترجمة مقابل المفاتيح التي تحتفظ بها العملية في مسجلات مفتاح

الحماية الخاصة بها عند الإشارة إلى صفحة ما، مما يسمح بالوصول إليها في حالة وجود المفتاح أو يؤدي إلى رفع خطأ حماية لنظام التشغيل.

يمكن للمفتاح فرض الأذونات أيضًا، إذ يمكن أن تحتوي إحدى العمليات مثلًا على مفتاح يمنح أذونات الكتابة ويمكن أن تحتوي عملية أخرى على مفتاح للقراءة فقط، مما يسمح بمشاركة مدخلات الترجمة بدقة وفي نطاق أوسع بكثير وصولًا إلى مستوى الصفحة الواحدة، ويؤدي ذلك إلى تحسينات محتملة كبيرة في أداء مخزن TLB.

ب. أداة إيتانيوم العتادية للمرور على جدول الصفحات Page-Table

يؤدي تبديل السياق إلى نظام التشغيل عند حل خطأ في مخزن TLB إلى إضافة عبء كبير إلى مسار معالجة الخطأ، إذ يواجه معالج إيتانيوم ذلك العبء من خلال السماح بخيار استخدام العتاد المُدمَج لقراءة جدول الصفحات وتحميل ترجمات الصفحة الوهمية إلى الصفحة الحقيقية في مخزن TLB تلقائيًا. تتجنب أداة المرور على جدول الصفحات العتادية Hardware Page-table Walker -أو HPW اختصارًا- عمليات الانتقال المكلفة إلى نظام التشغيل، ولكنه يتطلب أن تكون الترجمات بصيغة ثابتة ومناسبة للعتاد لتفهمه.

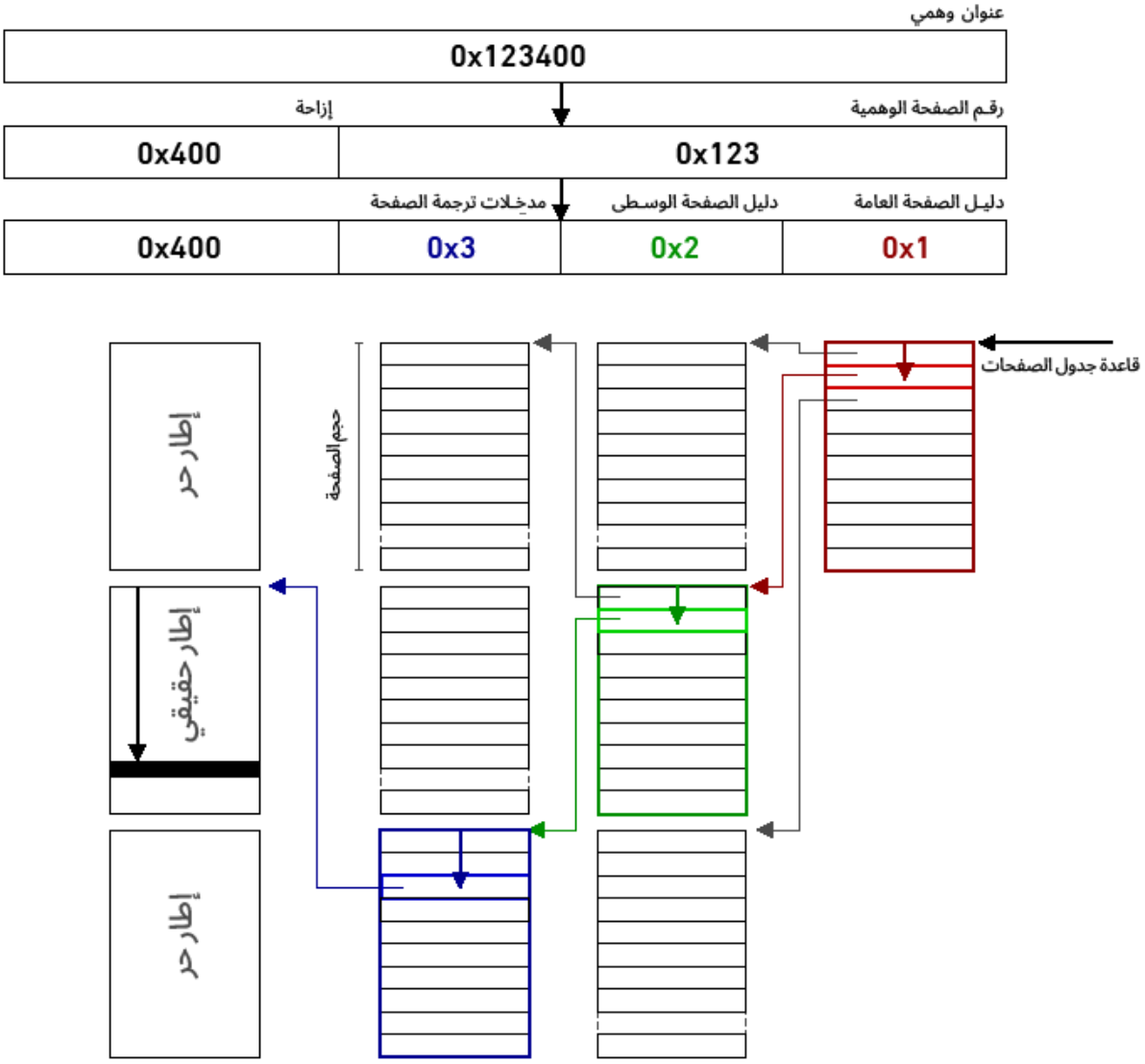
يُشار إلى أداة HPW الخاصة بمعالج إيتانيوم في توثيق إنتل على أنها أداة مُعمَّاة وهميًا للمرور على جدول الصفحات Virtually Hashed Page-table Walker أو VHPT walker اختصارًا. يمنح معالج إيتانيوم المطورين خيارين من تقديمات HPW الحصرية تبادليًا، إذ يعتمد أحدهما على جدول الصفحات الخطي الوهمي ويعتمد الآخر على جدول التعمية Hash Table.

تجدر الإشارة إلى أنه يمكن العمل بدون أداة عتادية للمرور على جدول الصفحات، حيث يحل نظام التشغيل كل خطأ TLB ويصبح المعالج معمارية محمَّلة برمجيًا، ولكن يُعَد تأثير تعطيل HPW على الأداء كبيرًا جدًا دون الحصول على أيّ فائدة.

جدول الصفحات الخطي الوهمي Virtual Linear Page-Table

يشار إلى تقديم جدول الصفحات الخطي الوهمي في التوثيقات على أنه الصيغة القصيرة لجدول الصفحات المُعمَّاة وهميًا Short Format Virtually Hashed Page-table -أو SF-VHPT اختصارًا، وهو نموذج HPW الافتراضي الذي يستخدمه لينكس على معالج إيتانيوم.

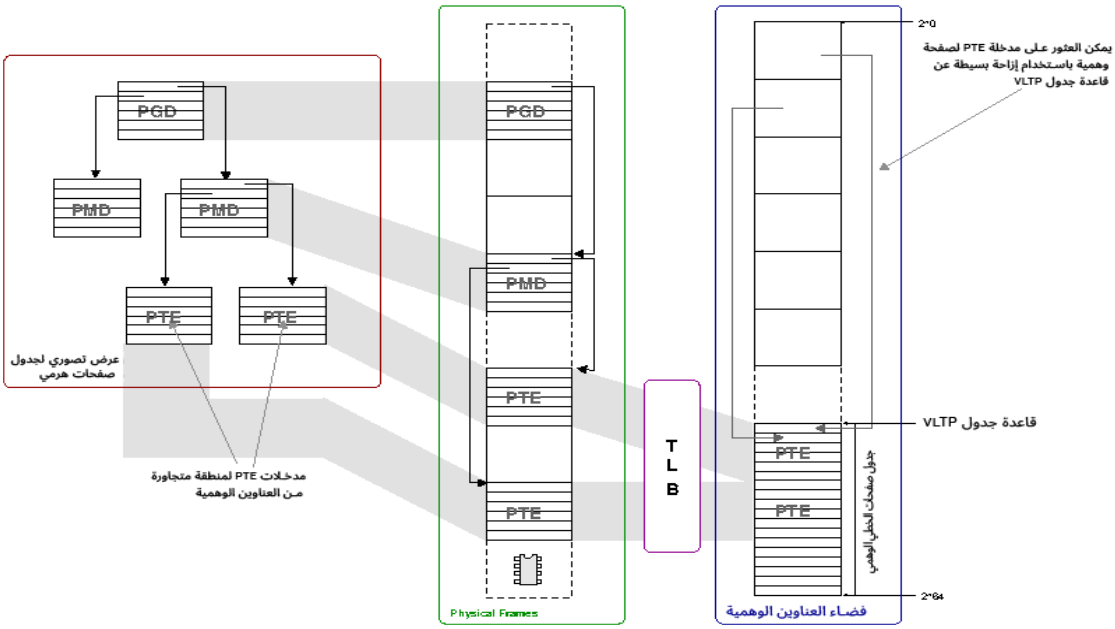
الحل المعتاد هو استخدام جدول صفحات متعدد المستويات أو هرمي، حيث تُستخدَم البتات التي تتكون من رقم الصفحة الوهمية بوصفها فهرسًا إلى مستويات وسيطة من جدول الصفحات. لا توجد مناطق فضاء عناوين وهمية فارغة في جدول الصفحات الهرمي. تُهدَر مساحة صغيرة نسبيًا في الحِمل الإضافي بالنسبة للحالة الواقعية لفضاء العناوين المُجمَّعة Clustered بإحكام والمملوءة بصورة ضئيلة بالموازنة مع جدول الصفحات الخطي، ولكن العيب الرئيسي هو مراجع الذاكرة المتعددة المطلوبة للبحث.



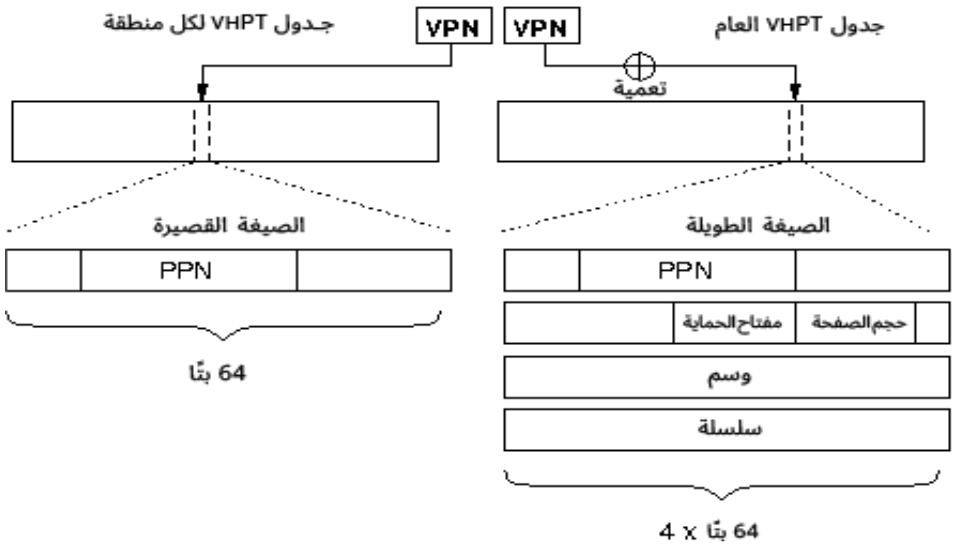
شكل 33: رسم توضيحي لجدول صفحات هرمي

يأخذ الجدول الخطي الذي حجمه 512 جيبي بايت GiB مع فضاء عناوين 64 بت ما مقداره 0.003% فقط من 16 إكسابايت المتاحة، وبالتالي يمكن إنشاء جدول صفحات خطي وهمي Virtual Linear Page-table -أو VLPT اختصارًا- في منطقة متجاورة من فضاء العناوين الوهمية.

يستخدم العتاد عند حدوث خطأ في مخزن TLB رقم الصفحة الوهمية للإزاحة عن قاعدة جدول الصفحات تمامًا كما هو الحال بالنسبة لجدول صفحات خطي حقيقي. إذا كانت هذه المدخلة صحيحة، فستُقرَأ الترجمة وتُدرَج مباشرةً في مخزن TLB، ولكن يكون عنوان مدخلة الترجمة في حد ذاتها عنوانًا وهميًا باستخدام جدول VLPT، وبالتالي هناك احتمال أن تكون الصفحة الوهمية التي توجد بها غير موجودة في مخزن TLB، وسيُرفَع خطأ متداخل Nested Fault إلى نظام التشغيل في هذه الحالة. يجب على البرمجيات بعد ذلك تصحيح هذا الخطأ عن طريق ربط الصفحة التي تحتوي على مدخلة الترجمة مع جدول VLPT.



شكل 34: تقديم جدول VHPT بصيغة قصيرة في معالج إيتانيوم



شكل 35: صيغ مدخلة PTE في معالج إيتانيوم

يمكن جعل هذه العملية مباشرةً إذا احتفظ نظام التشغيل بجدول صفحات هرمي، حيث تحتوي الصفحة التي تمثل ورقة من جدول صفحات هرمي على مدخلات ترجمة لمنطقة متجاورة وهميًا من العناوين، وبالتالي يمكن ربطها باستخدام مخزن TLB لإنشاء جدول VLPT كما هو موضح في الشكل السابق.

تحدث ميزة جدول VLPT الرئيسية عندما يطلب أحد التطبيقات وصولًا متكررًا أو متواصلًا إلى الذاكرة. ضع في حساباتك أن الخطأ الأول في عملية المرور على الذاكرة المتجاورة وهميًا سيؤدي إلى ربط صفحة مليئة

بمدخلات الترجمة مع جدول الصفحات الخطي الوهمي. سيتطلب الوصول اللاحق إلى الصفحة الوهمية التالية تحميلَ مدخلة الترجمة التالية في مخزن TLB، والتي تتوفر الآن في جدول VLPT، وبالتالي تُحمَّل بسرعة كبيرة دون استدعاء نظام التشغيل. سيكون ذلك ميزةً عند الاستفادة من تكلفة الخطأ المتداخل الأولي على عمليات مرور HPW الناجحة اللاحقة.

العيب الرئيسي هو أن جدول VLPT يتطلب الآن مدخلات مخزن TLB، مما يؤدي إلى زيادة الضغط عليه. يتطلب كل فضاء عناوين جدول صفحاتٍ خاص به، لذلك تصبح التكلفة أكبر كلما أصبح النظام أكثر نشاطًا، ولكن يجب أن تكون أيّ زيادة في أخطاء الوصول إلى مخزن TLB أكبر من الفائدة الحاصلة عند انخفاض تكاليف إعادة الملء من أداة المرور العتادية الفعالة. لاحظ أن الحالة السيئة يمكن أن تتخطى مدخلات بمقدارٍ يساوي نتيجة قسمة حجم الصفحة `page_size` على حجم الترجمة `translation_size`، مما يتسبب في حدوث أخطاء متداخلة ومتكررة، ولكنه يُعَد نمط وصول غير مُحتمَل.

تتوقع أداة المرور العتادية hardware walker أن تكون مدخلات الترجمة بصيغة معينة كما هو موضح على يسار الشكل السابق، حيث يتطلب جدول VLPT ترجمات بصيغة قصيرة مؤلَّفة من 8 بايتات. إذا استخدم نظام التشغيل جدول الصفحات الخاص به بوصفه دعمًا لجدول VLPT كما في الشكل "تقديم جدول VHPT بصيغة قصيرة في معالج إيتانيوم"، فيجب أن تستخدم هذه صيغة الترجمة القصيرة. تتجاهل المعمارية عددًا محدودًا من البتات في هذه الصيغة وبالتالي تكون متاحة لتستخدمها البرمجيات مع عدم احتمال حدوث تعديلات كبيرة.

يعتمد جدول الصفحات الخطي linear page-table على فكرة حجم الصفحة الثابت، ويُعَد دعم أحجام الصفحات المتعددة مشكلةً لأنه يعني أن ترجمة صفحة وهمية معينة لم تَعُد عند إزاحة ثابتة، ولكن يمكن حل هذه المشكلة من خلال احتواء كل منطقة من المناطق الثمانية في فضاء العناوين -كما هو موضّح في الشكل "رسم توضيحي للمناطق ومفاتيح الحماية في معالج إيتانيوم"- على جدول VLPT منفصل يربط عناوين تلك المنطقة فقط. يمكن إعطاء حجم الصفحة الوهمية لكل منطقة، حيث تُخصَّص منطقة واحدة لصفحات أكبر (باستخدام مخزن HugeTLB في لينكس) ولكن لا يمكن استخدام أحجام صفحات متعددة ضمن المنطقة الواحدة.

جدول التعمية الوهمي Virtual Hash Table

يمكن أن يكون استخدام مدخلات مخزن TLB لمحاولة تقليل تكاليف إعادة تعبئته -كما هو الحال مع جدول SF-VHPT- مقايضة فعّالة أو يمكن ألّا يكون كذلك. يطبّق معالج إيتانيوم جدول صفحات مُعمَّى hashed page-table مع إمكانية خفض تكاليف مخزن TLB، حيث يعمّي المعالج في هذا المخطط عنوانًا وهميًا للعثور على إزاحة في جدول مجاور.

يُعَد جدول الصفحات الخطي الذي ناقشناه سابقًا جدول صفحات مُعمَّى باستخدام تعمية Hash مثالية لن ينتج عنها تضاربٌ أبدًا، ولكن يتطلب ذلك مقايضة غير عملية لمناطق ضخمة من الذاكرة الحقيقية المتجاورة. يزيد تقييد متطلبات الذاكرة لجدول الصفحات من احتمال حدوث تضاربات عند تعمية عنوانين وهميين إلى

الإزاحة نفسها. تتطلب الترجمات المتضاربة مؤشر سلسلة Chain Pointer لإنشاء قائمة مترابطة من المدخلات البديلة الممكنة. يتطلب تمييز المدخلة الصحيحة في القائمة المترابطة وسمًا Tag مشتقًا من العنوان الوهمي الوارد.

تؤدي المعلومات الإضافية المطلوبة لكل مدخلة ترجمة إلى ظهور اسم بديل بصيغة جدول VHPT الطويلة - أو LF-VHPT اختصارًا. تنمو مدخلات الترجمة إلى 32 بايت كما هو موضح على الجانب الأيمن من الشكل "صيغ مدخلة PTE في معالج إيتانيوم".

الميزة الرئيسية لهذه الطريقة هي أن جدول التعمية العام يمكن تثبيته باستخدام مدخلة TLB واحدة. تشترك جميع العمليات في الجدول، لذا يجب أن ينمو حجمه بطريقة أفضل من صيغة SF-VHPT، إذ تتطلب كل عملية أعدادًا متزايدة من مدخلات صفحات جدول VLPT في مخزن TLB. لكن تكون المدخلات الأكبر أقل ملاءمة للذاكرة المخبئية، إذ يمكننا ملاءمة أربعة مدخلات ذات صيغة قصيرة بحجم 8 بايتات مع كل مدخلة ذات صيغة طويلة بحجم 32 بايت. يمكن أن تساعد الذواكر المخبئية الكبيرة جدًا الموجودة على معالج إيتانيوم في تخفيف هذا التأثير.

تتمثل إحدى مزايا صيغة SF-VHPT في أن نظام التشغيل يمكنه الاحتفاظ بالترجمات في جدول صفحات هرمي، ويمكنه ربط الصفحات الورقية في هذه البنية الهرمية مباشرةً مع جدول VLPT مع الاحتفاظ بصيغة الترجمة العتادية. بينما يجب على نظام التشغيل باستخدام صيغة LF-VHPT إما استخدام جدول التعمية بوصفه مصدرًا أساسيًا لمدخلات الترجمة أو الاحتفاظ بجدول التعمية بوصفه ذاكرة مخبئية لمعلومات الترجمة الخاصة به. يُعَد الاحتفاظ بجدول التعمية بصيغة LF-VHPT بوصفه ذاكرة مخبئية دون المستوى الأمثل إلى حد ما بسبب زيادة الحِمل في مسارات الأخطاء الأساسية في الوقت المناسب، ولكن تُكتسَب الفوائد من الجدول الذي يتطلب مدخلة واحدة فقط في مخزن TLB.

دورة علوم الحاسوب

مميزات الدورة

- بناء معرض أعمال قوي بمشاريع حقيقية
- وصول مدى الحياة لمحتويات الدورة
- تحديثات مستمرة على الدورة مجانًا
- شهادة معتمدة من أكاديمية حسوب
- إرشادات من المدربين على مدار الساعة
- من الصفر دون الحاجة لخبرة مسبقة

اشترك الآن

7. سلسلة الأدوات Toolchain

ناقشنا حتى الآن كيفية تحميل البرنامج في تحميل البرنامج في الذاكرة الوهمية، وسوف نبدأ في هذا الفصل بالتعرف على عملية يتعقّبها نظام التشغيل ويتفاعل معها باستخدام استدعاءات النظام وهي عملية التصريف Compiling. كما سنتعرّف في هذا الفصل على الخطوات الثلاث المتبعة لإنشاء ملف قابل للتنفيذ، ولكننا سنبدأ أولًا بالتعرّف على أهم الفروقات بين البرامج المُصرَّفة Compiled Programs والبرامج المُفسَّرة Interpreted Programs.

7.1 البرامج المصرفة Compiled والبرامج المفسرة Interpreted

يجب أن يكون البرنامج الذي يمكن تحميله مباشرةً في الذاكرة بصيغة ثنائية binary format، حيث تُسمَّى عملية تحويل الشيفرة البرمجية المكتوبة بلغةٍ مثل لغة C إلى ملف ثنائي جاهز للتنفيذ بعملية التصريف التي تُطبَّق باستخدام مصرِّف Compiler، والمثال الأكثر انتشارًا هو المصرِّف gcc.

للبرامج المُصرَّفة بعض العيوب في تطوير البرمجيات الحديثة، إذ يجب استدعاء المصرِّف لإعادة إنشاء الملف القابل للتنفيذ في كل مرة يُجري فيها المطور تعديلًا -لو بسطيًا- وفي المقابل يمكن منطقيًا وبناءً على ذلك تصميم برنامجٍ مُصرَّف يمكنه قراءة برنامج آخر وتنفيذ شيفرته البرمجية سطرًا سطرًا، ونسمي هذا النوع من البرامج المُصرَّفة بالبرامج المُفسَّرة Interpreter لأنها تفسّر كل سطر من ملف الدخل وتنفّذه بوصفه شيفرة برمجية، بحيث لا تكون هناك حاجة لتصريف البرنامج وستظهر أي تعديلات جديدة مضافة في المرة التالية التي يشغّل فيها المفسّر الشيفرة البرمجية.

تعمل البرامج المفسَّرة عادةً بصورة أبطأ من نظيرتها المُصرَّفة، حيث يمكن مصادفة حِمل البرنامج في قراءة وتفسير الشيفرة البرمجية مرةً واحدة فقط في البرامج المُصرَّفة، بينما يصادف البرنامج المفسَّر هذا الحِمل في كل مرة يُشغَّل فيها. لكن تمتلك اللغات المفسَّرة العديد من الجوانب الإيجابية، حيث تعمل العديد من اللغات

المفسرة فعليًا في آلة افتراضية virtual machine مُجرَّدة من العتاد الأساسي. تُعَد لغة البرمجة بايثون Python ولغة Perl 6 من اللغات التي تستخدم آلةً افتراضية تفسّر الشيفرة البرمجية.

7.1.1 الآلات الافتراضية Virtual Machines

يعتمد البرنامج المُصرَّف كليًا على عتاد الآلة التي يُصرَّف من أجلها، إذ يجب أن يكون هذا العتاد قادرًا على نسخ البرنامج في الذاكرة وتنفيذه، حيث تُعَد الآلة الافتراضية Virtual Machine تجريدًا برمجيًا للعتاد.

تستخدم لغة جافا Java مثلًا نهجًا هجينًا يجمع بين التصريف والتفسير، فهي لغة مُصرَّفة جزئيًا ومُفسَّرة جزئيًا. تُصرَّف شيفرة جافا في برنامج يعمل ضمن آلة جافا الافتراضية Java Virtual Machine أو يشار إليها اختصارًا، وبالتالي يمكن تشغيل البرنامج المُصرَّف على أيّ عتاد يحتوي على آلة JVM خاصة به، أي يمكنك أن تكتب شيفرتك البرمجية مرة واحدة وتشغّلها في أيّ مكان.

7.2 بناء ملف قابل للتنفيذ

هناك ثلاث خطوات منفصلة تتضمنها عملية إنشاء ملف قابل للتنفيذ عندما نتحدث عن المُصرِّفات وهذه الخطوات هي:

1. التصريف Compiling
2. التجميع Assembling
3. الربط Linking

تسمَّى جميع المكونات المتضمنة في هذه العملية بسلسلة الأدوات Toolchain، إذ تكون هذه الأدوات على شكل سلسلة بحيث يكون خرج إحداها دخلًا للأخرى حتى الوصول إلى الخرج النهائي. يأخذ كل رابط في السلسلة الشيفرةَ البرمجية تدريجيًا بحيث تكون أقرب إلى كونها شيفرة برمجية ثنائية مناسبةً للتنفيذ.

7.3 التصريف Compiling

تتمثل الخطوة الأولى لتصريف ملف مصدري إلى ملف قابل للتنفيذ في تحويل الشيفرة البرمجية من لغة عالية المستوى يفهمها الإنسان إلى شيفرة تجميع Assembly Code تعمل مباشرةً مع التعليمات والمسجلات التي يوفرها المعالج.

تُعَد عملية التصريف أكثر الخطوات تعقيدًا لعدة أسباب أولها أنه لا يمكن التنبؤ بتصرفات البشر، فلديهم شيفرتهم البرمجية الخاصة بأشكال مختلفة. يهتم المصرِّف بالشيفرة البرمجية الفعلية فقط، ولكن يحتاج البشر لأشياء إضافية مثل التعليقات والمسافات البيضاء (الفراغات ومسافات الجدولة Tab والمسافات البادئة وما

إلى ذلك) لفهم هذه الشيفرة البرمجية. تسمى العملية التي يتخذها المصرِّف لتحويل الشيفرة البرمجية التي يكتبها الإنسان إلى تمثيلها الداخلي بعملية التحليل Parsing.

هناك خطوة قبل تحليل الشيفرة البرمجية في الشيفرة المكتوبة بلغة C، حيث تسمَّى هذه الخطوة بالمعالجة المُسبَقة أو التمهيدية يقوم بها المعالج المسبق Pre-processor، وهو عبارة عن برنامج لاستبدال النصوص، حيث يُستبدَل مثلًا المتغير `variable` المُصرَّح عنه بالشكل `#define variable text` بالنص `text`، ثم تُمرَّر هذه الشيفرة البرمجية المعالَجة مسبقًا إلى المصرِّف.

7.3.1 الصياغة

لكل لغة برمجة صياغة معينة تمثّل قواعد اللغة، بحيث يعرف المبرمج والمصرِّف قواعد الصياغة ليفهما بعضهما البعض ويسير كل شيء على ما يرام. ينسى البشر القواعد أو يكسرونها في أغلب الأحيان، مما يجعل المصرِّف غير قادر على فهم ما يقصده المبرمج، فإن لم تضع قوس الإغلاق لشرط `if` مثلًا، فلن يعرف المصرِّف مكان الشرط فعليًا.

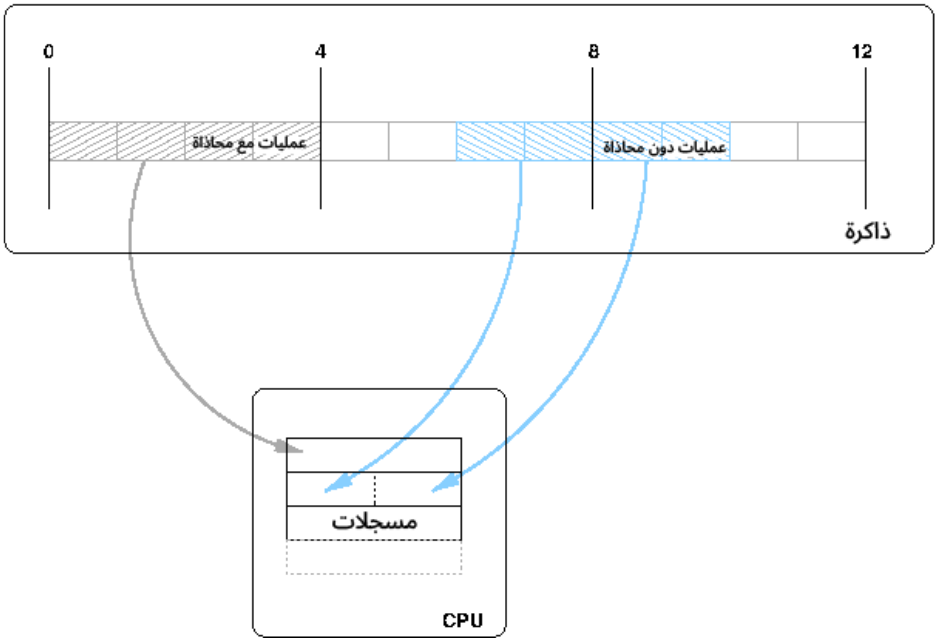
تُوصَف الصياغة في صيغة باكوس نور Backus-Naur Form -أو BNF اختصارًا- في أغلب الأحيان، وهي لغة يمكنك من خلالها وصف اللغات، والشكل الأكثر شيوعًا منها هو صيغة باكور نور الموسَّعة Extended Backus-Naur Form -أو EBNF اختصارًا- التي تسمح ببعض القواعد الإضافية الأكثر ملاءمة للغات الحديثة.

7.3.2 توليد شيفرة التجميع Assembly Generation

وظيفة المصرِّف هي ترجمة لغة عالية المستوى higher level language إلى شيفرة تجميع مناسبة للهدف من التصريف، فلكل معمارية مجموعة تعليمات مختلفة وأعداد مختلفة من المسجلات وقواعد مختلفة للتشغيل الصحيح.

ا. المحاذاة Alignment

تُعَد محاذاة المتغيرات في الذاكرة أمرًا مهمًا للمصرِّفات، إذ يحتاج مبرمجو الأنظمة أن يكونوا على دراية بقيود المحاذاة لمساعدة المصرِّف على إنشاء أكثر شيفرة برمجية فعّالة ممكنة.



شكل 36: المحاذاة في الذاكرة

لا تستطيع وحدات المعالجة المركزية CPU تحميل قيمة في المسجل من موقع ذاكرة عشوائي، إذ يتطلب ذلك أن تحاذي المتغيرات حدودًا معينة. يمكننا أن نرى في الشكل السابق كيفية تحميل قيمة 32 بت (4 بايتات) في مسجل على آلة تتطلب محاذاة بمقدار 4 بايتات للمتغيرات.

يمكن تحميل المتغير الأول في المسجل مباشرةً، حيث يقع بين حدود 4 بايتات، ولكن يجتاز المتغير الثاني حدود 4 بايتات، مما يعني أنه ستكون هناك حاجة إلى عمليتي تحميل على الأقل للحصول على المتغير في مسجل واحد إحداهما للنصف السفلي أولًا ثم النصف العلوي.

يمكن لبعض المعماريات مثل معمارية x86 التعاملَ مع عمليات التحميل التي تكون دون محاذاة في العتاد مع انخفاض في الأداء، حيث يطبّق العتاد العمل الإضافي للحصول على القيمة في المسجل، بينما لا يمكن أن يكون هناك انتهاك لقواعد المحاذاة في المعماريات الأخرى وسترفع استثناءً يكتشفه نظام التشغيل الذي يتعين عليه بعد ذلك تحميل المسجل يدويًا على أجزاء، مما يتسبب في مزيد من الحِمل.

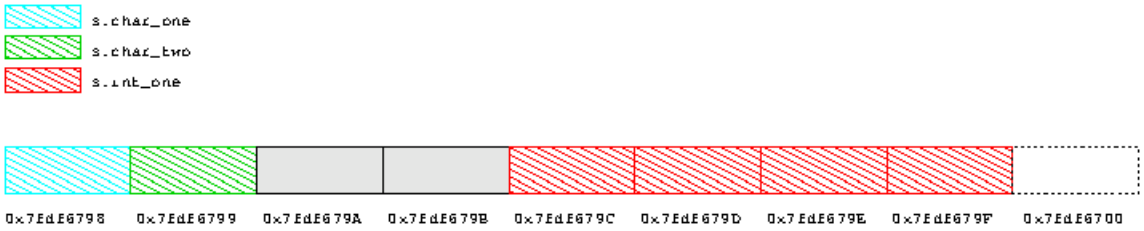
حاشية البنية Structure Padding

يجب أن يأخذ المبرمجون المحاذاة في الحسبان خاصةً عند إنشاء البنى `struct`، حيث يمكن للمبرمجين في بعض الأحيان أن يتسببوا في سلوك دون المستوى الأمثل، بينما يعرف المصرِّف قواعد المحاذاة للمعماريات التي يبنيها. ينص معيار C99 على أن البنى ستُرتَّب في الذاكرة بالترتيب المُحدَّد في التصريح نفسه، وستكون جميع العناصر بالحجم نفسه في مصفوفة من البنى.

إليك مثال عن حاشية بنية Struct Padding:

```
$ cat struct.c
#include <stdio.h>

struct a_struct {
  char char_one;
  char char_two;
  int int_one;
};

int main(void)
{

  struct a_struct s;

  printf("%p : s.char_one\n" \
    "%p : s.char_two\n" \
    "%p : s.int_one\n", &s.char_one,
    &s.char_two, &s.int_one);

  return 0;

}

$ gcc -o struct struct.c

$ gcc -fpack-struct -o struct-packed struct.c

$ ./struct
0x7fdf6798 : s.char_one
0x7fdf6799 : s.char_two
0x7fdf679c : s.int_one

$ ./struct-packed
0x7fcd2778 : s.char_one
0x7fcd2779 : s.char_two
0x7fcd277a : s.int_one
```

أنشأنا في المثال السابق بنية تحتوي على بايتين من النوع `char` متبوعين بعدد صحيح بحجم 4 بايتات من النوع `int`. يضيف المصرِّف حاشية للبنية كما يلي:



شكل 37: محاذاة المتغيرات

نوجّه في المثال السابق المصرِّف إلى عدم حشو البنى، وبالتالي يمكننا أن نرى أن العدد الصحيح يبدأ مباشرةً بعد قيمتين من النوع `char`.

محاذاة خط الذاكرة المخبئية Cache line alignment

تحدثنا سابقًا عن استخدام الأسماء البديلة في الذاكرة المخبئية، وكيف يمكن ربط عدة عناوين مع سطر الذاكرة المخبئية نفسه. يجب أن يتأكد المبرمجون من أنهم لا يتسببون في ارتداد Bouncing في خطوط الذاكرة المخبئية عندما يكتبون برامجهم.

يحدث هذا الموقف عندما يصل البرنامج باستمرار إلى منطقتين من الذاكرة ترتبطان مع خط الذاكرة المخبئية نفسه، مما يؤدي إلى هدر هذا الخط، حيث يُحمَّل ويُستخدَم لفترة قصيرة ثم يجب إزالته وتحميل خط الذاكرة المخبئية الآخر في المكان نفسه من الذاكرة المخبئية. يؤدي تكرار هذا الموقف إلى تقليل الأداء بصورة كبيرة، ولكن يمكن تخفيفه من خلال تنظيم البيانات المتعارضة بطرق مختلفة لتجنب تعارض خطوط الذاكرة المخبئية.

إحدى الطرق الممكنة لاكتشاف هذا النوع من المواقف هي التشخيص Profiling الذي يمثّل مراقبة الشيفرة البرمجية لتحليل مساراتها التي يمكن استخدامها والمدة المُستغرقَة لتنفيذها. يمكن للمصرِّف باستخدام التحسين المُوجَّه بالتشخيص Profile Guided Optimisation -أو PGO اختصارًا- وضعَ بتات إضافية خاصة من الشيفرة البرمجية في أول ثنائية binary يبنيها ويشغّلها ويسجّل الفروع المأخوذة منها وغير ذلك. يمكنك بعد ذلك إعادة تصريفها مع المعلومات الإضافية لإنشاء ثنائية binary مع أداء أفضل، وإلّا فيمكن للمبرمج أن ينظر إلى خرج عملية التشخيص ويكتشف مواقفًا أخرى مثل ارتداد خط الذاكرة المخبئية.

المقايضة بين المساحة والسرعة

يمكن المقايضة مع ما فعله المصرِّف سابقًا باستخدام ذاكرة إضافية لتحسين السرعة عند تشغيل شيفرتنا البرمجية. يعرف المصرِّف قواعد المعمارية ويمكنه اتخاذ قرارات بشأن أفضل طريقة لمحاذاة البيانات عن طريق مقايضة كميات صغيرة من الذاكرة المهدورة لزيادة الأداء أو للوصول إلى الأداء الصحيح فقط.

لا يجب أبدًا -بصفتك مبرمجًا- وضع افتراضات حول طريقة ترتيب المصرِّف للمتغيرات والبيانات، لأنها لا تُعَد قابلةً للنقل، إذ يكون للمعماريات المختلفة قواعدٌ مختلفة ويمكن أن يتخذ المُصرِّف قرارات مختلفة بناءً على أوامر أو مستويات تحسين صريحة.

وضع الافتراضات

يجب أن تكون -بصفتك مبرمجًا بلغة C- على دراية بما يمكنك افتراضه بشأن ما سيفعله المصرِّف وما يمكن أن يكون متغيرًا. ذُكِر بالتفصيل ما يمكنك أن تفترضه بالضبط وما لا يمكنك افتراضه في معيار C99، حيث إذا كنت مبرمجًا بلغة C، فلا بد أن يكون التعرف على القواعد جديرًا بالعناء لتجنب كتابة شيفرة برمجية غير قابلة للنقل.

إليك مثال عن محاذاة المكدس Stack Alignment:

```
$ cat stack.c
#include <stdio.h>

struct a_struct {
  int a;
  int b;
};

int main(void)
{
  int i;
  struct a_struct s;
  printf("%p\n%p\ndiff %ld\n", &i, &s, (unsigned long)&s - (unsigned
long)&i);
  return 0;
}
$ gcc-3.3 -Wall -o stack-3.3 ./stack.c
$ gcc-4.0 -o stack-4.0 stack.c

$ ./stack-3.3
0x60000fffffc2b510
0x60000fffffc2b520
diff 16

$ ./stack-4.0
```

```
0x60000fffff89b520
0x60000fffff89b524
diff  4
```

يمكننا أن نرى في المثال السابق المأخوذ من آلة إيتانيوم Itanium أن حاشية ومحاذاة المكدس تغيرت بصورة كبيرة بين إصدارات المصرِّف gcc، وهذا أمر متوقع ويجب على المبرمج مراعاته. كما يجب عليك التأكد من عدم وضع افتراضات حول حجم الأنواع أو قواعد المحاذاة.

مفاهيم لغة C الخاصة بالمحاذاة

هناك عدد من تسلسلات الشيفرة البرمجية الشائعة التي تتعامل مع المحاذاة، ويجب أن تضعها معظم البرامج في حساباتها. يمكن أن ترى "مفاهيم الشيفرة البرمجية" في العديد من الأماكن خارج النواة Kernel عند التعامل مع البرامج التي تعالج أجزاءً من البيانات بصيغة أو بأخرى، لذا فإن الأمر يستحق البحث.

يمكننا أخذ بعض الأمثلة من نواة لينكس Linux kernel التي يتعين عليها في أغلب الأحيان التعامل مع محاذاة صفحات الذاكرة ضمن النظام. إليك مثال عن التعامل مع محاذاة الصفحات:

```
[ include/asm-ia64/page.h ]

/*
 * يحدّد PAGE_SHIFT حجم صفحة النواة الفعلي
 */
#if defined(CONFIG_IA64_PAGE_SIZE_4KB)
# define PAGE_SHIFT     12
#elif defined(CONFIG_IA64_PAGE_SIZE_8KB)
# define PAGE_SHIFT     13
#elif defined(CONFIG_IA64_PAGE_SIZE_16KB)
# define PAGE_SHIFT     14
#elif defined(CONFIG_IA64_PAGE_SIZE_64KB)
# define PAGE_SHIFT     16
#else
# error Unsupported page size!
#endif

#define PAGE_SIZE               (__IA64_UL_CONST(1) << PAGE_SHIFT)
#define PAGE_MASK               (~(PAGE_SIZE - 1))
#define PAGE_ALIGN(addr)        (((addr) + PAGE_SIZE - 1) & PAGE_MASK)
```

يمكننا أن نرى في المثال السابق أن هناك عددًا من الخيارات المختلفة لأحجام الصفحات داخل النواة الـتي تـتراوح من 4 كيلوبايتـات إلى 64 كيلوبـايت. يُعـد المـاكرو `PAGE_SIZE` واضـحًا إلى حـد مـا، فهـو يعطي حجم الصفحة الحالي المحدّد ضمن النظام عن طريق انزياح قيمته 1 باستخدام رقم الانزياح المُعطَى، ويعادل ذلـك `2n` حيث `n` هو انزياح الصفحة `PAGE_SHIFT`.

لدينا بعد ذلك تعريف قناع الصفحة `PAGE_MASK` الذي يسمح لنـا بـالعثور على تلـك البتـات الموجـودة في الصفحة الحالية فقط، أي إزاحة `offset` العنوان في صفحته.

7.3.3 التحسين Optimisation

يريد المصرِّف بمجرد الحصول على تمثيل داخلي للشيفرة البرمجية إيجادَ أفضل خـرج بلغـة التجميـع لـدخل الشيفرة البرمجيـة المُحـدَّد. هـذه مشـكلة كبـيرة ومتنوعـة وتتطلب معرفـة كـل شـيء من الخوارزميـات الفعالـة المعتمَدة في علوم الحاسوب إلى المعرفة العميقة بالمعالج الذي ستعمل الشيفرة البرمجية عليه.

هناك بعض التحسينات الشائعة التي يمكن أن ينظر إليها المصرِّف عند توليد الخرج، وهناك العديد والعديد من الاستراتيجيات لإنشاء الشيفرة البرمجية الأفضل، ويُعَد ذلك مجال بحث غـني. يمكن للمصـرِّف أن يـرى في كثير من الأحيان أنه لا يمكن استخدام جـزء معين من الشـيفرة البرمجيـة، لـذا يتركـه لتحسـين بنيـة لغـة معينـة وينتقل إلى شيء أصغر يوصل للنتيجة نفسها.

ا. فك الحلقات Unrolling Loops

إذا احتوت الشيفرة البرمجية على حلقة مثل حلقة `for` أو `while` وكان لدى المُصرِّف فكرة عن عدد المرات التي ستنفّذ فيها، فسيكون فك الحلقة أكثر فاعلية بحيث تُنفَّذ تسلسليًا، إذ تُكرَّر شيفرة الحلقة الداخلية لتنفيذها عدد المرات ذاك أخرى بدلًا من تنفيذ الجزء الداخلي من الحلقة ثم العودة إلى البداية لتكرار العملية.

تزيد هذه العملية من حجم الشيفرة البرمجية، إذ يمكن أن تسمح للمعالج بتنفيـذ التعليمـات بفعاليـة، حيث يمكن أن تتسبب الفروع في تقليل كفاءة خط أنابيب التعليمات الواردة إلى المعالج.

ب. الدوال المضمنة Inlining Functions

يمكن وضع دوال مُضمَّنة لاستدعائها ضمن المستدعي callee، ويمكن للمبرمج تحديد ذلـك للمصـرِّف من خلال وضع الكلمة `inline` في تعريف الدالة، ويمكنك مقايضة حجم الشـيفرة البرمجيـة بتسلسـل تنفيـذها من خلال ذلك.

ج. توقع الفرع Branch Prediction

إذا صادف الحاسوب تعليمة `if`، فهناك نتيجتان محتملتان إما صحيحة أو خاطئـة. يريـد المعـالج الاحتفـاظ بأنابيبه الواردة ممتلئة قدر الإمكان، لذا لا يمكنـه انتظـار نتيجـة الاختبـار قبـل وضـع الشـيفرة البرمجيـة في خـط

الأنابيب، وبالتالي يمكن للمصرِّف أن يتنبأ بالطريقة التي يُحتمَل أن يسير بها الاختبار. وهناك بعض القواعد البسيطة التي يمكن أن يستخدمها المصرِّف لتخمين هذه الأمور، على سبيل المثال لا يُحتمَل أن تكون التعليمة `if (val == -1)` صحيحةً، لأن القيمة `-1` تشير عادةً إلى رمز خطأ ونأمل ألّا تُشغَّل هذه التعليمة كثيرًا.

يمكن لبعض المصرِّفات تصريف البرنامج، وجعل المستخدم يشغّله ليلاحظ الطريق الذي تسير به الفروع في ظل ظروف واقعية، ويمكنه بعد ذلك إعادة تصريفه بناءً على ما شاهده.

7.4 المجمع Assembler

تبقى شيفرة التجميع التي أخرجها المصرِّف في صيغة يمكن أن يقرأها الإنسان إذا كنت على معرفة بتفاصيل شيفرة التجميع الخاصة بالمعالج. يُلقي المطورون في أغلب الأحيان نظرة خاطفة على خرج التجميع للتحقق يدويًا من أن الشيفرة البرمجية هي الأفضل أو لاكتشاف أخطاء المصرِّف، ويُعَد ذلك أكثر شيوعًا مما هو متوقع خاصةً عندما يكثِر المصرِّف من التحسينات.

المجمِّع assembly هو عملية آلية لتحويل شيفرة التجميع إلى صيغة ثنائية. يحتفظ المجمّع بجدول كبير لكل تعليمة ممكنة ولنظيرها الثنائي الذي يسمى شيفرة العملية Op Code. يدمج المجمّع شيفرات العمليات مع المسجلات المحدَّدة في شيفرة التجميع لإنتاج ملف ثنائي بوصفه خرجًا.

يُطلق على هذه الشيفرة بشيفرة التعليمات المُصرَّفة Object Code، وهي شيفرة غير قابلة للتنفيذ في هذه المرحلة، وتُعد مجرد تمثيل ثنائي للدخل الذي يمثل شيفرة برمجية مصدرية. يُفضَّل ألّا يضع المبرمج الشيفرة المصدرية بأكملها في ملفٍ واحد.

7.5 الرابط Linker

ستُقسَم في أغلب الأحيان الشيفرة البرمجية في برنامج كبير إلى ملفات متعددة لتكون الدوال ذات الصلة مع بعضها بعضًا. يمكن تصريف كل ملفٍ من هذه الملفات إلى شيفرة تعليمات مُصرَّفة ولكن هدفك النهائي هو إنشاء ملف قابل للتنفيذ. يجب أن يكون هناك طريقة ما لدمجها في ملف واحد قابل للتنفيذ، حيث نسمي هذه العملية بالربط Linking.

لاحظ أنه لا يزال يجب ربط برنامجك بمكتبات نظام معينة للعمل بصورة صحيحة حتى إن كان برنامجك مناسبًا لملفٍ واحد، إذ يكون الاستدعاء `printf` مثلًا في مكتبة يجب دمجها مع ملفك القابل للتنفيذ ليعمل، لذا لا تزال هناك بالتأكيد عملية ربط تحدث لإنشاء ملفك القابل للتنفيذ بالرغم من أنه لا داعي للقلق صراحةً بشأن الربط في هذه الحالة.

سنشرح فيما يلي بعض المصطلحات الأساسية لفهم عملية الربط.

7.5.1 الرموز Symbols

لجميع المتغيرات والدوال أسماء في الشيفرة المصدرية، إذ نشير إليها باستخدام هذه الأسماء. تتمثل إحدى طرق التفكير في تعليمة التصريح عن متغير `int a` في أنك تخبر المصرِّف بأن يحجز حيزًا من الذاكرة بحجم `sizeof(int)`، وبالتالي كلما استخدمت اسم المتغير `a`، فسيشير إلى هذه الذاكرة المخصَّصة، وكذلك الأمر بالنسبة للدالة التي تخبر المصرِّف بأن يحزّن هذه الشيفرة البرمجية في الذاكرة، ثم ينتقل إليها وينفّذها عند استدعاء الدالة `function()`. وبالتالي نستدعي الرمزين `a` و`function` لأنهما يُعَدان تمثيلًا رمزيًا لمنطقةٍ من الذاكرة.

تساعد هذه الرموز البشر على فهم البرمجة. لكن يمكنك القول أن المهمة الأساسية لعملية التصريف هي إزالة هذه الرموز، إذ لا يعرف المعالج ما يمثله الرمز `a`، فكل ما يعرفه هو أن لديه بعض البيانات في عنوان ذاكرة معين. تحوِّل عملية التصريف التعليمة `a += 2` إلى العبارة "زيادة القيمة الموجودة في العنوان `0xABCDE` من الذاكرة بمقدار 2".

I. إمكانية رؤية الرموز Symbol Visibility

لا بد أنك شاهدت في البرامج المكتوبة بلغة C المصطلحين `static` و`extern` مع المتغيرات، إذ يمكن أن تؤثر هذه المعدِّلات Modifiers على ما نسميه إمكانية رؤية الرموز.

لنفترض أنك قسمت برنامجك إلى ملفين، ولكن تريد بعضُ الدوال مشاركةَ متغيرٍ ما. نريد تعريفًا Definition أو موقعًا واحدًا فقط في الذاكرة للمتغير المشترك وإلا فلا يمكن مشاركته، ولكن يجب أن يشير كلا الملفين إليه. يمكن ذلك من خلال التصريح عن المتغير في ملف واحد، ثم نصرّح في الملف الآخر عن متغير بالاسم نفسه مع البادئة `extern` التي ترمز إلى أنه خارجي External وترمز للمبرمج بأن هذا المتغير مُصرَّحٌ عنه في مكان آخر.

تخبر الكلمة `extern` المصرِّف أنه لا ينبغي تخصيص أي مساحة في الذاكرة لهذا المتغير، ويجب ترك هذا الرمز في التعليمات المُصرَّفة لإصلاحه لاحقًا. لا يمكن للمصرِّف أن يعرف مكان تعريف الرمز فعليًا ولكن الرابط Linker يمكنه ذلك، فوظيفته هي النظر في جميع ملفات التعليمات المُصرَّفة ودمجها في ملف واحد قابل للتنفيذ. لذا سيرى الرابط هذا الرمز في الملف الثاني، وسيقول: "رأيت هذا الرمز مسبقًا في الملف 1، وأعلم أنه يشير إلى موقع الذاكرة `0x12345`"، وبالتالي يمكن تعديل قيمة الرمز لتكون قيمة الذاكرة للمتغير الموجود في الملف الأول.

تُعَد الكلمة ساكن `static` عكس خارجي `extern` تقريبًا، لأنها تضع قيودًا على رؤية الرمز الذي نريد تعديله. إذا صرّحتَ عن متغير بأنه ساكن `static`، فهذا يعني للمصرّف بألا يترك أيّ رموز لهذا المتغير في شيفرة التعليمات المصرَّفة، وبالتالي لن يرى الرابط هذا الرمز أبدًا عندما يربط ملفات التعليمات المُصرَّفة مع بعضها البعض، أي لا يمكنه القول بأنه رأى هذا الرمز سابقًا. يُعَد استخدام الكلمة `static` مفيدًا للفصل بين

الرموز وتقليل التعارضات بينها، إذ يمكنك إعادة استخدام اسم المتغير المُصرَّح عنه بأنه `static` في ملفات أخرى دون وجود تعارضات بين الرموز. يمكن القول بأننا نقيّد رؤية الرمز، لأننا لا نسمح للرابط برؤيته بعكس الرمز الذي لم يُصرَّح عنه بأنه `static` ويمكن للرابط رؤيته.

7.5.2 عملية الربط

تتكون عملية الربط من خطوتين هما: دمج جميع ملفات التعليمات المُصرَّفة في ملف واحد قابل للتنفيذ ثم الانتقال إلى كل ملف لتحليل الرموز. يتطلب ذلك تمريرين، أحدهما لقراءة جميع تعريفات الرموز وتدوين الرموز التي لم تُحلَّل والثاني لإصلاح تلك الرموز التي لم تُحلَّل في المكان الصحيح.

يجب أن يكون الملف القابل للتنفيذ النهائي بدون رموز غير مُحلَّلة، إذ سيفشل الرابط مع وجود خطأ بسبب هذه الرموز. نسمي ذلك بالربط الساكن Static Linking، فالربط الديناميكي هو مفهوم مشابه يُطبَّق ضمن الملف القابل للتنفيذ في وقت التشغيل، حيث سنتطرق إليه لاحقًا.

تعرّفنا الفقرات السابقة على الخطوات الثلاث لبناء ملف قابل للتنفيذ هي: التصريف Compiling والتجميع Assembling والربط Linking، وسنطبّق في الفقرات التالية هذه الخطوات عمليًا لبناء ملف قابل للتنفيذ بلغة C.

7.6 تطبيق عملي لبناء برنامج تنفيذي من شيفرة مصدرية بلغة C

بعد أن تعرّفنا على الخطوات الثلاث لبناء ملف قابل للتنفيذ هي: التصريف Compiling والتجميع Assembling والربط Linking، وسنطبّق الآن هذه الخطوات عمليًا لبناء ملف قابل للتنفيذ.

تابع فيما الخطوات المتبعة لبناء تطبيق بسيط خطوة بخطوة. لاحظ أن الأمر `gcc` يشغّل برنامج تشغيلٍ driver program يخفي معظم الخطوات عنك، وهذا هو ما تريده بالضبط في ظل الظروف العادية، لأن الأوامر والخيارات الدقيقة للحصول على ملف قابلٍ للتنفيذ على نظام حقيقي يمكن أن تكون معقدة للغاية وخاصةً بكل معمارية على حدة.

سنشرح عملية التصريف في المثالين التاليين، حيث سنستخدم ملفين مصدريين مكتوبين بلغة C، إذ يعرّف أحدهما الدالة الرئيسية `main()` التي تُعَد نقطة الدخول الأولية، ويصرّح الملف الآخر عن دالة مساعدة، وهناك متغير عام واحد.

إليك مثال مرحبًا بالعالم Hello World:

```c
#include <stdio.h>

/* function() نحتاج نموذجًا أوليًا ليعرف المصرّف نوع الدالة */
int function(char *input);
```

```
/* بما أن هذا المتغير ساكن static، فيمكننا تعريفه في كلٍّ من الملفين hello.c وfunction.c */
static int i = 100;

/* هذا متغير عام */
int global = 10;

int main(void)
{
  /* يجب أن تعيد الدالة function() قيمة المتغير العام global */
  int ret = function("Hello, World!");
  exit(ret);
}
```

إليك مثال على دالة:

```
#include <stdio.h>

static int i = 100;

/* مُصرّح عنه بأنه خارجي extern لأنه مُعرَّف في الملف hello.c */
extern int global;

int function(char *input)
{
  printf("%s\n", input);
  return global;
}
```

7.6.1 التصريف Compiling

لكل المصرِّفات خيارٌ لتنفيذ الخطوة الأولى من التصريف فقط مثل استخدام الراية `-S` لوضع الخرج في ملف يحمل اسم ملف الدخل نفسه ولكن مع اللاحقة `.s`، وبالتالي يمكننا عرض الخطوة الأولى باستخدام الأمر `gcc -S` كما هو موضح في المثال التالي:

```
$ gcc -S hello.c
$ gcc -S function.c
$ cat function.s
```

```
  .file   "function.c"
  .pred.safe_across_calls p1-p5,p16-p63
  .section        .sdata,"aw",@progbits
  .align 4
  .type   i#, @object
  .size   i#, 4
i:
  data4   100
  .section        .rodata
  .align 8
.LC0:
  stringz "%s\n"
  .text
  .align 16
  .global function#
  .proc function#
function:
  .prologue 14, 33
  .save ar.pfs, r34
  alloc r34 = ar.pfs, 1, 4, 2, 0
  .vframe r35
  mov r35 = r12
  adds r12 = -16, r12
  mov r36 = r1
  .save rp, r33
  mov r33 = b0
  .body
  ;;
  st8 [r35] = r32
  addl r14 = @ltoffx(.LC0), r1
  ;;
  ld8.mov r37 = [r14], .LC0
  ld8 r38 = [r35]
  br.call.sptk.many b0 = printf#
  mov r1 = r36
  ;;
```

```
    addl r15 = @ltoffx(global#), r1
    ;;
    ld8.mov r14 = [r15], global#
    ;;
    ld4 r14 = [r14]
    ;;
    mov r8 = r14
    mov ar.pfs = r34
    mov b0 = r33
    .restore sp
    mov r12 = r35
    br.ret.sptk.many b0
    ;;
    .endp function#
    .ident  "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-11)"
```

تُعَد عملية التجميع Assembly معقدة قليلًا، ولكن يجب أن تكون قادرًا على معرفة مكان تعريف المتغير `i` بوصفه `data4` أي 4 بايتات أو 32 بت بحجم النوع `int`، ومكان تعريف الدالة `function` (بالشكل `function:`) واستدعاء الدالة `printf()`.

أصبح لدينا الآن ملفا تجميع جاهزين لتجميعهما في شيفرة الآلة البرمجية machine code.

7.6.2 التجميع Assembly

التجميع هو عملية مباشرة إلى حد ما، ويُطلَق على المجمّع `as` ويأخذ وسائطًا بطريقة مماثلة للأمر `gcc`.

إليك مثال عن التجميع:

```
$ as -o function.o function.s
$ as -o hello.o hello.s
$ ls
function.c  function.o  function.s  hello.c  hello.o  hello.s
```

تنتج عن عملية التجميع التعليمات المُصرَّفة Object Code، حيث تكون هذه الشيفرة جاهزةً لربطها مع بعضها البعض في الملف النهائي القابل للتنفيذ. يمكنك تخطي الاضطرار إلى استخدام المُجمِّع يدويًا من خلال استدعاء المصرِّف مع الراية `-c` التي تحوّل ملف الدخل مباشرةً إلى شيفرة كائن، وتضعها في ملف له البادئة نفسها ولكن مع اللاحقة `.o`.

لا يمكننا فحص شيفرة التعليمات المُصرَّفة مباشـرةً لأنهـا في صـيغة ثنائيـة، ولكن يمكننـا اسـتخدام بعض الأدوات لفحص ملفـات التعليمـات المُصـرَّفة مثـل الأداة `readelf --symbols` الـتي سـتعرض الرمـوز الموجودة في ملف الكائن كما يلي:

```
$ readelf --symbols ./hello.o

Symbol table '.symtab' contains 15 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
0000000000000000     0 SECTION LOCAL  DEFAULT    1
0000000000000000     0 SECTION LOCAL  DEFAULT    3
0000000000000000     0 SECTION LOCAL  DEFAULT    4
0000000000000000     0 SECTION LOCAL  DEFAULT    5
0000000000000000     4 OBJECT  LOCAL  DEFAULT    5 i
0000000000000000     0 SECTION LOCAL  DEFAULT    6
0000000000000000     0 SECTION LOCAL  DEFAULT    7
0000000000000000     0 SECTION LOCAL  DEFAULT    8
0000000000000000     0 SECTION LOCAL  DEFAULT   10
0000000000000004     4 OBJECT  GLOBAL DEFAULT    5 global
0000000000000000    96 FUNC    GLOBAL DEFAULT    1 main
0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND function
0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit

$ readelf --symbols ./function.o

Symbol table '.symtab' contains 14 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS function.c
0000000000000000     0 SECTION LOCAL  DEFAULT    1
0000000000000000     0 SECTION LOCAL  DEFAULT    3
0000000000000000     0 SECTION LOCAL  DEFAULT    4
0000000000000000     0 SECTION LOCAL  DEFAULT    5
0000000000000000     4 OBJECT  LOCAL  DEFAULT    5 i
0000000000000000     0 SECTION LOCAL  DEFAULT    6
```

```
0000000000000000     0 SECTION LOCAL  DEFAULT    7
0000000000000000     0 SECTION LOCAL  DEFAULT    8
0000000000000000     0 SECTION LOCAL  DEFAULT   10
0000000000000000   128 FUNC    GLOBAL DEFAULT    1 function
0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND global
```

يُعَد هذا الخرج معقدًا للغاية، ولكن يجب أن تكون قادرًا على فهم الكثير منه مثل:

- لاحظ الرمز الذي يحمل الاسم `i` في الخرج `hello.o`، حيث يُسبَق هذا الرمز بالكلمة `LOCAL` أي أنه محلي، لأننا صرّحنا عنه بأنه ساكن `static`، وبالتالي يُميَّز على أنه محلي لملف الكائن.
- لاحظ المتغير `global` في الخرج نفسه المُعرَّف على أنه متغير عام `GLOBAL`، مما يعني أنه مرئي خارج هذا الملف، وتكون الدالة الرئيسية `main()` مرئية من خارج الملف.
- لاحظ أن الرمز `function` له النوع `UND` أو غير مُعرَّف Undefined من أجل استدعاء الدالة `function()`، أي أن الأمر متروك للرابط Linker للعثور على عنوان الدالة.
- لاحظ الرموز الموجودة في الملف `function.c` وكيفية ملاءمتها مع الخرج.

7.6.3 الربط Linking

يُعَد استدعاء الرابط المُسمَّى `ld` عمليةً معقدة للغاية على نظام حقيقي، لذلك نترك عملية الربط للأمر gcc، ولكن يمكننا التعرّف على ما يفعله داخليًا باستخدام الراية `-v` التي ترمز إلى Verbose أي مُفصَّلة.

إليك مثال عن عملية الربط:

```
/usr/lib/gcc-lib/ia64-linux/3.3.5/collect2 -static
/usr/lib/gcc-lib/ia64-linux/3.3.5/../../../crt1.o
/usr/lib/gcc-lib/ia64-linux/3.3.5/../../../crti.o
/usr/lib/gcc-lib/ia64-linux/3.3.5/crtbegin.o
-L/usr/lib/gcc-lib/ia64-linux/3.3.5
-L/usr/lib/gcc-lib/ia64-linux/3.3.5/../../..
hello.o
function.o
--start-group
-lgcc
-lgcc_eh
-lunwind
```

```
-lc
--end-group
/usr/lib/gcc-lib/ia64-linux/3.3.5/crtend.o
/usr/lib/gcc-lib/ia64-linux/3.3.5/../../../crtn.o
```

أول شيء تلاحظه هو استدعاء برنامج بالاسم `collect2` وهو عبارة عن مُغلِّف للرابط `ld`، ويستخدم الأمر `gcc` داخليًا. الشيء الآخر الذي ستلاحظه هو ملفات الكائنات التي تبدأ بالرمز `crt` أي أنها مُحدَّدة للرابط. يُوفّر الأمر `gcc` ومكتبات النظام هذه الدوال التي تحتوي على الشيفرة البرمجية المطلوبة لبدء البرنامج. لا تُعَد الدالة الرئيسية `main()` أول دالة مُستدعاة عند تشغيل البرنامج، بل تُستدعَى أولًا الدالة `_start` الموجودة في ملفات الكائنات `crt`، حيث تضبط هذه الدالة بعض الإعدادات العامة التي لا يجب أن يقلق مبرمجو التطبيقات بشأنها.

يُعَد تسلسل المسار الهرمي معقدًا للغاية، ولكن يمكننا أن نرى أن الخطوة الأخيرة هي ربط بعض ملفات الكائنات الإضافية وهي:

- `crt1.o`: توفره مكتبات النظام libc، ويحتوي على الدالة `_start` التي تُعَد أول شيء يُستدعَى في البرنامج.
- `crti.o`: توفّره مكتبات النظام.
- `crtbegin.o`
- `crtsaveres.o`
- `crtend.o`
- `crtn.o`

يمكنك أن ترى بعد ذلك أننا نربط ملفي الكائنات `hello.o` و `function.o`، ثم نحدّد بعض المكتبات الإضافية باستخدام رايات `-l`، حيث تُعَد هذه المكتبات خاصةً بالنظام ومطلوبة لكل برنامج. الراية الرئيسية هي الراية `-lc` التي تجلب مكتبة C التي تحتوي على جميع الدوال المشتركة مثل الدالة `printf()`. نربط بعد ذلك مرة أخرى بعض ملفات كائنات النظام التي تطبّق بعض عمليات التنظيف بعد انتهاء البرامج. تُعَد هذه التفاصيل معقدة، إلا أن مفهومها واضح ومباشر.

سنربط بعدها جميع ملفات التعليمات المُصرَّفة مع بعضها بملف واحد قابل للتنفيذ وجاهز للتشغيل.

7.6.4 الملف القابل للتنفيذ Executable

سندخل في مزيد من التفاصيل حول الملف القابل للتنفيذ لاحقًا، ولكن يمكننا إجراء فحص بطريقة مماثلة لملفات الكائنات لمعرفة ما يحدث.

إليك مثال عن ملف قابل للتنفيذ:

```
ianw@lime:~/programs/csbu/wk7/code$ gcc -o program hello.c function.c
ianw@lime:~/programs/csbu/wk7/code$ readelf --symbols ./program

Symbol table '.dynsym' contains 11 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
6000000000000de0     0 OBJECT  GLOBAL DEFAULT  ABS _DYNAMIC
0000000000000000   176 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2 (2)
600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
0000000000000000   704 FUNC    GLOBAL DEFAULT  UND exit@GLIBC_2.2 (2)
600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
6000000000000fe8     0 OBJECT  GLOBAL DEFAULT  ABS _GLOBAL_OFFSET_TABLE_
7: 60000000000010b0     0 NOTYPE  GLOBAL DEFAULT  ABS _end
0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
0000000000000000   544 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2
(2)
0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Symbol table '.symtab' contains 127 entries:
  Num:    Value          Size Type    Bind   Vis      Ndx Name
0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
40000000000001c8     0 SECTION LOCAL  DEFAULT    1
40000000000001e0     0 SECTION LOCAL  DEFAULT    2
4000000000000200     0 SECTION LOCAL  DEFAULT    3
4000000000000240     0 SECTION LOCAL  DEFAULT    4
4000000000000348     0 SECTION LOCAL  DEFAULT    5
40000000000003d8     0 SECTION LOCAL  DEFAULT    6
40000000000003f0     0 SECTION LOCAL  DEFAULT    7
4000000000000410     0 SECTION LOCAL  DEFAULT    8
4000000000000440     0 SECTION LOCAL  DEFAULT    9
40000000000004a0     0 SECTION LOCAL  DEFAULT   10
40000000000004e0     0 SECTION LOCAL  DEFAULT   11
40000000000005e0     0 SECTION LOCAL  DEFAULT   12
4000000000000b00     0 SECTION LOCAL  DEFAULT   13
4000000000000b40     0 SECTION LOCAL  DEFAULT   14
4000000000000b60     0 SECTION LOCAL  DEFAULT   15
4000000000000bd0     0 SECTION LOCAL  DEFAULT   16
4000000000000ce0     0 SECTION LOCAL  DEFAULT   17
```

```
6000000000000db8     0 SECTION LOCAL  DEFAULT   18
6000000000000dd0     0 SECTION LOCAL  DEFAULT   19
6000000000000dd8     0 SECTION LOCAL  DEFAULT   20
6000000000000de0     0 SECTION LOCAL  DEFAULT   21
6000000000000fc0     0 SECTION LOCAL  DEFAULT   22
6000000000000fd0     0 SECTION LOCAL  DEFAULT   23
6000000000000fe0     0 SECTION LOCAL  DEFAULT   24
6000000000000fe8     0 SECTION LOCAL  DEFAULT   25
6000000000001040     0 SECTION LOCAL  DEFAULT   26
6000000000001080     0 SECTION LOCAL  DEFAULT   27
60000000000010a0     0 SECTION LOCAL  DEFAULT   28
60000000000010a8     0 SECTION LOCAL  DEFAULT   29
0000000000000000     0 SECTION LOCAL  DEFAULT   30
0000000000000000     0 SECTION LOCAL  DEFAULT   31
0000000000000000     0 SECTION LOCAL  DEFAULT   32
0000000000000000     0 SECTION LOCAL  DEFAULT   33
0000000000000000     0 SECTION LOCAL  DEFAULT   34
0000000000000000     0 SECTION LOCAL  DEFAULT   35
0000000000000000     0 SECTION LOCAL  DEFAULT   36
0000000000000000     0 SECTION LOCAL  DEFAULT   37
0000000000000000     0 SECTION LOCAL  DEFAULT   38
0000000000000000     0 SECTION LOCAL  DEFAULT   39
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
```

```
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS abi-note.S
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS init.c
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS initfini.c
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
4000000000000670   128 FUNC    LOCAL  DEFAULT   12 gmon_initializer
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS initfini.c
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS /build/buildd/glibc-2.3.2
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS auto-host.h
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
6000000000000fc0     0 NOTYPE  LOCAL  DEFAULT   22 __CTOR_LIST__
6000000000000fd0     0 NOTYPE  LOCAL  DEFAULT   23 __DTOR_LIST__
6000000000000fe0     0 NOTYPE  LOCAL  DEFAULT   24 __JCR_LIST__
6000000000001088     8 OBJECT  LOCAL  DEFAULT   27 dtor_ptr
40000000000006f0   128 FUNC    LOCAL  DEFAULT   12 __do_global_dtors_aux
4000000000000770   128 FUNC    LOCAL  DEFAULT   12 __do_jv_register_classes
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
6000000000001090     4 OBJECT  LOCAL  DEFAULT   27 i
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS function.c
6000000000001098     4 OBJECT  LOCAL  DEFAULT   27 i
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS auto-host.h
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <command line>
0000000000000000     0 FILE    LOCAL  DEFAULT  ABS <built-in>
```

```
6000000000000fc8     0 NOTYPE  LOCAL  DEFAULT   22 __CTOR_END__
6000000000000fd8     0 NOTYPE  LOCAL  DEFAULT   23 __DTOR_END__
6000000000000fe0     0 NOTYPE  LOCAL  DEFAULT   24 __JCR_END__
6000000000000de0     0 OBJECT  GLOBAL DEFAULT  ABS _DYNAMIC
4000000000000a70   144 FUNC    GLOBAL HIDDEN   12 __do_global_ctors_aux
6000000000000dd8     0 NOTYPE  GLOBAL DEFAULT  ABS __fini_array_end
60000000000010a8     8 OBJECT  GLOBAL HIDDEN   29 __dso_handle
40000000000009a0   208 FUNC    GLOBAL DEFAULT   12 __libc_csu_fini
0000000000000000   176 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2
40000000000004a0    32 FUNC    GLOBAL DEFAULT   10 _init
4000000000000850   128 FUNC    GLOBAL DEFAULT   12 function
40000000000005e0   144 FUNC    GLOBAL DEFAULT   12 _start
6000000000001094     4 OBJECT  GLOBAL DEFAULT   27 global
6000000000000dd0     0 NOTYPE  GLOBAL DEFAULT  ABS __fini_array_start
40000000000008d0   208 FUNC    GLOBAL DEFAULT   12 __libc_csu_init
600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
40000000000007f0    96 FUNC    GLOBAL DEFAULT   12 main
6000000000000dd0     0 NOTYPE  GLOBAL DEFAULT  ABS __init_array_end
6000000000000dd8     0 NOTYPE  WEAK   DEFAULT   20 data_start
4000000000000b00    32 FUNC    GLOBAL DEFAULT   13 _fini
0000000000000000   704 FUNC    GLOBAL DEFAULT  UND exit@@GLIBC_2.2
600000000000109c     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
6000000000000fe8     0 OBJECT  GLOBAL DEFAULT  ABS _GLOBAL_OFFSET_TABLE_
60000000000010b0     0 NOTYPE  GLOBAL DEFAULT  ABS _end
6000000000000db8     0 NOTYPE  GLOBAL DEFAULT  ABS __init_array_start
6000000000001080     4 OBJECT  GLOBAL DEFAULT   27 _IO_stdin_used
60000000000010a0     8 OBJECT  GLOBAL DEFAULT   28 __libc_ia64_register_back
6000000000000dd8     0 NOTYPE  GLOBAL DEFAULT   20 __data_start
0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
0000000000000000   544 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

إليك بعض الأشياء التي يجب ملاحظتها:

- لاحظ طريقة بناء الملف القابل للتنفيذ السهلة.
- لاحظ وجود نوعين من جداول الرمـوز همـا: `dynsym` و`symtab`. سنشـرح كيفيـة عمـل رمـوز `dynsym` لاحقًا، ولكن لاحظ أن بعضها يحمل الرمز `@`.

- لاحظ الرموز العديدة المُضمَّنة من ملفات الكائنات الإضافية، حيث يبدأ الكثـير منهـا بـالرمز __ لتجنب التعارض مع الأسماء التي يختارها المبرمج. اقرأ واختر الرموز التي ذكرناهـا سـابقًا من ملفـات الكائنـات واكتشف إن تغيرت بأيّ شكل من الأشكال.

خمسات

لبيع وشراء الخدمات المصغرة

أكبر سوق عربي لبيع وشراء الخدمات المصغرة
اعرض خدماتك أو احصل على ما تريد بأسعار تبدأ من $5 فقط

تصفح الخدمات

8. ما وراء العملية

8.1 نظرة على الملفات القابلة للتنفيذ

يحتوي البرنامج الذي يعمل في الذاكرة على مكونين رئيسيين هما: الشيفرة البرمجية Code المعروفة أيضًا باسم النص Text والبيانات Data. لا يبقى الملف القابل للتنفيذ في الذاكرة، ولكنه يقضي معظم وقته بوصفه ملفًا على القرص الصلب ينتظر تحميله عند التشغيل. يُعَد الملف مجرد مصفوفة متجاورة من البتات، لذا تبتكر جميع الأنظمة طرقًا لتنظيم الشيفرة البرمجية والبيانات ضمن الملفات للتنفيذ عند الطلب، حيث يشار إلى هذه الصيغة من الملفات باسم ملف ثنائي Binary أو ملف قابل للتنفيذ Executable، وتكون البتات والبايتات الخاصة بالملف بصيغة جاهزة لوضعها في الذاكرة وتفسيرها مباشرةً بواسطة عتاد المعالج.

8.2 تمثيل الملفات القابلة للتنفيذ

يجب أن تحدّد أيّ صيغة لملفٍ قابل للتنفيذ executable file مكانَ وجود الشيفرة البرمجية والبيانات ضمن الملف الثنائي، حيث تُعَد الشيفرة البرمجية والبيانات القسمين الأساسيين لملفٍ قابل للتنفيذ، وأحد المكونات الإضافية التي لم نذكرها حتى الآن هو مساحة تخزين المتغيرات العامة غير المُهيَّأة uninitialised global variables.

إذا صرّحنا عن متغير وأعطيناه قيمة أولية، فيجب تخزين هذه القيمة في ملف قابل للتنفيذ بحيث يمكن تهيئته بالقيمة الصحيحة عند بدء البرنامج، ولكن هناك العديد من المتغيرات غير المهيأة أو التي قيمتها صفر عند تنفيذ البرنامج لأول مرة. يُعَد حجز مساحة لهذه المتغيرات في الملف القابل للتنفيذ ثم تخزين قيم صفرية أو فارغة NULL هدرًا للمساحة، مما يؤدي إلى تضخّم حجم الملف القابل للتنفيذ على القرص الصلب دون داع لذلك. تُعرّف معظم الصيغ الثنائية مفهوم القسم BSS الإضافي بوصفه حجمًا بديلًا للبيانات الصفرية غير المُهيَّأة. يمكن تخصيص الذاكرة الإضافية التي يحدّدها القسم BSS وضبطها على القيمة صفر عند تحميل

البرنامج. يرمز الاختصار BSS إلى العبارة Block Started by Symbol، وهو أمر بلغة تجميع حاسوب IBM القديم، ولكن يُرجَّح أن الاشتقاق الدقيق له ضاع مع الوقت.

8.2.1 الصيغة الثنائية Binary Format

يُنشَأ الملف القابل بالتنفيذ باستخدام سلسلة أدوات من الشيفرة المصدرية، حيث يجب أن يكون هذا الملف بصيغة محددة وواضحة بحيث يمكن للمصرِّف إنشاؤه ويمكن لنظام التشغيل تحديده وتحميله في الذاكرة وتحويله إلى عملية مُشغَّلة يمكن لنظام التشغيل إدارتها. يمكن أن تكون هذه الصيغة من الملفات القابلة للتنفيذ خاصة بنظام التشغيل، إذ لا نتوقع تنفيذ برنامج مُصرَّف لنظامٍ ما على نظام آخر مثل أن تعمل برامج ويندوز على نظام لينكس أو أن تعمل برامج لينكس على نظام macOS.

لكن خيط المعالجة Thread المشترك بين جميع صيغ الملفات القابلة للتنفيذ هو أنها تتضمن ترويسة معيارية مُعرَّفة مسبقًا توضّح كيفية تخزين شيفرة وبيانات البرنامج في بقية الملف، حيث يمكن أن تشرح ذلك بالكلمات مثل أن نقول: "تبدأ شيفرة البرنامج من 20 بايت في هذا الملف، ويبلغ طولها 50 كيلوبايت، وتتبعها بيانات البرنامج ويبلغ طولها 20 كيلوبايت".

هناك صيغة معينة أصبحت في الآونة الأخيرة معيارًا لتمثيل الملفات القابلة للتنفيذ في الأنظمة الحديثة القائمة على نظام يونكس، ويطلَق على هذا التنسيق بصيغة الرابط والملفات القابلة للتنفيذ Executable and Linker Format -أو ELF اختصارًا، حيث سنشرحها بمزيد من التفصيل لاحقًا.

8.2.2 تاريخ الصيغة الثنائية

سنوضح فيما يلي صيغتين للملفات الثنائية سبقت ظهور صيغة ملفات ELF هما a.out و COFF.

I. a.out

لم تكن صيغة ملفات ELF المعيار دائمًا، إذ استخدمت أنظمة يونكس الأصلية صيغة ملف بالاسم `a.out`. يمكننا أن نرى آثار ذلك عند تصريف برنامج بدون الخيار `-o` لتحديد اسم ملف الخرج، حيث سينشأ الملف القابل للتنفيذ بالاسم الافتراضي `a.out` الذي يُعَد اسم ملف الخرج الافتراضي الناتج عن الرابط Linker. يستخدم المصرِّف Compiler أسماء الملفات المُنشَأة عشوائيًا بوصفها ملفات وسيطة لشيفرة التجميع والشيفرة المُصرَّفة.

`a.out` هو صيغة ترويسة بسيطة تسمح فقط بقسم واحد للبيانات والشيفرة وBSS، وهذا غير كافٍ للأنظمة الحديثة ذات المكتبات الديناميكية.

ب. COFF

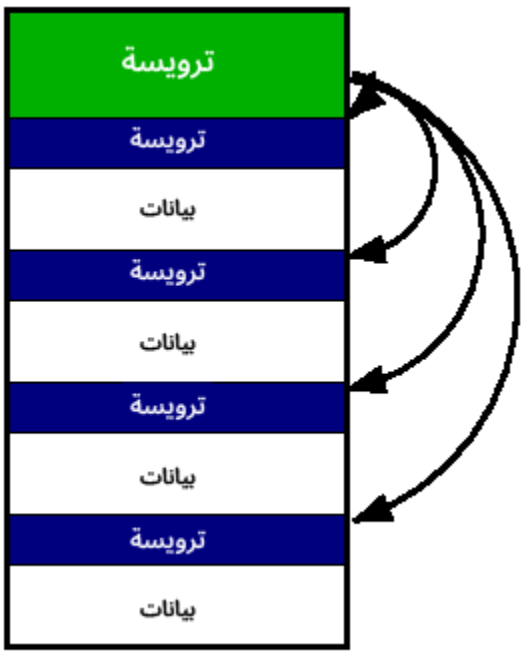
كانت صيغة ملف التعليمات المُصرَّفة المشترك Common Object File Format -أو COFF اختصارًا- مقدمة لظهور صيغة ملفات ELF، حيث كانت صيغة ترويستها أكثر مرونة، مما يسمح بمزيد -ولكن محدود- من الأقسام في الملف.

تواجه صيغة COFF صعوبات في دعم المكتبات المشتركة، لذا اختيرت صيغة ELF بوصفها تقديمًا Implementation بديلًا على نظام لينكس. لكن توجد صيغة COFF في مايكروسوفت ويندوز بوصفها صيغة ملفات قابلة للتنفيذ والنقل Portable Executable -أو PE اختصارًا- التي تُعَد بالنسبة إلى ويندوز مثل صيغة ملفات ELF في لينكس.

8.3 صيغة ملفات ELF

تُعَد صيغة ملفات ELF صيغةً مرنة لتمثيل الشيفرة الثنائية في النظام، حيث يمكنك باتباع معيار ELF تمثيل النواة Kernel ثنائيًا بسهولة مثل تمثيل ملف قابل للتنفيذ أو مكتبة نظام عادية. يمكن استخدام الأدوات نفسها لفحص وتشغيل جميع ملفات ELF ويمكن للمطورين الذين يفهمون صيغة ملفات ELF الاستفادة من مهاراتهم في معظم الأنظمة الحديثة المبنية على يونكس.

توسّع الصيغة ELF صيغة الملفات COFF وتمنح الترويسة مرونة كافية لتحديد عدد عشوائي من الأقسام، بحيث يكون لكل منها خاصياته الخاصة، مما يسهّل الربط الديناميكي وتنقيح الأخطاء Debugging.



شكل 38: نظرة عامة على ELF

8.3.1 ترويسة ملفات ELF

يحتوي الملف على ترويسة ملف File Header تصِف الملف، ثم يحتوي على مؤشرات لكل قسم من الأقسام التي يتكون منها الملف. يوضح المثال التالي الوصف على النحو الوارد في توثيق واجهة برمجة تطبيقات ELF32 (نموذج 32 بت من صيغة ملفات ELF)، وهو تخطيط لبنية لغة C الذي يعرّف ترويسة ELF:

```
typedef struct {
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;
  Elf32_Half    e_machine;
  Elf32_Word    e_version;
  Elf32_Addr    e_entry;
  Elf32_Off     e_phoff;
  Elf32_Off     e_shoff;
  Elf32_Word    e_flags;
  Elf32_Half    e_ehsize;
  Elf32_Half    e_phentsize;
  Elf32_Half    e_phnum;
  Elf32_Half    e_shentsize;
  Elf32_Half    e_shnum;
  Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

إليك مثال عن ترويسة ELF كما هو موضح باستخدام الأداة `readelf`:

```
$ readelf --header /bin/ls

ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           PowerPC
  Version:                           0x1
```

```
  Entry point address:               0x10002640
  Start of program headers:          52 (bytes into file)
  Start of section headers:          87460 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28

  [...]
```

يوضّح المثال السابق نموذجًا سهل القراءة على الإنسان كما مولدًا باستخدام برنامج `readelf`، وهو جزء من أدوات Binutils في GNU.

تُوجَد المصفوفة `e_ident` في بداية أيّ ملف ELF، وتبدأ دائمًا بمجموعة بايتات سحرية. البايت الأول هو `0x7F` ثم الثلاثة بايتات التالية هي "ELF". يمكنك فحص ملف ELF الثنائي لترى ذلك بنفسك باستخدام الأمر `hexdump`.

يفحص المثال التالي عدد ELF السحري:

```
ianw@mingus:~$ hexdump -C /bin/ls | more
7f 45 4c 46 01 02 01 00  00 00 00 00 00 00 00 00  |.ELF............|

... (يتبع ذلك بقية البرنامج) …
```

لاحظ وجود البت `0x7F` في البداية ثم سلسلة آسكي ASCII المُشفَّرة "ELF". ألقِ نظرة على المعيار وشاهد ما تعرّفه بقية المصفوفة وما هي القيم الموجودة في الملف الثنائي. لدينا بعد ذلك بعض الرايات Flags لنوع الجهاز الذي أُنشِئ هذا الملف الثنائي من أجله. لاحظ أن صيغة ELF تعرّف إصدارات مختلفة من الأحجام مثل إصدارات 32 بت و64 بت، حيث سنشرح هنا الإصدار 32 بت. يكمن الاختلاف في أنه يجب الاحتفاظ بالعناوين على أجهزة 64 بت في متغيرات بحجم 64 بت. يمكننا أن نرى أن الملف الثنائي أُنشِئ للجهاز الذي يستخدم صيغة Big Endian (تخزين البتات الأقل أهمية أولًا)، حيث يستخدم هذا الجهاز المكمل الثنائي لتمثيل الأعداد السالبة. لاحظ بعد ذلك أن الخاصية `Machine` تخبرنا أنه جهاز PowerPC الثنائي.

يبدو أن عنوان نقطة الدخول واضح وصريح بدرجة كافية، وهو العنوان الموجود في الذاكرة الذي تبدأ منه شيفرة البرنامج. يُقال لمبرمجي لغة C المبتدئين أن الدالة الرئيسية `main()` هي أول برنامج يُستدعَى في برامجهم، ولكن يمكننا التحقق من أنه ليس كذلك باستخدام عنوان نقطة الدخول كما يلي:

```
$ cat test.c
#include <stdio.h>

int main(void)
{
  printf("main is : %p\n", &main);
  return 0;
}

$ gcc -Wall -o test test.c

$ ./test
main is : 0x10000430

$ readelf --headers ./test | grep 'Entry point'
  Entry point address:               0x100002b0

$ objdump --disassemble ./test | grep 100002b0
100002b0 <_start>:
100002b0:       7c 29 0b 78     mr      r9,r1
```

لاحظ في المثال السابق أنه يمكننا أن نرى أن نقطة الدخول هي دالة تسمى `_start`. لم يعرّف برنامجنا هذه الدالة على الإطلاق، ويشير الخط السفلي في بداية اسم الدالة إلى أنها موجودة في فضاء أسماء منفصل. سنشرح لاحقًا كيفية بدء البرنامج بالتفصيل.

تحتوي الترويسة بعد ذلك على مؤشرات إلى المكان الموجود في الملف الذي تبدأ فيه الأجزاء المهمة الأخرى من ملف ELF مثل جدول المحتويات.

8.3.2 الرموز Symbols والمنقولات Relocation

توفر مواصفات ملف ELF جداول رموز Symbol Tables تربط بين السلاسل النصية أو الرموز ومواقع في الملف. تُعَد الرموز مطلوبة للربط Linking، فمثلًا يمكن أن يتطلب إسنادُ قيمة للمتغير `foo` المُصرَّح عنه بالشكل `extern int foo` رابطًا للعثور على عنوان المتغير `foo`، والذي يمكن أن يتضمن البحث عن الكلمة "foo" في جدول الرموز وإيجاد العنوان.

ترتبط المنقولات Relocations ارتباطًا وثيقًا بالرموز، حيث يُعَد الانتقال مساحةً فارغة تُترَك لإصلاحها لاحقًا، إذ لا يمكن استخدام المتغير `foo` في المثال السابق حتى معرفة عنوانه، ولكن نعلم في نظام 32 بت أن عنوان المتغير `foo` يجب أن يكون بقيمة 4 بايتات، لذلك يمكن للمصرِّف ببساطة ترك مساحة فارغة بمقدار 4 بايتات والاحتفاظ بانتقالٍ Relocation يخبر الرابط بأن يضع القيمة الحقيقية للمتغير `foo` في هذه المساحة التي مقدارها 4 بايتات في هذا العنوان في أيّ وقت يحتاج فيه المصرِّف استخدامَ هذا العنوان لإسناد قيمة مثلًا، ولكن يتطلب ذلك تحليل الرمز "foo".

8.3.3 المقاطع Segments والأقسام Sections

تحدد صيغة ELF عرضين لملف ELF، حيث يُستخدَم أحدهما للربط والآخر للتنفيذ، مما يوفر مرونة كبيرة لمصممي الأنظمة. سنتحدث عن الأقسام الموجودة في شيفرة الكائن التي تنتظر أن تُربَط بملف قابل للتنفيذ، ويُربَط قسم واحد أو أكثر مع مقطعٍ ما في الملف القابل للتنفيذ.

ا. المقاطع Segments

من الأسهل في بعض الأحيان النظر إلى المستوى الأعلى من التجريد abstraction المتمثل بالمقاطع قبل فحص الطبقات السفلية. يحتوي ملف ELF على ترويسة تصف تخطيط الملف العام، حيث تشير ترويسة ELF إلى مجموعة أخرى من الترويسات تسمى ترويسات البرامج Program Headers، حيث تصِف هذه الترويسات لنظام التشغيل أيّ شيء يمكن أن يكون مطلوبًا لتحميل الملف الثنائي في الذاكرة وتنفيذه. كما تصف ترويسات البرامج المقاطع، ولكن هناك بعض الأشياء الأخرى المطلوبة لتشغيل الملف القابل للتنفيذ.

إليك مثال عن ترويسة برنامج:

```
typedef struct {
  Elf32_Word p_type;
  Elf32_Off  p_offset;
  Elf32_Addr p_vaddr;
  Elf32_Addr p_paddr;
  Elf32_Word p_filesz;
  Elf32_Word p_memsz;
```

```
    Elf32_Word p_flags;
    Elf32_Word p_align;
} 
```

يوضح المثال السابق تعريف ترويسة البرنامج. لا بد أنك لاحظت من تعريف ترويسة ELF سابقًا وجود الحقول e_phoff وe_phnum وe_phentsize التي تمثّل الإزاحة في الملف حيث تبدأ ترويسات البرامج وعدد ترويسات البرامج الموجودة وحجم كل ترويسة برنامج، وبالتالي يمكنك العثور على ترويسات البرامج وقراءتها بسهولة باستخدام هذه الأجزاء الثلاثة من المعلومات.

تُعَد ترويسات البرامج أكثر من مجرد مقاطع، حيث يعرّف الحقل p_type ما تعرّفه ترويسة البرنامج، فمثلًا إذا كان هذا الحقل هو PT_INTERP، فستُعرَّف الترويسة بأنها مؤشر سلسلة نصية يؤشّر إلى مفسّر Interpreter الملف الثنائي. ناقشنا سابقًا الفرق بين اللغات المُصرَّفة Compiled واللغات المُفسَّرة Interpreted وميّزنا المُصرِّف بأنه ينشئ ملفًا ثنائيًا يمكن تشغيله بطريقة مستقلة. لكن لا بد أنك تتساءل عن سبب حاجتنا لمفسّر! حسنًا، ترغب الأنظمة الحديثة في المرونة عند تحميل الملفات القابلة للتنفيذ، لذا لا يمكن الحصول على بعض المعلومات بصورة كافية إلّا في الوقت الفعلي الذي يُعَد فيه البرنامج للتشغيل، وهذا ما يسمى بالربط الديناميكي Dynamic Linking الذي سنتحدث عنه لاحقًا، وبالتالي يجب إجراء بعض التغييرات الطفيفة على البرنامج الثنائي للسماح له بالعمل بصورة صحيحة في وقت التشغيل. لذا يُعَد مفسّر الملف الثنائي المعتاد هو المحمّل الديناميكي Dynamic Loader، لأنه يأخذ الخطوات النهائية لإكمال تحميل الملف القابل للتنفيذ وإعداد الصورة الثنائية للتشغيل.

تصف القيمة PT_LOAD في الحقل p_type المقاطع، ثم تصف الحقول الأخرى في ترويسة البرنامج كلّ مقطع منها. يخبرك الحقل p_offset بمقدار بُعد بيانات المقطع عن الملف الموجود على القرص الصلب. بينما يخبرك الحقل p_vaddr بالعنوان الذي يجب أن توجد عنده البيانات في الذاكرة الوهمية Virtual Memory، حيث يصف الحقل p_addr العنوان الحقيقي Physical Address الذي يُعَد مفيدًا للأنظمة المدمَجة الصغيرة التي لا تطبّق الذاكرة الوهمية. تخبرك الرايتان p_filesz و p_memsz بحجم المقطع الموجود على القرص الصلب وكم يجب أن يكون حجمه في الذاكرة. إذا كان حجم الذاكرة أكبر من حجم القرص الصلب، فيجب ملء التداخل بينهما بالأصفار، وبالتالي يمكنك توفير مساحة كبيرة في ملفاتك الثنائية من خلال عدم الاضطرار إلى هدر مساحة للمتغيرات العامة الفارغة. أخيرًا، يشير الحقل p_flags إلى أذونات المقطع، حيث يمكن تحديد أذونات التنفيذ والقراءة والكتابة، فمثلًا يجب تمييز مقاطع الشيفرة البرمجية بأنها للقراءة والتنفيذ فقط، وتمييز أقسام البيانات للقراءة والكتابة فقط بدون تنفيذ.

هناك عدد من أنواع المقاطع الأخرى المُعرَّفة في ترويسات البرامج الموصوفة كاملةً في مواصفات المعايير.

ب. الأقسام Sections

تشكّل الأقسام مقاطعًا، حيث تُعَد الأقسام طريقة لتنظيم الملف الثنائي في مناطق منطقية لتوصيل المعلومات بين المصرِّف والرابط. تُستخدَم الأقسام في بعض الملفات الثنائية الخاصة مثل نواة لينكس Linux Kernel بطرق أكثر تحديدًا سنوضحها لاحقًا.

رأينا كيف تصل المقاطع في النهاية إلى كتلة بيانات في ملف على القرص الصلب مع بعض المواصفات حول المكان الذي يجب تحميلها فيه والأذونات التي تمتلكها. تمتلك الأقسام ترويسةً مماثلة لترويسة المقاطع كما هو موضح في المثال التالي:

```
typedef struct {
  Elf32_Word sh_name;
  Elf32_Word sh_type;
  Elf32_Word sh_flags;
  Elf32_Addr sh_addr;
  Elf32_Off  sh_offset;
  Elf32_Word sh_size;
  Elf32_Word sh_link;
  Elf32_Word sh_info;
  Elf32_Word sh_addralign;
  Elf32_Word sh_entsize;
}
```

تحتوي الأقسام على عدد من الأنواع المُعرَّفة للحقل `sh_type` مثل تعريف قسم من النوع `SH_PROGBITS` بوصفه قسمًا يحتوي على بيانات ثنائية يستخدمها البرنامج. تشير الرايات الأخرى إلى ما إذا كان هذا القسم جدولَ رموز يستخدمه الرابط أو منقح الأخطاء مثلًا أو يمكن أن يكون شيئًا ما خاصًا بالمحمّل الديناميكي. كما توجد سمات إضافية مثل سمة التخصيص Allocate التي تشير إلى أن هذا القسم سيحتاج إلى ذاكرة مخصصة له.

سنختبر الآن البرنامج الموضح في المثال التالي:

```
#include <stdio.h>

int big_big_array[10*1024*1024];

char *a_string = "Hello, World!";
```

```
int a_var_with_value = 0x100;

int main(void)
{
  big_big_array[0] = 100;
  printf("%s\n", a_string);
  a_var_with_value += 20;
}
```

يوضح المثال التالي خرج الأداة `readelf` مـع بعض الأجـزاء الأخـرى، حيث يمكننـا باسـتخدام هـذا الخـرج تحليل كل جزء من برنامجنا البسيط السابق ومعرفة ما سيحدث به في خرج الملف الثنائي النهائي:

```
$ readelf --all ./sections
ELF Header:
  ...
    Size of section headers:           40 (bytes)
    Number of section headers:         37
    Section header string table index: 34

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf
Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0
0
  [ 1] .interp           PROGBITS        10000114 000114 00000d 00   A  0   0
1
  [ 2] .note.ABI-tag     NOTE            10000124 000124 000020 00   A  0   0
4
  [ 3] .hash             HASH            10000144 000144 00002c 04   A  4   0
4
  [ 4] .dynsym           DYNSYM          10000170 000170 000060 10   A  5   1
4
  [ 5] .dynstr           STRTAB          100001d0 0001d0 00005e 00   A  0   0
1
  [ 6] .gnu.version      VERSYM          1000022e 00022e 00000c 02   A  4   0
2
  [ 7] .gnu.version_r    VERNEED         1000023c 00023c 000020 00   A  5   1
4
  [ 8] .rela.dyn         RELA            1000025c 00025c 00000c 0c   A  4   0
4
  [ 9] .rela.plt         RELA            10000268 000268 000018 0c   A  4  25
4
```

```
  [10] .init             PROGBITS        10000280 000280 000028 00  AX  0   0
4
  [11] .text             PROGBITS        100002b0 0002b0 000560 00  AX  0   0
16
  [12] .fini             PROGBITS        10000810 000810 000020 00  AX  0   0
4
  [13] .rodata           PROGBITS        10000830 000830 000024 00   A  0   0
4
  [14] .sdata2           PROGBITS        10000854 000854 000000 00   A  0   0
4
  [15] .eh_frame         PROGBITS        10000854 000854 000004 00   A  0   0
4
  [16] .ctors            PROGBITS        10010858 000858 000008 00  WA  0   0
4
  [17] .dtors            PROGBITS        10010860 000860 000008 00  WA  0   0
4
  [18] .jcr              PROGBITS        10010868 000868 000004 00  WA  0   0
4
  [19] .got2             PROGBITS        1001086c 00086c 000010 00  WA  0   0
1
  [20] .dynamic          DYNAMIC         1001087c 00087c 0000c8 08  WA  5   0
4
  [21] .data             PROGBITS        10010944 000944 000008 00  WA  0   0
4
  [22] .got              PROGBITS        1001094c 00094c 000014 04 WAX  0   0
4
  [23] .sdata            PROGBITS        10010960 000960 000008 00  WA  0   0
4
  [24] .sbss             NOBITS          10010968 000968 000000 00  WA  0   0
1
  [25] .plt              NOBITS          10010968 000968 000060 00 WAX  0   0
4
  [26] .bss              NOBITS          100109c8 000968 2800004 00  WA  0   0
4
  [27] .comment          PROGBITS        00000000 000968 00018f 00      0   0
1
  [28] .debug_aranges    PROGBITS        00000000 000af8 000078 00      0   0
8
  [29] .debug_pubnames   PROGBITS        00000000 000b70 000025 00      0   0
1
  [30] .debug_info       PROGBITS        00000000 000b95 0002e5 00      0   0
1
  [31] .debug_abbrev     PROGBITS        00000000 000e7a 000076 00      0   0
1
  [32] .debug_line       PROGBITS        00000000 000ef0 0001de 00      0   0
1
```

```
  [33] .debug_str        PROGBITS        00000000 0010ce 0000f0 01  MS  0   0
1
  [34] .shstrtab         STRTAB          00000000 0011be 00013b 00      0   0
1
  [35] .symtab           SYMTAB          00000000 0018c4 000c90 10     36  65
4
  [36] .strtab           STRTAB          00000000 002554 000909 00      0   0
1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.
 ...

Symbol table '.symtab' contains 201 entries:
  Num:    Value  Size Type    Bind   Vis      Ndx Name
...
100109cc 0x2800000 OBJECT  GLOBAL DEFAULT   26 big_big_array
...
10010960     4 OBJECT  GLOBAL DEFAULT   23 a_string
...
10010964     4 OBJECT  GLOBAL DEFAULT   23 a_var_with_value
...
10000430    96 FUNC    GLOBAL DEFAULT   11 main
```

لنلقِ أولًا نظرة على المتغير `big_big_array` الذي -كما يوحي الاسم- هو مصفوفة عامة كبيرة إلى حد مـا، وإذا انتقلنا إلى جدول الرموز، فيمكننا أن نرى أن هـذا المتغـير موجـود في الموقـع `0x100109cc` الـذي يمكننـا ربطه بالقسم `.bss` في قائمة الأقسام لأنه يبدأ تحته مباشرةً عند الموقع `0x100109c8`، ولاحظ حجمه الكبير.

ذكرنا أن القسم BSS هو جزء معياري من صورة ثنائية، لأنـه ليس منطقيًـا أن تطلب أن يكـون لملـفٍ ثنـائي على القرص الصلب 10 ميجابايتات من المساحة المخصَّصة له عندما تكـون كـل هـذه المسـاحة قيمًـا صـفرية. لاحظ أن هذا القسم يحتوي على النوع `NOBITS`، مما يعني أنه لا يحتوي على أيّ بايت على القرص الصـلب. لـذا يُعرَّف القسم `.bss` للمتغيرات العامة التي يجب أن تكون قيمتها صفرًا عند بـدء البرنـامج. رأينـا كيـف يمكن أن يختلف حجم الذاكرة عن حجم القرص الصلب عند مناقشتنا للمقاطع، فوجود المتغيرات في القسم `.bss` دليـل على أنها ستُعطَى قيمة صفرية عند بدء البرنامج.

يوجد المتغير `a_string` في القسم `.sdata` الذي يمثّل البيانات الصغيرة Small Data، حيث يُعَد هـذا القسـم وقسـم `.sbss` المقابـل لـه أقسـامًا متـوفرة في بعض المعماريـات حيث يمكن الوصـول إلى البيانـات

باستخدام الإزاحة عن بعض المؤشرات المعروفة، وهذا يعني أنه يمكن إضافة قيمة ثابتة إلى العنـوان الأساسـي، مما يجعل الوصول إلى البيانات في الأقسام أسر ع نظـرًا لعـدم وجـود عمليـات بحث مطلوبـة إضـافية وتحميـل للعنـاوين في الـذاكرة. تقتصـر معظم المعماريـات على حجم القيم الفوريـة Immediate Value الـتي يمكنـك إضافتها إلى المسجل مثل القيمة الفورية 70 عند تطبيق التعليمـة `r1 = add r2, 70;` على عكس جمـع قيمتين مخزنتين في مسجلين `r1 = add r2,r3`، وبالتالي يمكن تطبيق إزاحة بمقدار مسافة صغيرة معينة عن العنوان. يمكننا أيضًا أن نرى أن المتغير `a_var_with_value` يوجد في المكان نفسه.

بينما توجد الدالة الرئيسية `main` في القسم `.text`. تذكر أن "النص Text" و"الشيفرة Code" يُسـتخدَمان للإشارة إلى برنامج في الذاكرة.

ج. الأقسام والمقاطع مع بعضها بعضًا

إليك مثال يحتوي على الأقسام والمقاطع مع بعضها بعضًا:

```
$ readelf --segments /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x100026c0
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x10000034 0x10000034 0x00100 0x00100 R E 0x4
  INTERP         0x000154 0x10000154 0x10000154 0x0000d 0x0000d R   0x1
    [Requesting program interpreter: /lib/ld.so.1]
  LOAD           0x000000 0x10000000 0x10000000 0x14d5c 0x14d5c R E 0x10000
  LOAD           0x014d60 0x10024d60 0x10024d60 0x002b0 0x00b7c RWE 0x10000
  DYNAMIC        0x014f00 0x10024f00 0x10024f00 0x000d8 0x000d8 RW  0x4
  NOTE           0x000164 0x10000164 0x10000164 0x00020 0x00020 R   0x4
  GNU_EH_FRAME   0x014d30 0x10014d30 0x10014d30 0x0002c 0x0002c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4

 Section to Segment mapping:
  Segment Sections...
    00
.interp
.interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_ r
.rela.dyn .rela.plt .init .text .fini .rodata .eh_frame_hdr
.data .eh_frame .got2 .dynamic .ctors .dtors .jcr .got .sdata .sbss .p lt .bss
```

```
.dynamic
.note.ABI-tag
.eh_frame_hdr
    07
```

يوضح المثال السابق كيف تظهِر الأداة `readelf` ربط المقاطع والأقسام في ملف ELF مع الملف الثنـائي `./bin/ls`.

انتقل إلى نهاية الخرج حيث يمكننا أن نرى الأقسام المنقولة إلى المقـاطع، فمثلًا يُوضَـع القسـم `.interp` في المقطع الذي له الرايـة `INTERP`. لاحـظ أن الأداة `readelf` تخبرنـا بطلب المفسّـر `/lib/ld.so.1`، وهـو الرابط الديناميكي الذي يُشغَّل لإعداد الملف الثنائي للتنفيذ.

يمكننا أن نرى الفرق بين النص والبيانات بالنظر إلى مقطعي `LOAD`. لاحـظ أن المقطـع الأول لديـه أذونـات القراءة والتنفيذ فقط، بينما يكون للمقطع الآخر أذونات القراءة والكتابة والتنفيذ، أي أن مقطع الشيفرة له أذونات القراءة والكتابة (r/w) ومقطع البيانـات لـه أذونـات القـراءة والكتابـة والتنفيـذ (r/w/e)، ولكن لا يجب أن تكـون البيانات قابلة للتنفيذ. لن يُميَّز قسم البيانات في معظم المعماريات مثل المعمارية x86 الأكـثر شـيوعًا على أنـه يحتوي على قسم بيانات قابل للتنفيذ. لكن المثال السابق مأخوذ من معمارية PowerPC التي لها نموذج برمجة مختلف قليلًا وهو واجهة التطبيق الثنائية Application Binary Interface -أو ABI اختصارًا- التي تتطلب أن يكون قسم البيانات قابلًا للتنفيذ. هذه هي حياة مبرمج الأنظمة، إذ وُضِعت القواعد لكسرها.

تستدعي واجهة ABI في معمارية PowerPC شيفرات اختباريـة Stubs للـدوال في المكتبـات الديناميكيـة مباشرةً في جدول الإزاحة العام Global Offset Table -أو GOT اختصارًا- بـدلًا من جعلهـا ترتـد بين مـدخلات منفصلة من جدول PLT، وبالتالي يحتاج المعالج إلى أذونات تنفيذ للقسم GOT الذي يمكنك رؤيتـه مضـمَّنًا في مقطع البيانات.

الشيء الآخر الذي يجب ملاحظته هو أن حجم الملف هـو حجم الـذاكرة نفسـه لمقطـع الشـيفرة، ولكن حجم الذاكرة أكبر من حجم ملف مقطع البيانات، ويأتي ذلك من القسم BSS الذي يحتوي على متغيرات عامة صفرية.

8.4 واجهات ABI

تُعَد واجهة ABI مصطلحًا ستسمع عنه كثيرًا عند العمل مع برمجة الأنظمة، وهو مختلـف عن مصـطلح API الذي يُعَد واجهات يراها المبرمج في شيفرته البرمجية. تشـير ABI إلى واجهـات المسـتوى الأدنى الـتي يجب أن يتفق عليها المصرِّف ونظام التشغيل والمعالج إلى حد ما للتواصل مع بعضها بعضًا. سنقدم فيمـا يلي عـددًا من المفاهيم المهمة لفهم واجهات ABI.

8.4.1 ترتيب البايتات

تُرتَّب البايتات باستخدام ترتيب Endianess الذي يحتوي على نوعين هما: Big-endian أي تخزين البتـات الأقل أهمية أولًا، و Little-endian أي تخزين البتات الأكثر أهمية أولًا.

8.4.2 العرف المتبع في الاستدعاءات

يمكن تنفيذ الاستدعاءات بطريقتين هما: تمرير المعـاملات باسـتخدام المسـجلات registers أو المكـدس stack وواصفات الدوال.

بخصوص واصفات الدوال، لا تُستدعَى الدالة في العديد من المعماريات مباشرةً، بل تُستدعَى عـبر واصـف دالة Function Descriptor. يتكون واصف الدالـة في المعماريـة IA64 مثلًا من مكـونين همـا: عنـوان الدالـة (يُمثَّل بقيمة مقدارها 64 بت أو 8 بايتات) وعنوان المؤشر العام Global Pointer أو gp اختصارًا. تحدد واجهـة ABI أن المسـجل `r1` يجب أن يحتـوي دائمًـا على قيمـة المؤشـر gp الخـاص بالدالـة، وهـذا يعـني أن مهمـة المستدعي عند استدعاء دالة هي حفظ قيمة المؤشر gp الخاصة به وضبط المسجل r1 على القيمة الجديدة من واصف الدالة ثم استدعاء هذه الدالة.

تُعَد واصفات الدوال مفيدة للغاية كما سترى لاحقًـا. يمكن أن تأخـذ تعليمـة الجمـع `add` في المعـالج IA64 قيمة فورية ذات حجم بحد أقصى 22 بت بسبب الطريقة التي يحزُم بها المعالج IA64 التعليمـات، حيث تُوضَـع ثلاثة تعليمات في كل حزمة، ولا يوجد سـوى مسـاحة كافيـة للاحتفـاظ بقيمـة 22 بت للحفـاظ على الحزمـة مـع بعضها البعض. القيمة الفورية Immediate Value هي القيمة المحـددة مباشـرةً وليس القيمـة الموجـودة في المسجل، إذ تُعَد القيمة 100 في التعليمة `add r1 + 100` هي القيمة الفورية.

يمكن أن تتمكن 22 بت من تمثيل 4194304 بايت أو 4 ميجابايتات، وبالتالي يمكن إزاحة كل دالة مباشـرة في حيّز ذاكرة كبير مقـداره 4 ميجابايتـات دون الحاجـة إلى تحمـل عنـاء تحميـل أيّ قيمٍ في المسـجل. إذا اتفـق المصرِّف والرابط والمحمِّل على ما يشير إليه المؤشر العام كما هو محدد في واجهـة ABI، فيمكن تحسـين الأداء من خلال تقليل عمليات التحميل.

8.5 المكتبات

لقد سئم المطورون من الاضطرار إلى كتابة كل شيء من البداية، لذلك كانت المكتبـات من أولى اختراعـات علوم الحاسوب، فالمكتبة هي ببساطة مجموعة من الدوال التي يمكنك استدعاؤها من برنامجك. تتمتع المكتبـة بالعديد من المزايا مثل أنه يمكنك توفير الكثير من الوقت عن طريق إعادة استخدام العمل الـذي أنجـزه شـخص آخر، وتكون أكثر ثقة في أنها تحتوي على أخطاء أقل بسـبب وجـود أشـخاص آخـرين اسـتخدموا هـذه المكتبـات مسبقًا، وبالتالي ستستفيد من عثـورهم على الأخطـاء وإصـلاحها. تشـبه المكتبـة الملـف القابـل للتنفيـذ تمامًـا باستثناء استدعاء دوال المكتبة باستخدام معاملات من ملفك القابل للتنفيذ بدلًا من تشغيلها مباشرةً.

8.5.1 المكتبات الساكنة Static Libraries

الطريقة الأكثر مباشرة لاستخدام دالة المكتبة هي ربط ملفات الكائنات من المكتبة مباشرة بملفك النهائي القابل للتنفيذ كما هو الحال مع تلك الملفات التي صرَّفتها بنفسك، وعندها تُسمَّى المكتبة مكتبة ساكنة، لأن المكتبة ستبقى دون تغيير ما لم يُعاد تصريف البرنامج. تُعَد هذه الطريقة لاستخدام مكتبة الطريقة الأسهل لأن النتيجة النهائية هي ملف قابل للتنفيذ بسيط بدون اعتماديات.

تُعَد المكتبة الساكنة مجموعةً من ملفات الكائنات، حيث يُحتفَظ بملفات الكائنات في سجل Archive، مما يؤدي إلى استخدام لاحقتها المعتادة `.a`. يمكنك التفكير في هذه السجلات بوصفها ملفًا مضغوطًا ولكن بدون ضغط. يوضّح المثال التالي كيفية إنشاء مكتبة ساكنة بسيطة ويقدم بعض الأدوات الشائعة للتعامل مع المكتبات:

```
$ cat library.c
/* دالة مكتبة */
int function(int input)
{
  return input + 10;
}

$ cat library.h
/* تعريف الدالة */
int function(int);

$ cat program.c
#include <stdio.h>
/* ترويسة ملف المكتبة */
#include "library.h"

int main(void)
{
  int d = function(100);

  printf("%d\n", d);
}

$ gcc -c library.c
```

```
$ ar rc libtest.a library.o
$ ranlib ./libtest.a
$ nm --print-armap ./libtest.a

Archive index:
function in library.o

library.o:
T function

$ gcc -L . program.c -ltest -o program

$ ./program
110
```

أولًا، نصرّف مكتبتنا إلى ملف كائن كما رأينا سابقًا. لاحظ أننا نحدد واجهة API الخاصة بالمكتبة في ترويسة الملف، حيث تتكون واجهة API من تعريفات الدوال الموجودة في المكتبة حتى يعرف المُصرِّف أنواع الدوال عند إنشاء ملفات الكائنات التي تشير إلى المكتبة مثل الملف `program.c` الذي يُضمَّن باستخدام `#include` في ترويسة الملف.

ننشئ سجل مكتبة باستخدام الأمر `ar` الذي يمثل اختصارًا للكلمة "سجل Archive". تُسبَق أسماء ملفات المكتبة الساكنة بالبادئة `lib` ويكون لها اللاحقة `.a` حسب العرف المتَّبع. يخبر الوسيطُ `c` البرنامجَ بإنشاء السجل Archive، ويخبر `a` السجل بإضافة ملفات الكائنات المحددة في ملف المكتبة. تنبثق السجلات المُنشَأة باستخدام الأمر `ar` في أماكن مختلفة من أنظمة لينكس بخلاف إنشاء مكتبات ساكنة. أحد التطبيقات المستخدمة على نطاق واسع هي التطبيقات المُستخدَمة في صيغة حزم `.deb` مع أنظمة دبيان Debian وأوبنتو Ubuntu وبعض أنظمة لينكس الأخرى، حيث تستخدم ملفات `deb` السجلات للاحتفاظ بجميع ملفات التطبيق مع بعضها بعضًا في ملف حزمة واحد. تستخدم حزم RPM صيغةً بديلةً ولكنها مشابهة لصيغة `deb` وتُسمَّى cpio. يُعَد ملف `tar` التطبيقَ الأساسي لحفظ الملفات مع بعضها بعضًا، وهو صيغة شائعة لتوزيع الشيفرة المصدرية.

نستخدم بعد ذلك تطبيق `ranlib` لإنشاء ترويسة في المكتبة باستخدام رموز محتويات ملف الكائن، مما يساعد المصرِّف على الإشارة إلى الرموز بسرعة، إذ يمكن أن تبدو هذه الخطوة زائدة في حالة وجود رمز واحد فقط ، ولكن يمكن أن تحتوي مكتبة كبيرة على آلاف الرموز مما يعني أن الفهرس يمكن أن يسار ع بصورة كبيرة في العثور على المراجع. نفحص هذه الترويسة الجديدة باستخدام تطبيق `nm`. لاحظ وجود الرمز `function` الخاص بالدالة `function()` عند إزاحة بمقدار صفر كما هو متوقع.

يمكنك بعد ذلك تحديد المكتبة للمصرِّف باستخدام الخيار `-lname` حيث يكون الاسم هو اسم ملف المكتبة بدون البادئة `lib`. كما نوفر مجلد بحث إضافي للمكتبات وهو المجلد الحالي (`-L .`)، لأنه لا يمكن البحث عن المكتبات في المجلد الحالي افتراضيًا. النتيجة النهائية هي ملف قابل للتنفيذ مع المكتبة الجديدة المُضمَّنة.

ا. عيوب الربط الساكن

يُعَد الربط الساكن أمرًا سهلًا للغاية، ولكن له عدد من العيوب، فهناك نوعان من العيوب الرئيسية أولهما أنه يجب عليك إعادة تصريف برنامجك إلى ملف تنفيذي جديد عند تحديث شيفرة المكتبة لإصلاح خطأ مثلًا، وثانيهما احتواء كل برنامج يستخدم تلك المكتبة في النظام على نسخة في ملفه القابل للتنفيذ. يُعَد ذلك غير فعال وخاصة إذا وجدت خطأ واضطررت إلى إعادة تصريفه.

تُضمَّن مكتبة C التي هي glibc مثلًا في جميع البرامج، وتوفر جميع الدوال الشائعة مثل `printf`.

8.5.2 المكتبات المشتركة

تُعَد المكتبات المشتركة طريقةً للتغلب على المشاكل التي تشكّلها المكتبات الساكنة. تُحمَّل المكتبة المشتركة ديناميكيًا في وقت التشغيل لكل تطبيق يحتاجها، حيث يستخدم التطبيق مؤشرات تتطلب مكتبة معينة، وتُحمَّل المكتبة في الذاكرة وتُنفَّذ عند استدعاء الدالة. إن حُمِّلت المكتبة لتطبيق آخر، فيمكن مشاركة الشيفرة البرمجية بين التطبيقين، مما يوفر موارد كبيرة مع المكتبات شائعة الاستخدام.

يُعَد الربط الديناميكي الذي تحدثنا عنه سابقًا أحد الأجزاء الأكثر تعقيدًا في نظام التشغيل الحديث.

8.6 مفاهيم متقدمة متعلقة بصيغة ملفات ELF

تعرفنا في الفقرات السابقة على الملفات القابلة للتنفيذ في نظام التشغيل وتمثيلها باستخدام الصيغة ELF وسنوضح في الفقرات التالية بعض المفاهيم المتعلقة بصيغة ملفات ELF مثل تنقيح الأخطاء Debugging وكيفية إنشاء أقسام مخصصة فيها وسكربتات الرابط Linker Scripts التي يستخدمها الرابط لبناء الأقسام Sections المُكوِّنة للمقاطع Segments، ولكن لنتعرّف أولًا على مفهوم ملفات ELF القابلة للتنفيذ.

تُعَد الملفات القابلة للتنفيذ أحد الاستخدامات الأساسية لصيغة ELF. يحتوي الملف الثنائي على كل ما هو مطلوب لنظام التشغيل لتنفيذ الشيفرة البرمجية بالطريقة المطلوبة، حيث صُمِّم الملف التنفيذي لتشغيله في عملية ذات فضاء عناوين فريد، لذا يمكن للشيفرة البرمجية وضع افتراضات حول مكان تحميل أجزاء البرنامج المختلفة في الذاكرة.

يوضح المثال الآتي اختبار أجزاء ملفٍ قابل للتنفيذ باستخدام أداة `readelf`. يمكننا أن نرى العناوين الوهمية التي يجب وضع مقاطع `LOAD` فيها، حيث يمكننا أن نرى أنّ أحد هذه المقاطع مخصصٌ للشيفرة

البرمجية ويمتلك أذونات القراءة والتنفيذ فقط، وهناك مقطع آخر مخصّص للبيانات ولديه أذونات القراءة والكتابة دون وجود أذونات التنفيذ، فبدونها لن تُميَّز الصفحات التي تدعم خطأ ما بأن لها أذونات التنفيذ حتى إن سمح هذا الخطأ للمهاجم بإدخال بيانات عشوائية، وبالتالي لن تسمح معظم المعالجات بأيّ تنفيذ للشيفرة البرمجية في تلك الصفحات.

```
$ readelf --segments /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x4046d4
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R E    8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000019ef4 0x0000000000019ef4  R E    200000
  LOAD           0x000000000001a000 0x000000000061a000 0x000000000061a000
                 0x000000000000077c 0x0000000000001500  RW     200000
  DYNAMIC        0x000000000001a028 0x000000000061a028 0x000000000061a028
                 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x000000000000021c 0x000000000040021c 0x000000000040021c
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000017768 0x0000000000417768 0x0000000000417768
                 0x00000000000006fc 0x00000000000006fc  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8

Section to Segment mapping:
  Segment Sections...

.interp
.interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata
.eh_frame_hdr .eh_frame
```

```
.ctors .dtors .jcr .dynamic .got .got.plt .data .bss
.dynamic
.note.ABI-tag .note.gnu.build-id
.eh_frame_hdr
    07
```

يجب تحميل مقاطع البرنامج على هذه العناوين، حيث تتمثل الخطوة الأخيرة للرابط Linker في تحليل معظم المنقولات Relocations وتصحيحها باستخدام العناوين المطلقة المُفترَضة، ثم تجاهل البيانات التي تصف الانتقال في الملف الثنائي النهائي دون وجود طريقة لإيجاد هذه المعلومات بعد الآن.

تحتوي الملفات القابلة للتنفيذ عمومًا على اعتماديات Dependencies خارجية للمكتبات المشتركة Shared Libraries أو أجزاء من الشيفرة البرمجية المشتركة يمكن تجريدها ومشاركتها بين أجزاء النظام بأكمله، حيث تتعلق جميع الأجزاء الغريبة في المثال السابق باستخدام المكتبات المشتركة التي سنوضّحها لاحقًا.

8.6.1 تنقيح الأخطاء Debugging

يُشار تقليديًا إلى الطريقة الأساسية لتنقيح أخطاء ما بعد التعطل باسم التفريغ الأساسي Core Dump، حيث يأتي مصطلح الأساسي Core من الخصائص الفيزيائية الأصلية للذاكرة المغناطيسية الأساسية التي تستخدم اتجاه الحلقات المغناطيسية الصغيرة لتخزين الحالة. يُعَد التفريغ الأساسي لقطة كاملة للبرنامج عند عمله في وقت معين، ويمكن بعد ذلك استخدام منقح أخطاء Debugger لفحص هذا التفريغ وإعادة بناء حالة البرنامج. يوضح المثال التالي نموذجًا لبرنامج يكتب في موقع ذاكرة عشوائية لغرض التعطل، حيث ستتوقف العمليات ويُسجَّل تفريغ للحالة الحالية:

```
$ cat coredump.c
int main(void) {
  char *foo = (char*)0x12345;
  *foo = 'a';

  return 0;
}

$ gcc -Wall -g -o coredump coredump.c

$ ./coredump
Segmentation fault (core dumped)

$ file ./core
```

```
./core: ELF 32-bit LSB core file Intel 80386, version 1 (SYSV), SVR4-
style, from './coredump'


$ gdb ./coredump
...
(gdb) core core
[New LWP 31614]
Core was generated by `./coredump'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483c4 in main () at coredump.c:3
*foo = 'a';
(gdb)
```

وبالتالي فإن ملف التفريغ الأساسي هو مجرد ملف ELF يحتوي على مجموعة من الأقسام التي يفهمها منقح الأخطاء لتمثيل أجزاء من البرنامج المُشغَّل.

8.6.2 الرموز ومعلومات تنقيح الأخطاء

يتطلب منقح الأخطاء gdb الملف القابل للتنفيذ الأصلي وملف التفريغ الأساسي لإعادة بناء بيئة جلسة تنقيح الأخطاء. لاحظ أن الملف القابل للتنفيذ الأصلي أُنشئ باستخدام الراية `-g` التي توجه المصرِّف Compiler لتضمين جميع معلومات الأخطاء، حيث يجري الاحتفاظ بالمعلومات المتعلقة بعملية تنقيح الأخطاء الإضافية في أقسام خاصة من ملف ELF، إذ تصف هذه المعلومات بالتفصيل أشياءً مثل قيم المسجّل التي تحتوي حاليًا على المتغيرات المستخدمة في الشيفرة البرمجية وحجم المتغيرات وطول المصفوفات وغير ذلك. تكون هذه المعلومات بصيغة DWARF المعيارية التي تُعَد مرادفًا لصيغة ELF تقريبًا.

يمكن أن يؤدي تضمين معلومات تنقيح الأخطاء إلى جعل الملفات والمكتبات القابلة للتنفيذ كبيرة جدًا، إذ لا تزال تشغل مساحة كبيرة على القرص الصلب بالرغم من أنها ليست مطلوبة في الذاكرة للتشغيل الفعلي، وبالتالي يجب إزالة هذه المعلومات من ملف ELF. يمكن نقل كل من الملفات التي أُزيلت منها هذه المعلومات والملفات التي لم تُزال منها هذه المعلومات، ولكن توفّر معظم طرق توزيع أو نشر الملفات الثنائية binary distribution الحالية معلومات لتنقيح الأخطاء في ملفات منفصلة. يمكن استخدام أداة objcopy لاستخراج معلومات تنقيح الأخطاء (`--only-keep-debug`) ثم إضافة رابط في الملف القابل للتنفيذ الأصلي إلى هذه المعلومات المُزالَة (`--add-gnu-debuglink`)، ثم سيكون هناك قسم خاص بالاسم `.gnu_debuglink` موجود في الملف القابل للتنفيذ الأصلي ويحتوي على قيمة فريدة بحيث يمكن لمنقح الأخطاء عند بدء جلسات تنقيح الأخطاء التأكدَ من أنه يربط معلومات تنقيح الأخطاء الصحيحة بالملف التنفيذي الصحيح.

يوضح المثال التالي إزالة معلومات تنقيح الأخطاء إلى ملفات منفصلة باستخدام الأداة objcopy:

```
$ gcc -g -shared -o libtest.so libtest.c
$ objcopy --only-keep-debug libtest.so libtest.debug
$ objcopy --add-gnu-debuglink=libtest.debug libtest.so
$ objdump -s -j .gnu_debuglink libtest.so

libtest.so:     file format elf32-i386

Contents of section .gnu_debuglink:
6c696274 6573742e 64656275 67000000  libtest.debug...
52a7fd0a                             R...
```

تشغل الرموز مساحة أقل بكثير، ولكنها تُعَد هدفًا للإزالة من الخرج النهائي، إذ لن تكـون هنـاك حاجـة لبقـاء معظم الرموز بمجرد ربط ملفات التعليمات المُصرَّفة object files لملف قابل للتنفيـذ بالصـورة النهائيـة. تُعَـد الرموز مطلوبة لإصلاح مدخلات المنقولات Relocation، ولكن لن تكون الرموز بعد ذلك ضرورية تمامًا لتشـغيل البرنامج النهائي. توفر سلسلة أدوات GNU لتجريد البرنـامج في نظـام لينكس خيـاراتٍ لإزالـة الرمـوز. لاحـظ أنـه يجب تحليل بعض الرموز في وقت التشغيل (للربط الديناميكي Dynamic Linking)، ولكنها تُوضَع في جداول رموز ديناميكية منفصلة حتى لا تُزال وتجعل الخرج النهائي عديم الفائدة.

8.6.3 التفريغ الأساسي Coredump

يُعَد التفريغ الأساسي مجرد ملف ELF. يوضح المثال التالي مرونة صيغة ELF بوصفها صيغة ثنائية:

```
$ readelf --all ./core
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              CORE (Core file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
```

```
  Size of program headers:           32 (bytes)
  Number of program headers:         15
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0

There are no sections in this file.

There are no sections to group in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NOTE           0x000214 0x00000000 0x00000000 0x0022c 0x00000     0
  LOAD           0x001000 0x08048000 0x00000000 0x01000 0x01000 R E 0x1000
  LOAD           0x002000 0x08049000 0x00000000 0x01000 0x01000 RW  0x1000
  LOAD           0x003000 0x489fc000 0x00000000 0x01000 0x1b000 R E 0x1000
  LOAD           0x004000 0x48a17000 0x00000000 0x01000 0x01000 R   0x1000
  LOAD           0x005000 0x48a18000 0x00000000 0x01000 0x01000 RW  0x1000
  LOAD           0x006000 0x48a1f000 0x00000000 0x01000 0x153000 R E 0x1000
  LOAD           0x007000 0x48b72000 0x00000000 0x00000 0x01000     0x1000
  LOAD           0x007000 0x48b73000 0x00000000 0x02000 0x02000 R   0x1000
  LOAD           0x009000 0x48b75000 0x00000000 0x01000 0x01000 RW  0x1000
  LOAD           0x00a000 0x48b76000 0x00000000 0x03000 0x03000 RW  0x1000
  LOAD           0x00d000 0xb771c000 0x00000000 0x01000 0x01000 RW  0x1000
  LOAD           0x00e000 0xb774d000 0x00000000 0x02000 0x02000 RW  0x1000
  LOAD           0x010000 0xb774f000 0x00000000 0x01000 0x01000 R E 0x1000
  LOAD           0x011000 0xbfeac000 0x00000000 0x22000 0x22000 RW  0x1000

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.

No version information found in this file.

Notes at offset 0x00000214 with length 0x0000022c:
  Owner                 Data size   Description
  CORE                  0x00000090   NT_PRSTATUS (prstatus structure)
```

```
  CORE                 0x0000007c   NT_PRPSINFO (prpsinfo structure)
  CORE                 0x000000a0   NT_AUXV (auxiliary vector)
  LINUX                0x00000030   Unknown note type: (0x00000200)

$ eu-readelf -n ./core

Note segment of 556 bytes at offset 0x214:
  Owner          Data size  Type
  CORE                 144  PRSTATUS
    info.si_signo: 11, info.si_code: 0, info.si_errno: 0, cursig: 11
    sigpend: <>
    sighold: <>
    pid: 31614, ppid: 31544, pgrp: 31614, sid: 31544
    utime: 0.000000, stime: 0.000000, cutime: 0.000000, cstime: 0.000000
    orig_eax: -1, fpvalid: 0
    ebx:     1219973108  ecx:     1243440144  edx:              1
    esi:              0  edi:              0  ebp:     0xbfecb828
    eax:          74565  eip:     0x080483c4  eflags:  0x00010286
    esp:     0xbfecb818
    ds: 0x007b  es: 0x007b  fs: 0x0000  gs: 0x0033  cs: 0x0073  ss: 0x007b
  CORE                 124  PRPSINFO
    state: 0, sname: R, zomb: 0, nice: 0, flag: 0x00400400
    uid: 1000, gid: 1000, pid: 31614, ppid: 31544, pgrp: 31614, sid: 31544
    fname: coredump, psargs: ./coredump
  CORE                 160  AUXV
    SYSINFO: 0xb774f414
    SYSINFO_EHDR: 0xb774f000
    HWCAP: 0xafe8fbff  <fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov clflush dts acpi mmx fxsr sse sse2 ss tm pbe>
    PAGESZ: 4096
    CLKTCK: 100
    PHDR: 0x8048034
    PHENT: 32
    PHNUM: 8
    BASE: 0
    FLAGS: 0
    ENTRY: 0x8048300
    UID: 1000
    EUID: 1000
```

```
    GID: 1000
    EGID: 1000
    SECURE: 0
    RANDOM: 0xbfecba1b
    EXECFN: 0xbfecdff1
    PLATFORM: 0xbfecba2b
    NULL
  LINUX                48  386_TLS
    index: 6, base: 0xb771c8d0, limit: 0x000fffff, flags: 0x00000051
    index: 7, base: 0x00000000, limit: 0x00000000, flags: 0x00000028
    index: 8, base: 0x00000000, limit: 0x00000000, flags: 0x00000028
```

يمكننا أن نرى في المثال السابق اختبارًا باستخدام أداة `readelf` أولًا للملف الأساسي الناتج عن مثال إنشاء ملف تفريغ أساسي واستخدامه باستخدام منقح الأخطاء gdb. لا توجد أقسام أو منقولات أو معلومات أخرى غريبة في هذا الملف يمكن أن تكون مطلوبة لتحميل ملف قابل للتنفيذ أو مكتبة، إذ يتكون من سلسلة من ترويسات البرامج التي تمثل مقاطع `LOAD` التي هي عمليات تفريغ بيانات أولية أنشأتها النواة Kernel لتخصيصات الذاكرة الحالية.

المكون الآخر لملف التفريغ الأساسي هو أقسام الملاحظات `NOTE` التي تحتوي على البيانات الضرورية لتنقيح الأخطاء ولكن ليس بالضرورة أن تُلتقَط في لقطة مباشرة لتخصيصات الذاكرة. يوفّر برنامج `eu-readelf` المُستخدَم في الجزء الثاني من المثال السابق رؤيةً أكثر اكتمالًا للبيانات من خلال فك تشفيرها.

تقدّم الملاحظة `PRSTATUS` مجموعة من المعلومات حول العملية أثناء تشغيلها، فمثلًا يمكننا أن نرى من قيمة `cursig` أن البرنامج تلقى إشارة قيمتها 11 تمثل خطأ تقطيع segmentation fault. كما تتضمن بالإضافة إلى معلومات رقم العملية ملف تفريغ لجميع المسجلات الحالية. يمكن لمنقح الأخطاء بالنظر إلى قيم المسجّل إعادة بناء حالة المكدس وبالتالي توفير تعقب خلفي Backtrace، حيث يمكن لمنقح الأخطاء

جنبًا إلى جنب مع الرمز ومعلومات تنقيح الأخطاء من الملف الثنائي الأصلي إظهار كيفية الوصول إلى نقطة التنفيذ الحالية.

من المخرجات الأخرى المتجه المساعد الحالي Auxiliary Vector أو `AUXV` اختصارًا. تصف `TLS_386` مدخلات جدول الواصفات العام المستخدمة في تقديم Implementation معمارية x86 لمخزن محلي قائم على الخيوط thread-local storage. سيكون هناك مدخلات مكررة لكل خيط قيد التشغيل بالنسبة للتطبيق متعدد الخيوط، حيث سيفهم منقح الأخطاء ذلك وهذه هي الطريقة التي يطبّق بها منقح الأخطاء gdb الأمر `thread` لإظهار الخيوط والتبديل بينها.

تنشئ النواة ملف التفريغ الأساسي ضمن حدود إعدادات `ulimit` الحالية، إذ يمكن أن يؤدي البرنامج الذي يستخدم قدرًا كبيرًا من الذاكرة إلى وجود ملف تفريغ كبير جدًا، ويُحتمَل أن يملأ القرص الصلب ويزيد المشاكل سوءًا، ويُضبَط `ulimit` على مستوى منخفض أو حتى على القيمة الصفر لأن معظم الأشخاص الذين ليسوا مطورين لديهم استخدام ضئيل لملف التفريغ الأساسي. لكن يظل التفريغ الأساسي هو الطريقة الأكثر فائدة لتنقيح أخطاء حالة غير متوقعة بعد التعطل.

8.6.4 إنشاء أقسام مخصصة

يُعَد تنظيم الشيفرة والبيانات والرموز شيئًا يمكن للمبرمج ترك إعدادات سلسلة أدواته الافتراضية كما هي، ولكن يمكن في بعض الأحيان توسيع أو تخصيص الأقسام ومحتوياتها مثل وحدات نواة لينكس التي تُستخدَم لتحميل المشغّلات والميزات الأخرى ديناميكيًا في نواة التشغيل. تُعَد هذه الوحدات غير قابلة للنقل، فهي تعمل فقط مع إصدار بناء نواة ثابت واحد، لذلك يمكن أن تكون الواجهة بين الوحدات والنواة مرنةً وغير مرتبطة بمعايير معينة، وبالتالي يمكن تعريف طرق تخزين مثل تخزين معلومات الترخيص والتأليف والاعتماديات والمعاملات الخاصة بالوحدات بصورة فريدة وكاملة باستخدام النواة.

يمكن لأداة `modinfo` فحص هذه المعلومات في وحدةٍ ما وتقديمها للمستخدم. يوضح المثال التالي استخدام مثال عن وحدة نواة لينكس `fuse` التي تسمح لمكتبات مساحة المستخدم Userspace بتوفير تقديمات نظام الملفات للنواة:

```
$ cd /lib/modules/$(uname -r)

$ sudo modinfo ./kernel/fs/fuse/fuse.ko
filename:       /lib/modules/3.2.0-4-amd64/./kernel/fs/fuse/fuse.ko
alias:          devname:fuse
alias:          char-major-10-229
license:        GPL
description:    Filesystem in Userspace
author:         Miklos Szeredi <miklos@szeredi.hu>
depends:
intree:         Y
vermagic:       3.2.0-4-amd64 SMP mod_unload modversions
parm:           max_user_bgreq:Global limit for the maximum number of
backgrounded requests an unprivileged user can set (uint)
parm:           max_user_congthresh:Global limit for the maximum congestion
threshold an unprivileged user can set (uint)

$ objdump -s -j .modinfo ./kernel/fs/fuse/fuse.ko
```

```
./kernel/fs/fuse/fuse.ko:     file format elf64-x86-64

Contents of section .modinfo:
616c6961 733d6465 766e616d 653a6675  alias=devname:fu
73650061 6c696173 3d636861 722d6d61  se.alias=char-ma
6a6f722d 31302d32 32390070 61726d3d  jor-10-229.parm=
6d61785f 75736572 5f636f6e 67746872  max_user_congthr
6573683a 476c6f62 616c206c 696d6974  esh:Global limit
20666f72 20746865 206d6178 696d756d   for the maximum
20636f6e 67657374 696f6e20 74687265   congestion thre
73686f6c 6420616e 20756e70 72697669  shold an unprivi
6c656765 64207573 65722063 616e2073  leged user can s
65740070 61726d74 7970653d 6d61785f  et.parmtype=max_
 00a0 75736572 5f636f6e 67746872 6573683a  user_congthresh:
 00b0 75696e74 00706172 6d3d6d61 785f7573  uint.parm=max_us
 00c0 65725f62 67726571 3a476c6f 62616c20  er_bgreq:Global
 00d0 6c696d69 7420666f 72207468 65206d61  limit for the ma
 00e0 78696d75 6d206e75 6d626572 206f6620  ximum number of
 00f0 6261636b 67726f75 6e646564 20726571  backgrounded req
75657374 7320616e 20756e70 72697669  uests an unprivi
6c656765 64207573 65722063 616e2073  leged user can s
65740070 61726d74 7970653d 6d61785f  et.parmtype=max_
75736572 5f626772 65713a75 696e7400  user_bgreq:uint.
6c696365 6e73653d 47504c00 64657363  license=GPL.desc
72697074 696f6e3d 46696c65 73797374  ription=Filesyst
656d2069 6e205573 65727370 61636500  em in Userspace.
61757468 6f723d4d 696b6c6f 7320537a  author=Miklos Sz
65726564 69203c6d 696b6c6f 7340737a  eredi <miklos@sz
65726564 692e6875 3e000000 00000000  eredi.hu>.......
 01a0 64657065 6e64733d 00696e74 7265653d  depends=.intree=
 01b0 59007665 726d6167 69633d33 2e322e30  Y.vermagic=3.2.0
 01c0 2d342d61 6d643634 20534d50 206d6f64  -4-amd64 SMP mod
 01d0 5f756e6c 6f616420 6d6f6476 65727369  _unload modversi
 01e0 6f6e7320 00                          ons .
```

تحلّل الأداة `modinfo` القسم `.modinfo` المُضمَّن في ملف الوحدة لتقديم تفاصيلها. يوضح المثال التالي كيفيـــة وضـــع حقـــل "المؤلـــف Author" في الوحـــدة، حيث تـــأتي هـــذه الشـــيفرة البرمجيـــة غالبًـــا من `include/linux/module.h`:

```
/*
 * ابدأ من الأسفل ثم انتقل إلى الأعلى
 */

 /* وحدات الماكرو غير المباشرة مطلوبة للصق الوسيط المُوسَّع مثل الماكرو __LINE__ */
#define ___PASTE(a,b) a##b
#define __PASTE(a,b) ___PASTE(a,b)

#define __UNIQUE_ID(prefix) __PASTE(__PASTE(__UNIQUE_ID_, prefix),
__COUNTER__)

/*  تحويل الماكرو غير المباشر إلى سلسلة نصية. يسمح إنشاء مستويين للمعامل بأن يكون
ماكرو بحد ذاته، حيث يتحوّل __stringify(FOO) مثلًا إلى السلسلة "bar" عند التصريف
-DFOO=bar باستخدام */

#define __stringify_1(x...)     #x
#define __stringify(x...)       __stringify_1(x)

#define __MODULE_INFO(tag, name, info)
static const char __UNIQUE_ID(name)[]
  __used __attribute__((section(".modinfo"), unused, aligned(1)))
  = __stringify(tag) "=" info

/* tag = "info" معلومات عامة للصيغة */
#define MODULE_INFO(tag, info) __MODULE_INFO(tag, tag, info)

/*
* استخدم للمؤلفين المتعددين الذين يستخدمون "Name <email>" أو "Name" فقط
* تعليمات أو سطور MODULE_AUTHOR() متعددة
*/
#define MODULE_AUTHOR(_author) MODULE_INFO(author, _author)

/* ---- */

MODULE_AUTHOR("Your Name <your@name.com>");
```

لنبدأ من الجزء السفلي حيث نرى أن الوحدة `MODULE_AUTHOR` تغلّـف المـاكرو الأعم `__MODULE_INFO`، ويمكننـا أن نـرى أننـا نبـني متغـيرًا من النـوع `static const char []` ليحتـوي على السلسـلة النصـية

"author=Your Name <your@name.com>". لاحظ أن المتغير لديه معامل إضافي __attribute__((section(".modinfo"))) يخبر المصرّف بعدم وضع هذا المتغير في قسم البيانات data مع المتغيرات الأخرى، ولكن يمكن إخفاؤه في قسم ELF الخاص به الذي اسمه .modinfo. توقِف المعاملات الأخرى المتغـير الذي يجري تحسينه لأنه يبدو غير مُستخدَم وللتأكد من أننا نضع المتغيرات بجانب بعضها بعضًا من خلال تحديد المحاذاة.

هناك استخدام واسع النطاق لتحويل وحدات الماكرو إلى سلاسـل نصـية Stringification Macros، وهي حيل تُستخدَم في معالج لغة C المسـبق لضـمان أن السلاسـل النصـية والتعـاريف يمكن أن تكـون مـع بعضـها بعضًا. يوفّر المصرّف gcc التعريف الخاص __COUNTER__ الذي يوفر قيمة فريـدة ومتزايـدة في كـل اسـتدعاء، مما يسمح باستدعاءات وحدة MODULE_AUTHOR متعددة في ملف واحد دون استخدام اسم المتغير نفسه.

يمكننا فحص الرموز الموضوعة في الوحدة النهائية لمعرفة النتيجة النهائية كما يلي:

```
$ objdump --syms ./fuse.ko | grep modinfo

l    d  .modinfo      0000000000000000 .modinfo
l     O .modinfo      0000000000000013 __UNIQUE_ID_alias1
l     O .modinfo      0000000000000018 __UNIQUE_ID_alias0
000000000000002b l     O .modinfo  0000000000000011 __UNIQUE_ID_alias8
000000000000003c l     O .modinfo  000000000000000e __UNIQUE_ID_alias7
000000000000004a l     O .modinfo  0000000000000068
__UNIQUE_ID_max_user_congthresh6
00000000000000b2 l     O .modinfo  0000000000000022
__UNIQUE_ID_max_user_congthreshtype5
00000000000000d4 l     O .modinfo  000000000000006e
__UNIQUE_ID_max_user_bgreq4
l     O .modinfo      000000000000001d __UNIQUE_ID_max_user_bgreqtype3
000000000000015f l     O .modinfo  000000000000000c __UNIQUE_ID_license2
000000000000016b l     O .modinfo  0000000000000024 __UNIQUE_ID_description1
000000000000018f l     O .modinfo  000000000000002a __UNIQUE_ID_author0
00000000000001b9 l     O .modinfo  0000000000000011 __UNIQUE_ID_alias0
00000000000001d0 l     O .modinfo  0000000000000009 __module_depends
00000000000001d9 l     O .modinfo  0000000000000009 __UNIQUE_ID_intree1
00000000000001e2 l     O .modinfo  000000000000002f __UNIQUE_ID_vermagic0
```

8.6.5 سكربتات الرابط Linker Scripts

تتمثل وظيفة الرابط في بناء الأقسام Sections لتشكيل المقاطع Segments من خلال استخدام سـكربت الرابط الذي يصِف مكان بدء المقاطع والأقسام الموجودة فيها ويحدد المعاملات الأخرى.

يوضح المثال الآتي مقتطفًا من سكربت الرابط الافتراضي الذي سيعرضه الرابط عند إعطاء الراية التفصيلية Verbose باستخدام الرايتين `-Wl` و `--verbose` مع gcc. السكربت الافتراضي مُضمَّنٌ في الرابط ويعتمد على تعريفات واجهة API المعيارية لإنشاء برامج عاملة لمساحة مستخدمٍ خاصة بمنصة البناء.

```
$ gcc -Wl,--verbose -o test test.c
GNU ld (GNU Binutils for Debian) 2.26
...
using internal linker script:
==================================================
OUTPUT_FORMAT("elf64-x86-64", "elf64-x86-64",
        "elf64-x86-64")
OUTPUT_ARCH(i386:x86-64)
ENTRY(_start)
SEARCH_DIR("=/usr/local/lib/x86_64-linux-gnu"); ...
SECTIONS
{
  /* أقسام للقراءة فقط مُدمَجة في مقطع النص :text segment */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x400000)); . =
SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .note.gnu.build-id : { *(.note.gnu.build-id) }
  .hash           : { *(.hash) }
  .gnu.hash       : { *(.gnu.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
  .gnu.version_d  : { *(.gnu.version_d) }
  .gnu.version_r  : { *(.gnu.version_r) }
  .rela.dyn       :
    {
    ...
    }
  PROVIDE (etext = .);
  .rodata         : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
  .rodata1        : { *(.rodata1) }
...
```

يمكنك أن ترى في المثال السابق كيف يحدد سكربت الرابط أمورًا متعددة مثل مواقع البدء والأقسام المراد تجميعها في مقاطع مختلفة. تُستخدَم الراية `-Wl` لتمرير الراية `--verbose` إلى الرابط عبر gcc، إذ يمكن توفـير

سكربتات مخصصة للرابط باستخدام الرايات. ليس مُحتمَلًا أن يحتاج مطورو مساحة المستخدم العادية إلى تجاوز سكربت الرابط الافتراضي، ولكن تتطلب التطبيقات المخصَّصة جدًا مثل عمليات بناء النواة سكربتات مخصصة للرابط في أغلب الأحيان.

8.7 بدء العمليات

ذكرنا سابقًا أن البرنامج لا يبدأ بالدالة الرئيسية `main()`، وسنختبر في هذا الفصل ما يحدث لبرنامج مرتبط ديناميكيًا عند تحميله وتشغيله ونشرح العمليات التي تسبق بدء تنفيذ برنامج في نظام التشغيل.

تخصّص النواة أولًا البنى لعملية جديدة وتقرأ ملف ELF المُحدَّد من القرص الصلب استجابةً لاستدعاء النظام exec. ذكرنا أن صيغة ELF لديها حقل لمفسّر Interpreter البرنامج هو `PT_INTERP` الذي يمكن ضبطه لتفسير البرنامج، حيث يكون المفسّر بالنسبة للتطبيقات المرتبطة ديناميكيًا هو الرابط الديناميكي Dynamic Linker -أو ld.so- الذي يسمح بإجراء بعض عمليات الربط مباشرةً قبل بدء البرنامج.

كما تقرأ النواة شيفرة الرابط الديناميكي، وتبدأ البرنامج من عنوان نقطة الدخول entry point الذي تحدده. سنختبر دور الرابط الديناميكي بالتفصيل لاحقًا، ولكن يكفي أن نقول أنه يضبط بعض الإعدادات مثل تحميل المكتبات التي يتطلبها التطبيق كما هو محدد في القسم الديناميكي من الملف الثنائي، ثم يبدأ تنفيذ البرنامج الثنائي عند عنوان نقطة الدخول أي الدالة `_init`.

8.7.1 اتصال النواة بالبرامج

تحتاج النواة Kernel إلى توصيل بعض الأشياء للبرامج عند بدء تشغيلها مثل وسائط البرنامج arguments ومتغيرات البيئة الحالية environment variables وبنية خاصة اسمها المتجه المساعد Auxiliary Vector أو `auxv` اختصارًا. يمكنك أن تطلب من الرابط الديناميكي أن يُظهر لك بعضًا من خرج تنقيح الأخطاء من البنية `auxv` من خلال تحديد قيمة البيئة كما يلي `LD_SHOW_AUXV=1`. تسمح الوسائط والبيئة والأشكال المختلفة من استدعاء النظام `exec` بتحديد هذه الأشياء للبرنامج التي يمكن للنواة توصيلها من خلال وضع جميع المعلومات المطلوبة في المكدس ليلتقطها البرنامج المُنشَأ حديثًا، وبالتالي يمكن للبرنامج عند بدء تشغيله استخدام مؤشر المكدس الخاص به للعثور على جميع معلومات بدء التشغيل المطلوبة.

المتجه المساعد هو بنية خاصة لنقل المعلومات مباشرةً من النواة إلى البرنامج المُشغَّل حديثًا، ويحتوي على معلومات خاصة بالنظام يمكن أن تكون مطلوبة مثل الحجم الافتراضي لصفحة الذاكرة الوهمية على النظام أو إمكانات العتاد، وهذه هي الميزات التي تحددها النواة للعتاد الأساسي ويمكن أن تستفيد منها برامج مساحة المستخدمين.

ا. مكتبة النواة Kernel Library

ذكرنا سابقًا أن استدعاءات النظام بطيئة وأن الأنظمة الحديثة لديها آليات لتجنب الحِمل الناتج عن استدعاء مصيدة Trap للمعالج، حيث يمكن تنفيذ ذلك في نظام لينكس من خلال استخدام حيلة بين المحمل الديناميكي والنواة المتصلَين ببنية AUXV، إذ تضيف النواة مكتبة مشتركة صغيرة إلى فضاء العناوين لكل عملية مُنشَأة حديثًا وتحتوي على دالة تجري استدعاءات النظام نيابة عنك. يكمن جمال هذا النظام في أنه إذا دعم العتاد الأساسي آلية استدعاء نظام سريعة، فيمكن للنواة استخدامها لكونها منشئة المكتبة، وإلّا فيمكنها استخدام النظام القديم لإنشاء مصيدة. تسمى هذه المكتبة linux-gate.so.1 لأنها بوابة إلى عمل النواة الداخلي.

تضيف النواة مدخلةً إلى البنية auxv تسمَّى AT_SYSINFO_EHDR عندما تبدأ الرابط الديناميكي، وهذه المدخلة هي العنوان الموجود في الذاكرة الذي توجد فيه مكتبة النواة الخاصة. يمكن للرابط الديناميكي عندما يبدأ البحثَ عن المؤشر AT_SYSINFO_EHDR، فإن وُجد، فستُحمَّل تلك المكتبة للبرنامج. لا يملك البرنامج أيّ فكرة عن وجود هذه المكتبة، لأنها تُعَد ترتيبًا خاصًا بين الرابط الديناميكي والنواة.

ذكرنا أن المبرمجين يجرون استدعاءات النظام بطريقة غير مباشرة من خلال استدعاء الدوال في مكتبات النظام libc التي يمكنها التحقق مما إذا كان ملف النواة الثنائي الخاص محمَّلًا أم لا، فإذا كان الأمر كذلك، فيجب استخدام الدوال الموجودة ضمنه لإجراء استدعاءات النظام. إذا حددت النواة أن العتاد يمتلك القدرة المطلوبة، فيجب استخدام طريقة استدعاء النظام السريع.

8.7.2 بدء البرنامج

تمرّر النواةُ المفسّرَ بعد تحميله إلى نقطة الدخول كما هو مذكور في ملف المفسّر (لاحظ عدم اختبار كيفية بدء الرابط الديناميكي). سيقفز الرابط الديناميكي إلى عنوان نقطة الدخول كما هو مذكور في ملف ELF الثنائي.

يوضح المثال التالي نتيجة تفكيك Disassembley بدء تشغيل البرنامج:

```
$ cat test.c

int main(void)
{
  return 0;
}

$ gcc -o test test.c

$ readelf --headers ./test | grep Entry
```

```
  Entry point address:               0x80482b0

$ objdump --disassemble ./test

[...]

080482b0 <_start>:
  80482b0:       31 ed                   xor    %ebp,%ebp
  80482b2:       5e                      pop    %esi
  80482b3:       89 e1                   mov    %esp,%ecx
  80482b5:       83 e4 f0                and    $0xfffffff0,%esp
  80482b8:       50                      push   %eax
  80482b9:       54                      push   %esp
  80482ba:       52                      push   %edx
  80482bb:       68 00 84 04 08          push   $0x8048400
  80482c0:       68 90 83 04 08          push   $0x8048390
  80482c5:       51                      push   %ecx
  80482c6:       56                      push   %esi
  80482c7:       68 68 83 04 08          push   $0x8048368
  80482cc:       e8 b3 ff ff ff          call   8048284
<__libc_start_main@plt>
  80482d1:       f4                      hlt
  80482d2:       90                      nop
  80482d3:       90                      nop

<main>:
push   %ebp
e5                   mov    %esp,%ebp
  804836b:       83 ec 08                sub    $0x8,%esp
  804836e:       83 e4 f0                and    $0xfffffff0,%esp
b8 00 00 00 00          mov    $0x0,%eax
c0 0f                add    $0xf,%eax
c0 0f                add    $0xf,%eax
  804837c:       c1 e8 04                shr    $0x4,%eax
  804837f:       c1 e0 04                shl    $0x4,%eax
c4                   sub    %eax,%esp
b8 00 00 00 00          mov    $0x0,%eax
c9                      leave
  804838a:       c3                      ret
```

```
  804838b:       90                      nop
  804838c:       90                      nop
  804838d:       90                      nop
  804838e:       90                      nop
  804838f:       90                      nop

<__libc_csu_init>:
push   %ebp
e5                  mov    %esp,%ebp
  [...]

<__libc_csu_fini>:
push   %ebp
  [...]
```

يمكننا أن نرى في المثال البسيط السابق باستخدام أداة `readelf` أن نقطة الدخول هي الدالة `_start` في الملف الثنائي، ويمكننا أن نرى في عملية التفكيك دفع بعض القيم إلى المكدس. تمثل القيمة الأولى `0x8048400` الدالة `__libc_csu_fini`، وتمثل القيمة `0x8048390` الدالة `__libc_csu_init`، وتمثل القيمة `0x8048368` الدالة الرئيسية `main()`، ثم تُستدعَى قيمة الدالة `__libc_start_main`.

إن الدالة `__libc_start_main` مُعرَّفة ضمن مصادر المكتبة glibc والموجودة ضمن المسار `sysdeps/generic/libc-start.c`، وتُعَد معقدةً جدًا ومخفيةً بين عدد كبير من التعريفات، حيث يجب أن تكون قابلة للنقل عبر عدد كبير جدًا من الأنظمة والمعماريات التي يمكن لمكتبة glibc العمل عليها. تطبّق هذه الدالة عددًا من الأشياء المحدَّدة المتعلقة بإعداد مكتبة C والتي لا يحتاج المبرمج العادي للقلق بشأنها. النقطة التالية التي تستدعي فيها المكتبةُ البرنامجَ هي عند التعامل مع شيفرة `init`.

تعد الدالتان `init` و`fini` مفهومين خاصين يستدعيان أجزاءً من الشيفرة البرمجية الموجودة في المكتبات المشتركة والتي يمكن أن تحتاج لاستدعائها قبل أن تبدأ المكتبة أو عند إلغاء تحميل المكتبة على التوالي. يمكنك أن ترى كيف يمكن أن يكون ذلك مفيدًا لمبرمجي المكتبات لإعداد المتغيرات عند بدء تشغيل المكتبة أو لتنظيفها في النهاية. كان البحث عن الدالتين `_init` و`_fini` في المكتبة ممكنًا سابقًا، ولكن أصبح ذلك مقيدًا إلى حد ما حيث كان كل شيء مطلوبًا في هاتين الدالتين. سنوضح فيما يلي كيفية عمل الدالتين `init` و`fini` فقط.

يمكننا أن نرى الآن أن الدالة `__libc_start_main` ستتلقى عددًا من معاملات الدخل في المكدس stack، إذ سيكون بإمكانها أولًا الوصول إلى وسائط البرنامج ومتغيرات البيئة والمتجه المساعد من النواة، ثم ستدفع دالة التهيئة إلى عناوين المكدس الخاصة بالدوال للتعامل مع الدالتين `init` أو `fini` ثم عنوان الدالة الرئيسية نفسها.

نحتاج طريقةً ما للإشارة إلى أنه يجب استدعاء دالـةٍ مـا باسـتخدام `init` أو `fini` في الشـيفرة المصـدرية. نستخدم مع gcc سمات Attributes لتمييز دالتين بأنهمـا بانيتـان Constructors أو هادمتـان Destructors في برنامجنا الرئيسي. تُستخدَم هذه المصطلحات بصورة أكثر شيوعًا مـع اللغـات كائنيـة التوجـه لوصـف دورات حياة الكائن.

إليك مثال عن دالة البناء ودالة الهدم:

```
$ cat test.c
#include <stdio.h>

void __attribute__((constructor)) program_init(void)  {
  printf("init\n");
}

void  __attribute__((destructor)) program_fini(void) {
  printf("fini\n");
}

int main(void)
{
  return 0;
}

$ gcc -Wall  -o test test.c

$ ./test
init
fini

$ objdump --disassemble ./test | grep program_init
<program_init>:

$ objdump --disassemble ./test | grep program_fini
080483b0 <program_fini>:

$ objdump --disassemble ./test

[...]
<_init>:
```

```
push   %ebp
e5                    mov    %esp,%ebp
ec 08                 sub    $0x8,%esp
e8 79 00 00 00           call   8048304 <call_gmon_start>
  804828b:       e8 e0 00 00 00          call   8048370 <frame_dummy>
e8 2b 02 00 00           call   80484c0 <__do_global_ctors_aux>
c9                       leave
c3                       ret
[...]

080484c0 <__do_global_ctors_aux>:
  80484c0:       55                      push   %ebp
  80484c1:       89 e5                   mov    %esp,%ebp
  80484c3:       53                      push   %ebx
  80484c4:       52                      push   %edx
  80484c5:       a1 2c 95 04 08          mov    0x804952c,%eax
  80484ca:       83 f8 ff                cmp    $0xffffffff,%eax
  80484cd:       74 1e                   je     80484ed
<__do_global_ctors_aux+0x2d>
  80484cf:       bb 2c 95 04 08          mov    $0x804952c,%ebx
  80484d4:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
  80484da:       8d bf 00 00 00 00       lea    0x0(%edi),%edi
  80484e0:       ff d0                   call   *%eax
  80484e2:       8b 43 fc                mov    0xfffffffc(%ebx),%eax
  80484e5:       83 eb 04                sub    $0x4,%ebx
  80484e8:       83 f8 ff                cmp    $0xffffffff,%eax
  80484eb:       75 f3                   jne    80484e0
<__do_global_ctors_aux+0x20>
  80484ed:       58                      pop    %eax
  80484ee:       5b                      pop    %ebx
  80484ef:       5d                      pop    %ebp
  80484f0:       c3                      ret
  80484f1:       90                      nop
  80484f2:       90                      nop
  80484f3:       90                      nop

$ readelf --sections ./test
There are 34 section headers, starting at offset 0xfb0:

Section Headers:
```

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf
Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0
0
  [ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0
1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0
4
  [ 3] .hash             HASH            08048148 000148 00002c 04   A  4   0
4
  [ 4] .dynsym           DYNSYM          08048174 000174 000060 10   A  5   1
4
  [ 5] .dynstr           STRTAB          080481d4 0001d4 00005e 00   A  0   0
1
  [ 6] .gnu.version      VERSYM          08048232 000232 00000c 02   A  4   0
2
  [ 7] .gnu.version_r    VERNEED         08048240 000240 000020 00   A  5   1
4
  [ 8] .rel.dyn          REL             08048260 000260 000008 08   A  4   0
4
  [ 9] .rel.plt          REL             08048268 000268 000018 08   A  4  11
4
  [10] .init             PROGBITS        08048280 000280 000017 00  AX  0   0
4
  [11] .plt              PROGBITS        08048298 000298 000040 04  AX  0   0
4
  [12] .text             PROGBITS        080482e0 0002e0 000214 00  AX  0   0
16
  [13] .fini             PROGBITS        080484f4 0004f4 00001a 00  AX  0   0
4
  [14] .rodata           PROGBITS        08048510 000510 000012 00   A  0   0
4
  [15] .eh_frame         PROGBITS        08048524 000524 000004 00   A  0   0
4
  [16] .ctors            PROGBITS        08049528 000528 00000c 00  WA  0   0
4
  [17] .dtors            PROGBITS        08049534 000534 00000c 00  WA  0   0
4
  [18] .jcr              PROGBITS        08049540 000540 000004 00  WA  0   0
4
  [19] .dynamic          DYNAMIC         08049544 000544 0000c8 08  WA  5   0
4
  [20] .got              PROGBITS        0804960c 00060c 000004 04  WA  0   0
4
  [21] .got.plt          PROGBITS        08049610 000610 000018 04  WA  0   0
4
```

```
  [22] .data             PROGBITS        08049628 000628 00000c 00  WA  0   0
4
  [23] .bss              NOBITS          08049634 000634 000004 00  WA  0   0
4
  [24] .comment          PROGBITS        00000000 000634 00018f 00      0   0
1
  [25] .debug_aranges    PROGBITS        00000000 0007c8 000078 00      0   0
8
  [26] .debug_pubnames   PROGBITS        00000000 000840 000025 00      0   0
1
  [27] .debug_info       PROGBITS        00000000 000865 0002e1 00      0   0
1
  [28] .debug_abbrev     PROGBITS        00000000 000b46 000076 00      0   0
1
  [29] .debug_line       PROGBITS        00000000 000bbc 0001da 00      0   0
1
  [30] .debug_str        PROGBITS        00000000 000d96 0000f3 01  MS  0   0
1
  [31] .shstrtab         STRTAB          00000000 000e89 000127 00      0   0
1
  [32] .symtab           SYMTAB          00000000 001500 000490 10     33  53
4
  [33] .strtab           STRTAB          00000000 001990 000218 00      0   0
1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

$ objdump --disassemble-all --section .ctors ./test

./test:     file format elf32-i386

Contents of section .ctors:
ffffffff 98830408 00000000           ............
```

كـانت دالـة التهيئـة `__libc_csu_init` هي القيمـة الأخـيرة المدفوعـة إلى المكـدس من أجـل الدالـة `__libc_start_main`. إذا اتبعنا سلسلة الاستدعاءات ابتداءً من `__libc_csu_init`، فيمكننا أن نرى أنها تجري بعض الإعدادات ثم تستدعي الدالة `_init` في الملف القابل للتنفيذ. تستدعي الدالة `_init` في النهايـة دالة تسمى `__do_global_ctors_aux`، حيث إذا نظرنا إلى تفكيك هذه الدالة، فيمكننا أن نرى أنها تبـدأ من العنوان `0x804952c` ثم تتكـرر وتقـرأ قيمـة وتسـتدعيها. هـذا العنـوان الـذي يمثـل البدايـة موجـود في القسـم

`.ctors` من الملف، حيث إذا ألقينا نظرة عليه، فسنرى أنه يحتوي على القيمة الأولى 1- وعنوان الدالة بصيغة Big Endian أي تخزين البتات الأقل أهمية أولًا والقيمة صفر.

العنوان بصيغة Big Endian هو `0x08048398` أو عنوان الدالة `program_init`، لذا فإن صيغة القسم `.ctors` هي 1- أولًا ثم عنوان الدوال المطلوب استدعاؤها عند التهيئة، وأخيرًا القيمة صفر للإشارة إلى اكتمال القائمة. ستُستدعَى كل مدخلة، ولدينا في هذه الحالة دالة واحدة فقط.

أخيرًا، تستدعي الدالة `__libc_start_main` الدالةَ الرئيسية `main()` بمجرد اكتمالها باستدعاء الدالة `_init`. تذكر أن هذه الدالة تمتلك إعداد المكدس الأولي باستخدام الوسائط ومؤشرات البيئة من النواة، وهذه هي الطريقة التي تحصل بها الدالة الرئيسية على الوسائط `argc, argv[], envp[]`. ستعمل العملية بعد ذلك وتكتمل مرحلة الإعداد.

تحدث عملية مماثلة مع القسم `.dtors` للهادمين Destructors عند إنهاء البرنامج، حيث تستدعيها الدالة `__libc_start_main` عند اكتمال الدالة الرئيسية `main()`.

لاحظ تطبيق الكثير من العمل قبل أن يبدأ البرنامج وحتى بعد أن تعتقد أنه انتهى بقليل.

9. مفهوم الربط الديناميكي

9.1 مشاركة الشيفرة

يشرح هذا الفصل طريقة مشاركة الشيفرة بين التطبيقات والمكتبات المشتركة، ويشرح مفهوم الربط الديناميكي في تنفيذ التطبيقات حيث تُعَد شيفرة نظام التشغيل البرمجية للقراءة فقط وتكون منفصلة عن البيانات، لذا إن لم تتمكن البرامج من تعديل الشيفرة البرمجية مع وجود كميات كبيرة من الشيفرة البرمجية فمن المنطقي مشاركة هذه الشيفرة بين العديد من الملفات القابلة للتنفيذ بدلًا من تكرارها لكل ملف منها.

يمكن القيام بذلك بسهولة بين التطبيقات والمكتبات المشتركة باستخدام الذاكرة الوهمية، إذ يمكن الرجوع بسهولة إلى صفحات الذاكرة الحقيقية التي جرى تحميل شيفرة المكتبة البرمجية إليها من خلال عدد من الصفحات الوهمية في عددٍ من فضاءات العناوين. لذا يمكن لكل عملية الوصول إلى شيفرة المكتبة البرمجية باستخدام أيّ عنوان وهمي تريده، بينما يكون لديك نسخة حقيقية واحدة فقط من هذه الشيفرة في ذاكرة النظام.

وبذلك توصل المبرمجون بسرعة إلى فكرة المكتبة المشتركة Shared Library التي -كما يوحي الاسم- يمكن مشاركتها بين العديد من الملفات القابلة للتنفيذ. يحتوي كل ملف قابل للتنفيذ على مرجع يقول: "أحتاج مكتبة Foo مثلًا"، حيث يُترَك الأمر للنظام عند تحميل البرنامج للتحقق من وجود برنامج آخر حمّل شيفرة هذه المكتبة في الذاكرة ثم مشاركتها من خلال ربط صفحات الملف القابل للتنفيذ مع الذاكرة الحقيقية، أو يمكنه تحميل المكتبة في ذاكرة الملف القابل للتنفيذ. تسمى هذه العملية بالربط الديناميكي Dynamic Linking، لأنها تطبّق جزءًا من عملية الربط مباشرةً عند تنفيذ البرامج في النظام.

9.1.1 تفاصيل المكتبة الديناميكية

تشبه المكتبات إلى حدٍ كبير برنامجًا لا يُشغَّل أبدًا، إذ لديها قسم الشيفرة البرمجية وقسم البيانات (الدوال والمتغيرات) تمامًا مثل الملفات القابل للتنفيذ، ولكن لا يمكن تشغيلها، فهي توفر فقط مكتبة من الدوال

للمطورين لاستدعائها. لذا يمكن ملف ELF أن يمثل مكتبة ديناميكية تمامًا كما يمثل ملفًا قابلًا للتنفيذ مع وجود بعض الاختلافات الأساسية مثل عدم وجود مؤشر للمكان الذي يجب أن يبدأ فيه التنفيذ، ولكن تُعَد جميع المكتبات المشتركة مجرد كائنات بصيغة ELF مثل أي ملف آخر قابل للتنفيذ.

تحتوي ترويسة ملف ELF على رايتين حصريتين هما `ET_EXEC` و `ET_DYN` لتمييز ملف ELF بوصفه ملفًا قابلًا للتنفيذ أو ملفَ كائن مشترك.

9.1.2 تضمين المكتبات في ملف قابل للتنفيذ

يمكن تضمين المكتبات في ملف قابل للتنفيذ ولكن يجب أن نراعي مسألتين مهمتين متعلقتين بالمصرّف والرابط الديناميكي كما سنوضح الآن.

ا. التصريف Compilation

تمتلك ملفات الكائنات مراجعًا إلى دوال المكتبة تمامًا كما هو الحال مع أيّ مرجع خارجي آخر عندما تصرّف برنامجك الذي يستخدم مكتبة ديناميكية. يجب تضمين ترويسة المكتبة ليعرف المصرّف الأنواع المحددة للدوال التي تستدعيها، إذ يحتاج المصرّف فقط معرفةَ الأنواع المرتبطة بالدالة (مثل أن تأخذ الدالة النوع `int` وتعيد النوع `char*`) بحيث يمكنه تخصيص مساحة لاستدعاء الدالة بصورة صحيحة.

لم يكن هذا هو الحال دائمًا مع معايير لغة C، إذ افترضت المصّرفات سابقًا أن أيّ دالة غير معروفة تعيد قيمة من النوع `int`. يكون لحجم المؤشر في نظام 32 بت حجم النوع `int` نفسه، لذلك لا توجد مشكلة في ذلك، ولكن يكون حجم المؤشر ضعف حجم `int` في نظام 64 بت، لذلك إذا أعادت الدالة مؤشرًا، فستُدمَّر قيمتها. يُعَد ذلك الأمر غير مقبول، لأن المؤشر لن يؤشّر إلى ذاكرة صالحة، ولكن تغيّر معيار C99 بحيث يُطلَب منك تحديد أنواع الدوال المُضمَّنة.

ب. الربط Linking

يطبّق الرابط الديناميكي الكثير من العمل للمكتبات المشتركة، ولكن لا يزال الرابط التقليدي يلعب دورًا هامًا في إنشاء الملف القابل للتنفيذ، إذ يجب أن يضع الرابط التقليدي مؤشرًا في الملف القابل للتنفيذ حتى يعرف الرابط الديناميكي المكتبة التي ستحقّق الاعتماديات Dependencies في وقت التشغيل. يتطلب القسم `dynamic` من الملف القابل للتنفيذ مدخلة مطلوبة `NEEDED` لكل مكتبة مشتركة يعتمد عليها الملف القابل للتنفيذ.

يمكننا فحص هذه الحقول باستخدام برنامج `readelf`. سنلقي فيما يلي نظرة على ملف ثنائي معياري `/bin/ls` يمثل تحديد المكتبات الديناميكية:

```
$ readelf --dynamic /bin/ls
```

```
Dynamic segment at offset 0x22f78 contains 27 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [librt.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libacl.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
 0x000000000000000c (INIT)               0x4000000000001e30
 ... snip ...
```

يمكنك أن ترى أن المثال السابق يحدد ثلاث مكتبات. المكتبة الأكثر شيوعًا التي تتشارك بها أغلبية البرامج الموجودة على النظام -إن لم تكن جميعها- هي مكتبة `libc`، وهناك بعض المكتبات الأخرى التي يحتاجها البرنامج ليعمل بصورة صحيحة.

تكون قراءة ملف ELF المباشرة مفيدةً أحيانًا، ولكن الطريقة المعتادة لفحص ملف قابل للتنفيذ مرتبط ديناميكيًا هي باستخدام أداة `ldd` التي تمر على اعتماديات المكتبات، حيث ستظهِر لك إذا كانت المكتبة معتمدة على مكتبة أخرى كما يلي:

```
$ ldd /bin/ls
        librt.so.1 => /lib/tls/librt.so.1 (0x2000000000058000)
        libacl.so.1 => /lib/libacl.so.1 (0x2000000000078000)
        libc.so.6.1 => /lib/tls/libc.so.6.1 (0x2000000000098000)
        libpthread.so.0 => /lib/tls/libpthread.so.0 (0x20000000002e0000)
        /lib/ld-linux-ia64.so.2 => /lib/ld-linux-ia64.so.2
(0x2000000000000000)
        libattr.so.1 => /lib/libattr.so.1 (0x2000000000310000)
$ readelf --dynamic /lib/librt.so.1

Dynamic segment at offset 0xd600 contains 30 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
 0x0000000000000001 (NEEDED)             Shared library: [libpthread.so.0]
 ... snip ...
```

يمكننا أن نرى في المثال السابق أن مكتبة `libpthread` مطلوبة من مكان ما، حيث إذا تعمّقنا قليلًا، فيمكننا أن نرى أنها مطلوبة من المكتبة `librt`.

9.2 الرابط الديناميكي Dynamic Linker

الرابط الديناميكي هو البرنامج الذي يدير المكتبات الديناميكية المشتركة بدلًا من الملف القابل للتنفيذ، ويعمل على تحميل المكتبات في الذاكرة وتعديل البرنامج في وقت التشغيل لاستدعاء الدوال الموجودة في

المكتبـة. يسـمح ملـف ELF للملفـات القابلـة للتنفيـذ بتحديـد المفسّـر Interpreter الـذي هـو برنـامج يجب استخدامه لتشغيل الملف القابل للتنفيذ. يضبط المصرِّف Compiler والرابـط السـاكن Static Linker مفسّـر الملفات القابلة للتنفيذ الذي يعتمد على المكتبات الديناميكية ليكون الرابط الديناميكي.

يوضَح المثال التالي كيفية التحقق من مفسّر البرنامج:

```
ianw@lime:~/programs/csbu$ readelf --headers /bin/ls

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x4000000000000040 0x4000000000000040
                 0x0000000000000188 0x0000000000000188  R E    8
  INTERP         0x00000000000001c8 0x40000000000001c8 0x40000000000001c8
                 0x0000000000000018 0x0000000000000018  R      1
      [Requesting program interpreter: /lib/ld-linux-ia64.so.2]
  LOAD           0x0000000000000000 0x4000000000000000 0x4000000000000000
                 0x0000000000022e40 0x0000000000022e40  R E    10000
  LOAD           0x0000000000022e40 0x6000000000002e40 0x6000000000002e40
                 0x0000000000001138 0x00000000000017b8  RW     10000
  DYNAMIC        0x0000000000022f78 0x6000000000002f78 0x6000000000002f78
                 0x0000000000000200 0x0000000000000200  RW     8
  NOTE           0x00000000000001e0 0x40000000000001e0 0x40000000000001e0
                 0x0000000000000020 0x0000000000000020  R      4
  IA_64_UNWIND   0x0000000000022018 0x4000000000022018 0x4000000000022018
                 0x0000000000000e28 0x0000000000000e28  R      8
```

يمكنك أن ترى في المثال السابق أن المفسّـر مضـبوط ليكـون /lib/ld-linux-ia64.so.2 أي يمثـل الرابـط الديناميكي. تتحقق النواة Kernel عندما تحمّل الملف الثنائي للتنفيذ مما إذا كان الحقل `PT_INTERP` موجودًا، حيث إذا كان موجودًا، فيجب تحميل ما يؤشّر إليه في الذاكرة وتشغيله.

ذكرنا أن للملفات القابلة للتنفيذ المرتبطة ديناميكيًا مراجع يجب إصلاحها باستخدام معلومـات غـير متـوفرة حـتى وقت التشـغيل مثـل عنـوان دالـة موجـودة في مكتبـة مشـتركة، وتسـمَّى هـذه المراجـع بالانتقالات Relocations.

9.2.1 الانتقالات Relocations

يتمثل الجزء الأساسي من الرابط الديناميكي في إصلاح العناوين في وقت التشغيل الذي يُعَد المرة الوحيدة التي يمكنك أن تعرف فيها مكان تحميلك في الـذاكرة بالضـبط. يمكن التفكـير في الانتقـالات بأنهـا ملاحظـة أن

عنوانًا معينًا يجب إصلاحه في وقت التحميل، حيث يجب قراءة جميع الانتقالات وإصلاح العناوين التي تشير إليها للإشارة إلى المكان الصحيح قبل أن تصبح الشيفرة البرمجية جاهزة للتشغيل. إليك مثال عن عملية انتقال:

العنوان	الحدث
0x123456	عنوان الرمز "x"
0x564773	الدالة X

جدول 24: مثال على الترحيل

هناك العديد من أنواع الانتقالات لكل معمارية، حيث يُوثَّق السلوك الدقيق لكل نوع بوصفه جزءًا من واجهة ABI الخاصة بالنظام. يُعَد تعريف الانتقالات واضحًا وسهلًا كما في المثال التالي:

```
typedef struct {
  Elf32_Addr    r_offset;  <--- address to fix
  Elf32_Word    r_info;    <--- symbol table pointer and relocation
type
}

typedef struct {
  Elf32_Addr    r_offset;
  Elf32_Word    r_info;
  Elf32_Sword   r_addend;
} Elf32_Rela
```

يشير الحقل `r_offset` إلى الإزاحة في الملف التي يجب إصلاحها، ويحدد الحقل `r_info` نوع الانتقال الذي يصِف بالضبط ما يجب تطبيقه لإصلاح هذه الشيفرة البرمجية. تُعَد قيمة الرمز أبسط عملية انتقال مُعرَّفة لمعماريةٍ ما، حيث يمكنك ببساطة في هذه الحالة استبدال عنوان الرمز في الموقع المحدد، وبالتالي سيكتمل إصلاح عملية الانتقال.

يوجد نوعان من الطرق المُستخدمة لتشغيل الانتقالات أحدهما مع قيمة مُضافة والآخر بدونها. القيمة المضافة هي ببساطة شيء يجب إضافته إلى العنوان الذي جرى إصلاحه للعثور على العنوان الصحيح، فمثلًا إذا كان الانتقال للرمز `i` مثلًا، فستُضبَط القيمة المُضافة على القيمة 8 لأن الشيفرة البرمجية الأصلية تطبّق شيئًا مثل `i[8]`، وهذا يعني العثور على عنوان `i` ثم تجاوزه بمقدار 8.

يجب تخزين هذه القيمة المضافة في مكان ما، فمثلًا تُخزَّن في صيغة `REL` القيمة المُضافة في شيفرة البرنامج ضمن المكان الذي يجب أن يكون فيه العنوان الذي جرى إصلاحه. لذا يجب إصلاح العنوان بصورة صحيحة من خلال قراءة الذاكرة التي تريد إصلاحها للحصول على أيّ قيمة مضافة وتخزينها والعثور على العنوان

الحقيقي وإضافة القيمة المضافة إليه ثم كتابته مرة أخرى فوق القيمة المضافة. بينما تحدد الصيغة RELA مكان القيمة المضافة في الانتقال مباشرةً.

هناك بعض المقايضات لكلٍ من هاتين الصيغتين، إذ يجب في صيغة REL استخدام مرجع ذاكرة إضافي للعثور على القيمة المضافة قبل الإصلاح، لكنك لا تهدر مساحةً في الملف الثنائي لأنك تستخدم الذاكرة الهدف لعملية الانتقال. بينما يمكنك في صيغة RELA الاحتفاظ بالقيمة المضافة مع الانتقال، ولكنك تهدر هذه المساحة في ملف القرص الصلب الثنائي. تستخدم معظم الأنظمة الحديثة انتقالات RELA.

ا. كيفية عمل الانتقالات

يوضح المثال التالي كيفية عمل الانتقالات، حيث سننشئ مكتبتين مشتركتين بسيطتين ونشير إلى إحدى المكتبتين من المكتبة الأخرى:

```
$ cat addendtest.c
extern int i[4];
int *j = i + 2;

$ cat addendtest2.c
int i[4];

$ gcc -nostdlib -shared -fpic -s -o addendtest2.so addendtest2.c
$ gcc -nostdlib -shared -fpic -o addendtest.so addendtest.c ./addendtest2.so

$ readelf -r ./addendtest.so

Relocation section '.rela.dyn' at offset 0x3b8 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name +
Addend
0000000104f8  000f00000027 R_IA64_DIR64LSB   0000000000000000 i + 8
```

لدينا عملية انتقال واحدة في addendtest.so من النوع R_IA64_DIR64LSB الذي إن بحثت عنه في واجهة IA64 ABI، فيمكن تقسيمه إلى ما يلي:

1. R_IA64: تبدأ جميع الانتقالات بهذه البادئة.
2. DIR64: انتقال من النوع 64 بت المباشر.
3. LSB: بما أن IA64 يمكنها أن تعمل في أنماط Big Endian (تخزين البتات الأقل أهمية أولًا) و Little Endian (تخزين البتات الأكثر أهمية أولًا)، فسيكون هذا الانتقال Little Endian (البايت الأقل أهمية Least Significant Byte).

تقول واجهة ABI أن هذا الانتقال يمثل قيمة الرمز الذي يؤشّر إليه مع أيّ قيمة مضافة. يمكننا أن نرى أن لدينا قيمة مضافة مقدارها 8 ، حيث أن حجم النوع `int` يساوي 4 أو `sizeof(int) == 4`، ونقلنا قيمتين من النوع `int` في المصفوفة أي `*j = i + 2`. لذا يمكن إصلاح هذا الانتقال في وقت التشغيل من خلال العثور على عنوان الرمز `i` ووضع قيمته بالإضافة إلى القيمة 8 في `0x104f8`.

9.2.2 استقلال المواقع

يُعطَى مقطع الشيفرة البرمجية ومقطع البيانات في ملف قابل للتنفيذ عنوانًا أساسيًا محددًا في الذاكرة الوهمية، لذا لا يمكن مشاركة شيفرة الملف القابل للتنفيذ الذي يحصل على فضاء عناوين جديد خاص به، وهذا يعني أن المصرِّف يعرف بالضبط مكان قسم البيانات ويمكنه الرجوع إليه مباشرةً.

لا تمتلك المكتبات مثل هذا الضمان، إذ يمكنها معرفة أن قسم البيانات الخاص بها سيكون إزاحة محددة عن العنوان الأساسي، ولكن لا يمكن بالضبط معرفة مكان ذلك العنوان الأساسي إلّا في وقت التشغيل. لذا يجب إنشاء جميع المكتبات باستخدام شيفرة برمجية يمكن تنفيذها بغض النظر عن مكان وضعها في الذاكرة، ويُعرَف ذلك باسم الشيفرة المستقلة عن الموقع Position Independent Code، أو PIC اختصارًا. لاحظ أن قسم البيانات لا يزال يمثل إزاحة ثابتة عن قسم الشيفرة البرمجية، ولكن يجب إضافة الإزاحة إلى عنوان التحميل للعثور على عنوان البيانات.

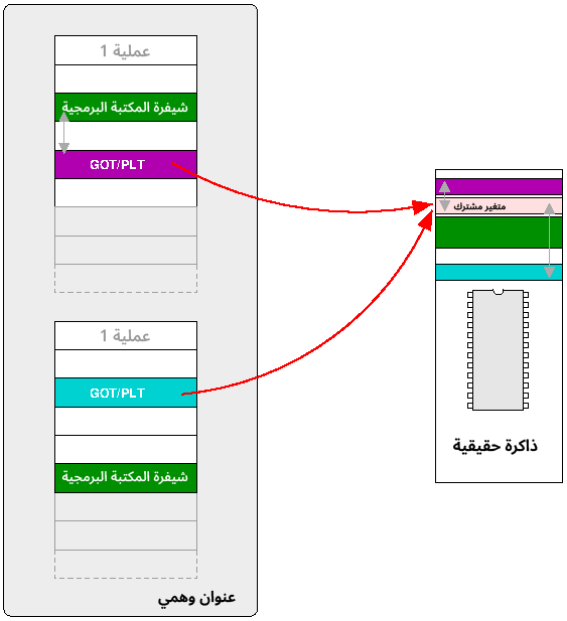
9.3 جدول الإزاحة العام Global Offset Table

لا بد أنك لاحظت مشكلة خطيرة في الانتقالات Relocations عند التفكير في أهداف المكتبة المشتركة، حيث ذكرنا سابقًا أن ميزة المكتبة المشتركة مع الذاكرة الوهمية هي أن البرامج المتعددة يمكنها استخدام الشيفرة البرمجية الموجودة في الذاكرة من خلال مشاركة الصفحات.

تنبع هذه المشكلة من حقيقة أن المكتبات ليس لديها أي ضمان حول مكان وضعها في الذاكرة، حيث سيجد الرابط الديناميكي المكان الأكثر ملاءمة في الذاكرة الوهمية لكل مكتبة مطلوبة ويضعها هناك. لكن إن لم يحدث ذلك، فستطلب كل مكتبة في النظام جزءها الخاص من الذاكرة الوهمية حتى لا تتداخل مع غيرها. تتطلب كل مكتبة جديدة في النظام تخصيصًا لها عند إضافتها، ويمكن كتابة مكتبة ضخمة لا تترك مساحةَ كافية للمكتبات الأخرى مع احتمالية ألّا يرغب برنامجك أبدًا في استخدام هذه المكتبة على أيّ حال. إذا عدّلتَ شيفرة مكتبة مشتركة لديها انتقال، فلن تصبح هذه الشيفرة البرمجية قابلةً للمشاركة، وسنفقد ميزة المكتبة المشتركة.

لنفترض أننا نأخذ قيمة رمزٍ ما، حيث سيكون لدينا باستخدام الانتقالات فقط رابط ديناميكي يبحث عن عنوان الذاكرة لهذا الرمز ويعيد كتابة الشيفرة البرمجية لتحميل هذا العنوان. يمكن تحسين هذا الموقف من خلال تخصيص مساحة في الملف الثنائي للاحتفاظ بعنوان هذا الرمز وجعل الرابط الديناميكي يضع العنوان هناك بدلًا من وضعه في الشيفرة البرمجية مباشرةً، وبالتالي لا نحتاج أبدًا إلى تعديل جزء الشيفرة البرمجية في الملف الثنائي.

تسمى المنطقة المخصصة لهذه العناوين بجدول الإزاحة العام Global Offset Table -أو GOT اختصـارًا- وهو يتواجد في القسم `.got` من ملف ELF، ويوضح الشكل التالي الوصول للذاكرة باستخدام جدول GOT:



شكل 39: الوصول إلى الذاكرة عبر GOT

يُعَد جدول GOT خاصًا بكل عملية، ويجب أن يكون للعملية أذونات للكتابة خاصة بها. بينمـا يمكن مشـاركة شيفرة المكتبة ويجب أن يكون للعملية فقط أذونات قراءة وتنفيذ الشيفرة البرمجيـة، وإلا فسـيكون هنـاك خـرق أمني خطير إذا تمكنت العملية من تعديل الشيفرة البرمجية.

9.3.1 كيفية عمل جدول GOT

يوضح المثال التالي كيفية استخدام جدول GOT:

```
$ cat got.c
extern int i;

void test(void)
{
  i = 100;
}

$ gcc -nostdlib  -shared -o got.so ./got.c

$ objdump --disassemble ./got.so
```

```
./got.so:     file format elf64-ia64-little

Disassembly of section .text:

<test>:
0d 10 00 18 00 21       [MFI]       mov r2=r12
00 00 02 00 c0                  nop.f 0x0
 41c:   81 09 00 90                         addl r14=24,r1;;
0d 78 00 1c 18 10       [MFI]       ld8 r15=[r14]
00 00 02 00 c0                  nop.f 0x0
 42c:   41 06 00 90                         mov r14=100;;
00 38 1e 90 11       [MIB]       st4 [r15]=r14
c0 00 08 00 42 80                  mov r12=r2
 43c:   08 00 84 00                         br.ret.sptk.many b0;;

$ readelf --sections ./got.so
There are 17 section headers, starting at offset 0x640:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
0     0     0
  [ 1] .hash             HASH             0000000000000120  00000120
       00000000000000a0  0000000000000004   A       2     0     8
  [ 2] .dynsym           DYNSYM           00000000000001c0  000001c0
       00000000000001f8  0000000000000018   A       3     e     8
  [ 3] .dynstr           STRTAB           00000000000003b8  000003b8
       000000000000003f  0000000000000000   A       0     0     1
  [ 4] .rela.dyn         RELA             00000000000003f8  000003f8
A       2     0     8
  [ 5] .text             PROGBITS         0000000000000410  00000410
AX       0     0     16
  [ 6] .IA_64.unwind_inf PROGBITS         0000000000000440  00000440
A       0     0     8
  [ 7] .IA_64.unwind     IA_64_UNWIND     0000000000000458  00000458
AL       5     5     8
  [ 8] .data             PROGBITS         0000000000010470  00000470
```

```
WA       0     0     1
  [ 9] .dynamic          DYNAMIC          0000000000010470  00000470
WA       3     0     8
  [10] .got              PROGBITS         0000000000010570  00000570
WAp      0     0     8
  [11] .sbss             NOBITS           0000000000010590  00000590
W        0     0     1
  [12] .bss              NOBITS           0000000000010590  00000590
WA       0     0     1
  [13] .comment          PROGBITS         0000000000000000  00000590
0     0     1
  [14] .shstrtab         STRTAB           0000000000000000  000005b6
       000000000000008a  0000000000000000           0     0     1
  [15] .symtab           SYMTAB           0000000000000000  00000a80
16    12     8
  [16] .strtab           STRTAB           0000000000000000  00000cd8
0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

يوضّح المثال السابق كيفية إنشاء مكتبة مشتركة بسيطة تشير إلى رمز خارجي. لا نعرف عنوان هذا الرمز في وقت التصريف، لذلك نتركه للرابط الديناميكي لإصلاحه في وقت التشغيل. لكننا نريد أن تبقى الشيفرة البرمجية قابلةً للمشاركة في حالة رغبة العمليات الأخرى في استخدام هذه الشيفرة، حيث يوضّح التفكيك Disassembly كيفية تطبيق ذلك باستخدام القسم `..got`. يُعرف المسجّل `r1` في معمارية IA64 التي صُرِّفت المكتبة من أجلها بالمؤشر العام Global Pointer الذي يؤشّر دائمًا إلى مكان تحميل القسم `.got` في الذاكرة.

إذا ألقينا نظرة على خرج الأداة `readelf`، فيمكننا أن نرى أن القسم `.got` يبدأ عند عنوان يبعد بمقدار 0x10570 بايت عن مكان تحميل المكتبة في الذاكرة. لذا إذا حُمِّلت المكتبة في الذاكرة عند العنوان 0x6000000000000000، فسيكون القسم `.got` موجودًا عند العنوان 0x6000000000010570، وسيؤشّر المسجّل `r1` دائمًا إلى هذا العنوان.

كما يمكننا أن نرى أننا نخزن القيمة 100 في عنوان الذاكرة الموجود في المسجّل `r15` الذي يحتوي على قيمة عنوان الذاكرة المخزن في المسجل `r14`، حيث حمّلنا هذا العنوان من خلال إضافة عدد صغير إلى المسجّل 1. يُعَد جدول GOT مجرد قائمة طويلة من المدخلات، حيث تكون كل مدخلة خاصةً بمتغير خارجي، مما يعني أن مدخلة جدول GOT الخاصة بالمتغير الخارجي `i` تخزّن 24 بايت أو 3 عناوين بحجم 64 بت.

```
$ readelf --relocs ./got.so

Relocation section '.rela.dyn' at offset 0x3f8 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000f00000027 R_IA64_DIR64LSB   0000000000000000 i + 0
```

كما يمكننا التحقق من انتقال مدخلة جدول GOT، حيث يمثل الانتقالُ استبدالَ القيمة عند الإزاحة 10588 بموقع الذاكرة الذي خُزِّن فيه الرمز `i`.

يبدأ القسم `.got` عند الإزاحة 0x10570 عن الخرج السابق، حيث رأينا كيف تحمّل الشيفرة البرمجية عنوانًا يبعد عن القسم `.got` بمقدار 0x18 (أو 24 في النظام العشري)، مما يمنحنا عنوانًا مقداره 0x18 + 0x10570 = 0x10588 يمثّل العنوان الذي طُبِّق الانتقال لأجله. لذا يجب أن يصلح الرابط الديناميكي الانتقال قبل أن يبدأ البرنامج للتأكد من أن قيمة الذاكرة عند الإزاحة 0x10588 هي عنوان المتغير العام `i`.

9.4 المزيد حول المكتبات

يمكن أن تحتوي المكتبات على العديد من الدوال، ويمكن أن يحتوي البرنامج على العديد من المكتبات لإنجاز عمله. يستخدم البرنامج دالة أو دالتين فقط من كل مكتبة من المكتبات المتعددة المتاحة، ويمكن أن تستخدم الشيفرة البرمجية بعض الدوال دون غيرها اعتمادًا على مسار وقت التشغيل.

تحتوي عملية الربط الديناميكي Dynamic Linking التي شرحناها في الفصل السابق على الكثير من العمليات الحسابية، لأنها تتضمن النظر والبحث عبر العديد من الجداول، لذا يمكن تحسين الأداء عند تطبيق تقنيات تساعد في تقليل هذا الحِمل الناتج عن هذه العمليات الحسابية الكثيرة وهو ما سنشرحه في الفقرات التالية.

9.4.1 جدول البحث عن الإجراءات Procedure Lookup Table

يسهّل جدول البحث عن الإجراءات Procedure Lookup Table -أو PLT اختصارًا- ما يسمى بالارتباط الكسول Lazy Binding في البرامج، حيث يُعَد الارتباط Binding مرادفًا لعملية إصلاح المتغيرات الموجودة في جدول GOT الموضحة سابقًا، إذ يُقال أن المدخلة مرتبطة بعنوانها الفعلي عند إصلاحها.

يتضمن البرنامج في بعض الأحيان دالةً من مكتبة، ولكنه لا يستدعيها أبدًا اعتمادًا على دخل المستخدم. تحتوي عملية الارتباط الخاصة بهذه الدالة الكثير من العمليات لتطبيقها، لأنها تتضمن تحميل الشيفرة البرمجية والبحث في الجداول والكتابة في الذاكرة، لذا تُعَد المتابعة في عملية ارتباط دالة غير مُستخدَمة مضيعة للوقت، حيث يؤجِّل الارتباط الكسول هذه العملية حتى يستدعي جدولُ PLT الدالةَ الفعلية.

لكل دالة في مكتبةٍ مدخلةٌ في جدول PLT تؤشّر في البداية إلى بعض الشيفرات البرمجية الوهمية Dummy Code الخاصة. إن استدعى البرنامج الدالة، فهذا يعني أنه يستدعي مدخلة من جدول PLT باستخدام الطريقة نفسها للإشارة إلى المتغيرات في جدول GOT نفسها.

تحمّل هذه الدالة الوهمية بعض المعاملات التي تريد تمريرها إلى الرابط الديناميكي لتتمكن من تحليل الدالة ثم استدعاء دالة بحث خاصة بالرابط الديناميكي. يجد الرابط الديناميكي عنوان الدالة الفعلي، ويكتب هذا الموقع في استدعاء الملف الثنائي في أعلى استدعاء الدالة الوهمية، وبالتالي يمكن تحميل العنوان دون الحاجة إلى العودة إلى المحمل الديناميكي مرة أخرى في المرة التالية لاستدعاء الدالة. إن لم تُستدعَى دالةٌ مطلقًا، فلن تُعدَّل مدخلة جدول PLT أبدًا ولكن لن يكون هناك وقت تشغيل إضافي.

ا. كيفية عمل جدول PLT

يجب أن تبدأ الآن في إدراك أن هناك قدرًا لا بأس به من العمل في تحليل رمز ديناميكي. لنطلع على تطبيق "hello World" البسيط الذي يجري استدعاء مكتبة واحد فقط هو استدعاء الدالة `printf` لعرض السلسلة النصية للمستخدم كما يلي:

```
$ cat hello.c
#include <stdio.h>

int main(void)
{
  printf("Hello, World!\n");
  return 0;
}

$ gcc -o hello hello.c

$ readelf --relocs ./hello

Relocation section '.rela.dyn' at offset 0x3f0 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name +
Addend
6000000000000ed8  000700000047 R_IA64_FPTR64LSB  0000000000000000
_Jv_RegisterClasses + 0
6000000000000ee0  000900000047 R_IA64_FPTR64LSB  0000000000000000
__gmon_start__ + 0

Relocation section '.rela.IA_64.pltoff' at offset 0x420 contains 3 entries:
```

```
  Offset          Info           Type           Sym. Value    Sym. Name +
Addend
6000000000000f10  000200000081 R_IA64_IPLTLSB    0000000000000000 printf + 0
6000000000000f20  000800000081 R_IA64_IPLTLSB    0000000000000000
__libc_start_main + 0
6000000000000f30  000900000081 R_IA64_IPLTLSB    0000000000000000
__gmon_start__ + 0
```

يمكننا أن نرى في المثال السابق أن لدينا الانتقال `R_IA64_IPLTLSB` للرمـز `printf` الـذي يمثـل وضـع عنوان رمز هذه الدالة في عنوان الذاكرة 0x6000000000000f10. يجب أن نبدأ في البحث بصورة أعمق للعثور على الإجراء الدقيق الذي يعطينا الدالة.

سنلقي في المثال التالي نظرة على تفكيك الدالة الرئيسية `main()` الخاصة بالبرنامج:

```
<main>:
00 08 15 08 80 05       [MII]       alloc r33=ar.pfs,5,4,0
20 02 30 00 42 60                   mov r34=r12
400000000000079c:       04 08 00 84                         mov r35=r1
40000000000007a0:       01 00 00 00 01 00       [MII]       nop.m 0x0
40000000000007a6:       00 02 00 62 00 c0                   mov r32=b0
40000000000007ac:       81 0c 00 90                         addl r14=72,r1;;
40000000000007b0:       1c 20 01 1c 18 10       [MFB]       ld8 r36=[r14]
40000000000007b6:       00 00 00 02 00 00                   nop.f 0x0
40000000000007bc:       78 fd ff 58                         br.call.sptk.many
b0=4000000000000520 <_init+0xb0>
40000000000007c0:       02 08 00 46 00 21       [MII]       mov r1=r35
40000000000007c6:       e0 00 00 00 42 00                   mov r14=r0;;
40000000000007cc:       01 70 00 84                         mov r8=r14
40000000000007d0:       00 00 00 00 01 00       [MII]       nop.m 0x0
40000000000007d6:       00 08 01 55 00 00                   mov.i ar.pfs=r33
40000000000007dc:       00 0a 00 07                         mov b0=r32
40000000000007e0:       1d 60 00 44 00 21       [MFB]       mov r12=r34
40000000000007e6:       00 00 00 02 00 80                   nop.f 0x0
40000000000007ec:       08 00 84 00                         br.ret.sptk.many
b0;;
```

يجب أن يكـون اسـتدعاء العنـوان 0x4000000000000520 هـو اسـتدعاء الدالـة `printf`، حيث يمكننـا معرفة مكان هذا العنوان من خلال الاطلاع الأقسام Sections باستخدام الأداة `readelf` كما يلي:

```
$ readelf --sections ./hello
There are 40 section headers, starting at offset 0x25c0:
```

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
0000000000000000           0     0     0
...
  [11] .plt              PROGBITS         40000000000004c0  000004c0
       00000000000000c0  0000000000000000  AX       0     0     32
  [12] .text             PROGBITS         4000000000000580  00000580
       00000000000004a0  0000000000000000  AX       0     0     32
  [13] .fini             PROGBITS         4000000000000a20  00000a20
0000000000000000  AX       0     0     16
  [14] .rodata           PROGBITS         4000000000000a60  00000a60
       000000000000000f  0000000000000000   A       0     0     8
  [15] .opd              PROGBITS         4000000000000a70  00000a70
0000000000000000   A       0     0     16
  [16] .IA_64.unwind_inf PROGBITS         4000000000000ae0  00000ae0
       00000000000000f0  0000000000000000   A       0     0     8
  [17] .IA_64.unwind     IA_64_UNWIND     4000000000000bd0  00000bd0
       00000000000000c0  0000000000000000  AL      12     c     8
  [18] .init_array       INIT_ARRAY       6000000000000c90  00000c90
0000000000000000  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       6000000000000ca8  00000ca8
0000000000000000  WA       0     0     8
  [20] .data             PROGBITS         6000000000000cb0  00000cb0
0000000000000000  WA       0     0     4
  [21] .dynamic          DYNAMIC          6000000000000cb8  00000cb8
       00000000000001e0  0000000000000010  WA       5     0     8
  [22] .ctors            PROGBITS         6000000000000e98  00000e98
0000000000000000  WA       0     0     8
  [23] .dtors            PROGBITS         6000000000000ea8  00000ea8
0000000000000000  WA       0     0     8
  [24] .jcr              PROGBITS         6000000000000eb8  00000eb8
0000000000000000  WA       0     0     8
  [25] .got              PROGBITS         6000000000000ec0  00000ec0
0000000000000000 WAp       0     0     8
  [26] .IA_64.pltoff     PROGBITS         6000000000000f10  00000f10
0000000000000000 WAp       0     0     16
```

```
  [27] .sdata            PROGBITS         6000000000000f40  00000f40
0000000000000000 WAp       0     0     8
  [28] .sbss             NOBITS           6000000000000f50  00000f50
0000000000000000  WA       0     0     8
  [29] .bss              NOBITS           6000000000000f58  00000f50
0000000000000000  WA       0     0     8
  [30] .comment          PROGBITS         0000000000000000  00000f50
       00000000000000b9  0000000000000000           0     0     1
  [31] .debug_aranges    PROGBITS         0000000000000000  00001010
0000000000000000           0     0     16
  [32] .debug_pubnames   PROGBITS         0000000000000000  000010a0
0000000000000000           0     0     1
  [33] .debug_info       PROGBITS         0000000000000000  000010c5
       00000000000009c4  0000000000000000           0     0     1
  [34] .debug_abbrev     PROGBITS         0000000000000000  00001a89
0000000000000000           0     0     1
  [35] .debug_line       PROGBITS         0000000000000000  00001bad
       00000000000001fe  0000000000000000           0     0     1
  [36] .debug_str        PROGBITS         0000000000000000  00001dab
       00000000000006a1  0000000000000001  MS       0     0     1
  [37] .shstrtab         STRTAB           0000000000000000  0000244c
       000000000000016f  0000000000000000           0     0     1
  [38] .symtab           SYMTAB           0000000000000000  00002fc0
       0000000000000b58  0000000000000018          39    60     8
  [39] .strtab           STRTAB           0000000000000000  00003b18
0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

يوجد هذا العنوان في القسم `.plt` كما هـو متوقـع حيث يوجـد اسـتدعاؤها في جـدول PLT. لكن لنواصـل البحث أكثر ولنفكك القسم `.plt` لنرى ما يفعله هذا الاستدعاء كما يلي:

```
40000000000004c0 <.plt>:
40000000000004c0:       0b 10 00 1c 00 21       [MMI]       mov r2=r14;;
40000000000004c6:       e0 00 08 00 48 00                   addl r14=0,r2
40000000000004cc:       00 00 04 00                         nop.i 0x0;;
40000000000004d0:       0b 80 20 1c 18 14       [MMI]       ld8 r16=[r14],8;;
```

```
40000000000004d6:       10 41 38 30 28 00                   ld8 r17=[r14],8
40000000000004dc:       00 00 04 00                         nop.i 0x0;;
40000000000004e0:       11 08 00 1c 18 10       [MIB]       ld8 r1=[r14]
40000000000004e6:       60 88 04 80 03 00                   mov b6=r17
40000000000004ec:       60 00 80 00                         br.few b6;;
40000000000004f0:       11 78 00 00 00 24       [MIB]       mov r15=0
40000000000004f6:       00 00 00 02 00 00                   nop.i 0x0
40000000000004fc:       d0 ff ff 48                         br.few
40000000000004c0 <_init+0x50>;;
11 78 04 00 00 24       [MIB]       mov r15=1
00 00 00 02 00 00                   nop.i 0x0
400000000000050c:       c0 ff ff 48                         br.few
40000000000004c0 <_init+0x50>;;
11 78 08 00 00 24       [MIB]       mov r15=2
00 00 00 02 00 00                   nop.i 0x0
400000000000051c:       b0 ff ff 48                         br.few
40000000000004c0 <_init+0x50>;;
0b 78 40 03 00 24       [MMI]       addl r15=80,r1;;
00 41 3c 70 29 c0                   ld8.acq r16=[r15],8
400000000000052c:       01 08 00 84                         mov r14=r1;;
11 08 00 1e 18 10       [MIB]       ld8 r1=[r15]
60 80 04 80 03 00                   mov b6=r16
400000000000053c:       60 00 80 00                         br.few b6;;
0b 78 80 03 00 24       [MMI]       addl r15=96,r1;;
00 41 3c 70 29 c0                   ld8.acq r16=[r15],8
400000000000054c:       01 08 00 84                         mov r14=r1;;
11 08 00 1e 18 10       [MIB]       ld8 r1=[r15]
60 80 04 80 03 00                   mov b6=r16
400000000000055c:       60 00 80 00                         br.few b6;;
0b 78 c0 03 00 24       [MMI]       addl r15=112,r1;;
00 41 3c 70 29 c0                   ld8.acq r16=[r15],8
400000000000056c:       01 08 00 84                         mov r14=r1;;
11 08 00 1e 18 10       [MIB]       ld8 r1=[r15]
60 80 04 80 03 00                   mov b6=r16
400000000000057c:       60 00 80 00                         br.few b6;;
```

إذًا لنمر على التعليمات، حيث أضفنا أولًا القيمة 80 إلى القيمة الموجودة في المسجّل r1، وخزّناها في المسجّل r15. سيؤشّر المسجل r1 إلى جدول GOT، مما يعني تخزين المسجل r15 الذي يحتوي على 80 بايت في جدول GOT. ثانيًا، حمّلنا القيمة المخزنة في هذا الموقع من جدول GOT إلى المسجّل r16، ثم زدنا القيمة الموجودة في المسجل r15 بمقدار 8 بايتات. ثالثًا، خزّنا المسجّل r1 -أو موقع جدول GOT- في

المسجّل r14 وضبطنا القيمة الموجودة في المسجل r1 لتكون القيمة الموجودة في 8 بايتات التالية للمسجّل r15، ثم نتفرّع إلى المسجل r16.

ناقشنا سابقًا كيفية استدعاء الدوال باستخدام واصف الدالة Function Descriptor الذي يحتوي على عنوان الدالة وعنوان المؤشر العام. يمكننا أن نرى أن مدخلة جدول PLT تحمّل أولًا قيمة الدالة، مما يؤدي إلى الانتقال بمقدار 8 بايتات إلى الجزء الثاني من واصف الدالة ثم تحميل تلك القيمة في مسجّل العملية Op Register قبل استدعاء الدالة.

نعلم أن المسجل r1 سيؤشّر إلى جدول GOT، ثم سنذهب بمقدار 80 بايت بعد جدول GOT أي بمقدار (0x50).

```
$ objdump --disassemble-all ./hello
Disassembly of section .got:

6000000000000ec0 <.got>:
        ...
6000000000000ee8:       80 0a 00 00 00 00                   data8 0x02a000000
6000000000000eee:       00 40 90 0a                         dep r0=r0,r0,63,1
6000000000000ef2:       00 00 00 00 00 40       [MIB] (p20) break.m 0x1
6000000000000ef8:       a0 0a 00 00 00 00                   data8 0x02a810000
6000000000000efe:       00 40 50 0f                         br.few
6000000000000ef0 <_GLOBAL_OFFSET_TABLE_+0x30>
6000000000000f02:       00 00 00 00 00 60       [MIB] (p58) break.m 0x1
6000000000000f08:       60 0a 00 00 00 00                   data8 0x029818000
6000000000000f0e:       00 40 90 06                         br.few
6000000000000f00 <_GLOBAL_OFFSET_TABLE_+0x40>
Disassembly of section .IA_64.pltoff:

6000000000000f10 <.IA_64.pltoff>:
6000000000000f10:       f0 04 00 00 00 00       [MIB] (p39) break.m 0x0
6000000000000f16:       00 40 c0 0e 00 00                   data8 0x03b010000
6000000000000f1c:       00 00 00 60                         data8 0xc000000000
6000000000000f20:       00 05 00 00 00 00       [MII] (p40) break.m 0x0
6000000000000f26:       00 40 c0 0e 00 00                   data8 0x03b010000
6000000000000f2c:       00 00 00 60                         data8 0xc000000000
6000000000000f30:       10 05 00 00 00 00       [MIB] (p40) break.m 0x0
6000000000000f36:       00 40 c0 0e 00 00                   data8 0x03b010000
6000000000000f3c:       00 00 00 60                         data8 0xc000000000
```

إذا أضـــــفنا القيمـــــة 0x50 إلى العنـــــوان 0x6000000000000ec0، فسنصـــــل إلى العنـــــوان 0x6000000000000f10 أو القسم ..IA_64.pltoff

يمكننا فك شيفرة خرج البرنامج objdump لنتمكّن من رؤية ما جرى تحميله بالضبط. يـؤدي تبـديل تـرتيب البايت لأول 8 بايتات f0 04 00 00 00 00 00 40 إلى الحصول على العنوان 0x4000000000004f0، إذ يبـدو هـذا العنـوان مألوفًـا، حيث إذا نظرنـا إلى الـوراء في نـاتج التجميـع الخـاص بجـدول PLT ، فسـنرى ذلك العنوان.

أولًا تضع الشيفرة البرمجية الموجودة عند العنوان 0x4000000000004f0 قيمة صفرية في المسجل r15، ثم تتفر ع مرة أخرى إلى العنوان 0x40000000000004c0، ولكن يُعَـد هـذا العنـوان بدايـة القسـم PLT. يمكننـا تتبّـع هـذه الشـيفرة البرمجيـة، إذ نحفـظ أولًا قيمـة المؤشـر العـام في المسـجل r2، ثم نحمـل ثلاث قيم بحجم 8 بايتات في المسجلات r16 وr17 وr1، ثم نتفر ع إلى العنوان الموجود في المسجل r17، حيث يمثّـل تلـك العملية الاستدعاء الفعلي للرابط الديناميكي.

يجب أن نتعمق قليلًا في فهم واجهة ABI التي تعطينا مفهومين لنفهم بالضبط ما يجري تحميله الآن، وهذا المفهومـان همـا أنـه يجب أن تحتـوي الـبرامج المرتبطـة ديناميكيًـا على قسـم خـاص يسـمى القسـم DT_IA_64_PLT_RESERVE الذي يمكنه الاحتفاظ بثلاث قيم بحجم 8 بايتات، ويوجـد مؤشـر في مكـان وجـود هذه المنطقة المحجوزة في المقطع الديناميكي للملف الثنائي الموضّح في المثال التالي:

```
Dynamic segment at offset 0xcb8 contains 25 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
 0x000000000000000c (INIT)               0x4000000000000470
 0x000000000000000d (FINI)               0x4000000000000a20
 0x0000000000000019 (INIT_ARRAY)         0x6000000000000c90
 0x000000000000001b (INIT_ARRAYSZ)       24 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x6000000000000ca8
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x0000000000000004 (HASH)               0x4000000000000200
 0x0000000000000005 (STRTAB)             0x4000000000000330
 0x0000000000000006 (SYMTAB)             0x4000000000000240
 0x000000000000000a (STRSZ)              138 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000070000000 (IA_64_PLT_RESERVE)  0x6000000000000ec0 --
0x6000000000000ed8
 0x0000000000000003 (PLTGOT)             0x6000000000000ec0
 0x0000000000000002 (PLTRELSZ)           72 (bytes)
```

```
0x0000000000000014 (PLTREL)             RELA
0x0000000000000017 (JMPREL)             0x4000000000000420
0x0000000000000007 (RELA)               0x40000000000003f0
0x0000000000000008 (RELASZ)             48 (bytes)
0x0000000000000009 (RELAENT)            24 (bytes)
0x000000006ffffffe (VERNEED)            0x40000000000003d0
0x000000006fffffff (VERNEEDNUM)         1
0x000000006ffffff0 (VERSYM)             0x40000000000003ba
0x0000000000000000 (NULL)               0x0
```

لاحظ أننا حصلنا على قيمة جدول GOT نفسه، وهذا يعني أن أول ثلاث مدخلات بحجم 8 بايتات في جدول GOT تمثل المنطقة المحجوزة، وبالتالي سيُؤشَّر إليها دائمًا باستخدام المؤشر العام.

يجب أن يملأ الرابط الديناميكي هذه القيم عند بدء تشغيله، حيث تحدّد واجهة ABI أنه يجب ملء القيمة الأولى بواسطة الرابط الديناميكي الذي يمنح هذه الوحدة معرفًا فريدًا، والقيمة الثانية هي قيمة المؤشر العام للرابط الديناميكي، والقيمة الثالثة هي عنوان الدالة التي تبحث عن الرمز وتصلحه.

يوضّح المثال التالي شيفرة برمجية في الرابط الديناميكي لإعداد قيم خاصة من المكتبة libc أو من sysdeps/ia64/dl-machine.h:

```
/*
* إعداد الكائن المحمَّل الموصوف باستخدام المتغير L حتى تقفز مدخلات جدول PLT التي
* ليس لها انتقالات إلى شيفرة الإصلاح البرمجية عند الطلب في ملف .dl-runtime.c
*/
static inline int __attribute__ ((unused, always_inline))
elf_machine_runtime_setup (struct link_map *l, int lazy, int profile)
{
  extern void _dl_runtime_resolve (void);
  extern void _dl_runtime_profile (void);

  if (lazy)
    {
      register Elf64_Addr gp __asm__ ("gp");
      Elf64_Addr *reserve, doit;

        /*
         * احذر من تبديل الأنواع Typecast هنا أو ستُضاف عناصر مؤشر l-l_addr
         */
```

```
        reserve = ((Elf64_Addr *)
                   (l->l_info[DT_IA_64 (PLT_RESERVE)]->d_un.d_ptr + l-
                      >l_addr));
      /* تعريف هذا الكائن المشترك  */
       reserve[0] = (Elf64_Addr) l;

      /* Relocation ستُستدعَى هذه الدالة لتطبيق الانتقال */
       if (!profile)
         doit = (Elf64_Addr) ((struct fdesc *) &_dl_runtime_resolve)-
               >ip;
       else
         {
           if (GLRO(dl_profile) != NULL
              && _dl_name_match_p (GLRO(dl_profile), l))
             {
             /* هذا هو الكائن الذي نبحث عنه. لنفترض أننا نريد استخدام التشخيص
             Profiling مع بدء المؤقتات */
               GL(dl_profile_map) = l;
             }
           doit = (Elf64_Addr) ((struct fdesc *) &_dl_runtime_profile)-
               >ip;
         }

       reserve[1] = doit;
       reserve[2] = gp;
     }

   return lazy;
 }
```

يمكننا أن نرى كيفية إعداد هذه القيم بواسطة الرابط الديناميكي من خلال النظر في الدالة التي تطبّق ذلك للملف الثنائي. يُضبَط المتغير `reserve` من مؤشر القسم `PLT_RESERVE` في الملف الثنائي. تمثل القيمة الفريدة الموضوعة في `reserve[0]` عنوان خارطة الربط Link Map لهذا الكائن، حيث تُعَد خارطة الربط التمثيل الداخلي ضمن مكتبة `glibc` للكائنات المشتركة.ثم نضع بعد ذلك عنوان الدالة `_dl_runtime_resolve` في القيمة الثانية بافتراض أننا لا نستخدم عملية التشخيص Profiling، ثم تُضبط قيمة `reserve[2]` على `gp` التي يمكن العثور عليها في المسجل `r2` باستخدام الاستدعاء `__asm__`.

إذا عدنا إلى الوراء في واجهة ABI، فسنرى أنه يجب وضع فهرس انتقال للمدخلة في المسجل `r15` ويجب تمرير المعرّف الفريد في المسجل `r16`. ضُبِط المسجل `r15` مسبقًا في الشيفرة الاختبارية Stub Code قبل العودة إلى بداية جدول PLT. ألقِ نظرة على المدخلات، ولاحظ كيف تحمِّل كل مدخلة في جدول PLT المسجل `r15` مع قيمة متزايدة، إذ لا ينبغي أن يكون ذلك مفاجئًا إذا نظرت إلى عمليات الانتقال، حيث يكون لانتقال الدالة `printf` العدد صفر.

نحمّل المسجل `r16` من القيم التي هيّأها الرابط الديناميكي، ثم يمكننا تحميل عنوان الدالة والمؤشر العام والفرع في الدالة، ثم نشغّل دالة الرابط الديناميكي `_dl_runtime_resolve` التي تعثر على الانتقال. يستخدم الانتقال اسم الرمز الذي حدّده للعثور على الدالة الصحيحة، حيث يمكن يتضمن ذلك تحميل المكتبة من القرص الصلب إن لم تكن موجودة في الذاكرة، وإلّا فيجب مشاركة الشيفرة البرمجية.

يوفر سجلُ الانتقال للرابط الديناميكي العنوانَ الذي يجب إصلاحه، حيث كان هذا العنوان موجودًا في جدول GOT ثم حمّلته شيفرة PLT الاختبارية، وهذا يعني أنه يمكن الحصول على عنوان الدالة المباشر أو ما يسمى بتقصير دورة الرابط الديناميكي بعد المرة الأولى التي تُستدعَى فيها الدالة أي في المرة الثانية لتحميلها.

رأينا الآلية الدقيقة لعمل جدول PLT والعمل الداخلي للرابط الديناميكي. النقاط المهمة التي يجب تذكرها هي:

- تستدعي استدعاءات المكتبة في برنامجك الشيفرة الاختبارية في جدول PLT الخاص بالملف الثنائي.
- تحمّل هذه الشيفرة الاختبارية عنوانًا وتقفز إليه.
- يؤشّر هذا العنوان إلى دالةٍ في الرابط الديناميكي قادرةٍ على البحث عن الدالة الحقيقية من خلال النظر إلى المعلومات الواردة في مدخلة الانتقال لتلك الدالة.
- يعيد الرابط الديناميكي كتابة العنوان الذي تقرأه الشيفرة الاختبارية، بحيث تنتقل الدالة مباشرة إلى العنوان الصحيح في المرة التالية لاستدعائها.

9.5 عمل الرابط الديناميكي مع المكتبات

يوفر وجود الربط الديناميكي Dynamic Linking بعض المزايا التي يمكننا الاستفادة منها وبعض المشاكل الإضافية التي يجب حلها للحصول على نظام فعّال.

9.5.1 إصدارات المكتبات

إحدى المشاكل المُحتمَلة هي وجود إصدارات مختلفة للمكتبات. لكن هناك احتمال أقل بكثير لوجود مشاكل عند استخدام المكتبات الساكنة، حيث تُدمَج شيفرة المكتبة البرمجية مباشرةً في الملف الثنائي الخاص بالتطبيق. إن أردتَ استخدام إصدار جديد من المكتبة، فيجب إعادة تصريفها في ملف ثنائي جديد لتحل محل

الإصدار القديم. يُعَد ذلك أمرًا غير عملي إلى حد ما بالنسبة للمكتبات الشائعة وأكثرها شيوعًا مكتبه libc والمُضمَّنة في معظم التطبيقات. إذا كانت المكتبة متوفرة فقط بوصفها مكتبة ساكنة، فيجب إعادة بناء كل تطبيق في النظام عند أي تعديل فيها.

يمكن أن تسبّب التعديلات في طريقة عمل المكتبة الديناميكية مشكلات متعددة. تكون التعديلات في أحسن الأحوال متوافقة تمامًا دون تغيير أيّ شيء مرئي خارجيًا، ولكن يمكن أن تتسبب التعديلات في تعطل التطبيق مثل تغير الدالة التي تأخذ النوع `int` لتأخذ النوع `int *`. الأسوأ من ذلك هو أن يغيّر إصدارُ المكتبة الجديد الدلالات ويعيد قيمًا مختلفة وخاطئة فجأةً. يمكن أن يكون هذا خطأً يصعب تعقّبه، حيث إن تعطل أحد التطبيقات، فيمكنك استخدام منقّح أخطاء Debugger لعزل مكان حدوث الخطأ، بينما يمكن أن يظهر تلف البيانات أو تعديلها فقط في أجزاء أخرى من التطبيق.

يتطلب الرابط الديناميكي طريقة لتحديد إصدار المكتبات في النظام بحيث يمكن التعرّف على التعديلات الأحدث. هناك عدد من الأنظمة التي يمكن للرابط الديناميكي الحديث استخدامها للعثور على الإصدارات الصحيحة من المكتبات التي سنوضّحها فيما يلي.

ا. نظام sonames

يُستخدَم نظام `sonames` لإضافة بعض المعلومات الإضافية إلى مكتبة للمساعدة في تحديد الإصدارات. يسرد التطبيق المكتبات التي يريدها في الحقول `DT_NEEDED` ضمن القسم الديناميكي للملف الثنائي، وتوجد المكتبة الفعلية في ملف على القرص الصلب ضمن المجلد `/lib` لمكتبات النظام الأساسية أو المجلد `/usr/lib` للمكتبات الاختيارية.

يتطلب وجودُ إصدارات متعددة من المكتبة على القرص الصلب استخدامَ أسماء ملفات مختلفة. لذا يستخدم نظام `sonames` مجموعة من الأسماء وروابطًا إلى نظام الملفات لبناء تسلسل هرمي من المكتبات من خلال تقديم مفهوم التعديلات الرئيسية Major والثانوية Minor للمكتبة. يُعَد التعديل الثانوي تعديلًا متوافقًا مع إصدار سابق من المكتبة، ويتكون من إصلاحاتٍ للأخطاء فقط. بينما يُعَد التعديل الرئيسي أي تعديل غير متوافق مثل تغيير دخل الدوال أو الطريقة التي تتصرف بها الدالة.

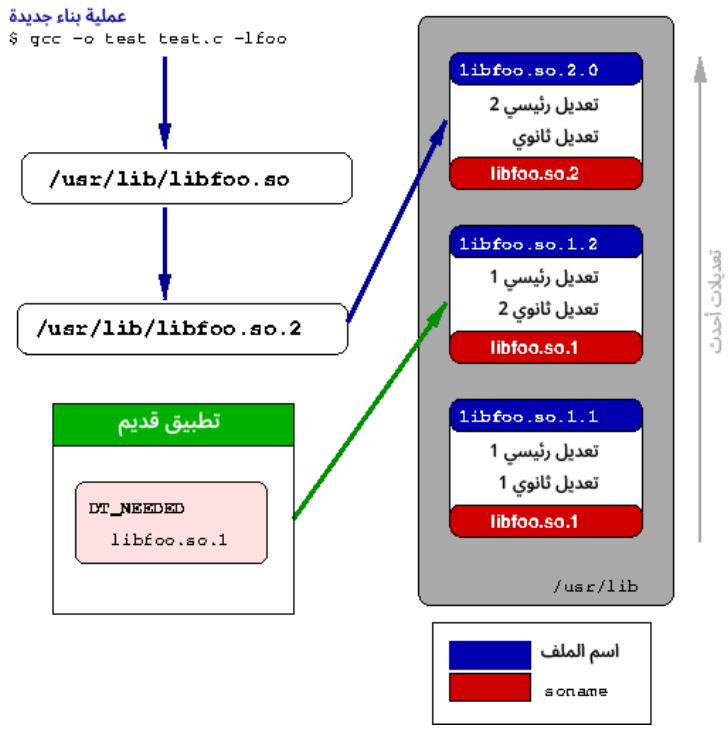
تشكّل الحاجة إلى الاحتفاظ بكل تعديل مكتبة رئيسي أو ثانوي في ملف منفصل على القرص الصلب أساسَ تسلسل المكتبات الهرمي. يكون اسم المكتبة هو `libNAME.so.MAJOR.MINOR` حسب العِرف المتبع، حيث يمكنك اختياريًا الحصول على إطلاق Release بوصفه معرفًا نهائيًا بعد العدد الثانوي، ويكفي ذلك لتمييز جميع إصدارات المكتبة المختلفة. لكن إذا رُبِط كل تطبيق بهذا الملف مباشرةً، فسنواجه المشكلة نفسها التي واجهناها مع المكتبة الساكنة، إذ يجب إعادة بناء التطبيق للإشارة إلى المكتبة الجديدة في كل مرة يحدث فيها تعديل ثانوي. ما نريده هو أن نشير إلى ما يمثله العدد الرئيسي Major من المكتبة الذي إن تغير، فيجب إعادة تصريف Recompile تطبيقنا، لأننا نحتاج إلى التأكد من أن برنامجنا لا يزال متوافقًا مع المكتبة الجديدة.

يكـون soname بالشـكل libNAME.so.MAJOR، ويجب ضـبطه في الحقـل DT_SONAME من القسـم الديناميكي لمكتبة مشتركة، حيث يمكن لمؤلف المكتبة تحديد هذا الإصدار عند إنشاء المكتبة.

يمكن أن يحدّد كل ملف مكتبة للإصدار الثانوي على القرص الصلب رقمَ الإصدار الرئيسي نفسه في الحقـل DT_SONAME، مما يسمح للرابط الديناميكي بمعرفة أن ملف المكتبة يطبّق تعديلًا رئيسـيًا معينًـا لواجهـتي API و ABI الخاصتين بالمكتبة.

لذا يُشغَّل تطبيق اسمه ldconfig لإنشاء روابط رمزية للإصدار الرئيسي إلى أحدث إصدار ثانوي على النظام. يعمـل تطـبيق ldconfig من خلال تشـغيل جميـع المكتبـات الـتي تطبّـق رقم إصـدار رئيسـي معين، ثم يختـار المكتبة التي تحتوي على أعلى رقم تعديل ثـانوي، ثم ينشِـئ رابطًـا رمزيًـا من libNAME.so.MAJOR إلى ملـف المكتبة الفعلي الموجود على القرص الصلب مثل libNAME.so.MAJOR.MINOR.

الجزء الأخير من التسلسل الهرمي هو اسم تصريف Compile Name المكتبة. إن أردت تصريف برنامجك لربطه بمكتبة، فيمكنك استخدام الراية -lNAME التي تبحث عن الملف libNAME.so في مسار بحث المكتبة. لاحظ أننا لم نحدد أي رقم إصدار، لأننا نريد فقط الربط بأحدث مكتبة على النظام. يعود الأمـر إلى إجـراء التثـبيت الخاص بالمكتبة لإنشاء رابط رمزي بين اسم التصريف libNAME.so وأحدث شيفرة مكتبة على النظام، ويمكن التعامل مع ذلك باستخدام نظام الحزم dpkg أو rpm. لا يُعَد ذلك عملية آلية، إذ يُحتمَل ألّا تكون أحـدث مكتبـة على النظام هي المكتبة التي ترغب في تصريفها دائمًا، فمثلًا يمكن أن تكون أحدث مكتبة مُثبَّتة إصدارًا تطويريًـا غير مناسب للاستخدام العام، ويوضح الشكل التالي العملية العامة لنظام sonames:



شكل 40: نظام sonames

كيف يبحث الرابط الديناميكي عن المكتبات

يبحث الرابط الديناميكي في الحقل DT_NEEDED للعثور على المكتبات المطلوبة عند بدء تشغيل التطبيق، حيث يحتوي هذا الحقل على اسم soname الخاص بالمكتبة، لذا فالخطوة التالية هي أن يمر الرابط الديناميكي على جميع المكتبات في مسار بحثه بحثًا عن المكتبة المطلوبة.

تتضمن هذه العملية من الناحية النظرية خطوتين. أولًا، يجب أن يبحث الرابط الديناميكي في جميع المكتبات للعثور على تلك المكتبات التي تطبّق نظام soname المحدد. ثانيًا، يجب موازنة أسماء الملفات الخاصة بالتعديلات الثانوية للعثور على أحدث إصدار والذي يكون جاهزًا للتحميل لاحقًا.

ذكرنا سابقًا أن هناك رابطًا رمزيًا أعدّه برنامج ldconfig بين اسم soname الخاص بالمكتبة والتعديل الثانوي الأخير، وبالتالي يجب أن يتبع الرابط الديناميكي هذا الرابط فقط للعثور على الملف الصحيح المراد تحميله بدلًا من الاضطرار إلى فتح جميع المكتبات الممكنة وتحديد المكتبات التي تريد استخدامها في كل مرة يكون التطبيق مطلوبًا فيها.

يُعَد الوصول إلى نظام الملفات بطيئًا جدًا، لذا ينشئ برنامج ldconfig ذاكرة مخبئية للمكتبات المُثبَّتة في النظام، حيث تكون هذه الذاكرة المخبئية ببساطة قائمةً بأسماء soname الخاصة بالمكتبات المتاحة للرابط الديناميكي ومؤشرًا لرابط الإصدار الرئيسي على القرص الصلب، مما يوفر على الرابط الديناميكي قراءة مجلدات كاملة مليئة بالملفات لتحديد الرابط الصحيح. يمكنك تحليل ذلك باستخدام /sbin/ldconfig -p الموجود ضمن الملف /etc/ldconfig.so.cache. إن لم يُعثَر على المكتبة في الذاكرة المخبئية، فسيعود الرابط الديناميكي إلى الخيار الأبطأ المتمثل في المرور على نظام الملفات، وبالتالي يجب إعادة تشغيل برنامج ldconfig عند تثبيت مكتبات جديدة.

9.5.2 البحث عن الرموز

ناقشنا كيف حصل الرابط الديناميكي على عنوان دالة المكتبة ووضعه في جدول PLT ليستخدمه البرنامج، ولكننا لم نناقش حتى الآن كيف يجد الرابط الديناميكي عنوان الدالة. تُسمَّى هذه العملية بالارتباط Binding، لأن اسم الرمز مرتبط بالعنوان الذي يمثله.

يحتوي الرابط الديناميكي على أجزاء من المعلومات مثل الرمز الذي يبحث عنه وقائمة المكتبات التي يمكن أن يكون هذا الرمز فيها كما هو محدَّد باستخدام حقول DT_NEEDED في الملف الثنائي. تحتوي كل مكتبة كائنات مشتركة على قسم يسمى .dynsym مميَّز على أنه SHT_DYNSYM، حيث يُعَد هذا القسم الحد الأدنى من مجموعة الرموز المطلوبة للربط الديناميكي، وهو أيّ رمز في المكتبة يمكن أن يستدعيه برنامج خارجي.

ا. جدول الرموز الديناميكي

هناك ثلاثة أقسام تلعب جميعها دورًا في وصف الرمـوز الديناميكيـة. لنلـقِ أولًا نظـرة على تعريـف رمـز من مواصفات ملف ELF كما يلي:

```
typedef struct {
  Elf32_Word    st_name;
  Elf32_Addr    st_value;
  Elf32_Word    st_size;
  unsigned char st_info;
  unsigned char st_other;
  Elf32_Half    st_shndx;
} Elf32_Sym;
```

الحقل	القيمة
st_name	فهرس إلى جدول السلاسل النصية
st_value	القيمة الموجودة في كائن مشترك قابل للنقل، حيث تحتفظ هذه القيمة بالإزاحة عن قسم الفهرس المعطى في الحقل
st_size	أي حجم مرتبط بالرمز
st_info	معلومات عن ارتباط Binding الرمز الذي سنشرحه لاحقًا ويكون نوع هذا الرمز دالة أو كائن أو غير ذلك
st_other	غير مُستخدَم حاليًا
st_shndx	فهرس القسم الذي يوجد فيه الرمز (اطّلع على الحقل st_value)

جدول 25: حقول رمز ELF

تكون السلسلة النصية الفعلية لاسم الرمز ضمن قسم منفصل هو القسم .dynstr حيث تحتوي المدخلـة في هذا القسم فهرسًا إلى قسم السلاسل النصـية فقـط، ممـا يـؤدي إلى ظهـور مسـتوًى معين من الحِمـل على الرابط الديناميكي، إذ يجب أن يقرأ الرابط الديناميكي جميع مدخلات الرموز في القسم .dynstr ثم يتبع مؤشر الفهرس للعثور على اسم الرمز للمقارنة.

يمكن تسريع هذه العملية من خلال تقديم قسم ثالث يسمى .hash يحتوي على جدول تعمية Hash Table لأسماء رموز مدخلات جدول الرموز. يُحسَب جدول التعمية مسبقًا عند إنشاء المكتبة ويسمح للرابط الديناميكي بالعثور على مدخلة الرمز بصورة أسرع باستخدام عملية بحث واحدة أو اثنتين فقط.

ب. ارتباط الرموز Symbol Binding

تشير عملية العثور على عنوان رمز إلى عملية ارتباط هذا الرمز، ولكن ارتباط الرموز Symbol Binding له معنًى منفصل، إذ تفرض عملية ارتباط الرموز رؤيتها خارجيًا أثناء عملية الربط الديناميكي. يُعَد الرمز المحلي Local Symbol غير مرئي خارج ملف الكائن المُعرَّف ضمنه، بينما يُعَد الرمز العام Global Symbol مرئيًا لملفات الكائنات الأخرى ويمكن أن يحقِّق المراجعَ غير المُعرَّفة في كائنات أخرى. يكون المرجع الضعيف Weak Reference نوعًا خاصًا من المراجع العامة ذات الأولوية المنخفضة، مما يعني أنه مُصمَّم لتجاوزه كما سنرى لاحقًا.

يوضح المثال التالي برنامجًا بلغة سي C نحلّله لفحص ارتباطات الرموز:

```
$ cat test.c
static int static_variable;

extern int extern_variable;

int external_function(void);

int function(void)
{
  return external_function();
}

static int static_function(void)
{
  return 10;
}

#pragma weak weak_function
int weak_function(void)
{
  return 10;
}
```

```
$ gcc -c test.c
$ objdump --syms test.o

test.o:     file format elf32-powerpc

SYMBOL TABLE:
l    df *ABS*  00000000 test.c
l    d  .text  00000000 .text
l    d  .data  00000000 .data
l    d  .bss   00000000 .bss
l     F .text  00000024 static_function
l    d  .sbss  00000000 .sbss
l     O .sbss  00000004 static_variable
l    d  .note.GNU-stack        00000000 .note.GNU-stack
l    d  .comment       00000000 .comment
g     F .text  00000038 function
*UND*  00000000 external_function
0000005c  w    F .text  00000024 weak_function

$ nm test.o
         U external_function
T function
t static_function
s static_variable
0000005c W weak_function
```

لاحظ استخدام `#pragma` لتعريف الرمز الضعيف، حيث يُعَد `pragma` طريقة لإيصال معلومات إضافية إلى المصرِّف Compiler واستخدامه غير شائع، ولكن يكون في بعض الأحيان مطلوبًا لإخراج المصرِّف من العمليات المعتادة.

يمكن فحص الرموز باستخدام أداتين مختلفتين كما هو موضَّح في المثال السابق، حيث يظهر الارتباط في العمود الثاني في كلتا الحالتين، ويجب أن تكون الشيفرات البرمجية واضحة تمامًا.

تجاوز الرموز Overriding Symbols

يجب أن يكون المبرمج قادرًا على تجاوز رمز في مكتبة، مما يعني تخريب الرمز العادي بتعريفٍ مختلف. ذكرنا أن ترتيب البحث في المكتبات مُحدَّدٌ حسب ترتيب حقول `DT_NEEDED` داخل المكتبة، ولكن يمكن

إدخال مكتبات لتكون المكتبات الأخيرة التي يجري البحث عنها، وهذا يعني أنه سيُعثَر على أيّ رموز ضمنها بوصفها مرجعًا نهائيًا. يمكن تحقيق ذلك باستخدام متغير بيئة يسمى LD_PRELOAD يحدد المكتبات التي يجب أن يحمّلها الرابط في النهاية كما في المثال التالي:

```
$ cat override.c
#define _GNU_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <dlfcn.h>

pid_t getpid(void)
{
  pid_t (*orig_getpid)(void) = dlsym(RTLD_NEXT, "getpid");
  printf("Calling GETPID\n");

  return orig_getpid();
}

$ cat test.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  printf("%d\n", getpid());
}

$ gcc -shared -fPIC -o liboverride.so override.c -ldl
$ gcc -o test test.c
$ LD_PRELOAD=./liboverride.so ./test
Calling GETPID
15187
```

تجاوزنا في المثال السابق الدالة `getpid` لطباعة عبارة صغيرة عند استدعائها. نستخدم الدالة `dlysm` الـتي توفرها مكتبة `libc` مع وسيط يخبرها بالاستمرار والعثور على الرمز التالي المسمَّى `getpid`.

الرموز الضعيفة

الرمز الضعيف هو الرمز المُميَّز بأنه لـه أولويـة أقـل ويمكن تجـاوزه برمـز آخـر، حيث إن لم يُعثَـر على تقـديم Implementation آخـر أبـدًا، فسـيكون الرمـز الضـعيف هـو الرمـز المُسـتخدَم. لـذا يجب أن يحمّـل المحمـل الديناميكي جميـع المكتبـات ويتجاهـل الرمـوز الضـعيفة الموجـودة في تلـك المكتبـات لصـالح الرمـوز العاديـة الموجودة في مكتبات أخرى، حيث كانت هذه هي الطريقة المتبعة لتقديم معالجـة الرمـوز الضـعيفة في لينكس باستخدام مكتبة `glibc` سابقًا.

لكن كان ذلك غير صحيح بالنسبة لنص معيار يونكس في ذلك الوقت (SysVr4) الذي يفرض أنــه يجب أن يتعامل الرابط الساكن مع الرموز الضعيفة التي يجب أن تظل بعيدة عن الرابط الديناميكي. تطابق تقديم لينكس الخاص بجعل الرابط الديناميكي يتجاوز الرموز الضعيفة مع منصة IRIX الخاصة بشركة SGI واختلف عن الأنظمة الأخرى مثل Solaris و AIX في ذلك الوقت. لذا لغى المطورون هذا السلوك عنـدما أدركـوا أنـه ينتهـك المعيـار، وتغير السلوك القديم ليتطلّب ضبط راية بيئة خاصة (`LD_DYNAMIC_WEAK`).

تحديد ترتيب الارتباط

رأينا كيف يمكننا تجاوز دالة في مكتبة من خلال التحميل المسبق لمكتبة مشتركة أخرى لهـا الرمـز المحـدد نفسه. يُعَد الرمز الذي يُحلَّل بوصفه الرمز الأخير بأن له المرتبة الأخـيرة في تـرتيب تحميـل المحمـل الـديناميكي للمكتبات، حيث تُحمَّل المكتبات بالترتيب المحدَّد في الراية `DT_NEEDED` الخاصة بالملف االثنائي، ويُحدَّد هـذا الترتيب بدوره من خلال ترتيب تمرير المكتبات في سطر الأوامر عنـد بنـاء الكـائن. يبـدأ الرابـط الـديناميكي عنـد تحديد موقع الرموز بآخر مكتبة مُحمَّلة ويعمل بصورة عكسية حتى العثور على الرمز المطلوب.

لكن تحتاج بعض المكتبات المشتركة إلى طريقة لتجاوز هذا السلوك، إذ يجب أن تخبر هذه المكتبات الرابط الديناميكي بأنه يجب أن ينظر أولًا بداخلها عن هذه الرموز بدلًا من العمل بصورة عكسية من آخـر مكتبـة مُحمَّلـة. يمكن للمكتبات ضبط الرايـة `DT_SYMBOLIC` في ترويسـة القسـم الـديناميكي للحصـول على هـذا السـلوك، إذ يمكن ضبط هذه الراية من خلال تمرير الراية `-Bsymbolic` عبر سطر أوامـر الروابـط السـاكنة عنـد بنـاء المكتبـة المشتركة، حيث تتحكم هذه الراية برؤية الرمز Symbol Visibility. لا يمكن تجاوز الرموز الموجودة في المكتبـة، لذا يمكن عَدُّها خاصةً بالمكتبة المُحمَّلة.

لكن يؤدي ذلك إلى فقدان قدر كبير من التفاصيل نظـرًا لتمييـز المكتبـة بهـذا السـلوك أو عـدم تمييزهـا، إذ سيسمح النظام الأفضل بجعل بعض الرموز خاصة وبعض الرموز عامة.

تحديد إصدار الرموز Symbol Versioning

يأتي النظام الأفضل من خلال استخدام تحديد إصدار الرموز، حيث يمكننا تحديد بعض المدخلات الإضـافية للرابط الساكن لمنحه بعض المعلومات الإضافية حول الرموز في المكتبة المشتركة كما يلي:

```
$ cat Makefile
all: test testsym

clean:
	rm -f *.so test testsym

liboverride.so : override.c
	$(CC) -shared -fPIC -o liboverride.so override.c

libtest.so : libtest.c
	$(CC) -shared -fPIC -o libtest.so libtest.c

libtestsym.so : libtest.c
	$(CC) -shared -fPIC -Wl,-Bsymbolic -o libtestsym.so libtest.c

test : test.c libtest.so liboverride.so
	$(CC) -L. -ltest -o test test.c

testsym : test.c libtestsym.so liboverride.so
	$(CC) -L. -ltestsym -o testsym test.c

$ cat libtest.c
#include <stdio.h>

int foo(void) {
  printf("libtest foo called\n");
  return 1;
}

int test_foo(void)
{
  return foo();
```

```
}

$ cat override.c
#include <stdio.h>

int foo(void)
{
  printf("override foo called\n");
  return 0;
}

$ cat test.c
#include <stdio.h>

int main(void)
{
  printf("%d\n", test_foo());
}

$ cat Versions
{global: test_foo;  local: *; };

$ gcc -shared -fPIC -Wl,-version-script=Versions -o libtestver.so
libtest.c

$ gcc -L. -ltestver -o testver test.c

$ LD_LIBRARY_PATH=. LD_PRELOAD=./liboverride.so ./testver
libtest foo called

l     F .text 00000054              foo
000005c8 g     F .text    00000038              test_foo
```

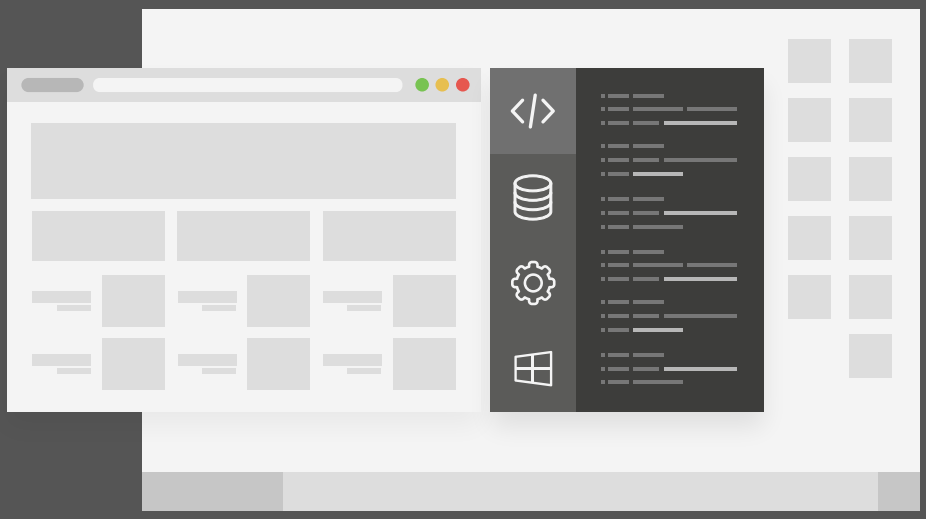
يمكننا ذكر ما إذا كان الرمز عامًا أم محليًا في أبسط الحالات على النحو الوارد في المثال السابق. تكون الدالة `foo` دالة دعم للدالة `test_foo`، ويمكن أن نكون سعداء بتجاوز الوظيفة الكلية للدالة `test_foo`، ولكن إن استخدمنا إصدار المكتبة المشتركة، فيجب الوصول إليها دون تعديل، إذ لا ينبغي لأحدٍ تعديل دالة الدعم.

يسمح ذلك بالحفاظ على فضاء أسمائنا منظمًا بطريقة أفضل، إذ يمكن أن ترغب العديـدُ من المكتبـات في تقديم شـيء يمكن تسـميته باسـم دالـة شـائعة مثـل `read` أو `write`، ولكن إن فعلت ذلـك، فيمكن أن يكـون الإصدار الفعلي الممنوح للبرنامج خاطئًا تمامًا. يمكن للمطور من خلال تحديد الرموز بأنها محلية التأكـدُ من عـدم تعارض أي شيء مع هذا الاسم الداخلي دون أن يؤثر الاسم الذي يختاره على أيّ برنامج آخر.

جاء مفهوم تحديد إصدار الرمـوز Symbol Versioning من تلـك الفكـرة، حيث يمكنـك تحديـد إصـدارات متعددة من الرمز نفسه ضمن المكتبة نفسها. يُلحِق الرابط الساكن بعض معلومات الإصدار بعد اسم الرمز مثـل `@VER` الذي يصف الإصدار المعطى للرمز.

إن قدّم المطور دالة لها الاسم نفسـه تقـديمًا ثنائيًـا أو برمجيًـا مختلفًـا، فيمكنـه زيـادة رقم الإصـدار. تلتقـط التطبيقات الجديدة أحدث إصدار من الرمز عند بنائها بمقابل المكتبة المشتركة. لكن ستطلب التطبيقات المبنية بمقابل الإصدارات السابقة من المكتبة نفسها إصدارات أقدم، فمثلًا سيكون لهـا سلاسـل `@VER` أقـدم في اسـم الرمز الذي تطلبه، وبالتالي ستحصل على التقديم الأصلي.

دورة علوم الحاسوب



دورة تدريبية متكاملة تضعك على بوابة الاحتراف
في تعلم أساسيات البرمجة وعلوم الحاسوب

التحق بالدورة الآن

10. قائمة المصطلحات

- **واجهة برمجة التطبيقات Application Programming Interface أو API اختصارًا**: مجموعة المتغيرات والدوال المُستخدَمة للتواصل بين أجزاء البرامج المختلفة.
- **واجهة التطبيق الثنائية Application Binary Interface أو ABI اختصارًا**: وصف تقني لكيفية تعامل نظام التشغيل مع العتاد.
- **لغة التوصيف الموسَّعة Extensible Markup Language**: تعرّف على هذه اللغة أكثر في أكاديمية حسوب.
- **لغة التوصيف المعياري العام Standardised Generalised Markup Language**: تُعَد الأب الأكبر لجميع المستندات.
- **الإقصاء المتبادل Mutually Exclusive**: يمكن أن يكون عنصر واحد فقط صالحًا في كل مرة عند تطبيق الإقصاء المتبادل على عدد من الأشياء، إذ يؤدي كون أحد الأشياء صالحًا إلى جعل الأشياء الأخرى غير صالحة.
- **MMU**: مكون وحدة إدارة الذاكرة Memory Management Unit في معمارية العتاد.
- **المصدر المفتوح Open Source**: تُوزَّع البرمجيات بصيغة مصدر بموجب تراخيص تضمن لأي شخص حقوق الاستخدام والتعديل وإعادة توزيع الشيفرة البرمجية بحرية.
- **الصدفَة Shell**: الواجهة المستخدمة للتفاعل مع نظام التشغيل.

أحدث إصدارات أكاديمية حسوب