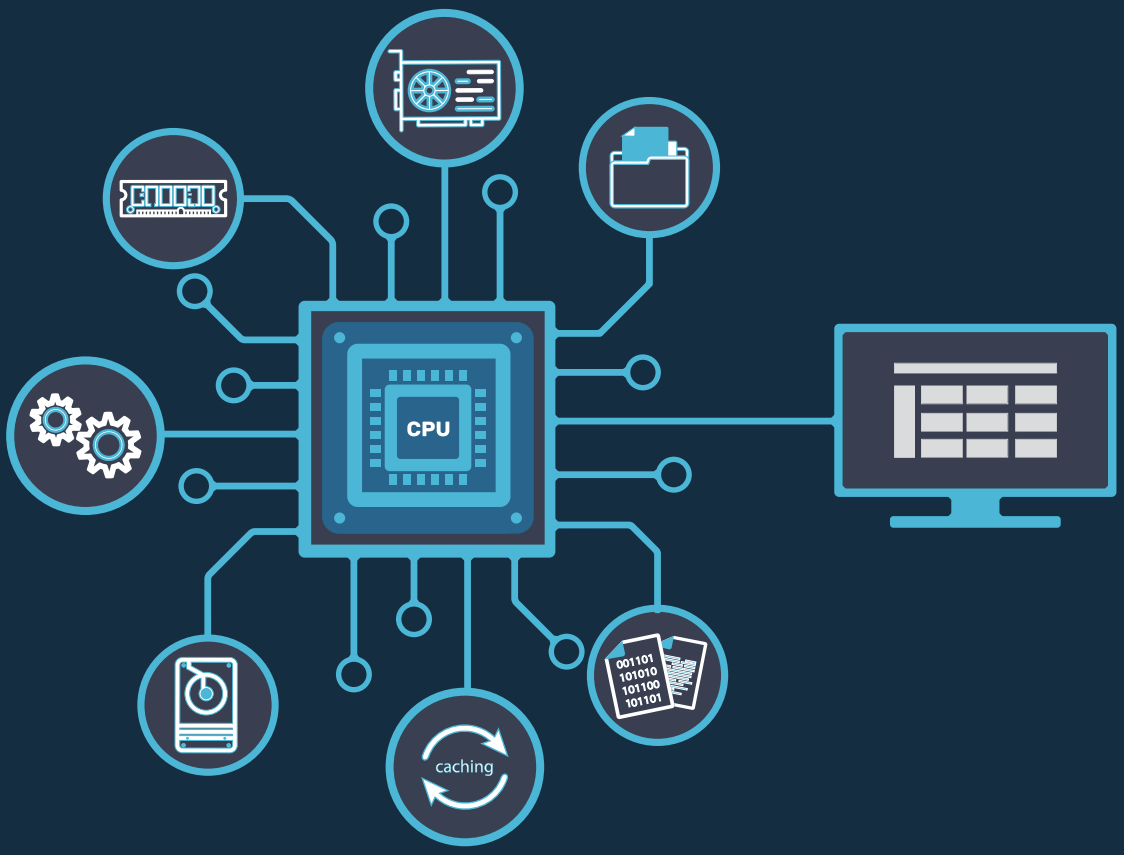


أنظمة التشغيل للمبرمجين

آلن داوني

أكاديمية حسوب

أنظمة التشغيل للمبرمجين

مقدمة مختصرة إلى أنظمة التشغيل بما يهم المبرمج

تأليف

آلن داوني

ترجمة

علا عباس

تحرير وإشراف

جميل بيلوني

النسخة الأولى v1.0

عن الناشر

أُنتج هذا الكتاب برعاية شركة حسوب وأكاديمية حسوب.

تهدف أكاديمية حسوب إلى توفير دروس وكتب عالية الجودة في مختلف المجالات وتقدم دورات شاملة لتعلُّم البرمجة بأحدث تقنياتها معتمدةً على التطبيق العملي الذي يؤهل الطالب لدخول سوق العمل بثقة.

حسوب مجموعة تقنية في مهمة لطوير العالم العربي. تبني حسوب منتجات تركِّز على تحسين مستقبل العمل، والتعليم، والتواصل. تدير حسوب أكبر منصتي عمل حر في العالم العربي، مستقل وخمسات ويعمل في فيها فريق شاب وشغوف من مختلف الدول العربية.

جدول المحتويات

تمهيد

تُعد أنظمة التشغيل موضوعًا متقدمًا في العديد من برامج علوم الحاسوب، حيث يتعرف المتعلم على كيفية البرمجة بلغة C بحلول الوقت الذي يتعرف فيه على أنظمة التشغيل، وربما يأخذ المتعلم فصلًا دراسيًا في معمارية الحاسوب Computer Architecture قبل ذلك، فيصبح الهدف من تقديم هذا الموضوع عادةً هو عرض تصميم وتنفيذ أنظمة التشغيل للطلاب مع افتراض ضمني أن بعضهم سيجري بحثًا في هذا المجال، أو يكتب جزءًا من نظام تشغيل.

هذا الكتاب مخصصٌ لجمهورٍ مختلف وله أهدافٌ مختلفة، حيث طُوّر لفصلٍ دراسي في كلية أولين Olin College يدعى أنظمة البرمجيات Software Systems، إذ تعلّم معظم الطلاب الذين يحضرون هذا الفصل البرمجة بلغة بايثون، لذا فإن أحد الأهداف هو مساعدتهم على تعلّم لغة C، فاستُخدِم كتاب C First Head للكاتبين دايفد غريفث David Griffiths وداون غريفث Dawn Griffiths من O'Reilly Media لهذا الغرض، ويهدف هذا الكتاب إلى استكماله والبناء عليه.

سيكتب عددٌ قليل من أولئك الطلاب نظام تشغيل، ولكن العديد منهم سيكتبون تطبيقاتٍ منخفضة المستوى بلغة C أو يعملون على أنظمةٍ مدمجة embedded systems. يتضمن الفصل الدراسي الذي يدعى أنظمة البرمجيات -من جامعة أولين- مواد من أنظمة التشغيل والشبكات وقواعد البيانات والأنظمة المدمجة، ولكنه يركز على الموضوعات التي يحتاج المبرمجون إلى معرفتها وهو ما يدور حوله الكتاب.

حول الكتاب

هذا الكتاب مترجم عن الكتاب Think OS لكاتبه آلن داوني Allen B. Downey والذي يعد مسودة أولية، حيث لم يحوي الإصدار الحالي منه الأشكال بعد، لذلك ستتحسّن عدة أفكار بصورة كبيرة عندما تصبح الأشكال جاهزة وتضاف إليه.

لا يفترض هذا الكتاب أنك قد درست معمارية الحاسوب، فيجب أن يمنحك فهمًا أفضل أثناء قراءته عن الحاسوب ومعماريته وكيف يعمل المعالج والذاكرة فيه وكيف تُدار العمليات وتُخزَّن الملفات وما يحدث عند تشغيل البرامج، وما يمكنك القيام به لجعل البرامج تعمل بصورة أفضل وأسرع بوصفك مجبرمجًا.

يشرح الفصل الأول بعض الاختلافات بين اللغات المُصرَّفة compiled واللغات المُفسَّرة interpreted، مع بعض الأفكار حول كيفية عمل المصرِّفات compilers، يُوصى هنا بقراءة الفصل الأول من كتاب Head First C. ويشرح الفصل الثاني كيف يستخدم نظام التشغيل العمليات لحماية البرامج قيد التشغيل من التداخل مع بعضها البعض. ويشرح الفصل الثالث الذاكرة الوهمية virtual memory وترجمة العناوين، وهنا يُوصى بقراءة الفصل الثاني من كتاب Head First C. ويتحدث الفصل الرابع عن أنظمة الملفات ومجرى البيانات، ويُوصى بقراءة الفصل الثالث من كتاب Head First C. ويصف الفصل الخامس كيفية تشفير الأرقام والأحرف والقيم الأخرى، ويشرح أيضًا العامِلات الثنائية bitwise operators.

أما الفصل السادس، فيشرح كيفية استخدام إدارة الذاكرة الديناميكية وكيفية عملها، ويُوصى بقراءة الفصل السادس من كتاب Head First C. ويدور الفصل السابع حول التخبئة caching وهرمية الذاكرة. ويشرح الفصل الثامن تعدد المهام multitasking والجدولة scheduling. ويدور الفصل التاسع حول خيوط POSIX وكائنات المزامنة mutexes، وفيه يُوصى بقراءة الفصل الثاني عشر من كتاب Head First C والفصلين الأول والثاني من كتاب Little Book of Semaphores. ويشرح الفصل العاشر المتغيرات الشرطية POSIX ومشكلة المنتج / المستهلك، وفيه يُوصى بقراءة الفصلين الثالث والرابع من كتاب Little Book of Semaphores. ويدور الفصل الحادي عشر حول استخدام متغيرات تقييد الوصول POSIX وتطبيقها في لغة C.

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 github.com/AllenDowney/ThinkOS حيث 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.

1. مفهوم التصريف Compilation

1.1 اللغات المصرفة Compiled واللغات المفسرة Interpreted

تندرج لغات البرمجة تحت صنفين اثنين: إما مُصرَّفة compiled أو مُفسَّرة interpreted، فيعني المصطلح لغة مُصرَّف compiled ترجمة البرامج إلى لغة الآلة machine language لينفذها العتاد hardware، أما مصطلح لغة مُفسَّرة interpreted فيعني وجود برنامج يدعى «المفسِّر» interpreter يقرأ البرامج وينفذها مباشرةً وآنيًا.

تُعَد لغة البرمجة C على سبيل المثال لغة مُصرَّفة compiled عادةً، بينما تُعَد لغة Python لغة مُفسَّرة interpreted، لكنّ التمييز بين المصطلحين غير واضح دائمًا حيث:

أولًا يمكن للغات البرمجة المُفسَّرة أن تكون مُصرَّفة والعكس صحيح، فلغة C مثلًا هي لغة مصرَّفة ولكن يوجد مفسِرات لها تجعلها لغة مفسَّرةً أيضًا والأمر مماثل للغة Python المفسَّرة التي يمكن أن تكون مصرَّفة أيضًا.

ثانيًا توجد لغات برمجة، جافا Java مثلًا، تستخدم نهجًا هجينًا hybrid approach يجمع بين التصريف والتفسير، حيث يبدأ هذا النهج بترجمة البرنامج إلى لغة وسيطة intermediate language عبر مصرِّف ثم تنفيذ البرنامج عبر مُفسِّر. تَستخدم لغة Java لغةً وسيطةً intermediate language تُدعى جافا بايتكود Java bytecode شبيهة بلغة الآلة، لكنها تُنفَّذ باستخدام مُفسِّر برمجيات يدعى بآلة جافا الافتراضية Java virtual machine وتختصر إلى JVM.

وسمُ لغة البرمجة بكونها لغة مفسَّرة أو مصرَّفة لا يكسبها خاصية جوهرية، على كل حال توجد اختلافات عامة بين اللغتين المُصرَّفة والمُفسَّرة.

1.2 الأنواع الساكنة Static Types

تدعم العديد من اللغات المُفسَّرة الأنواع الديناميكية Dynamic Types، وتقتصر اللغات المُصرَّفة على الأنواع الساكنة Static Types. فيمكن في اللغات ساكنة النوع معرفة أنواع المتغيرات بمجرّد قراءة شيفرة البرنامج أي تكون أنواع المتغيرات محدَّدة قبل تنفيذ البرنامج، بينما تكون أنواع المتغيرات في اللغات التي توصف بأنها ديناميكية النوع غير معروفة قبل التنفيذ وتحدد وقت تنفيذ البرنامج. ويشير مصطلح ساكن Static إلى الأشياء التي تحدث في وقت التصريف Compile time أي عند تصريف شيفرة البرنامج إلى شيفرة التنفيذ، بينما يشير مصطلح Dynamic إلى الأشياء التي تحدث في وقت التشغيل run time، أي عندما يُشغَّل البرنامج.

يمكن كتابة الدالة التالية في لغة Python على سبيل المثال:

```
def add(x, y):
    return x + y
```

لا يمكن معرفة نوع المتغيرين y وx بمجرد قراءة الشيفرة السابقة حيث لا يحدَّد نوعهما حتى وقت تنفيذ البرنامج، لذلك يمكن استدعاء هذه الدالة عدة مرات بتمرير قيمة بنوع مختلف إليها في كل مرة، وستعمل عملًا صحيحًا ما دام نوع القيمة المُمرَّرة إليها مناسبًا لتطبيق عملية الجمع عليها، وإلا سترمي الدالة اعتراضًا exception أو خطأً وقت التشغيل.

يمكن كتابة نفس الدالة السابقة في لغة البرمجة C كما يلي:

```
int add(int x, int y) {
    return x + y
}
```

يتضمَّن السطر الأول من الدالة تصريحًا واضحًا وصريحًا بنوعي القيمتين التي يجب تمريرهما إلى الدالة ونوع القيمة التي تعيدها الدالة أيضًا، حيث يُصرَّح عن y وx كأعداد صحيحة integers، وهذا يعني أنه يمكن التحقق في وقت التصريف compiled time فيما إذا كان مسموحًا استخدام عامل الجمع مع النوع integer أم لا إنه مسموح حقًا، ويُصرَّح عن القيمة المُعادة كعدد صحيح integer أيضًا. وعندما تُستدعى الدالة السابقة في مكان آخر من البرنامج يستطيع المصرِّف compiler باستخدام التصريحات أن يتحقق من صحة نوع الوسطاء arguments الممررة للدالة، ومن صحة نوع القيمة التي تعيدها الدالة أيضًا.

يحدث التحقق في اللغات المصرَّفة قبل بدء تنفيذ البرنامج لذلك يمكن إيجاد الأخطاء باكرًا، ويمكن إيجاد الأخطاء أيضًا في أجزاء البرنامج التي لم تُشغَّل على الإطلاق وهو الشيء الأهم. علاوةً على ذلك لا يتوجب على هذا التحقق أن يحدث في وقت التشغيل runtime، وهذا هو أحد الأسباب التي تجعل تنفيذ اللغات المُصرَّفة أسرع من اللغات المُفسَّرة عمومًا. يحافظ التصريح عن الأنواع في وقت التصريف compile time على مساحة الذاكرة في اللغات ساكنة النوع أيضًا، بينما تُخزَّن أسماء المتغيرات في الذاكرة عند تنفيذ البرنامج في اللغات

ديناميكية النوع التي لا تحوي تصريحات واضحة لأنواعها وتكون أسماء هذه المتغيرات قابلة للوصول من قبل البرنامج. توجد دالة مبنيّة مسبقًا في لغة Python هي `locals`، تعيد هذه الدالة قاموسًا dictionary يتضمن أسماء المتغيرات وقيمها.

ستجد تاليًا مثالًا عن مفسِّر Python:

```
>>> x = 5
>>> print locals()
{'x': 5, '__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__', '__doc__': None, '__package__': None}
```

يبيّن المثال السابق أنه يُخزَّن اسم المتغير في الذاكرة عند تنفيذ البرنامج مع بعض القيم الأخرى التي تُعَد جزءًا من بيئة وقت التشغيل الافتراضية. بينما تتواجد أسماء المتغيرات في اللغات المُصرَّفة في الذاكرة في وقت التصريف compile time ولا تتواجد في وقت التشغيل runtime. حيث يختار المصرّف موقعًا في الذاكرة لكل متغير ويسجل هذه المواقع كجزء من البرنامج المُصرَّف سنتطرّق إلى مزيد من التفاصيل عن ذلك لاحقًا. يدعى موقع المتغير في الذاكرة عنوانًا address حيث تُخزَّن قيمة كل متغير في عنوانه، ولا تُخزَّن أسماء المتغيرات في الذاكرة على الإطلاق في وقت التشغيل ولكن هذا شيء اختياري للمصرف فيمكن أن يضيف المصرِّف compiler أسماء المتغيرات إلى الذاكرة في وقت التشغيل بهدف تنقيح الأخطاء debugging، أي لمعرفة أماكن تواجد الأخطاء في البرنامج.

1.3 عملية التصريف compilation process

يجب أن يفهم المبرمج فهمًا تامًا ما يحدث خلال عملية التصريف compilation، فإذا فُهِمت هذه العملية جيدًا سيساعد ذلك في تفسير رسائل الخطأ وتنقيح الأخطاء في الشيفرة وأيضًا في تجنُّب الزلات الشائعة. للتصريف خطوات هي:

1. المعالجة المسبقة Preprocessing: تتضمن لغة البرمجة C موجِّهات معالجة مسبقة preprocessing directives والتي تدخل حيز التنفيذ قبل تصريف البرنامج، فمثلًا يسبِّب الموجِّه `#include` إدراج شيفرة مصدرية source code خارجية موضع استعماله.
2. التحليل Parsing: يقرأ المُصرِّف compiler أثناء هذه الخطوة الشيفرة المصدرية source code ويبني تمثيلًا داخليًّا لها internal representation للبرنامج يُدعى بشجرة الصيغة المجردة abstract syntax tree. تُسمى عادةً الأخطاء المكتشفة خلال هذه الخطوة بأخطاء صياغية syntax errors.

3. التحقق الساكن Static checking: يتحقق المصرِّف من صحة نوع المتغيرات والقيم وفيما إذا أُستدعيت الدوال بعدد ونوع وسطاء صحيحين وغير ذلك من التحققات. يُدعى اكتشاف الأخطاء في هذه الخطوة أحيانًا بالأخطاء الدلالية الساكنة static semantic errors.

4. توليد الشيفرة Code generation: يقرأ المصرِّف التمثيل الداخلي internal representation للبرنامج ويولّد شيفرة الآلة machine code أو الشيفرة التنفيذية byte code للبرنامج.

5. الربط Linking: إذا استخدم البرنامج قيمًا ودوالًا مُعرَّفة في مكتبة، فيجب أن يجد المُصرِّف المكتبة المناسبة وأن يُضمِّن include الشيفرة المطلوبة المتعلقة بتلك المكتبة.

6. التحسين Optimization: يحسّن المصرف دومًا خلال عملية التصريف من الشيفرة ليصبح تنفيذها أسرع أو لجعلها تستهلك مساحةً أقل من الذاكرة. معظم هذه التحسينات هي تغييرات بسيطة توفر من الوقت والمساحة، ولكن تطبِّق بعض المُصرِّفات compilers تحسيناتٍ أعقد.

ينفذ المصرف كل خطوات التصريف ويولّد ملفًا تنفيذيًا executable file عند تشغيل الأداة `gcc`. المثال التالي هو شيفرة بلغة C:

```c
#include <stdio.h>
int main()
{
    printf("Hello World\n");
}
```

إذا حُفِظت الشيفرة السابقة في ملف اسمه `hello.c` فيمكن تصريفها ثم تشغيلها كما يلي:

```
$ gcc hello.c
$ ./a.out
```

تخزّن الأداة `gcc` الشيفرة القابلة للتنفيذ executable code في ملف يدعى افتراضيًا `a.out`، والذي يعني في الأصل خرج مُجمِّع assembler output. ينفذ السطر الثاني الملف التنفيذي، حيث تخبر البادئة `./` الصدفة shell لتبحث عن الملف التنفيذي في المجلّد directory الحالي.

من الأفضل استخدام الراية `-o` لتوفير اسم أفضل للملف التنفيذي، حيث يُعطى الملف التنفيذي الناتج بعد عملية التصريف اسمًا افتراضيًا `a.out` بدون استخدام الراية `-o`، ولكن يُعطى اسمًا محددًا باستخدام الراية `-o` كما يلي:

```
$ gcc hello.c -o hello
$ ./hello
```

1.4 التعليمات المُصرَّفة Object code

تخبر الراية `-c` الأداة `gcc` بأن تصرِّف البرنامج وتولّد شيفرة الآلة machine code فقط، بدون أن تربط link البرنامج أو تولّد الملف التنفيذي.

```
$ gcc hello.c -c
```

النتيجة هي توليد ملف يُدعى `hello.o`، حيث يرمز حرف o إلى object code وهو البرنامج المُصرّف. والتعليمات المُصرَّفة object code غير قابلة للتنفيذ لكن يمكن ربطها بملف تنفيذي. يقرأ الأمر `nm` في UNIX ملف التعليمات المُصرَّفة object file ويولّد معلومات عن الأسماء التي يُعرِّفها ويستخدمها الملف، فمثلًا:

```
$ nm hello.o
0000000000000000 T main
                 U puts
```

يشير الخرج السابق إلى أن `hello.o` يحدد اسم التابع الرئيسي `main` ويستخدم دالة تدعى `puts`، والتي تشير إلى put string. وتطّبق `gcc` تحسينًا optimization عن طريق استبدال `printf` وهي دالة كبيرة ومعقدة بالدالة `puts` البسيطة نسبيًا. يمكن التحكم بمقدار التحسين الذي تقوم به `gcc` مع الراية `-O`، حيث تقوم `gcc` بإجراء تحسينات قليلة جدًا افتراضيًا مما يجعل تنقيح الأخطاء debugging أسهل. بينما يفعَّل الخيار `-O1` التحسينات الأكثر شيوعًا وأمانًا، وإذا استخدمنا مستويات أعلى أي O2 وما بعده فستفعِّل تحسينات إضافية، ولكنها تستغرق وقت تصريف أكبر.

لا ينبغي أن يغير التحسين من سلوك البرنامج من الناحية النظرية بخلاف تسريعه، ولكن إذا كان البرنامج يحتوي خللًا دقيقًا subtle bug فيمكن أن تحمي عملية التحسين أثره أو تزيل عملية التحسين هذا الخلل. إيقاف التحسين فكرة جيدة أثناء مرحلة التطوير عادةً، وبمجرد أن يعمل البرنامج ويجتاز الاختبارات المناسبة يمكن تفعيل التحسين والتأكد من أن الاختبارات ما زالت ناجحة.

1.5 الشيفرة التجميعية Assembly code

تتشابه الرايتان `-S` و `-c`، حيث أن الراية `-S` تخبر الأداة `gcc` بأن تصرف البرنامج وتولد الشيفرة التجميعية assembly code، والتي هي بالأساس نموذج قابل للقراءة تستطيع شيفرة الآلة machine code قراءته.

```
$ gcc hello.c -S
```

ينتج ملف يدعى `hello.s` والذي يبدو كالتالي

```
        .file "hello.c"
        .section .rodata
.LC0:
        .string "Hello World"
        .text
        .globl main
        .type main, @function
main:
.LFB0:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        movl $.LC0, %edi
        call puts
        movl $0, %eax
        popq %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section .note.GNU-stack,"",@progbits
```

تُضبَط gcc عادةً لتولد الشيفرة للآلة التي تعمل عليها، ففي حالتي، يقول المؤلف، وُلِّدت شيفرة لغة آلة لمعمارية x86 للمعالجات والتي يمكن تنفيذها على شريحة واسعة من معالجات Intel ومعالجات AMD وغيرهما وفي حال استهداف معمارية مختلفة، فستولد شيفرة أخرى مختلفة عن تلك التي تراها الآن.

1.6 المعالجة المسبقة Preprocessing

يمكن استخدام الراية -E لتشغيل المعالج المُسبق preprocessor فقط بدون الخطوات الأخرى من عملية التصريف:

```
$ gcc hello.c -E
```

سينتج خرج من المعالج المسبق فقط. يحتوي المثال السابق تلقائيًا على الشيفرة المُضمَّنة included code المبنية مسبقًا والمتعلقة بالمكتبة stdio.h المذكورة في بداية البرنامج، وبالتالي يتضمن كل الملفات المُضمَّنة المتعلقة بتلك المكتبة، وكل الملفات الفرعية التابعة للملفات السابقة والملفات الموجودة في الملفات الفرعية أيضًا وهكذا. فعلى حاسوبي، يقول المؤلف، وصل العدد الإجمالي للشيفرة الإجمالية المضمنة إلى 800 سطر، ونظرًا أن كل برنامج C يتضمّن ملف الترويسات stdio.h تقريبًا، لذلك تُضمَّن تلك الأسطر في كل برنامج مكتوب بلغة C.

وتتضمّن العديد من برامج C المكتبة stdlib.h أيضًا، وبالتالي ينتج أكثر من 1800 سطر إضافي من الشيفرة يجب تصريفها جميعًا.

1.7 فهم الأخطاء

أصبح فهم رسائل الخطأ أسهل بعد معرفة خطوات عملية التصريف، فمثلًا عند وجود خطأ في الموجّه #include ستصل رسالة من المعالج المسبق هي:

```
hello.c:1:20: fatal error: stdioo.h: No such file or directory
compilation terminated.
```

أما عند وجود خطأ صياغي syntax error متعلق بلغة البرمجة، ستصل رسالة من المُصرِّف compiler هي:

```
hello.c: In function 'main':
hello.c:6:1: error: expected ';' before '}' token
```

عند استخدام دالة غير معرَّفة في المكتبات القياسية ستصل رسالة من الرابط linker هي:

```
/tmp/cc7iAUbN.o: In function `main':
hello.c:(.text+0xf): undefined reference to `printff'
collect2: error: ld returned 1 exit status
```

ld هو اسم رابط UNIX ويشير إلى تحميل loading، حيث أن التحميل هو خطوة أخرى من عملية التصريف ترتبط ارتباطًا وثيقًا بخطوة الربط linking.

تجري لغة C تحققًا سريعًا جدًا ضمن وقت التشغيل بمجرد بدء البرنامج، لذلك من المحتمل أن ترى بعضًا من أخطاء وقت التشغيل runtime errors فقط وليس جميعها، مثل خطأ القسمة على صفر divide by zero، أو تطبيق عملية عدد عشري غير مسموحة وبالتالي الحصول على اعتراض عدد عشري Floating point exception، أو الحصول على خطأ تجزئة Segmentation fault عند محاولة قراءة أو كتابة موقع غير صحيح في الذاكرة.

2. العمليات Processes

2.1 التجريد Abstraction والوهمية Virtualization

ينبغي معرفة مصطلحين مهمين قبل الخوض في الحديث عن العمليات processes هما التجريد والوهمية.

أما **التجريد abstraction** فهو تمثيلٌ مُبسّط لشيء معقد. عند قيادة سيارة ما على سبيل المثال يُفَهم أن توجيه عجلة القيادة يسارًا يوجه السيارة يسارًا والعكس صحيح. ترتبط عجلة القيادة بسلسلة من الأنظمة الميكانيكية والهيدروليكية، حيث توجّه هذه الأنظمة العجلات لتتحرك على الطريق. يمكن أن تكون هذه الأنظمة معقدة، ولكنّ السائق غير ملزَمٍ بالاكتراث بأي من تلك التفاصيل التي تجري داخل أنظمة السيارة، فالسائق يقود السيارة بالفعل وذلك بسبب امتلاكه نموذجًا ذهنيًا mental model بسيطًا عن عملية قيادة السيارة، وهذا النموذج البسيط هو التجريد abstraction بعينه.

استخدام متصفح الويب web browser هو مثال آخر عن التجريد، فعند النقر على ارتباط link يعرض المتصفح الصفحة المرتبطة بهذا الارتباط. لا شكّ أن البرمجيات software وشبكات الاتصال network communication التي تجعل ذلك ممكنًا معقدة، ولكن لا يتوجب على المستخدم معرفة تفاصيل تلك الأمور المعقدة.

جزء كبير من هندسة البرمجيات هو تصميم التجريدات التي تسمح للمستخدمين والمبرمجين استخدام أنظمة معقدة بطريقة سهلة دون الحاجة إلى معرفة تفاصيل تنفيذ هذه الأنظمة.

وأما **الوهمية Virtualization** فهي نوع من التجريد الذي يخلق وهمًا illusion بوجود شيء فعليًا في حين أنه موجود وهميًا فقط. فمثلًا تشارك العديد من المكتبات العامة في تعاون بينها يسمح باستعارة الكتب من

بعضها البعض. عندما تطلب كتابًا يكون الكتاب على رف من رفوف مكتبتك أحيانًا، ولكن يجب نقله من مكان آخر عند عدم توافره لديك، ثم سيصلك إشعار عندما يُتاح الكتاب للاستلام في كلتا الحالتين. ليس هناك حاجة أن تعرف مصدر الكتاب ولا أن تعرف الكتب الموجودة في مكتبتك. إذًا يخلق النظام وهمًا بأن مكتبتك تحتوي على كتب العالم جميعها. قد تكون مجموعة الكتب الموجودة في مكتبتك فعليًا صغيرة، لكن مجموعة الكتب المتاحة لك وهميًا هي كل كتاب موجود في تلك المكتبات المتشاركة.

الإنترنت Internet هو مثال آخر عن الوهمية وهو مجموعة من الشبكات والبروتوكولات التي تعيد توجيه forward الحزم packets من شبكةٍ لأخرى. يتصرف نظام كل حاسوب كأنه متصل بكل حاسوب آخر على الإنترنت من وجهة نظر المستخدم أو المبرمج، حيث يكون الاتصال الفعلي أو الفيزيائي بين كل حاسوب وآخر قليلًا، أما الاتصال الوهمي كبير جدًا.

يُستخدم المصطلح وهمي virtual ضمن عبارة آلة وهمية virtual machine أكثر الأحيان. والآلة الوهمية تعني البرمجية التي تمنح المستخدم القدرة على إنشاء نظام تشغيل على حاسوب يشغّل نظام تشغيل مختلف، وبذلك تخلق هذه الآلة وهمًا بأن هذا النظام المنشَأ يعمل على حاسوب مستقل بذاته، ولكن في الحقيقة يمكن تشغيل عدة أنظمة تشغيل وهمية على حاسوب واحد بنفس الوقت وكأنّ كل نظام تشغيل يعمل على حاسوب مختلف. وبالتالي يُدعى ما يحدث فعليًا physical وما يحدث وهميًا logical أو abstract.

2.2 العزل Isolation

العزل هو أحد أهم مبادئ الهندسة، فعزل المكونات عن بعضها البعض هو أمر جيد عند تصميم نظام متعدد المكونات من أجل ألّا يؤثر مكون على المكونات الأخرى تأثيرًا غير مرغوب به. عزل كل برنامج قيد التشغيل عن البرامج الأخرى من أهم أهداف نظام التشغيل، فبذلك لا يضطر المبرمجون إلى الاهتمام باحتمالية حدوث تفاعلات بين البرامج المشغّلة، وبالتالي يحتاج نظام التشغيل إلى كائن برمجي يحقق هذا العزل ألا وهو العملية process. ولكن ما هي العملية؟

العملية process هي كائنٌ برمجي يمثل برنامجًا مشغلًا، وبالتالي يُمثَّل كل برنامج بعملية معينة. حيث يُقصَد بعبارة «كائن برمجي» بكائنٍ له روح البرمجة كائنية التوجه object-oriented programming، حيث يتضمن كل كائن بيانات data وتوابع methods تعمل على هذه البيانات. فالعملية هي كائن ولكن يتضمن البيانات التالية:

- نص البرنامج: وهو سلسلة من تعليمات لغة الآلة عادةً.
- بيانات مرتبطة بالبرنامج: والتي تنقسم إلى نوعين: بيانات ساكنة static data تُخصَّص مواقعها في الذاكرة في وقت التصريف و بيانات ديناميكية dynamic data تُخصَّص مواقعها في الذاكرة في وقت التشغيل.

- حالة عمليات الإدخال/الإخراج input/output المعلَّقة: مثل انتظار العملية قراءة بيانـات من القـرص، أو انتظار وصول حزمة عن طريق الشبكة، فحالات عمليات الانتظار هذه هي جزء من العملية نفسها.
- حالـة عتـاد البرنـامج hardware state: الـتي تتضـمن البيانـات المخزنـة في المسـجلات registers، ومعلومات الحالة status information، وعداد البرنامج الذي يحدد أية تعليمة ستنفّذ حاليًا. تشغّل كل عملية برنامجًا واحدًا في أغلب الأحيـان، ولكن يمكن لعمليـة مـا أن تحمّـل وتشـغّل برنامجًـا آخـر أيضًـا. ويمكن أن تشغّل عدة عمليات نفس البرنامج، ففي هذه الحالة تتشـارك العمليـات بنص البرنـامج ولكن ببيانات وحالات عتاد مختلفة.

توفر معظم أنظمة التشغيل مجموعة أساسية من القدرات لعزل العمليات عن بعضها البعض هي:

- تعدد المهام Multitasking: تتمتع معظم أنظمة التشغيل بالقدرة على مقاطعة تنفيذ أيـة عمليـة في أي وقت تقريبًا مع حفظ حالة عتاد العملية المُقاطعة، ثم استئناف تشغيل العملية لاحقًـا. على كـل حـال لا يضطر المبرمجون إلى التفكير كثيرًا في هذه المقاطعات interruptions، حيث يتصرف البرنامج كما لـو أنه يعمل باستمرار على معالج مخصص له فقط دون غيره، ولكن ينبغي التفكير بالوقت الفاصل بين كل تعليمة وأخرى من البرنامج فهو وقت لا يمكن التنبؤ به.
- الـذاكرة الوهميـة Virtual memory: تخلـق معظم أنظمـة التشـغيل وهمًـا بـأن كـل عمليـة لها قطعتها chunk من الذاكرة وهذه القطعة معزولة عن قطع العمليات الأخرى. يمكن القول مرة أخرى بـأن المبرمجين غير مضطرين لمعرفة كيفيـة عمـل الـذاكرة الوهميـة، فيمكنهم المتابعـة في كتابـة بـرامجهم معتبرين كل برنامجٍ له جزءٌ مخصص من الذاكرة.
- تجريد الجهاز Device abstraction: تتشارك العمليات العاملة على نفس الحاسوب بمحرك الأقراص أو قرص التخزين disk drive وبطاقة الشبكة network interface وبطاقـة الرسـوميات graphics card ومكونات عتادية أخرى أيضًا. إذا تعاملت العمليات مع تلك المكونات العتادية للحاسوب مباشـرةً ودون تنسيق من نظام التشغيل، سيؤدي ذلك إلى فوضـى عارمـة. فيمكن لعمليـة مـا أن تقـرأ بيانـات شـبكة عمليةٍ أخرى على سبيل المثال، أو يمكن أن تحاول عمليات متعـددة تخـزين بيانـات في الموقـع نفسـه على محرك القرص الصلب hard drive. الأمر متروك لنظام التشغيل في النهايـة ليحافـظ على النظـام من خلال توفير تجريدات مناسبة.

لا يحتاج المبرمج لمعرفة الكثير عن كيفية عمل قدرات نظام التشغيل لعـزل العمليـات عن بعضـها البعض، ولكن إذا دفعك الفضول لمعرفة المزيد عن ذلك فهو أمر جيـد، فالإبحـار في معرفـة مزيـد من التفاصـيل يصـنع منك مبرمجًا أفضل.

2.3 عمليات أنظمة يونكس UNIX

تخيل السيناريو التالي:

العمليـة الـتي تعيهـا عنـد اسـتخدامك الحاسـوب للكتابـة هي محـرر النصـوص، وإذا حـرّكت الفـأرة على الطرفية terminal يتنبه مدير النوافذ window manager وينبّه terminal التي بدورها تنبّه الصدفة shell، فإذا كتبت الأمر make فإن shell تنشئ عملية جديدة لتنفيذ الأمر Make الـتي بـدورها تنشـئ عمليـة أخـرى لتنفيـذ LaTeX وهكذا يستمر إنشاء العمليات حتى عرض نتـائج تنفيـذ الأمـر make. يمكن أن تبـدل إلى سـطح المكتب desktop إذا أردت البحث عن شيء ما، فيؤدي ذلك إلى تنبيه مدير النوافـذ أيضًـا. ويتسـبب نقـرك على أيقونـة متصفح الويب في إنشاء عملية تشغّل المتصفح. تنشئ بعض المتصفحات، Chrome مثلًا، عمليات لكل نافذة وتبويب جديدين. ولكن توجـد عمليـات أخـرى تعمـل في الخلفيـة background في ذات الـوقت، تتعلـق تلـك العمليات في معظمها بنظام التشغيل.

يمكن استخدام الأمر ps في UNIX لمعرفة معلومات عن العمليات التي تعمل حاليًا، فيظهـر الخـرج التـالي عند تنفيذ الأمر:

```
PID    TTY      TIME      CMD
2687   pts/1    00:00:00  bash
2801   pts/1    00:01:24  emacs
24762  pts/1    00:00:00  ps
```

يمثل العمود الأول معرِّف العملية الفريد ID، بينمـا يمثـل العمـود الثـاني الطرفيـة terminal الـتي أنشـأت العمليـة حيث يشـير TTY إلى عبـارة teletypewriter وهي جهـاز قـديم اُسـتخدِم لإرسـال و اسـتقبال الرسـائل المكتوبة من خلال قنوات اتصال مختلفـة، ويمثـل العمـود الثـالث الـزمن الإجمـالي المسـتغرَق خلال اسـتخدام العملية للمعالج ويكون بالشكل التـالي: سـاعات، دقـائق، ثـواني، ويمثّـل العمـود الرابـع والأخـير اسـم البرنـامج المشغَّل، حيث bash هو اسم الصدفة Shell التي قاطعت الأوامـر المكتوبـة في الطرفيـة terminal، وemacs هو محرر النصوص المستخدَم، ويعد ps هو البرنامج الذي ولّد الخرج السابق. فخـرج الأمـر ps هـو قائمـة تحـوي العمليـات المتعلقـة بالطرفيـة terminal الحاليـة فقـط، ولكن باسـتخدام الرايـة e- مـع ps أو باسـتخدام الراية aux التي هي خيارٌ آخر وشائع فستظهر كل العمليات بما في ذلك عمليات المسـتخدمين الآخـرين والـذي برأيي، يقول الكاتب، هو ثغرة أمنية . يوجد على نظامي التشغيلي مثلًا، يقول الكاتب، 233 عملية حاليًا فيما يلي بعض منها:

```
PID TTY TIME      CMD
1   ?   00:00:17  init
2   ?   00:00:00  kthreadd
3   ?   00:00:02  ksoftirqd/0
```

```
4   ?   00:00:00  kworker/0:0
8   ?   00:00:00  migration/0
9   ?   00:00:00  rcu_bh
10  ?   00:00:16  rcu_sched
47  ?   00:00:00  cpuset
48  ?   00:00:00  khelper
49  ?   00:00:00  kdevtmpfs
50  ?   00:00:00  netns
51  ?   00:00:00  bdi-default
52  ?   00:00:00  kintegrityd
53  ?   00:00:00  kblockd
54  ?   00:00:00  ata_sff
55  ?   00:00:00  khubd
56  ?   00:00:00  md
57  ?   00:00:00  devfreq_wq
```

أول عملية تنشأ عند بدء نظام التشغيل هي `init` التي تنشئ العديد من العمليات ثم تبقى خاملة بلا عمل حتى تنتهي تلك العمليات التي أنشأتها. أما `kthreadd` فهي العملية التي يستخدمها نظام التشغيل لإنشاء خيوط threads جديدة سنتكلم عن الخيوط لاحقًا ولكن يمكن القول أن الخيط هو نوع معين من العمليات، ويشير k في بداية `kthreadd` إلى نواة kernel، وهي جزء نظام التشغيل المسؤول عن قدرات نظام التشغيل الأساسية مثل إنشاء الخيوط threads، ويشير حرف `d` الإضافي إلى عفريت daemon، وهو اسم آخر للعملية التي تعمل في الخلفية وتوفر خدمات نظام التشغيل. وبالنسبة للعملية `ksoftirqd` فهي عفريت للنواة kernel daemon أيضًا وعملها معالجة طلبات المقاطعة البرمجية software interrupt requests أو soft IRQ. أما `kworker` فهي عملية تنشئها النواة للعمل على عمليات معالجة خاصة بها.

توجد عمليات متعددة تشغّل خدمات النواة، ففي حالتي، يقول الكاتب، توجد 8 عمليات `ksoftirqd` وعدد 35 عملية `kworker`. لن نخوض في تفاصيل العمليات المتبقية، ولكن إذا كنت مهتمًا يمكنك البحث عن معلومات عنها وتطبيق الأمر `ps` لترى العمليات المشغلة على نظامك.

3. الذاكرة الوهمية Virtual memory

3.1 نظرية بت المعلومات A bit of information theory

البت هو رقم ثنائي ووحدة معلومات أيضًا، فبت واحد يعني احتمالًا من اثنين إما 0 أو 1، أما وجود بتين يعني وجود 4 تشكيلات محتملة: 00 و 01 و 10 و 11. وإذا كان لديك b بت فهذا يعني وجود 2^b قيمة محتملة، حيث يتكون البايت مثلًا من 8 بتات أي $2^8=256$ قيمة محتملة. في الاتجاه المقابل، أي إذا علمت عدد القيم المحتملة ولكنك لا تعلم عدد البتات المناسبة، افترض أنك تريد تخزين حرف واحد من حروف الأبجدية التي تتكون من 26 حرفًا فكم بتًا تحتاج؟ لديك 16 قيمة محتملة 4 بتات $2^4=256$ وبالتالي هذا غير كافٍ لتخزين 26 حرفًا. وتحصل على 32 قيمة محتملة بقيمة 5 بتات وهو كافٍ لتخزين كل الحروف مع قيم فائضة أيضًا. لذلك إذا أردت الحصول على قيمة واحدة من أصل N قيمة محتملة يجب عليك اختيار أصغر قيمة من b التي تحقق $2^b \geq N$ ، وبأخذ اللوغاريتم الثنائي للطرفين ينتج $\log_2(N) \leq b$.

تعطيك نتيجة رمي قطعة نقود بتًا واحدًا من المعلومات لأن قطعة النقود تملك وجهين وبالتالي احتمالين فقط. أما نتيجة رمي حجر نرد فتعطيك $\log_2(6)$ بتًا من المعلومات لأن حجر النرد له ستة أوجه. حيث إذا كان احتمال النتيجة هو 1 من N فذلك يعني أن النتيجة تحوي $\log_2(N)$ بتًا من المعلومات عمومًا، وإذا كان احتمال النتيجة هو p مثلًا فبذلك تحوي النتيجة $-\log_2(p)$ من المعلومات. تدعى هذه الكمية من المعلومات بالمعلومات الذاتية self-information للنتيجة، وهي تقيس مقدار التفاجؤ الذي تسببه تلك النتيجة، ويدعى هذا المقدار أيضًا surprisal. فإذا كان حصانك مشاركًا في سباق خيل على سبيل المثال ويملك فرصةً واحدة للفوز من أصل 16 فرصة ثم يفوز بالفعل، وبالتالي تعطيك تلك النتيجة 4 بتات من المعلومات $\log_2(16)=4$ ، أما إذا فاز حصان ما بنسبة 75% من المرات، فيتضمن ذلك الفوز الأخير 0.42 بتًا من المعلومات فقط. حيث

تحمل النتائج غير المتوقعة معلومات أكثر، أما عند تأكّدك من حدوث شيء ما فلن يعطيك حدوثه بالفعل إلّا كمية قليلة من المعلومات.

ينبغي عليك أن تكون على معرفة بالتحويل بين عدد البتات الذي نرمز له ب b وعدد القيم N التي تشفِّرها encode تلك البتات بحيث $N=2^b$.

3.2 الذاكرة والتخزين

تُحفَظ معظم بيانات عملية ما في الذاكرة الرئيسية main memory ريثما تنفّذ تلك العملية، حيث أن الذاكرة الرئيسية هي نوع من الذواكر العشوائية random access memory وتختصر إلى RAM. للذاكرة الرئيسية هي ذاكرة متطايرة volatile على معظم الحواسيب، والتي تعني أن محتوياتها تُفقد عند إغلاق الحاسوب. يملك الحاسوب المكتبي النموذجي ذاكرة تتراوح بين 4 و8 جيبي بايت وربما أكثر بكثير من ذلك، حيث GiB تشير إلى جيبي بايت gibibyte وهي 2^{30} بايتًا.

إذا قرأت وكتبت عملية ما ملفات فإن هذه الملفات تُخزّن على القرص الصلب hard disk drive -ويختصر إلى HDD- أو على solid state drive -ويختصر إلى SSD-. وسائط التخزين هذه غير متطايرة non-volatile)، لذلك تُستخدم للتخزين طويل الأمد. يحتوي الحاسوب المكتبي حاليًا HDD بسعة تتراوح بين 500 جيجا بايت و2 تيرا بايت، حيث GB هي جيجا بايت وتقابل 10^9 بايتًا بينما تشير TB إلى تيرا بايت وتساوي 10^{12} بايتًا.

لابد أنك لاحظت استخدام وحدة النظام الثنائي الجيبي بايت، أي التي تعد الكيلوبايت مثلًا مساويًا 1024 بايتًا حيث أساسها العدد 2، لقياس حجم الذاكرة الرئيسية واستخدام وحدتي النظام العشري الجيجا بايت والتيرا بايت، أي التي تعد الكيلو بايت مثلًا مساويًا 1000 بايتًا حيث أساسها العدد 10، لقياس حجم HDD. يقاس حجم الذاكرة بالوحدات الثنائية وحجم القرص الصلب بالوحدات العشرية وذلك لأسباب تاريخية وتقنية، ولكن تُستخدم الجيجا بايت واختصارها GB استخدامًا مبهمًا لذلك يجب أن تنتبه لذلك. يُستخدم مصطلح ذاكرة memory أحيانًا للدلالة على HDDs وSSDs وRAM، ولكن خصائص هذه الأجهزة الثلاث مختلفة جدًا، ويشار إلى HDDs وSSDs بتخزين دائم storage.

3.3 أحياز العنونة Address spaces

يُحدد كل بايت في الذاكرة الرئيسية بعدد صحيح يدعى عنوانًا حقيقيًا physical address، حيث تدعى مجموعة العناوين الحقيقية الصالحة بحيز العنونة الحقيقية physical address space وتتراوح تلك العناوين بين 0 و N - 1 حيث N هو حجم الذاكرة الرئيسية. أعلى قيمة عنوان صالحة في نظام ب 1 جيبي بايت ذاكرة حقيقية هو $2^{30}-1$ ، أي 1,073,741,823 في نظام العد العشري و 0x3fff ffff في نظام العد الست عشري حيث تحدد السابقة 0x أنه عدد ست عشري.

توفر معظم أنظمة التشغيل ذاكرةً وهميةً virtual memory أيضًا، والتي تعني أن البرامج لا تتعامل أبدًا مع عناوين حقيقية physical addresses وليست ملزمة بمعرفة كمية الذاكرة الحقيقية المتوفرة. وبدلًا من ذلك تتعامل البرامج مع الذاكرة الوهمية والتي تتراوح قيمها بين 0 و1 - M، حيث M هو عدد العناوين الوهمية الصالحة. ويحدد نظام التشغيل والعتاد الذي يعمل عليه حجم حيّز العنونة الوهمية.

لا بد أنك سمعت الناس يتحدثون عن نظامي التشغيل 32 بت و 64 بت، حيث يحدد هذان المصطلحان حجم المسجلات والذي هو حجم العنوان الوهمي أيضًا. فيكون العنوان الوهمي 32 بتًا على نظام 32 بت والذي يعني أن حيز العنونة الوهمية يتراوح بين 0 و 0xffff ffff، أي حجم العنونة الوهمية هو 2^{32} بايتًا أو 4 جيبي بايت. أما على نظام 64 بت فحجم حيز العنونة الوهمية هو 2^{64} بايتًا أو $2^4 . 1024^6$ بايتًا، أي 16 إكسبي بايت exbibytes والذي هو أكبر من حجم الذاكرة الحقيقية الحالية بمليار مرة تقريبًا.

يولّد البرنامج عناوينَ وهمية بكل عملية قراءة أو كتابة في الذاكرة، ويترجمها العتاد إلى عناوين حقيقية بمساعدة نظام التشغيل قبل الوصول إلى الذاكرة الرئيسية، وتقوم هذه الترجمة على أساس per-process أي تعامَل كل عملية باستقلالية عن العمليات الأخرى، حيث حتى لو ولّدت عمليتان نفس العنوان الوهمي فسترتبطان بمواقع مختلفة من الذاكرة الحقيقية. إذًا الذاكرة الوهمية هي إحدى طرق نظام التشغيل لعزل العمليات عن بعضها البعض، حيث لا تستطيع عملية ما الوصول إلى بيانات عملية أخرى، فلا وجود لعنوان وهمي يستطيع توليد ارتباطات بذاكرة حقيقية مخصصة لعملية أخرى.

3.4 أجزاء الذاكرة

تُنظّم بيانات العملية المُشغّلة ضمن خمسة أجزاء:

- جزء الشيفرة code segment: ويتضمن نص البرنامج وهو تعليمات لغة الآلة التي تبني البرنامج.
- الجزء الساكن static segment: يتضمن القيم غير القابلة للتغيير، قيم السلاسل النصية مثلًا، حيث إذا احتوى برنامجك على سلسلة مثلًا "Hello, World" فستُخزَّن هذه الحروف في الجزء الساكن من الذاكرة.
- الجزء العام global segment: يتضمن المتغيرات العامة global variables والمتغيرات المحلية local variables التي يُصرَّح عنها كساكنة `static`.
- جزء الكومة heap segment: يتضمن قطع الذاكرة المخصصة في زمن التشغيل وذلك باستدعاء دالة مكتبة في لغة C هي `malloc` في أغلب الأحيان.
- جزء المكدس stack segment: يتضمن استدعاء المكدس وهو سلسلة من إطارات المكدس stack frames. يُخصَّص إطار المكدس ليتضمن المعاملات والمتغيرات المحلية الخاصة

بالدالة في كل مرة تستدعى فيها الدالة، ويزال إطار المكدس ذاك التابع لتلك الدالة من المكدس عنـدما تنتهي الدالة من عملها.

يتشارك المصرِّف مع نظام التشغيل في تحديد ترتيب الأجزاء السابقة، حيث تختلف تفاصيل ذلك الترتيب من نظام تشغيل لآخر ولكن الترتيب الشائع هو:

- يوجد جزء نص البرنامج أو جزء الشيفرة قرب قاع الذاكرة أي عند العناوين القريبة من القيمة 0.
- يتواجد الجزء الساكن غالبًا فوق جزء الشيفرة عند عناوين أعلى من عناوين جزء الشيفرة.
- ويتواجد الجزء العام فوق الجزء الساكن غالبًا.
- ويتواجد جزء الكومة فوق الجزء العام وإذا احتاج للتوسع أكثر فسيتوسع إلى عناوين أكبر.
- ويكون جزء المكدس قرب قمـة الـذاكرة top of memory أي قـرب العنـاوين الأعلى في حـيز العنونـة الوهمية، وإذا احتاج المكدس للتوسع فسيتوسع للأسفل باتجاه عناوين أصغر.

لتعرف ترتيب هذه الأجزاء على نظامك، نفّذ البرنامج التالي:

```c
#include <stdio.h>
#include <stdlib.h>
int global;
int main()
{
    int local = 5;
    void *p = malloc(128);
    char *s = "Hello, World";
    printf("Address of main is %p\n", main);
    printf("Address of global is %p\n", &global);
    printf("Address of local is %p\n", &local);
    printf("p points to %p\n", p);
    printf("s points to %p\n", s);
}
```

`main` هو اسم دالة ولكن عند استخدامها كمتغير فهي تشير إلى عنوان أول تعليمة لغة آلة في الدالـة `main` والتي من المتوقع أن تكون في جـزء الشـيفرة text segment، أمـا `global` فهـو متغـير عـام وبالتـالي يُتوقـع تواجده في الجزء العام، والمتغير `local` هو متغير محلي أي يتواجد في جزء المكدس.

ترمز `s` إلى سلسلة نصية وهي السلسلة التي تكون جزءًا من البرنامج، على عكس السلسلة التي تُقرَأ من ملف أو التي يدخلها المستخدم. ومن المتوقع أن يكون موقع هذه السلسلة هو الجزء الساكن، في حين يكون المؤشر `s` الذي يشير إلى تلك السلسلة متغيرًا محليًا.

أما `p` فيتضمن العنوان الذي يعيده تنفيذ الدالة `malloc` حيث أنها تخصص حيزًا في الكومة، وترمز `malloc` إلى memory allocate أي تخصيص حيز في الذاكرة. يخبر التنسيق التسلسلي `p%` الدالة `printf` بأن تنسق كل عنوان كمؤشر pointer فتكون النتيجة عبارة عن عدد ست عشري hexadecimal.

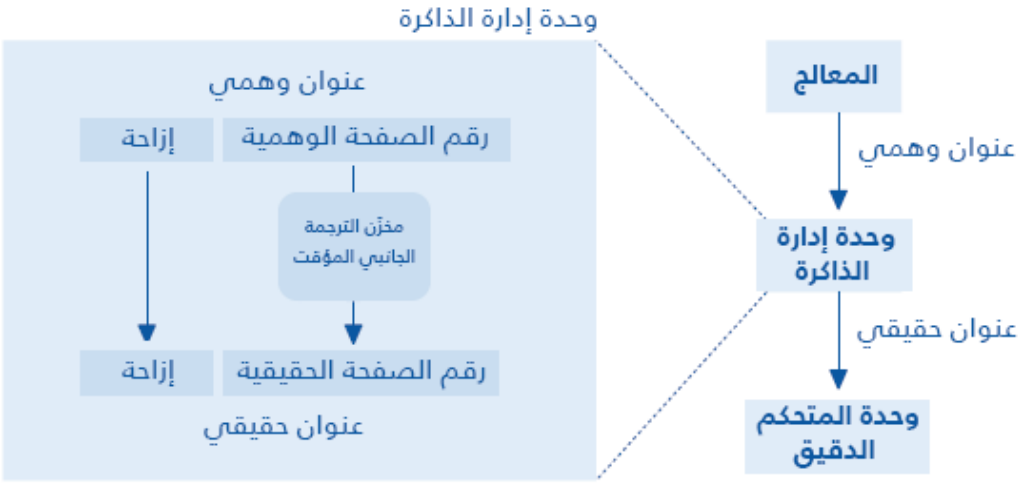
عند تنفيذ البرنامج السابق سيكون الخرج كما يلي:

```
Address of main is   0x 40057d
Address of global is 0x 60104c
Address of local is  0x7ffe6085443c
p points to          0x 16c3010
s points to          0x 4006a4
```

عنوان `main` هو الأقل كما هو متوقع، ثم يتبعه موقع سلسلة نصية، ثم موقع `global`، وبعده العنوان الذي يشير إليه المؤشر `p`، ثم عنوان `local` أخيرًا وهو الأكبر. يتكون عنوان `local` وهو العنوان الأكبر من 12 رقمًا ست عشريًا حيث يقابل كل رقم ست عشري 4 بتات وبالتالي يتكون هذا العنوان من 48 بتًا، ويدل ذلك أن القسم المُستخدم من حيز العنونة الوهمية هو 2^{48} بايتًا لأن حجم حيز العنونة الوهمية يكون مساويًا 2 حجم العنوان الوهمي بايتًا، وأكبر عنوان مستخدم في هذا المثال هو 48 بتًا.

جرب تنفيذ البرنامج السابق على حاسوبك وشاهد النتائج، أضف استدعاءً آخر للدالة `malloc` وتحقق فيما إذا أصبح عنوان كومة نظامك أكبر، وأضف أيضًا دالة تطبع عنوان متغير محلي وتحقق إذا أصبح عنوان المكدس يتوسع للأسفل.

يوضح المخطط التالي عملية ترجمة العناوين:



الشكل 1.3: رسم تخطيطي لعملية ترجمة العنوان.

3.5 المتغيرات المحلية الساكنة Static local variables

تدعى المتغيرات المحلية في المكدس بمتغيرات تلقائية automatic لأنها تُخصص تلقائيًا عند استدعاء الدالة وتحرر مواقعها تلقائيًا أيضًا عندما ينتهي تنفيذ الدالة. أما في لغة البرمجة C فيوجد نوع آخر من المتغيرات المحلية، تدعى ساكنة static، تخصص مواقعها في الجزء العام وتُهيّأ عند بدء تنفيذ البرنامج وتحافظ على قيمتها من استدعاءٍ لآخر للدالة.

تتتبّع الدالة التالية عدد مرات استدعائها على سبيل المثال:

```
int times_called()
{
    static int counter = 0;
    counter++;
    return counter;
}
```

تحدد الكلمة المفتاحية `static` أن المتغير `counter` هو متغير محلي ساكن، حيث أن تهيئة المتغير المحلي الساكن تحدث مرة واحدة فقط عند بدء تنفيذ البرنامج، ويخصص موقع المتغير `counter` في الجزء العام مع المتغيرات العامة وليس في جزء المكدس.

3.6 ترجمة العناوين Address translation

كيف يُترجم العنوان الوهمي VA إلى عنوان حقيقي PA؟

العملية الأساسية لتحقيق ذلك بسيطة، ولكن يمكن أن يكون التنفيذ البسيط أيضًا بطيئًا ويأخذ مساحة أكثر، لذلك يكون التنفيذ الحقيقي أعقد. توفر معظم العمليات وحدة إدارة الذاكرة memory management unit -وتختصر إلى MMU- التي تتموضع بين المعالج CPU والذاكرة الرئيسية وتطبّق ترجمة سريعة بين العناوين الوهمية والعناوين الحقيقية كما يلي:

1. يولّد المعالج CPU عنوانًا وهميًا VA عندما يقرأ أو يكتب البرنامج متغيرًا .
2. تقسّم MMU العنوان الوهمي إلى قسمين هما رقم الصفحة والإزاحة page number و offset. الصفحة page تعني قطعة ذاكرة ويعتمد حجم هذه الصفحة على نظام التشغيل والعتاد ولكن حجوم الصفحات الشائعة هي بين 1 و 4 كيبي بايت.
3. تبحث MMU عن رقم الصفحة في مخزَن الترجمة الجانبي المؤقت translation lookaside buffer، يُختصر إلى TLB، لتحصل MMU على رقم الصفحة الحقيقي المقابل ثم تدمج رقم الصفحة الحقيقي مع الإزاحة لينتج العنوان الحقيقي PA.
4. يمرَّر العنوان الحقيقي إلى الذاكرة الرئيسية التي تقرأ أو تكتب الموقع المطلوب.

يتضمن TLB نسخ بيانات مخبئية من جدول الصفحات page table والتي تُخزن في ذاكرة النواة، ويحتوي جدول الصفحات ربطًا بين أرقام الصفحات الوهمية و أرقام الصفحات الحقيقية. وبما أن لكل عملية جدول صفحاتها الخاص لذلك يجب على TLB أن يتأكد من أنه يستخدم مدخلات جدول صفحات العملية التي تنفّذ فقط. لفهم كيف تتم عملية الترجمة افترض أن العنوان الوهمي VA هو 32 بتًا والذاكرة الحقيقية 1 جيبي بايت مقسّمة إلى صفحات وكل صفحة ب 1 كيبي بايت:

- بما أن 1 جيبي بايت هي 2^{30} بايتًا و1 كيبي بايت هي 2^{10} بايت لذلك يوجد 2^{20} صفحةً حقيقية تدعى أحيانًا إطارات frames.
- حجم حيز العنونة الوهمية هو 2^{32} بايتًا وحجم الصفحة هو 2^{10} بايتًا لذلك يوجد 2^{22} صفحة وهمية.
- يحدد حجم الصفحة حجمَ الإزاحة وفي هذا المثال حجم الصفحة هو 210 بايتًا لذلك يتطلب 10 بتات لتحديد بايت من الصفحة.
- وإذا كان العنوان الوهمي 32 بتًا والإزاحة 10 بت فتشكّل 22 بتًا المتبقية رقم الصفحة الوهمية.
- وبما أنه يوجد 2^{20} صفحة حقيقية فكل رقم صفحة حقيقية هو 20 بتًا وأضف عليها إزاحة بمقدار 10 بت فتكون العناوين الحقيقية الناتجة بمقدار 30 بتًا.

يبدو كل شيء معقولًا حتى الآن، ولكن التنفيذ الأبسط لجدول الصفحات هو مصفوفة بمدخلة واحدة لكل صفحة وهمية، وتتضمن كل مدخلة رقم الصفحة الفعلية وهي 20 بتًا في هذا المثال، بالإضافة إلى بعض المعلومات الإضافية لكل إطار أو صفحة حقيقية، وبالتالي من 3 إلى 4 بايتات لكل مدخلة من الجدول ولكن مع 2^{22} صفحة وهمية يكون حجم جدول الصفحات 2^{24} بايتًا أو 16 ميبي بايت.

تحتاج كل عمليةٍ إلى جدول صفحات خاص بها لذلك إذا تضمن النظام 256 عملية مشغّلة فهو يحتاج 2^{32} بايتًا أو 4 جيبي بايت لتخزين جداول الصفحات فقط! وذلك مع عناوين وهمية بمقدار 32 بتًا وبالتالي مع عناوين وهمية بمقدار 48 و 64 بتًا سيكون حجم تخزين جداول الصفحات كبيرًا جدًا. لا نحتاج لتلك المساحة كلها لتخزين جدول الصفحات لحسن الحظ، لأن معظم العمليات لا تستخدم إلا جزءًا صغيرًا من حيز العنونة الوهمية الخاص بها، وإذا لم تستخدم العملية صفحةً وهمية فلا داعي لإنشاء مدخلة لها في جدول الصفحات.

يمكن القول بأن جداول الصفحات قليلة الكثافة مخوَّخَة sparse والتي تطبيقها باستخدام التنفيذات البسيطة، مثل مصفوفة مدخلات جدول الصفحات، هو أمر سيء. ولكن يمكن تنفيذ المصفوفات قليلة الكثافة المخوَّخَة sparse arrays بطرق أخرى أفضل لحسن الحظ. فأحد الخيارات هو جدول صفحات متعدد المستويات multilevel page table الذي تستخدمه العديد من أنظمة التشغيل، Linux مثلًا. الخيار الآخر هو الجدول الترابطي associative table الذي تتضمن كل مدخلة من مدخلاته رقم الصفحة الحقيقية والوهمية. يمكن أن يكون البحث في الجدول الترابطي بطيئًا برمجيًا، أما عتاديًا يكون البحث في كامل الجدول على التوازي parallel، لذلك تستخدم المصفوفات المترابطة associative arrays غالبًا في تمثيل مدخلات جدول الصفحات في TLB. يمكنك قراءة المزيد عن ذلك من خلال: en.wikipedia.org/wiki/Page_table.

كما ذُكر سابقًا أن نظام التشغيل قادر على مقاطعة أية عملية مُشغّلة ويحفظ حالتها ثم يشغّل عملية أخرى. تدعى هذه الآلية بتحويل السياق context switch. وبما أن كل عملية تملك جدول الصفحات الخاص بها فيجب على نظام التشغيل بالتعاون مع MMU أن يتأكد من أن كل عملية تحدث على جدول الصفحات الصحيح. وفي الآلات القديمة كان ينبغي أن تُستبدل معلومات جدول الصفحات الموجودة في MMU خلال كل عملية تحويل سياق، وبذلك كانت التكلفة باهظة. أما في الأنظمة الجديدة فإن كل مدخلة في جدول الصفحات ضمن MMU تتضمن معرّف عملية process ID لذلك يمكن أن تكون جداول صفحات عمليات متعددة موجودة في نفس الوقت في MMU.

4. فهم الملفات وأنظمتها

تُفقَد البيانات المخزَّنةٌ في الذاكرة الرئيسية لعمليةٍ ما عندما تكمل هذه العملية عملها أو تتعطل لسببٍ ما، ويُطلَق على البيانات المخزّنة في القرص الصلب hard disk drive -واختصاره HDD- والبيانات المخزّنة على أقراص التخزين ذات الحالة الثابتة solid state drive -وتختصر إلى SSD- ببياناتٍ دائمة persistent، أي أنها لا تُفقَد بعد اكتمال العملية حتى لو أغلِق الحاسوب. القرص الصلب HDD معقد، حيث تخزَّن البيانات ضمن كتل blocks التي تتواجد ضمن قطاعات sectors، وتشكّل القطاعاتُ مساراتٍ tracks ثم تنظَّم المسارات في دوائر متحدة المركز على أطباق platters القرص الصلب. أما أقراص التخزين ذات الحالة الثابتة HDD فهي أبسط إلى حدٍ ما لأنّ الكتل مرقّمة تسلسليًا، ولكنها تثير تعقيدًا مختلفًا فيمكن أن تُكتَب كل كتلة عددًا محدودًا من المرات قبل أن تصبح غير موثوقة للاستخدام مرةً أخرى.

والمبرمج غير مجبرٍ للتعامل مع تلك التعقيدات ولكن ما يحتاجه حقًا هو تجريدٌ مناسب لعتاد التخزين الدائم persistent storage hardware، وتجريد التخزين الدائم الأكثر شيوعًا هو نظام الملفات file system، فيمكن القول بتجريد أن:

- نظام الملفات ما هو إلا ربط mapping بين اسم الملف ومحتوياته، فإذا عُدَّت أسماء الملفات مفاتيحًا keys ومحتويات الملف قيمًا values فإن نظام الملفات مشابه لقاعدة البيانات ذات النوع مفتاح-قيمة key-value database
- الملف هو سلسلة من البايتات

تكون أسماء الملفات عادةً من النوع سلسلة string وتكون بنظام هرمي حيث يكون اسم الملف عبارة عن مسار يبدأ من المجلد الأعلى مستوى top-level directory مرورًا بمجلدات فرعية حتى الوصول إلى الملف المطلوب.

الاختلاف الرئيسي بين الآلية الأساسية underlying mechanism والتي هي التخزين الدائم وتجريدها والذي هو نظام الملفات هو أن الملفات تعمل على أساس البايت byte-based أما التخـزين الـدائم يعمـل على أسـاس الكتلة block-based.

يترجم نظام التشغيل عمليات الملف ذات الأساس البايتي في مكتبة C إلى عمليات ذات أساس كتلي على أجهزة التخزين الدائم، ويتراوح حجم الكتلة بين 1 و 8 كيبي بايت (KiB التي تساوي 2^{10} أو 1024 بـايت). فتفتح الشيفرة التالية ملفًا وتقرأ أول بايت:

```
FILE *fp = fopen("/home/downey/file.txt", "r");
char c = fgetc(fp);
fclose(fp);
```

يحدث ما يلي عند تشغيل الشيفرة السابقة:

1. تستخدم الدالة `fopen` اسم الملف لإيجاد المجلد الأعلى مسـتوى وهـو `/` ثم المجلـد الفـرعي `home` ثم المجلد الفرعي المتواجد ضمن `home` وهو `downey`.

2. لتجد بعد ذلك ملفًا اسمه `file.txt` وتفتحه للقراءة، وهذا يعني أن `fopen` تنشئ بنية بيانات تمثـل الملف المقروء، حيث تتتبّع بينةُ البيانات الكميـةَ المقـروءة من الملـف، وتـدعى هـذه الكميـة المقـروءة بموضع الملف file position. وتدعى بنية البيانات تلك بكتلة تحكم الملـف File Control Block في DOS، ولكنني أريد، يقول الكاتب، تجنّب ذلك المصطلح لأن له معنىً آخر في UNIX، وبالتـالي لا يوجـد اسمٌ جيد لبنية البيانات تلك في UNIX، وبما أنها مدخلة في جـدول الملـف المفتـوح لـذلك سأسـميها، يقول الكاتب، بمدخلة جدول الملف المفتوح OpenFileTableEntry.

3. يتحقق نظـام التشـغيل من وجـود الحـرف التـالي من الملـف مسـبقًا في الـذاكرة عنـد اسـتدعاء الدالـة `fgetc`، إذا كان موجود فإن الدالة `fgetc` تقرأ الحرف التالي وتقـدّم موضـع الملـف إلى الحـرف الـذي بعده ثم تعيد النتيجة.

4. أما إذا لم يوجـد الحـرف التـالي في الـذاكرة فيصـدر نظـام التشـغيل طلب إدخـال/إخـراج I/O request للحصول على الكتلة التالية. القرص الصلب بطيء لذلك تُقاطَع العملية التي تنتظـر وصـول بيانـاتٍ من القرص الصلب عادةً وتشغَّل عملية أخرى ريثما تصل تلك البيانات.

5. تُخزَّن الكتلة الجديدة من البيانات في الذاكرة عند اكتمال عمليـة الإدخـال/الإخـراج ثم تسـتأنف العمليـة عملها، حيث يُقرأ أول حرف ويُخزَّن كمتغير محلي.

6. يكمل نظام التشغيل أية عملية معلّقة pending operations أو يلغيهـا ثم يزيـل البيانـات المخزنـة في الذاكرة ويحرر مدخلة جدول الملف المفتوح OpenFileTableEntry عندما تغلق العمليةُ الملف.

عملية الكتابة في ملف مشابهة لعملية القراءة من ملف ولكن مع وجود خطوات إضافية، حيث يفتح المثال التالي ملفًا للكتابة ويغير أول حرف من الملف:

```
FILE *fp = fopen("/home/downey/file.txt", "w");
fputc('b', fp);
fclose(fp);
```

يحدث ما يلي عند تشغيل الشيفرة السابقة:

1. تستخدم الدالة `fopen` اسم الملف لإيجاده، فإذا كان الملف غير موجود مسبقًا فتنشئ الدالة `fopen` ملفًا جديدًا وتضيف مدخلةً في المجلد الأب `/home/downey`.
2. ينشئ نظام التشغيل مدخلة جدول الملف المفتوح (OpenFileTableEntry) التي تحدد أن الملف مفتوح للكتابة وتهيئ موضع الملف بالقيمة 0.
3. تحاول الدالة `fputc` كتابة أو إعادة كتابة البايت الأول من الملف، حيث إذا كان الملف موجودًا مسبقًا فيجب على نظام التشغيل تحميل الكتلة الأولى من الملف إلى الذاكرة، وإذا لم يوجد الملف فسيخصّص نظام التشغيل مكانًا للكتلة الجديدة في الذاكرة ويطلب كتلة جديدة من القرص الصلب.
4. يمكن ألّا تُنسَخ الكتلة المعدّلة في الذاكرة إلى القرص الصلب بعد تعديلها مباشرةً، حيث تُخزَّن البيانات المكتوبة في الملف تخزينًا مؤقتًا buffered أي أنها تُخزَّن في الذاكرة، ولكنها لا تُكتَب في القرص الصلب إلّا عند وجود كتلة واحدة على الأقل لكتابتها.
5. تُكتَب البيانات المخزنة تخزينًا مؤقتًا في القرص الصلب وتُحرَّر مدخلة جدول الملف المفتوح عند إغلاق الملف.

باختصار توفر مكتبة C تجريدًا هو نظام الملفات الذي يربط أسماء الملفات بمجرًى من البايتات، ويُبنى هذا التجريد على أجهزة التخزين الدائم التي تنظَّم ضمن كتل.

4.1 أداء القرص الصلب Disk performance

الأقراص الصلبة بطيئة حيث إن الوقت الوسطي لقراءة كتلة من القرص الصلب إلى الذاكرة يتراوح بين 5 و25 ميلي ثانية على الأقراص الصلبة الحالية HDDs (تعرّف على خصائص أداء القرص الصلب). أما SSDs فهي أسرع من HDDs، حيث تستغرق قراءة كتلة حجمها 4 كيبي بايت 25 ميكرو ثانية وتستغرق كتابتها 250 ميكرو ثانية (انظر مقال القرص الصلب: آلية تخزين البيانات على الأقراص).

وإذا وازنت الأرقام السابقة مع دورة ساعة المعالج clock cycle، حيث إن المعالج الذي يملك معدل ساعة clock rate مقداره 2 جيجا هرتز يكمل دورةَ ساعةٍ كل 0.5 نانو ثانية، والوقت اللازم لجلب بايت من الذاكرة إلى المعالج هو حوالي 100 نانو ثانية، وبالتالي إذا أكمل المعالج تعليمةً واحدةً في كل دورة ساعة (التي

مقدارها 0.5 نانو ثانية) فإنه سيكمل 200 تعليمة خلال انتظاره وصول بايت من الـذاكرة إليـه (200=100/0.5). وسيكمل المعالج 2000 تعليمة بدورة ساعة 1 ميكرو ثانية، وبالتالي يكمـل المعـالج 50,000 تعليمـة خلال وقت انتظار جلب بايت من SSD والذي يقدر بقيمة 25 ميكرو ثانية.

ويستطيع المعالج إكمال 2,000,000 تعليمة خلال ميلي ثانيـة وبـذلك يسـتطيع إكمـال 40 مليـون تعليمـة خلال وقت انتظار جلب بايت من القرص الصلب HDD والمقدّر بقيمة 20 ميلي ثانية. إذا لم يكن لـدى المعـالج أي عملٍ للقيام به خلال عملية انتظار جلب بيانات من القرص الصلب فإنـه يبقى خـاملًا بلا عمـل، لـذلك ينتقـل المعالج لتنفيذ عملية أخرى ريثما تصل البيانات من القرص الصلب.

أحد أهم التحديات التي تواجـه عمليـة تصـميم نظـام التشـغيل هـو الفجـوة في الأداء بين الـذاكرة الرئيسـية والتخزين الدائم، لذلك توفر أنظمة التشغيل والعتـاد مجموعـة خاصـيات الهـدف منهـا سـد هـذه الفجـوة وهـذه الخاصيات هي:

- تحويلات الكتلة Block transfers: يتراوح الوقت اللازم لتحميل بايت واحد من القـرص الصـلب بين 5 و 25 ميلي ثانية، بالمقابل فـإن الـوقت الإضـافي لتحميـل كتلـة حجمهـا 8 كيـبي بـايت KiB هـو وقت مهمل، لذلك تحاول أنظمة التشغيل قراءة كتل كبيرة الحجم في كل عملية وصول إلى القرص الصلب.
- الجلب المسبق Prefetching: يستطيع نظام التشغيل في بعض الأحيان توقّعَ أن العمليـة سـتقرأ كتلـةً ما ثم يبدأ بتحميل تلك الكتلة من القرص الصلب قبل أن تُطلَب. فإذا فتحتَ ملفًا وقرأت أول كتلة على سبيل المثال فهذا يؤدي إلى وجود احتمال كبير أنك ستقرأ الكتلة الثانية، لذلك سيحمّل نظـام التشـغيل كتل إضافية قبل أن تُطلَب.
- التخزين المؤقت Buffering: يخزّن نظام التشغيل بيانات الكتابة في ملفٍ ما في الذاكرة ولا يكتبهـا في القرص الصلب مباشرةً، لذلك إذا عدّلت الكتلة مراتٍ متعددة عند وجودها في الذاكرة فلن يكتبهـا نظـام التشغيل في القرص الصلب إلّا مرةً واحدة.
- التخبئة Caching: إذا استخدمت عمليةٌ كتلةً ما مؤخرًا فإنها ستستخدمها مرة أخـرى قريبًـا، وإذا احتفـظ نظام التشغيل بنسخة من هذه الكتلة في الذاكرة فإنه سيتعامل مـع الطلبـات المسـتقبلية لهـذه الكتلـة بسرعة الذاكرة.

تُطبَّق بعض الخاصيات السابقة عن طريق العتاد أيضًا، حيث توفر بعض الأقراص الصلبة ذاكرةً مخبئيةً على سبيل المثال وتُخزَّن فيها الكتل المستخدمة مؤخرًا، وتقرأ العديد من الأقراص الصلبة عدة كتل في نفس الـوقت على الرغم من وجود كتلة واحدة مطلوبة فقط.

تحسّن الآليات السابقة من أداء البرامج ولكنها لا تغير شيئًا من سلوكها، ولا يتوجب على المـبرمجين معرفـة الكثير عن تلك الآليات باستثناء حالتين هما: (1) إذا أصبح أداء البرنامج سـيئًا بشـكل غـير متوقـع فيجب عليـك معرفة هذه الآليات لتشخيص المشكلة. (2) يصبح تنقيح أخطاء debug البرنامج أصعب عنـدما تُخـزَّن البيانـات

تخزينًا مؤقتًا، فإذا طبع البرنامج قيمة ثم تعطّل البرنامج على سبيل المثال، فإن تلك القيمة لن تظهر لأنهـا يمكن أن تكون في مخزَن مؤقت buffer. وإذا كتب برنامجٌ ما بياناتٍ في القرص الصلب ثم أُغلق الحاسوب فجأةً قبـل كتابة البيانات في القرص الصلب فيمكن أن تُفقَد تلك البيانات إذا كانت في الذاكرة المخبئيـة cache ولم تنقَـل بعد إلى القرص الصلب.

4.2 بيانات القرص الصلب الوصفية Disk metadata

تكون الكتل التي تشكّل الملف منظمةً في القرص الصـلب بحيث قـد تكـون هـذه الكتـل مجـاورةً لبعضـها البعض وبـذلك يكـون أداء نظـام الملفـات أفضـل، ولكن قـد لا تحـوي معظم أنظمـة التشـغيل تخصيصًـا متجاورًا contiguous allocation للكتل، حيث يكون لأنظمة التشغيل كامل الحرية بوضع كتلةٍ ما في أي مكـان تريده على القرص الصلب وتستخدم بنى بيانات مختلفة لتتبع تلك الكتل.

تُدعى بنية البيانات في العديد من أنظمة ملفات يونكس UNIX باسم inode والتي ترمـز إلى عقـدة دليـل أو مؤشـر فهرسـة index node، وتـدعى معلومـات الملفـات مثـل موقـع كتـل هـذه الملفـات بالبيانـات الوصفية metadata، فمحتوى الملف هو بيانات data ومعلومات الملف بيانـات أيضًـا ولكنهـا بيانـات توصـف بيانات أخرى لذلك تدعى وصفية meta.

بما أن مؤشـرات الفهرسـة inodes تتوضـع على القـرص الصـلب مـع بقيـة البيانـات لـذلك فهي مصـممة لتتوافق مع كتل القرص الصلب بدقة. يتضمن المؤشر inode لنظام يونكس UNIX معلومات عن الملـف وهـذه المعلومـات هي معـرف ID المسـتخدم مالـك الملـف، ورايـات الأذونـات permission flags والـتي تحـدد مَن المسموح له قراءة أو كتابة أو تنفيذ الملف، والعلامات الزمنية timestamps التي تحدد آخر تعـديل وآخـر دخـول إلى الملف، وتتضمن أيضًا أرقام أول 12 كتلة من الكتل المشكّلة للملف.

فإذا كان حجم الكتلة الواحدة 8 كيبي بايت KiB فإن حجم أول 12 كتلة من الملف هو 96 كيـبي بـايت وهـذا الرقم كبير كفاية لأغلبية حجوم الملفات ولكنه ليس بكافٍ لجميع الملفات بالتأكيد، لـذلك تحتـوي inode مؤشـرًا إلى كتلة غير موجهة indirection block التي تتضمن مؤشّرات إلى كتلٍ أخرى فقط.

يعتمد عدد المؤشرات في كتلةٍ غير موجهةٍ على أحجام الكتل وعددها وهو 1024 كتلـةً عـادةً، حيث تسـطيع كتلةٌ غير موجهة عنونة 8 ميبي بايت (MiB التي تساوي 220 بايتًا) باستخدام 1024 كتلـة وبحجم 8 كيـبي بـايت لكل كتلة، وهذا رقمٌ كافٍ لجميع الملفات باستثناء الملفات الكبيرة أي أنه لا يزال غير كافٍ لجميع الملفات، لذلك يتضمن المؤشر inode أيضًا مؤشرًا إلى كتلة غير موجهة مضـاعفة double indirection block الـتي تتضـمن بدورها مؤشرات إلى كتـل غـير موجهـة، وبالتـالي يمكننـا عنونـة 8 جيـبي بـايت (GiB الـتي تسـاوي 2^{30} بايتًـا) باسـتخدام 1024 كتلـة غـير موجهـة. وإذا لم يكن ذلـك كافيًـا أيضًـا فتوجـد كتلـة غـير موجهة ثلاثية triple indirection block أخيرًا، والتي تتضمن مؤشرات إلى كتل غير موجهة مضاعفة وبذلك تكون كافية لملف حجمه 8 تيبي بايت (TiB التي تساوي 2^{40} بايتًا) كحدٍ أعلى.

بدا ذلك كافيًا لمدة طويلة عندما صُمِّمت inodes لنظام يونكس UNIX، ولكنها أضحت قديمةً الآن. تستخدم بعض أنظمة الملفات مثل FAT جدول تخصيص الملف File Allocation Table كبديل عن الكتل غير الموجهة، حيث يتضمن جدول التخصيص مدخلةً لكل كتلة وتدعى الكتلة هنا بعنقود cluster، ويتضمن المجلد الجذر root directory مؤشرًا لأول عنقود في كل ملف، وتشير مدخلة جدول FAT والتي تمثل عنقودًا إلى العنقود التالي في الملف بشكل مشابه للائحة المترابطة linked list.

4.3 تخصيص الكتلة Block allocation

ينبغي على أنظمة الملفات تتبّع الكتل التابعة لكل ملف وتتبّع الكتل المتاحة للاستخدام أيضًا، حيث يجد نظام الملفات كتلةً متوفرة لملفٍ ما ثم يخصصها له عندما يُنشَأ هذا الملف، ويجعل نظام الملفات كتل ملفٍ ما متاحةً لإعادة التخصيص عندما يُحذَف ذلك الملف. حيث أهداف تخصيص الكتلة هي ما يلي:

- السرعة Speed: يجب أن يكون تخصيص الكتل وتحريرها سريعًا.
- الحد الأدنى من استهلاك المساحة Minimal space overhead: يجب أن تكون بنى البيانات التي يستخدمها المخصّص allocator صغيرة الحجم بحيث تترك أكبر قدر ممكن من المساحة للبيانات.
- الحد الأدنى من التجزئة Minimal fragmentation: إذا وُجدت كتل غير مستخدمة نهائيًا أو مستخدمة جزئيًا فإن هذه المساحة غير المستخدمة تدعى تجزئة fragmentation.
- الحد الأعلى من التجاور Maximum contiguity: يجب أن تكون البيانات التي تُستخدم في الوقت ذاته مجاورة لبعضها البعض فيزيائيًا إذا كان ممكنًا وذلك لتحسين الأداء.

تصميم نظام ملفات يحقق هذه الأهداف أمرٌ صعب خاصةً أن أداء نظام الملفات معتمدٌ على خواص الحِمل workload characteristics مثل حجم الملفات وأنماط الوصول access patterns وغير ذلك، فنظام الملفات المتماشي جيدًا مع نوع حِمل قد لا يكون أداؤه جيدًا مع نوع حِملٍ آخر، ولهذا السبب تدعم معظم أنظمة التشغيل أنواعًا متعددة من أنظمة الملفات. يُعَد تصميم نظام الملفات مجالًا نشطًا للبحث والتطوير، حيث انتقلت أنظمة تشغيل Linux خلال العشر سنوات الماضية من ext2 والذي كان نظام ملفات UNIX التقليدي إلى ext3 والذي هو نظام ملفات مزودٌ بسجل journaling ومُعَد لتحسين السرعة speed والتجاور contiguity، ثم انتقلت بعد ذلك إلى ext4 الذي يستطيع التعامل مع ملفات وأنظمة ملفات أكبر، وقد يكون هناك انتقال migration آخر إلى نظام ملفات B-tree واختصاره Btrfs خلال السنوات القليلة القادمة.

انظر مقال"مقدمة إلى نظام ملفات لينكس EXT4" ومقال "دليل المستخدم للروابط في نظام ملفات لينكس" لدعم ما تحدثنا عنه.

4.4 هل كل شيء هو ملف؟

إن تجريد الملف حقيقةً هو تجريدٌ لمجرى من البايتات stream of bytes والذي اتضح أنه مفيدٌ لكثيرٍ من الأشياء وليس لأنظمة الملفات فقط، أنبوب pipe نظام UNIX هو مثالٌ على ذلك والذي هو نموذجٌ بسيط للاتصالات بين العمليات inter-process communication، فتكون العمليات مُعدّةً بحيث يكون خرج عمليةٍ ما دخلًا لعمليةٍ أخرى.

يتصرف الأنبوب على أساس أن أول عملية هي ملفٌ مفتوحٌ للكتابة بالتالي يمكنه أن يستخدم دوال مكتبة C مثل `fputs` و `fprintf` وأن العملية الأخرى ملفٌ مفتوحٌ للقراءة أي يمكنه استخدام الدالة `fgets` والدالة `fscanf`.

وتستخدم شبكة الاتصالات تجريد مجرى البايتات أيضًا، فمِقبس socket نظام UNIX هو بنية بيانات تمثل قناة اتصال بين العمليات الموجودة على حواسيب مختلفة عادةً، حيث تستطيع العمليات قراءة بيانات وكتابتها في مِقبس باستخدام دوال تتعامل مع الملفات.

تجعل إعادة استخدام تجريد الملف الأمور أسهل بالنسبة للمبرمجين، حيث إنهم غير ملزمين إلا بتعلم واجهة برمجة تطبيقات واحدة application program interface -واختصارها API، كما أنها تجعل البرامج متعددة الاستعمال بما أن البرنامج المجهّز ليعمل مع الملفات قادرٌ على العمل مع بيانات قادمة من الأنابيب ومصادر أخرى أيضًا.

5. مزيد من البتات والبايتات

5.1 تمثيل الأعداد الصحيحة integers

لا بدّ أنك تعلم أن الحواسيب تمثّل الأعداد بنظام العد ذو الأساس 2 (أي base 2) والمعروف أيضًا بالنظام الثنائي binary.

التمثيل الثنائي للأعداد الموجبة واضحٌ فعلى سبيل المثال التمثيل الثنائي للعدد 510 (أي للعدد 5 في نظام العد العشري) هو b101 (أي 101 بنظام العد الثنائي binary). بينما يستخدم التمثيلُ الأوضح للأعداد السالبة بتًا للإشارة sign bit لتحديد فيما إذا كان العدد موجبًا أو سالبًا، ولكن يوجد تمثيلٌ آخر يدعى بالمتمّم الثنائي two's complement وهو التمثيل الأكثر شيوعًا لأن العمل معه أسهل ضمن العتاد.

لإيجاد المتمم الثنائي للعدد السالب x- تجد التمثيل الثنائي للعدد x أولًا، ثم تقلب flip كل البتات أي تقلب الأصفار واحدات والواحدات أصفارًا، ثم تجمع 1 لناتج القلب، فلتمثيل العدد 510- بالنظام الثنائي على سبيل المثال، تبدأ بتمثيل العدد 510 بالنظام الثنائي بكتابته بنسخة 8 بت 8bit version وهو b00000101، ثم تقلب flip كل البتات وتضيف له 1 فينتج b11111011. يتصرف البت الموجود أقصى اليسار في المتمم الثنائي كبت إشارة، فهو 0 في الأعداد الموجبة و1 في الأعداد السالبة.

يجب إضافة أصفار للعدد الموجب وواحدات للعدد السالب عند تحويل عدد من النوع 8 بت إلى 16 بتًا، أي يجب نسخ قيمة بت الإشارة إلى البتات الجديدة وتدعى هذه العملية بامتداد الإشارة sign extension. كل أنواع الأعداد الصحيحة في لغة البرمجة C لها إشارة أي أنها قادرة على تمثيل الأعداد السالبة والموجبة إلّا إذا صرّحت عنهم كأعداد صحيحة بِلا إشارة unsigned، والاختلاف الذي يجعل هذا التصريح مهمًا هو أن عمليات الأعداد الصحيحة التي لا تملك إشارة unsigned integers لا تستخدم امتداد الإشارة.

5.2 العامِلات الثنائية Bitwise operators

يُصاب متعلمو لغة البرمجة C بالارتباك أحيانًا بالنسبة للعاملين الثنائيين & و | ، حيث يعامِل هذان العامِلان الأعدادَ الصحيحة integers كمتجهات من البتات bit vectors وتُحسَب العمليات المنطقية logical operations بتطبيقها على البتات المتناظرة، حيث تَحسب &عمليةAND بحيث ينتج 1 إذا كانت قيمة كلا المُعامَلين operands هي 1 وينتج 0 بخلاف ذلك.

المثال التالي هو تطبيق العامِل & على عددين مكونين من 4 بتات:

```
   1100
&  1010
   ----
   1000
```

وهذا يعني في لغة البرمجة C أن قيمة التعبير 12 & 10 هي 8.

ويحسب العامِل | عملية OR بحيث ينتج 1 إذا كانت قيمة أحد المعامَلين 1 وينتج 0 بخلاف ذلك كما في المثال التالي:

```
   1100
|  1010
   ----
   1110
```

أي قيمة التعبير 12 | 10 هي 14.

ويحسب العامل ^ عملية XOR بحيث ينتج 1 إذا كانت قيمة أحد المعامَلين 1 وليس كلاهما كما في المثال التالي:

```
   1100
^  1010
   ----
   0110
```

أي قيمة التعبير 12 ^ 10 هي 6.

يُستخدَم العامل & لتصفير clear مجموعة بتات من متجهة بتات، بينما يُستخدم العامل | لضبط (set) البتات، ويُستخدم العامل ^ لقلب flip أو تبديل toggle البتات كما يلي:

- تصفير البتات Clearing bits: قيمة x&0 هي 0 وقيمة x&1 هي x مهما كانت قيمة x، لذلك عند تطبيق العملية AND على متجهة مع العدد 3 فسيُختار البتان الموجودان أقصى اليمين فقط وتُضبَط بقية البتات بالقيمة 0 كما يلي:

```
   xxxx
&  0011
   ----
   00xx
```

حيث تُدعى القيمة 3 العشرية أو 0011 الثنائية بقناع mask لأنها تختار بعض البتات وتقنّع البتات الباقية.

- ضبط البتات Setting bits: قيمة x|0 هي x وقيمة x|1 هي 1 مهما كانت قيمة x، لذلك عند تطبيق OR على متجهة مع العدد 3 فستُضبط البتات الموجودة أقصى اليمين بينما ستُترَك بقية البتات كما هي:

```
   xxxx
|  0011
   ----
   xx11
```

- تبديل البتات Toggling bits: إذا طبّقت XOR مع العدد 3 فستُقلب البتات الموجودة أقصى اليمين وتُترك بقية البتات كما هي. جرّب حساب المتمم الثنائي للعدد 12 باستخدام ^ كتمرينٍ لك، تلميح: ما هو تمثيل المتمم الثنائي للعدد 1-؟

توفر لغة البرمجة C عاملات إزاحة أيضًا مثل << و >> التي تزيح البتات يمينًا ويسارًا، حيث تضاعف الإزاحةُ لليسار العددَ، فناتج 5 << 1 هو 10 أما ناتج 5 >> 2 هو 20، وتقسم الإزاحة لليمين العدد على 2 (ويكون الناتج مُقرّبًا للأسفل rounding down) حيث ناتج 5 >> 1 هو 2 وناتج 2 >> 1 هو 1.

انظر توثيق "معاملات الأعداد الثنائية" في موسوعة حسوب لمزيد من التفصيل والأمثلة.

5.3 تمثيل الأعداد العشرية floating-point numbers

تُمثَّل الأعداد العشرية باستخدام النسخة الثنائية للصيغة العلمية scientific notation، وتُكتَب الأعداد الكبيرة في الصيغة العشرية decimal notation كحاصل ضرب معامِلٍ مع 10 مرفوعة لأسّ، فسرعة الضوء المقدّرة بالمتر/ ثانية تساوي تقريبًا $2.998.10^8$ على سبيل المثال.

تستخدم معظم الحواسيب معيار IEEE standard لحساب الأعداد العشرية (معيار 754 IEEE للأعداد العشرية)، حيث يقابل النوع float في C المعيار IEEE المكون من 32 بتًا أما النوع double يقابل المعيار 64 بتًا. البت الموجود أقصى اليسار هو بت الإشارة sign bit ويرمز له s في معيار 32 بت، و تمثل 8 بتات التالية الأسَّ exponent و يرمز له q وآخر 23 بت هي المعامِل coefficient ويرمز له c، وبالتالي قيمة العدد العشري هي: $(-1)^s c.2^q$.

تلك القيمة صحيحة تقريبًا ولكن هناك شيء بسيط أيضًا، فالأعداد العشرية موحّدة بحيث يوجد رقمٌ واحد قبل الفاصلة، لذلك يُفضَّل في النظام العشري على سبيل المثال الصيغة $2.998.10^8$ على الصيغة $2.998.10^5$ أو على أي تعبيرٍ آخر مكافئ.

أما بالنظام الثنائي فالأعداد موحّدة بحيث يوجد الرقم 1 قبل الفاصلة الثنائية دومًا، وبما أن الرقم في هذا الموقع هو 1 دومًا لذلك يمكن توفير مساحة وذلك بعدم إدخال هذا الرقم ضمن التمثيل. فتمثيل العدد الصحيح 13_{10} هو b1101 على سبيل المثال، أما التمثيل العشري هو $1.101.2^3$ حيث 3 هو الأس وجزء المعامِل الذي يمكن أن يُخزَّن هو 101 (متبوعًا بمقدار 20 صفرًا). هذا صحيحٌ تقريبًا ولكن يوجد شيءٌ آخر أيضًا، وهو أن الأس يُخزّن مع معدل انحياز bias، وقيمة معدل الانحياز في معيار 32 بت هي 127، وبالتالي يمكن تخزين الأس 3 كَ 130.

تُستخدَم عمليات الاتحاد union operations والعمليات الثنائية bitwise operations لضغط وفك ضغط الأعداد العشرية في C كما في المثال التالي:

```c
union
{
    float f;
    unsigned int u;
} p;

p.f = -13.0;
unsigned int sign = (p.u >> 31) & 1;
unsigned int exp = (p.u >> 23) & 0xff;

unsigned int coef_mask = (1 << 23) - 1;
unsigned int coef = p.u & coef_mask;

printf("%d\n", sign);
printf("%d\n", exp);
printf("0x%x\n", coef);
```

يسمح النوع union بتخزين القيمة العشرية باستخدام المتغير p.f ثم قراءته كعدد صحيح بلا إشارة باستخدام المتغير p.u. وللحصول على بت الإشارة تُزاح البتات يمينًا بمقدار 31 ثم يُستخدم قناع 1 بت (أي تطبيق & على ناتج الإزاحة مع العدد 1) وذلك لاختيار البت الموجود أقصى اليمين فقط. وللحصول على الأس تُزاح البتات بمقدار 23 ثم تُختار 8 بتات الموجودة أقصى اليمين، حيث تملك القيمة الست عشرية 0xff ثمانية واحدات. وللحصول على المعامِل تحتاج لإزالة 23 بتًا الموجودين أقصى اليمين وتجاهل بقية البتات، ويمكن تحقيق ذلك من خلال إنشاء قناع مكون من واحدات في البتات 23 الموجودة أقصى اليمين وأصفارًا في البتات الموجودة على اليسار، والطريقة الأسهل لإنشاء هذا القناع هي إزاحة 1 يسارًا بمقدار 23 ثم يُطرَح من ناتج الإزاحة 1.

خرج البرنامج السابق هو كما يلي:

```
1
130
0x500000
```

بت الإشارة للعدد السالب هو 1 كما هو متوقع، والأس هو 130 متضمنًا معدل الانحياز، والمعامل هو 101 متبوعًا بمقدار 20 صفرًا ولكنه طُبع بالنظام الست عشري 0x500000.

5.4 أخطاء الاتحادات وأخطاء الذاكرة

يوجد استخدامان شائعان لاتحادات لغة البرمجة C أحدهما هو الوصول إلى التمثيل الثنائي للبيانات كما ذُكر سابقًا. والاستخدام الآخر هو تخزين البيانات غير المتجانسة heterogeneous data، فيمكنك استخدام اتحاد union لتمثيل عددٍ والذي من الممكن أن يكون عددًا صحيحًا integer أو عشريًا float أو مركبًا complex أو كسريًا rational.

الاتحادات معرضةٌ للخطأ على كل حال، ويعود الأمر للمبرمج لتتبع نوع البيانات الموجودة في الاتحاد، فإذا كتبت قيمة عشرية ثم فُسِّرت كعدد صحيح فستكون النتيجة لا معنًى لها، وهو أمرٌ مماثل لقراءة موقع من الذاكرة بصورة خاطئة مثل قراءة قيمة موقع من مصفوفة في حين تكون هذه المصفوفة انتهت أي قراءة قيمة من خارج حدود المصفوفة. تخصص الدالة التالية مكانًا للمصفوفة في المكدس وتملؤه بأعداد من 0 إلى 99:

```
void f1()
{
    int i;
    int array[100];

    for (i = 0; i < 100; i++)
```

```
    {
        array[i] = i;
    }
}
```

ثم تعرّف الدالة التالية مصفوفةً أصغر وتدخل عناصر قبل بداية المصفوفة وبعد نهايتها عمدًا:

```
void f2()
{
    int x = 17;
    int array[10];
    int y = 123;

    printf("%d\n", array[-2]);
    printf("%d\n", array[-1]);
    printf("%d\n", array[10]);
    printf("%d\n", array[11]);
}
```

تكون نتيجة استدعاء الدالة f1 ثم استدعاء الدالة f2 ما يلي:

```
17
123
98
99
```

تعتمد التفاصيل على المصرّف compiler الذي يرتّب المتغيرات في المكدّس، ويمكنك من خلال النتائج السابقة استنتاج أن المصرّف يضع المتغيرين x و y قرب بعضهما البعض أسفل المصفوفة أي في عناوين أسفل عنوان المصفوفة، وعندما تقرأ قيمةً خارج المصفوفة فكأنك تريد الحصول على قيمٍ متروكة من استدعاء دالة سابقة في المكدس. كل المتغيرات في المثال السابق أعدادٌ صحيحة integers لذلك سيكون سهلًا معرفة ما يحدث إلى حدٍ ما، ولكن يمكن أن يكون للقيم التي تقرؤها من خارج حدود المصفوفة أيّ نوع. إذا غيّرت في الدالة f1 بحيث تستخدم مصفوفة أعداد عشرية array of floats فالنتيجة هي:

```
17
123
1120141312
1120272384
```

آخر قيمتين من الخرج السابق هما ما تحصل عليه عندما تفسّر قيمة عشرية كعدد صحيح، وإذا صادفت هذا الخرج خلال عملية تنقيح الأخطاء debugging سيكون تفسير ما يحصل صعبًا جدًا.

5.5 تمثيل السلاسل strings

سلاسل لغة البرمجة C هي سلاسلٌ منتهية بالقيمة الخالية null-terminated، لذلك لا تنسَ البايت الإضافي في نهاية السلسلة وذلك عند تخصيص مكان لهذه السلسلة.

تُرمّز الحروف والأرقام في سلاسل C بواسطة ترميز ASCII (انظر الجدول كاملًا)، فترميز ASCII للأرقام من 0 إلى 9 هو من 48 إلى 57 وليس ترميزها من 0 إلى 9، فالرمز الآسكي 0 يمثل الحرف الخالي NUL الذي يحدد نهاية السلسلة، أما الرموز الآسكية من 1 إلى 9 فهي محارف خاصة تُستخدم في بعض بروتوكولات الاتصالات، والرمز الآسكي 7 هو جرس bell فينتج عن طباعته إصدارُ صوت في بعض الطرفيات. 65 هو الرمز الآسكي للحرف A وللحرف a هو 97 والتي تُكتب ثنائيًا كما يلي :

```
65 = b0100 0001
97 = b0110 0001
```

حيث ستلاحظ أنهما مختلفان فقط ببتٍ واحد إذا تمعّنت النظر قليلًا، ويُستخدم هذا النمط أيضًا لبقية الحروف، فيتصرّف البت السادس إذا ابتدأت العد من اليمين كبت حالة الحرف case bit فهو 0 للحروف الكبيرة و 1 للحروف الصغيرة.

جرّب كتابة دالة تحوّل الحرف الصغير إلى حرفٍ كبير وذلك من خلال قلب flipping البت السادس فقط، ويمكنك أيضًا صنع نسخة أسرع من الدالة وذلك من خلال قراءة سلسلة مكونة من 32 بتًا أو 64 بتًا وهذا أفضل من قراءة حرفٍ واحد فقط في كل مرة، حيث ينشَأ هذا التحسين بسهولة أكثر إذا كان طول السلسلة من مضاعفات 4 أو 8 بايتات.

إذا قرأت قيمةً خارج حدود المصفوفة ستظهر لك محارف غريبة، ولكن إذا كتبت سلسلةً ثم قرأتها كعدد صحيح int أو عشري float فسيكون تفسير interpret النتيجة صعبًا. وإذا شغلّت البرنامج التالي:

```c
char array[] = "allen";
float *p = array;
printf("%f\n", *p);
```

ستجد أن التمثيل الآسكي لأول 8 حروف من اسمي، يقول الكاتب، التي فُسّرت كعدد عشري مضاعف الدقة double-precision floating point number هو 6977971387880058545766 4.

6. إدارة الذاكرة

توفّر لغة البرمجة C أربع دوال تخصيصٍ ديناميكي للذاكرة هي:

- `malloc`: التي تأخذ وسيطًا نوعه عدد صحيح ويمثّل حجمًا بالبايتات وتعيد مؤشرًا إلى قطعة ذاكرةٍ مخصصة حديثًا حجمها يساوي الحجم المعطى على الأقل، وإذا لم تستوفِ الحجم المطلوب فإنها تعيد قيمة مؤشرٍ خاص هو NULL.
- `calloc`: وهي شبيهة بالدالة `malloc` باستثناء أنها تصفّر قطعة الذاكرة المخصصة حديثًا أيضًا أي أنها تضبط كل قيم بايتات القطعة بالقيمة 0.
- `free`: التي تأخذ وسيطًا هو مؤشر إلى قطعة ذاكرةٍ مخصصة سابقًا وتلغي تخصيصها deallocated أي تجعل حيز الذاكرة المشغول سابقًا متوفرًا لأي تخصيصٍ مستقبلي.
- `realloc`: والتي تأخذ وسيطين هما مؤشرٌ لقطعة ذاكرة مخصصة سابقًا وحجمٌ جديد، أي تخصص قطعة ذاكرة بحجمٍ جديد وتنسخ بيانات القطعة القديمة إلى القطعة الجديدة وتحرّر قطعة الذاكرة القديمة ثم تعيد مؤشرًا إلى قطعة الذاكرة الجديدة.

واجهة برمجة التطبيقات API لإدارة الذاكرة معرضةٌ للخطأ error-prone ولكنها غير متسامحة مع الخطأ في نفس الوقت، فإدارة الذاكرة هي أحد أهم التحديات التي تواجه تصميم أنظمة البرمجيات الكبيرة، وهي أحد أهم الأسباب التي تجعل لغات البرمجة الحديثة توفّر خصائصًا عالية المستوى لإدارة الذاكرة مثل خاصية كنس المهملات garbage collection.

6.1 أخطاء الذاكرة Memory errors

تشبه واجهة برمجة التطبيقات لإدارة الذاكرة في لغة البرمجة C إلى حدٍ مـا Jasper Beardly وهـو شخصـية ثانوية في برنامج الرسوم المتحركة التلفزيوني سيمبسون The Simpsons الذي ظهر في بعض الحلقـات كمعلّمٍ بديل حازمٍ حيث فرض عقوبةً جسدية لكل المخالفات أسماها paddlin. هناك بعض الأمور التي يحاول البرنـامج تنفيذها ليستحق بمحاولته تلك هذه العقوبة paddling، أي بمعنًى آخر إنها أمورٌ ممنوعة وهي:

- محاولة الوصول لقطعة ذاكرة لم تُخصّص بعد سواءً للقراءة أو للكتابة.
- محاولة الوصول إلى قطعة ذاكرة مخصَّصة محررةٌ مسبقًا.
- محاولة تحرير قطعة ذاكرة لم تُخصّص بعد.
- محاولة تحرير قطعة ذاكرة أكثر من مرة.
- استدعاء الدالة `realloc` مع قطعة ذاكرة لم تُخصّص بعد أو خُصصت ثم حُرّرت.

يمكن أن تجد أن اتباع القواعد السابقة ليس أمرًا صعبًا، ولكن يمكن أن تُخصّـص قطعـة ذاكـرة في جـزء من برنامجٍ كبير وتُستخدم في أجزاء أخـرى وتُحـرّر في جـزءٍ آخـر من البرنـامج، حيث يتطلب التغيـير في أحـد الأجـزاء تغييرًا في الأجزاء الأخرى أيضًا. ويمكن أن يوجد أيضًا العديد من الأسماء البديلة aliases أو المراجـع references التي تشير إلى نفس قطعة الذاكرة المخصصة في أجزاء مختلفة من البرنامج، لـذلك يجب ألّا تُحـرر تلـك القطعـة حتى تصبح كل المراجع التي تشير إليها غير مستخدَمة. يتطلب تحقيق ذلك تحليلًا لكـل أجـزاء البرنـامج بعنايـة، وهو أمرٌ صعب ومخالفٌ لمبادئ هندسة البرمجيات الأساسية.

يجب أن تتضمن كل الدوال التي تخصص الذاكرة معلوماتٍ عن كيفية تحرير تلك الذاكرة كجـزءٍ من الواجهـة الموثّقة documented interface في الحالة المثالية، حيث تقوم المكتبات الناضجة Mature libraries بـذلك جيدًا ولكن لا ترقى ممارسة هندسـة البرمجيـات الواقعيـة إلى تلـك المثاليـة. يمكن أن يكـون العثـور على أخطـاء الذاكرة صعبًا لأن أعراض تلك الأخطاء غير متنبأٍ بها مما يزيد الطين بلةً فمثلًا:

- إذا قرأت قيمةً من قطعة ذاكرةٍ غير مخصصة فقد يكتشف نظـام التشـغيل الخطـأ ثم ينبّـه trigger عن خطأ وقتٍ تشـغيلي والـذي يـدعى خطـأ تجزئـة segmentation fault ثم يوقـف البرنـامج، أو قـد يقـرأ البرنامج تلك القطعة غير المخصصة دون اكتشاف الخطأ، وفي هذه الحالة ستُخزّن القيمـة الـتي حصـل عليها البرنامج مهما كانت في موقع الذاكرة الذي وصل إليه، ولا يمكن التنبؤ بهـذا الموقـع لأنـه سـيتغير في كل مرة يُشغّل بها البرنامج.
- أما إذا كتبت قيمةً في قطعة ذاكرة غير مخصصة ولم تحصـل على خطـأ تجزئـة فسـتكون الأمـور أسـوأ، وسيمر وقتٌ طويل قبل أن تُقرأ تلك القيمة التي كتبتها في موقع غـير صـالح لعمليـة أخـرى أو جـزء مـا مسببةً مشاكل، وبالتالي سيكون إيجاد مصدر المشكلة صعبًا جدًا.

ويمكن أن تصبح الأمور أسوأ من ذلك أيضًا، فأحد أكثر مشاكل أسلوب C لإدارة الذاكرة شيوعًا هو أن بنى البيانات المستخدمة لتنفيذ الدالتين `malloc` و `free` تُخزّن مع قطع الذاكرة المخصصة غالبًا، لذلك إذا كتبت خارج نهاية قطعة الذاكرة المخصصة ديناميكيًا عن طريق الخطأ فهذا يعني أنك شوّهت mangle بنى البيانات تلك. ولن يكتشف النظام المشكلة حتى وقت متأخر وذلك عندما تستدعى الدالة `malloc` أو الدالة `free` وبالتالي تفشل هاتان الدالتان بطريقة مبهمة.

هناك استنتاجٌ يجب أن تستخلصه من ذلك وهو أن الإدارة الآمنة للذاكرة تتطلب تصميمًا وانضباطًا أيضًا، فإذا كتبت مكتبةً library أو نموذجًا module يخصّص ذاكرةً فيجب أن توفّر واجهةً interface لتحريرها، وينبغي أن تكون إدارة الذاكرة جزءًا من تصميم واجهة برمجة التطبيقات API منذ البداية.

إذا استخدمت مكتبةً تخصص ذاكرةً فيجب أن تكون منضبطًا في استخدامك لواجهة برمجة التطبيقات API، وإذا وفّرت المكتبة دوالًا لتخصيص وإلغاء تخصيص التخزين فيجب أن تستخدم تلك الدوال وألّا تستدعي الدالتين `free` و `malloc` لتحرير قطعة ذاكرة وتخصيصها على سبيل المثال، وينبغي أن تتجنب الاحتفاظ بمراجع متعددة تشير للقطعة ذاتها في أجزاء مختلفة من البرنامج.

توجد مقايضة trade-off بين الإدارة الآمنة للذاكرة والأداء أي لا يمكننا الحصول على الاثنين معًا بصورة تامة فمثلًا مصدر أخطاء الذاكرة الأكثر شيوعًا هو الكتابة خارج حدود مصفوفة، ويُستخدم التحقق من الحدود bounds checking لتلافي هذه المشكلة أي يجب التحقق فيما إذا كان الدليل index موجودًا خارج حدود المصفوفة في كل وصولٍ إلى تلك المصفوفة. تُجري المكتباتُ عالية المستوى High-level libraries والتي توفّر المصفوفات الشبيهة بالبنى structures تحققًا من الحدود على المصفوفات، ولكن لا تجري لغة البرمجة C ومعظم المكتبات منخفضة المستوى low-level libraries ذلك التحقق.

6.2 تسريب الذاكرة Memory leaks

يوجد خطأ ذاكرةٍ يمكن أن يستحق عقوبة ويمكن ألّا يستحقها وهو تخصيص قطعة ذاكرةٍ ثم عدم تحريرها نهائيًا وهذا ما يدعى بتسريب الذاكرة memory leak.

تسريب الذاكرة في بعض البرامج أمرٌ عادي فإذا خصص برنامجك ذاكرةً وأجرى حساباتٍ معينة عليها ثم غادر الذاكرة المخصصة، فمن الممكن أن يكون تحرير تلك الذاكرة المخصصة غير ضروري، حيث يلغي نظام التشغيل تخصيص ذاكرة برنامجٍ ما عند مغادرة هذا البرنامج من الذاكرة المخصصة له. وقد يؤدي تحرير الذاكرة مباشرةً أي قبل مغادرة البرنامج لذاكرته إلى الشعور بأن كل الأمور تحت السيطرة ولكنه مضيعةٌ للوقت على الأغلب. ولكن إذا اشتغل البرنامج لوقت طويل وسرّب ذاكرةً فإن مجمل ذاكرته المستخدمة ستزيد بصورةٍ غير محددة، وبالتالي قد تحدث مجموعة من الأمور هي:

- قد تَنفَد ذاكرة نظام التشغيل الحقيقية physical memory وبالتالي سيفشل استدعاء الدالة `malloc` التالي في أنظمة التشغيل التي لا تملك ذاكرة وهمية virtual memory، ثم تعيد الدالة القيمة NULL.

- بينما تستطيع أنظمة التشغيل التي تملك ذاكرةً وهمية نقلَ صفحات عملية أخرى من الذاكرة إلى القرص الصلب لتخصص حيّز ذاكرة أكبر للعملية المسرّبة.
- من الممكن أن يوجد حدٌّ لكمية الذاكرة التي تستطيع عمليةٌ ما تخصيصها، وبالتالي تعيد الدالة `malloc` القيمة NULL عند تجاوز هذا الحد.
- وقد تملأ عمليةٌ ما حيز العنونة الوهمية الخاص بها أي لا توجد عناوين أخرى لتخصيصها، وبالتالي تعيد الدالة `malloc` القيمة NULL أيضًا.

إذا أعادت الدالة `malloc` القيمة NULL ولكنك استمريت في تنفيذ البرنامج وحاولت الوصول إلى قطعة الذاكرة التي اعتقدت أنك خصصتها فستحصل على خطأ تجزئة segmentation fault، لذلك من الأفضل أن تتحقق من نتيجة تنفيذ الدالة `malloc` قبل استخدامها. أحد الخيارات هو أن تضيف شرطًا condition بعد كل استدعاء للدالة `malloc` كما يلي:

```
void *p = malloc(size);
if (p == NULL) {
perror("malloc failed");
exit(-1);
}
```

يُصرّح عن الدالة `perror` في ملف الترويسات `stdio.h` ومهمتها طباعة رسالة خطأ ومعلومات إضافية أيضًا عن آخر خطأ قد ظهر. أما الدالة `exit` فيصرّح عنها في ملف الترويسات `stdlib.h` والتي تسبب إنهاء العملية، ويدعى وسيط الدالة برمز الحالة status code الذي يحدد طريقة إنهاء العملية، حيث يحدد رمز الحالة 0 أنه إنهاءٌ عادي أما رمز الحالة 1- يدل على وجود خطأ في الشرط، ويوجد رموز حالة أخرى تدل على أنواع أخرى من الأخطاء الموجودة في الشرط.

الشيفرة المستخدمة للتحقق من الأخطاء Error-checking code مزعجةٌ وتجعل البرنامج صعب القراءة ولكن يمكنك التخفيف من ذلك من خلال استدعاء دوال المكتبة المغلّفة wrapping library function وشيفرات التحقق من الأخطاء الخاصة بها في دوالك الخاصة. ستجد مغلّف الدالة `malloc` الذي يتحقق من القيمة المعادة كما يلي:

```
void *check_malloc(int size)
{
    void *p = malloc(size);
    if (p == NULL)
    {
        perror("malloc failed");
```

```
            exit(-1);
        }
        return p;
    }
```

تسرّب معظمُ البرامج الكبيرة مثل متصفحات الويب الذاكرةَ وذلك لأن إدارة الذاكرة أمر صعبٌ جدًا، ويمكنك استخدام أداتي UNIX وهما `ps` و `top` لمعرفة البرامج التي تستخدم أكبر قدرٍ من الذاكرة على نظامك.

ننصحك بقراءة مقال "إدارة العمليات (Process) في لينكس باستخدام الطرفية" ومقال "مبادئ إدارة العمليات (Processes) على RedHat Enterprise Linux" لمزيد من التفاصيل عن عمليات لينكس.

6.3 التطبيق Implementation

يخصص نظام التشغيل حيّزًا لجزء نص البرنامج text segment وللبيانات المخصصة بصورة ساكنة data statically allocated وحيزًا آخر لجزء المكدس stack وحيزًا أيضًا للكومة heap والذي يتضمن البيانات المخصصة ديناميكيًا dynamically allocated data، وذلك عند بدء تشغيل عمليةٍ ما.

لا تخصص جميعُ البرامج البياناتِ الديناميكية لذلك يمكن أن يكون الحجم الابتدائي للكومة صغيرًا أو صفرًا، حيث تتضمن الكومة قطعةً واحدة حرّة فقط مبدئيًا. تتحقق الدالة `malloc` عند استدعائها فيما إذا كان هناك قطعةُ ذاكرةٍ حرة وكبيرة كفاية لها، فإذا لم تجد طلبها فإنها تطلب مزيدًا من الذاكرة من نظام التشغيل، حيث تُستخدم الدالة `sbrk` لهذا الغرض، وتضبط الدالة `sbrk` نهاية البرنامج break program الذي يُعَد مؤشرًا إلى نهاية الكومة.

يخصص نظامُ التشغيل صفحاتٍ جديدة من الذاكرة الحقيقية عند استدعاء الدالة `sbrk` ثم يحدّث جدول صفحات العملية ويضبط نهاية البرنامج، ويستطيع البرنامج استدعاء الدالة `sbrk` مباشرةً دون استخدام الدالة `malloc` وإدارة الكومة بنفسه، ولكن استخدام الدالة `malloc` أسهل كما أنها سريعة التنفيذ وتستخدم الذاكرة بكفاءة في معظم نماذج استخدام الذاكرة، فتستخدم معظم أنظمة تشغيل Linux الدالة `ptmalloc` لتطبيق واجهة برمجة التطبيقات لإدارة الذاكرة وهذه الواجهة هي الدوال `malloc` و `free` و `calloc` و `realloc`، حيث أن الدالة `ptmalloc` التي كتبها Doug Lea مرتكزةٌ على الدالة `dlmalloc`.

يتوفر بحثٌ قصير يشرح العناصر الأساسية للتنفيذ implementation، ولكن يجب أن يكون المبرمجون على دراية بالعناصر المهمة التالية:

- لا يعتمد الوقت التشغيلي للدالة `malloc` على حجم قطعة الذاكرة ولكنه يعتمد على عدد قطع الذاكرة الحرّة الموجودة. الدالة `free` سريعة عادةً بغض النظر عن عدد القطع الحرّة. يعتمد وقت التشغيل على حجم القطعة وعلى عدد القطع الحرّة لأن الدالة `calloc` تجعل جميع قيم بايتات القطعة أصفارًا. الدالة

`realloc` سريعة إذا كان الحجم الجديد أصغر من الحجم الحالي أو إذا كان حيّز الذاكرة متـوفرًا من أجـل توسيع قطعة الذاكرة الحالية، وإذا لم يتحقق ذلك فيجب على الدالة `realloc` نسخ البيانات من قطعة الذاكرة القديمة إلى قطعة الذاكرة الجديدة وبالتالي يعتمد وقت التشغيل في هذه الحالة على حجم قطعة الذاكرة القديمة.

- علامات الحدود Boundary tags: تضيف الدالة `malloc` حيّزًا في بداية ونهاية القطعة عند تخصـيص هذه القطعة وذلك لتخزين معلومات عن القطعة التي تتضمن حجم القطعـة وحالتهـا مخصصـة أو حـرّة وتدعى هذه المعلومات بعلامات الحدود Boundary tags، حيث تسـتطيع الدالـة `malloc` باسـتخدام هـذه العلامـات الانتقـال من أيـة قطعـة ذاكـرة إلى القطعـة السـابقة وإلى القطعـة التاليـة من الـذاكرة، بالإضـافة إلى أن قطـع الـذاكرة الحـرّة تكـون موصـولة ببعضـها بعضًـا ضـمن لائحـة مترابطة مضاعفة doubly-linked list حيث تتضمن كل قطعـة ذاكـرة حـرّة مؤشـرًا إلى القطعـة الـتي تسـبقها ومؤشرًا إلى القطعة التي تليها ضمن لائحة قطع الذاكرة الحرّة. تشكّل علامات الحـدود ومؤشـرات لائحـة القطع الحرة بنى البيانات الداخلية للدالة `malloc`، وتكون بنى البيانات هذه مبعثرةً مع بيانات البرنـامج لذلك يكون من السهل أن يتلفها خطأ برنامجٍ ما.

- كلفة حيز الذاكرة Space overhead: تشغَل علامـات الحـدود ومؤشـرات لائحـة القطـع الحـرّة حـيزًا من الذاكرة، فالحد الأدنى لحجم قطعة الذاكرة هو 16 بايتًا في معظم أنظمة التشـغيل، لـذلك ليسـت الدالـة `malloc` فعّالةً من حيث حيزالذاكرة بالنسبة لقطع الذاكرة الصـغيرة جـدًا، فـإذا تتطلب برنامجـك عـددًا كبيرًا من بنى البيانات الصغيرة فيكون تخصيصهم ضمن مصفوفات فعالًا أكثر.

- التجزئة Fragmentation: إذا خصصت وحررت قطع ذاكرة بأحجام مختلفة فإن الكومة تميل لأن تصـبح مجزّأة، وبالتالي يصبح حيز الذاكرة الحر مجزّأ إلى العديد من الأجزاء الصغيرة. تضيّع التجزئـة حـيز الـذاكرة وتبطّئ البرنامج أيضًا من خلال جعل الذواكر المخبئية أقل فعالية.

- التصنيف والتخبئة Binning and caching: تُخزّن لائحة القطع الحرة ضمن صناديق bins بحيث تكون مرتبة حسـب الحجم، حيث تعـرف الدالـة `malloc` في أي صـندوقٍ تبحث عنـدما تريـد الحصـول على قطعةٍ ذات حجم معين. وإذا حررت قطعةً ما ثم خصصت قطعة أخـرى بنفس الحجم مباشـرةً فسـتكون الدالة `malloc` أسرع عادةً.

7. فهم عملية التخبئة caching

7.1 كيف يُنفذ البرنامج؟

يجب أن تفهم كيف تنفّذ الحواسيب البرامج لفهم عملية التخبئة caching، وينبغي عليك دراسة معمارية الحاسوب لفهم التخبئة بصورةٍ أعمق.

تكون الشيفرة أو النص البرمجي ضمن القرص الصلب hard disk أو ضمن SSD عند بدء تشغيل البرنامج، وينشئ نظام التشغيل عمليةً جديدة لتشغيل البرنامج ثم ينسخ المحمِّل loader نص البرنامج من التخزين الدائم إلى الذاكرة الرئيسية ثم يبدأ البرنامج من خلال استدعاء الدالة `main`. تُخزَّن معظم بيانات البرنامج في الذاكرة الرئيسية أثناء تنفيذه، ولكن تكون بعض هذه البيانات موجودةً في مسجّلات registers والتي هي وحدات صغيرة من الذاكرة موجودة في المعالج CPU وتتضمن هذه المسجلات ما يلي:

- عداد البرنامج program counter واختصاره PC الذي يتضمن عنوان التعليمة التالية من البرنامج العنوان في الذاكرة.
- مسجِّل التعليمة instruction register ويختصر إلى IR ويتضمن شيفرة الآلة للتعليمة التي تنفّذ حاليًا.
- مؤشر المكدس stack pointer واختصاره SP الذي يتضمن عنوان إطار المكدس stack frame للدالة الحالية ويحتوي إطار المكدس معاملات الدالة ومتغيراتها المحلية.
- مسجلات ذات أغراض عامة General-purpose registers التي تحتفظ بالبيانات التي يعمل عليها البرنامج حاليًا.

- مسجل الحالة status register أو مسجل الراية flag register الذي يتضمن معلومات عن العمليات الحسابية الحالية، حيث يتضمن مسجل الراية عادةً بتًا، ويُضبَط هذا البت إذا كانت نتيجة العملية السابقة صفرًا على سبيل المثال.

ينفّذ المعالج الخطوات التالية عند تشغيل البرنامج وتدعى هذه الخطوات بدورة التعليمة instruction cycle:

- الجلب Fetch: تُجلَب التعليمة التالية من الذاكرة ثم تُخزَّن في مسجّل التعليمة.
- فك التشفير Decode: يَفك جزءُ المعالج الذي يدعى وحدة التحكم control unit تشفيرَ التعليمة ثم يرسل إشاراتٍ إلى الأجزاء الأخرى من المعالج.
- التنفيذ Execute: تسبب إشارات وحدة التحكم ظهور العمليات الحسابية المناسبة.

تستطيع معظم الحواسيب تنفيذ بضع مئات من التعليمات المختلفة تدعى بمجموعة التعليمات instruction set ولكن تندرج معظم التعليمات ضمن فئات عامة هي:

- تعليمات التحميل Load: تنقل قيمةً من الذاكرة إلى المسجل.
- التعليمات الحسابية أو المنطقية Arithmetic/logic: تحمّل المعامَلات operands من المسجّلات ثم تجري عمليات رياضية ثم تخزّن النتيجة في مسجّل.
- تعليمات التخزين Store: تنقل قيمةً من المسجّل إلى الذاكرة.
- تعليمات القفز وتعليمات الفرع Jump/branch: تسبب تغييراتُ عداد البرنامج قفزَ تدفق التنفيذ إلى موقعٍ آخر من البرنامج. تكون الفروع مشروطة عادةً وهذا يعني أن الفروع تتحقق من رايةٍ ما في مسجل الراية ثم تقفز إلى موقع آخر من البرنامج في حال ضُبطت هذه الراية فقط.

توفر بعضُ مجموعات التعليمات الموجودة ضمن معمارية نظام التشغيل واسعة الانتشار x86 تعليماتٍ تجمع بين عمليةٍ حسابيةٍ وعملية تحميل. تُقرَأ تعليمةٌ واحدةٌ من نص البرنامج خلال كل دورة تعليمة، ويُحمَّل حوالي نصف تعليمات البرنامج أو تخزن بياناتها، وتكمن هنا واحدة من المشاكل الأساسية في معمارية الحاسوب وهي مشكلة عنق زجاجة الذاكرة أو اختناق الذاكرة memory bottleneck.

نواة الحواسيب الحالية قادرةٌ على تنفيذ تعليمة في أقل من 1 نانو ثانية، ولكن يُقدّر الوقت اللازم لنقل بيانات من وإلى الذاكرة بحوالي 100 نانو ثانية، وإذا توجّب على المعالج الانتظار 100 نانو ثانية لجلب التعليمة التالية و100 نانو ثانية أخرى لتحميل البيانات فسيكمل المعالج التعليمات بصورة أبطأ ب 200 مرة مما هو متوقع، لذلك تُعَد الذاكرة في العديد من العمليات الحسابية هي عامل تحديد السرعة ولا يُعَد المعالج كذلك.

7.2 أداء الذاكرة المخبئية Cache performance

الـذاكرة المخبئيـة هي الحـل لمشـكلة اختنـاق الـذاكرة أو على الأقـل حـلٌ جـزئي لهـا، حيث أن الـذاكرة المخبئية ذاكرةٌ صغيرة الحجم وسريعة ومتواجدة قرب المعالج على نفس الشريحة عادةً.

تحـوي الحواسـيب الحديثـة مسـتويات متعـددة من الـذاكرة المخبئيـة هي: ذاكـرة مخبئيـة ذات مسـتوى أول Level 1 cache وهي الأصغر حجمًا والأسرع حيث يتراوح حجمها بين 1 و 2 ميبي بايت مع وقت وصول 1 نانو ثانية تقريبًا، أما الذاكرة المخبئية ذات المستوى الثاني Level 2 cache التي تملك وقت وصـول يسـاوي 4 نانو ثانية تقريبًا، وتملك الذاكرة المخبئية ذات المستوى الثالث وقت وصول يساوي 16 نانو ثانية.

يخزّن المعالج نسخةً من القيمة التي يحمّلها من الذاكرة في الذاكرة المخبئيـة، وإذا حُمّلت تلـك القيمـة مـرةً أخرى يجلب المعالج نسخة هذه القيمة من الذاكرة المخبئية وبالتالي لا يضطر المعالج إلى الانتظار لجلب القيمـة من الذاكرة، ولكن من الممكن أن تمتلئ الذاكرة المخبئية وبالتالي يجب إخـراج بعض القيم من الـذاكرة المخبئيـة عند إحضار قيم أخرى، لذلك إذا حمّل المعالج قيمةً ثم عـاد لتحميلهـا مـرةً أخـرى ولكن بعـد وقت طويـل فقـد لا تكون هذه القيمة موجودةً ضمن الذاكرة المخبئية.

إن أداء العديد من البرامج محدودٌ بمقدار فعالية الـذاكرة المخبئيـة، فـإذا كـانت التعليمـات والبيانـات الـتي يحتاج إليها المعالج موجودةً في الذاكرة المخبئية فإن البرنـامج يمكن أن ينفّـذ بسـرعةٍ قريبـة من سـرعة المعـالج الكاملة، بينمـا إذا احتـاج المعـالج بيانـاتٍ غـير موجـودةٍ في الـذاكرة المخبئيـة مـرارًا فسـيكون المعـالج محـدودًا بسرعة الذاكرة.

معدّل الإصابة hit rate للذاكرة المخبئية الذي يرمز له h هو جزء عمليات الوصول للذاكرة التي تجد البيانات في الذاكرة المخبئية، أما معدل الإخفاق miss rate والذي يرمز له m هو جزء عمليات الوصول للذاكرة التي يجب أن تذهب إلى الذاكرة لأنها لم تجد البيانات التي تريدها ضمن الـذاكرة المخبئيـة، فـإذا كـان وقت إصـابة الـذاكرة المخبئية هـو Th ووقت إخفـاق الـذاكرة المخبئيـة هـو Tm فـإن متوسـط وقت كـل عمليـة وصـولٍ للـذاكرة هـو: $h.T_h+m.T_m$ ، ويمكن تعريف عقوبة الإخفاق miss penalty كوقتٍ إضافي لمعالجة إخفاق الـذاكرة المخبئيـة والذي يساوي: $T_p=T_m-T_h$ وبالتالي متوسط وقت الوصول هو: $T_h+m.T_p$ يساوي متوسط وقت الوصـول تقريبًا Th عندما يكـون معـدل الإخفـاق منخفضًـا، وبالتـالي يـؤدي البرنـامج عملـه وكـأن الـذاكرة تعمـل بسـرعة الذاكرة المخبئية.

7.3 المحلية Locality

تحمّل الذاكرة المخبئية عادةً كتلةً أو سطرًا من البيانات الذي يتضمن البايت المطلوب وبعضًـا من البايتـات المجاورة له وذلك عندما يقرأ البرنامج بايتًا للمرة الأولى، وبالتالي إذا حـاول البرنـامج قـراءة إحـدى تلـك البايتـات المجاورة للبايت المطلوب لاحقًا فستكون موجودةً في الذاكرة المخبئية مسبقًا.

افترض أن حجم الكتلة هو 64 بايتًا مثل قراءة سلسلة طولها 64 بايتًا، حيث يمكن أن يقع أول بايت من السلسلة في بداية الكتلة، فستتحمّل عقوبة إخفاق miss penalty إذا حمّلت أول بايت ولكن ستوجد بقية السلسلة في الذاكرة المخبئية بعد ذلك، أي يساوي معدل الإصابة 63/64 بعد قراءة كامل السلسلة أي ما يعادل 98%، أما إذا كان حجم السلسلة يعادل كتلتين فستتحمّل عقوبتي إخفاق ويكون معدل الإصابة مساويًا 62/64 أو 97%، ويصبح معدل الإصابة 100% إذا قرأت السلسلة ذاتها مرة أخرى، ولكن سيصبح أداء الذاكرة المخبئية سيئًا إذا قفز البرنامج بصورة غير متوقعة محاولًا قراءة بيانات من مواقع متفرقة في الذاكرة أو إذا كان الوصول إلى نفس الموقع مرتين نادرًا.

يدعى ميل البرنامج لاستخدام البيانات ذاتها أكثر من مرة بالمحلية الزمانية temporal locality، ويدعى ميل البرنامج لاستخدام البيانات المتواجدة في مواقع قريبة من بعضها بعضًا بالمحلية المكانية spatial locality، حيث تقدّم البرامج نوعي المحلية كما يلي:

- تحوي معظم البرامج كتلًا من الشيفرة بدون تعليمات قفزٍ أو فروع، حيث تنفّذ التعليمات في هذه الكتل تسلسليًا، وبالتالي يملك نموذج الوصول access pattern محليةً مكانية spatial locality.
- تنفّذ البرامج نفس التعليمات مرات متعددة في الحلقات التكرارية loop، وبالتالي يملك نموذج الوصول access pattern محليةً زمانية temporal locality.
- تُستخدم نتيجة تعليمة ما كمُعامَل operand للتعليمة التالية مباشرةً، وبالتالي يملك نموذج وصول البيانات محليةً زمانية temporal locality.
- تُخزَّن معامِلات دالةٍ ومتغيراتها المحلية معًا في جزء المكدس عندما ينفّذ البرنامج هذه الدالة، أي يملك الوصول إلى هذه القيم محليةً مكانية spatial locality.
- أحد أكثر نماذج المعالجة شيوعًا هو قراءة أو كتابة عناصر مصفوفةٍ تسلسليًا وبالتالي تملك هذه النماذج محليةً مكانية spatial locality.

7.4 قياس أداء الذاكرة المخبئية

أحد برامجي المفضلة، يقول الكاتب، هو البرنامج الذي يتكرر خلال مصفوفة ويقيس متوسط وقت قراءة وكتابة عنصر، ويمكن استنتاج حجم الذاكرة المخبئية وحجم الكتلة وبعض الخصائص الأخرى من خلال تغيير حجم تلك المصفوفة، والجزء الأهم من هذا البرنامج هو الحلقة التكرارية loop التالية:

```
iters = 0;
do
{
    sec0 = get_seconds();
```

```
    for (index = 0; index < limit; index += stride)
        array[index] = array[index] + 1;

    iters = iters + 1;
    sec = sec + (get_seconds() - sec0);

} while (sec < 0.1);
```

حيث تمر حلقة `for` الداخلية على عناصر المصفوفة، ويحدد المتغير `limit` مقدار قيم المصفوفة التي تريد عبورها، ويحدد المتغير `stride` الخطوة أو عدد العناصر التي يجب تجاوزها في كل تكرار، فمثلًا إذا كانت قيمة المتغير`limit` هي 16 وقيمة المتغير `stride` هي 4 فإن الحلقة ستصل إلى القيم التالية: 0 و4 و8 و12.

يتتبّع المتغير `sec` وقت المعالج الذي تستخدمه حلقة `for` الداخلية، وتنفّذ الحلقة الخارجية حتى يتخطى المتغير `sec` حاجز 0.1 ثانية والذي هو وقتٌ كافٍ لحساب الوقت الوسطي بدقةٍ كافية. تستخدم الدالة `get_seconds` استدعاء النظام `clock_gettime` وتحوّل الوقت إلى ثوانٍ ثم تعيد النتيجة كعدد عشري مضاعف الدقة `double`:

```
double get_seconds()
{
    struct timespec ts;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
    return ts.tv_sec + ts.tv_nsec / 1e9;
}
```

يشغّل البرنامج حلقة أخرى لعزل وقت الوصول لعناصر المصفوفة، وهذه الحلقة مطابقة للحلقة في المثال السابق باستثناء أن الحلقة الداخلية لا تمس المصفوفة وإنما تزيد نفس المتغير كما يلي:

```
iters2 = 0;
do
{
    sec0 = get_seconds();

    for (index = 0; index < limit; index += stride)
        temp = temp + index;
```

```
        iters2 = iters2 + 1;
        sec = sec - (get_seconds() - sec0);
    } while (iters2 < iters);
```

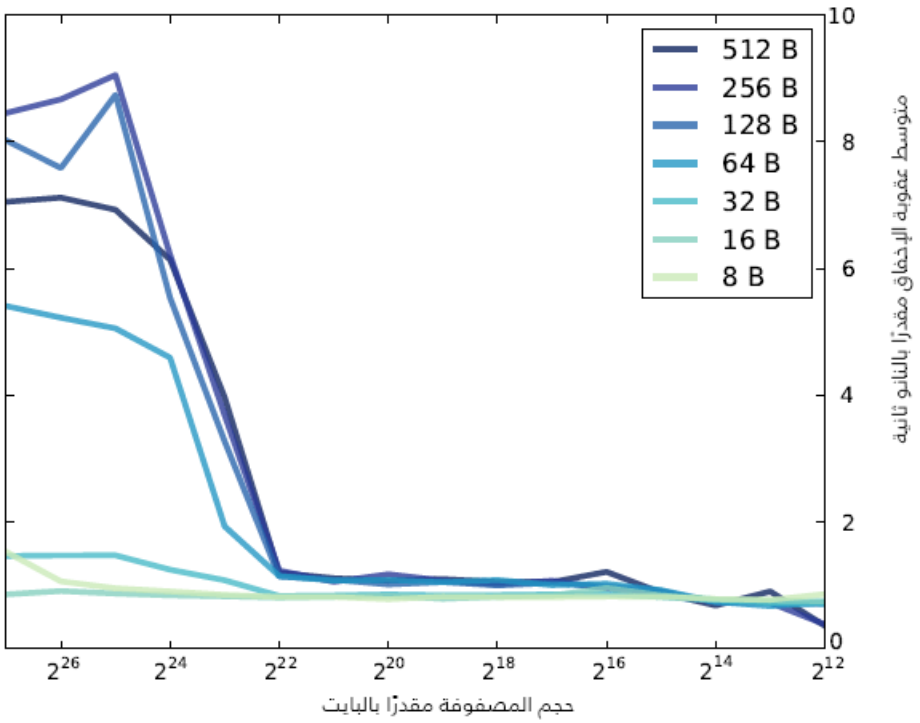
حيث تلاحظ أن الحلقة الثانية تنفّذ نفس عدد تكرارات الحلقة الأولى، وتطرح الوقت المستغرق بعد كل تكرار من المتغير `sec`، يتضمن المتغير `sec` عند اكتمال الحلقة الوقت الكلي لكل عمليات الوصول إلى المصفوفة مطروحًا منه الوقت الإجمالي لازدياد المتغير `temp`، وتتحمّل كل عمليات الوصول هذا الاختلاف الذي يمثّل عقوبة الإخفاق الكلية، ثم تُقسَم عقوبة الإخفاق الكلية على عدد عمليات الوصول للحصول على متوسط عقوبات الإخفاق في كل وصول مقدّرًا بالنانو ثانية كما يلي:

```
sec * 1e9 / iters / limit * stride
```

إذا صرّفت البرنامج السابق ثم شغّلته سيظهر الخرج التالي:

```
Size: 4096 Stride: 8 read+write: 0.8633 ns
Size: 4096 Stride: 16 read+write: 0.7023 ns
Size: 4096 Stride: 32 read+write: 0.7105 ns
Size: 4096 Stride: 64 read+write: 0.7058 ns
```

إذا كان لديك Python و `matplotlib` على جهازك فيمكنك استخدام `graph_data.py` لتحصل على رسمٍ بياني للنتائج كما في الشكل التالي الذي يظهر نتائج البرنامج عند تشغيله على Dell Optiplex 7010:



الشكل 7.1: متوسط عقوبة الإخفاق كدالة لحجم المصفوفة والخطوة

لاحظ أن حجم المصفوفة والخطوة stride مقدران بالبايت وليس بأعداد عناصر المصفوفة. إذا تمعنت النظر في الرسم البياني السابق سيمكنك استنتاج بعض الأشياء عن الذاكرة المخبئية والتي هي:

- يقرأ البرنامج من المصفوفة مراتٍ متعددة لذلك يكون لديه كثيرٌ من المحلية الزمانية، فإذا كانت المصفوفة بكاملها في الذاكرة المخبئية فهذا يعني أن متوسط عقوبة الإخفاق miss penalty تساوي 0 تقريبًا.
- يمكنك قراءة كل عناصر المصفوفة إذا كانت قيمة الخطوة stride هي 4 بايتات أي يكون لدى البرنامج كثيرٌ من المحلية المكانية spatial locality، وإذا كان حجم الكتلة مناسبًا لتضمين 64 عنصرًا على سبيل المثال فإن معدل الإصابة hit rate يساوي 63/64 على الرغم من عدم وجود المصفوفة في الذاكرة المخبئية.
- إذا كانت الخطوة الواحدة مساويةً لحجم الكتلة أو أكبر منها فإن قيمة المحلية المكانية صفر عمليًا، لأنك تصل إلى عنصرٍ واحد فقط في كل مرة تقرأ فيها كتلةً، أي من المتوقع أن تشاهد الحد الأعلى من عقوبة الإخفاق miss penalty في هذه الحالة.

من المتوقع الحصول على أداء جيد للذاكرة المخبئية إذا كانت المصفوفة أصغر من حجم الذاكرة المخبئية أو إذا كانت الخطوة أصغر من حجم الكتلة، بينما يقل الأداء فقط إذا كانت المصفوفة أكبر من الذاكرة المخبئية والخطوة كبيرة. أداء الذاكرة المخبئية في الرسم البياني السابق جيد بالنسبة لجميع الخطوات طالما أن المصفوفة أصغر من 2^{22} بايتًا، وبالتالي يمكنك استنتاج أن حجم الذاكرة المخبئية يقارب 4 ميبي بايت ولكن اعتمادًا على المواصفات فإن حجمها هو 3 ميبي بايت.

يكون أداء الذاكرة المخبئية جيدًا مع قيم الخطوة 8 و16 و32 بايتًا، ولكن يبدأ الأداء بالانخفاض عندما تصبح قيمة الخطوة 64 بايتًا، ويصبح متوسط عقوبة الإخفاق 9 نانو ثانية تقريبًا عندما تصبح الخطوة أكبر، وبالتالي يمكنك استنتاج أن حجم الكتلة يقارب 128 بايتًا. تستخدم العديد من المعالجات الذواكر المخبئية ذات المستويات المتعددة (multi-level caches) والتي تتضمن ذواكر مخبئية صغيرة وسريعة وذواكر مخبئية كبيرة وبطيئة، ويُلاحَظ أن عقوبة الإخفاق تزيد قليلًا عندما يصبح حجم المصفوفة أكبر من 2^{14} بايتًا، لذلك من المحتمل أن يكون للمعالج ذاكرةٌ مخبئية حجمها 16 كيلو بايت وبوقت وصول أقل من 1 نانو ثانية.

7.5 البرمجة والذاكرة المخبئية

تُطبَّق عمليات تخبئة الذاكرة على مستوى العتاد لذلك لا يتوجّب على المبرمجين معرفة الكثير عنها ولكن إذا عرفت كيفية عمل الذواكر المخبئية فيمكّنك ذلك من كتابة برامجٍ تستخدم الذاكرة المخبئية بفعاليةٍ أكثر، وإذا عملت مع مصفوفة كبيرة على سبيل المثال فمن الممكن أن تمر على عناصرها بسرعةٍ مرة واحدة ثم إجراء عملياتٍ متعددة على كل عنصر من المصفوفة وذلك أفضل من المرور على المصفوفة مراتٍ متعددة، وإذا تعاملت مع مصفوفة ثنائية الأبعاد 2D array فيمكن أن تُخزّن هذه المصفوفة كمصفوفة من الصفوف rows،

وإذا مررت على عناصر المصفوفة فسيكون المرور عليها من خلال الصفوف row-wise أسرع وذلك مع خطوةٍ مساويةٍ لحجم العنصر وهذا أفضل من المرور عليها من خلال الأعمدة column-wise مع خطوة مساوية لطول الصف الواحد.

لا تقدّم بنى البيانات المترابطة Linked data structures محليةً مكانية وذلك لأنه ليس ضروريًا أن تكون عقد هذه البنى المترابطة متجاورةً في الذاكرة، ولكن إذا خصصت عدة عقد في نفس الوقت فستكون موجودة في مواقع مشتركة من الكومة، وإذا خصصت مصفوفة العقد كلها مرةً واحدةً فستكون في مواقع متجاورة حتمًا.

تملك الاستراتيجيات التعاودية Recursive strategies مثل خوارزمية الفرز بالدمج mergesort سلوك ذاكرةٍ مخبئية جيدًا لأنها تقسّم المصفوفات الكبيرة إلى أجزاء صغيرة ثم تعمل على هذه الأجزاء الصغيرة، وتُضبَط هذه الخوارزميات في بعض الأحيان للاستفادة من سلوك الذاكرة المخبئية.

يمكن تصميم خوارزمياتٍ في التطبيقات التي يكون الأداء فيها مهمًا جدًا بحيث تكون هذه الخوارزميات مضبوطة من حيث حجم الذاكرة المخبئية وحجم الكتلة وخصائص عتاد أخرى، ويدعى هذا النوع من من الخوارزميات بالخوارزميات ذات الإدراك حسب الذاكرة المخبئية cache-aware، ولكن العائق الأهم لهذا النوع من الخوارزميات هو أنها خاصة بالعتاد hardware-specific.

7.6 هرمية الذاكرة

لا بدّ أنه خطر على بالك السؤال التالي: لماذا لم تُصنّع ذاكرة مخبئية كبيرة وبالتالي تُلغى الذاكرة الرئيسية نهائيًا بما أن الذاكرة المخبئية أسرع بكثير من الذاكرة الرئيسية؟

وجواب هذا السؤال هو وجود سببين أساسيين هما: سبب متعلق بالإلكترونيات والآخر سبب اقتصادي، فإن الذواكر المخبئية سريعة لأنها صغيرة وقريبة من المعالج وهذا يقلل التأخير بسبب سعة وانتشار الإشارة، وبالتالي إذا صنعت ذاكرة مخبئية كبيرة فستكون بطيئة حتمًا، وتشغَل الذواكر المخبئية حيّزًا على شريحة المعالج وكلما كانت الشريحة أكبر فسيصبح سعرها أكبر. أما الذاكرة الرئيسية فهي ذاكرة عشوائية ديناميكية dynamic random-access memory واختصارها DRAM والتي تستخدم ترانزستورًا ومكثفًا واحدًا لكل بت، وبالتالي يمكن ضغط مزيد من الذاكرة في نفس الحيّز، ولكن هذا التطبيق يجعل الذاكرة أبطأ من الطريقة التي تُطبَّق فيها الذواكر المخبئية.

تُحزَم الذاكرة الرئيسية عادةً ضمن وحدة الذاكرة ثنائية الخط dual in-line memory module واختصارها DIMM والتي تتضمن 16 شريحة أو أكثر، فشرائح صغيرة متعددة أرخص من شريحة واحدة كبيرة الحجم.

وجود تقايض بين السرعة والحجم والكلفة هو السبب الأساسي لصنع الذواكر المخبئية، فإذا وجدت تقنية سريعة وكبيرة ورخيصة للذاكرة فلسنا بحاجة أي شيء آخر. يطبّق نفس المبدأ بالنسبة للتخزين الدائم كما في الذاكرة الرئيسية، فالأقراص من النوع SSD سريعة ولكنها أغلى من الأقراص الصلبة HDD وبالتالي يجب أن

تكون أصغر من أجل التوفير في الكلفة، ومحركات الأشرطة Tape drives أبطأ من الأقراص الصلبة ولكنها تخزّن كميات كبيرة من البيانات وبسعر رخيص نسبيًا.

يظهر الجدول التالي وقت الوصول والحجم والكلفة لكل منهذه التقنيات:

الجهاز Device	وقت الوصول Access time	الحجم Typical size	الكلفة Cost
المسجل Register	يقدر ب 0.5 نانو ثانية	256 بايت	؟
الذاكرة المخبئية Cache	يقدر ب 1 نانو ثانية	2 ميبي بايت	؟
الذاكرة العشوائية الديناميكية DRAM	يقدر ب 100 نانو ثانية	4 جيبي بايت	10 دولار / جيبي بايت
قرص التخزين ذو الحالة الثابتة SDD	يقدر ب 10 ميكرو ثانية	100 جيبي بايت	1 دولار / جيبي بايت
القرص الصلب HDD	يقدر ب 5 ميلي ثانية	500 جيبي بايت	0.25 دولار / جيبي بايت
محرك الأشرطة Tape	دقائق	من 1 إلى 2 تيبي بايت	0.02 دولار / جيبي بايت

يعتمد عدد وحجم المسجلات على تفاصيل معمارية الحواسيب حيث تملك الحواسيب الحالية 32 مسجلًا ذو أغراضٍ عامة general-purpose registers ويخزن كل مسجل منها كلمة word واحدة حيث تساوي الكلمة 32 بتًا أو 4 بايتات في حواسيب 32 بت وتساوي 64 بتًا أو 8 بايتات في حواسيب 64 بت، وبالتالي يتراوح الحجم الإجمالي لملف المسجلات بين 100 إلى 300 بايت.

يصعب تحديد كلفة المسجلات والذواكر المخبئية لأنه تُجمَل كلفتهم مع كلفة الشرائح الموجودين عليها، وبالتالي لا يرى المستهلك كلفتهم بطريقة مباشرة، وبالنسبة للأرقام الأخرى الموجودة في الجدول فقد ألقيت النظر، يقول الكاتب، على العتاد النموذجي الموجود في المتاجر الإلكترونية لعتاد الحاسوب، وربما ستكون هذه الأرقام غير مستعملة في الوقت الذي ستقرأ فيه هذا الفصل ولكنها ستعطيك فكرةً عن الفجوة التي كانت موجودة بين الأداء والكلفة في وقتٍ ما.

تشكل التقنيات الموجودة في الجدول السابق هرمية الذاكرة memory hierarchy -والذي يتضمن التخزين الدائم storage أيضًا- حيث يكون كل مستوى من هذه الهرمية أكبر وأبطأ من المستوى الذي فوقه، وبمعنىً آخر يمكن عدّ كل مستوى هو ذاكرة مخبئية للمستوى الذي تحته، فيمكنك عدّ الذاكرة الرئيسية كذاكرة مخبئية للبرامج والبيانات التي تُخزَّن على أقراص SSD و HHD بصورة دائمة، وإذا عملت مع مجموعات بيانات كبيرة جدًا ومخزنة على محرك أشرطة tape فيمكنك استخدام القرص الصلب كذاكرة مخبئية لمجموعة بيانات فرعية واحدة في كل مرة.

7.7 سياسة التخبئة Caching policy

تقدّم هرمية الذاكرة إطار عملٍ للتخبئة caching من خلال الإجابة على أربعة أسئلة أساسية عن التخبئة في كل مستوٍ من هذه الهرمية وهي:

- من ينقل البيانات إلى الأعلى والأسفل ضمن هذه الهرمية؟ يخصص المصرّفُ compiler المسجّلَ في قمة هذه الهرمية، ويكون العتاد الموجود على المعالج مسؤولًا عن الذاكرة المخبئية، ينقل المستخدمون البيانات من التخزين الدائم إلى الذاكرة ضمنيًا عندما ينفّذون برامجًا وعندما يفتحون ملفات، ولكن ينقل نظام التشغيل البيانات أيضًا ذهابًا وإيابًا بين الذاكرة والتخزين الدائم، وينقل مسؤولو نظام التشغيل البيانات بين القرص الصلب ومحرك الأشرطة بوضوح وليس ضمنيًا في أسفل هذه الهرمية.
- ماذا يُنقَل؟ تكون أحجام الكتل صغيرةً في قمة الهرمية وكبيرة في أسفلها، فحجم الكتلة في الذاكرة المخبئية هو 128 بايتًا ويقدّر حجم الصفحات في الذاكرة ب 4 كيبي بايت، ولكن عندما يقرأ نظام التشغيل ملفًا من القرص الصلب فهو بذلك يقرأ عشرات أو مئات الكتل في كل مرة.
- متى تُنقَل البيانات؟ تُنقَل البيانات إلى الذاكرة المخبئية عندما تُستخدَم للمرة الأولى في معظم أنواع الذواكر المخبئية، ولكن تستخدم العديد من الذواكر المخبئية ما يسمّى بالجلب المسبق prefetching والذي يعني تحميل البيانات قبل أن تُطلَب صراحةً. وقد رأيت مسبقًا نموذجًا من الجلب المسبق هو تحميل الكتلة بأكملها عندما يُطلب جزء منها فقط.
- أين تذهب البيانات في الذاكرة المخبئية؟ لا تستطيع جلب أي شيء آخر إلى الذاكرة المخبئية عندما تمتلئ إلا إذا أخرجت شيئًا ما منها، لذلك يجب أن تبقي البيانات التي ستُستخدم مرة أخرى قريبًا وتستبدل البيانات التي لن تُستخدم قريبًا.

تشكّل أجوبة الأسئلة الأربعة السابقة ما يدعى بسياسة الذاكرة المخبئية cache policy، فيجب أن تكون سياسات الذاكرة المخبئية بسيطة في قمة هرمية الذاكرة لأنها يجب أن تكون سريعة وتُطبّق ضمن العتاد، أما في أسفل الهرمية فيوجد وقتٌ أكثر لاتخاذ القرارات حيث تصنع السياسات المصمَّمة جيدًا اختلافًا كبيرًا.

معظم سياسات الذاكرة المخبئية قائمةٌ على المبدأ الذي يقول أن التاريخ يعيد نفسه، فإذا ملكت معلومات عن الماضي القريب فيمكنك استخدامها لتنبؤ المستقبل القريب أيضًا، أي إذا أُستخدمت كتلة بيانات مؤخرًا فيمكنك توقع أنها ستُستخدم مرة أخرى قريبًا، وبالتالي يقدّم هذا المبدأ سياسة بديلة تدعى الأقل استخدامًا مؤخرًا least recently used واختصارها LRU والتي تحذف كتلة بيانات من الذاكرة المخبئية التي لم تُستخدم مؤخرًا تعرف على خوارزميات الذاكرة المخبئية (انظر توثيق الخوارزميات في موسوعة حسوب).

7.8 تبديل الصفحات Paging

يستطيع نظام التشغيل الذي يملك ذاكرة وهمية أن ينقل الصفحات ذهابًا وإيابًا بين الذاكرة والتخزين الـدائم وتدعى هذه الآلية بتبديل الصفحات paging أو بالتبديل swapping أحيانًا، وتجري هذه الآلية كما يلي:

1. افترض أن العملية A تستدعي الدالة `malloc` لتخصيص قطعةٍ من الذاكرة، فإذا لم يوجـد حيّـز حـر في الكومة بنفس الحجم المطلوب فإن الدالة `malloc` تسـتدعي الدالـة `sbrk` من أجـل طلب المزيـد من الذاكرة من نظام التشغيل.
2. يضيف نظام التشغيل صفحةً إلى جدول الصفحات الخاص بالعملية A عند وجود صفحةٍ حرّة في الذاكرة الحقيقية منشئًا بذلك مجالًا جديدًا من العناوين الوهمية الصالحة.
3. يختار نظام الصفحات صفحةً ضحية victim page تابعة للعملية B وذلك عند عدم وجود صفحات حرة، ثم ينسخ محتوى هذه الصـفحة الضـحية من الـذاكرة إلى القـرص الصـلب ثم يعـدّل جـدول الصـفحات الخاص بالعملية B ليشير إلى أن هذه الصفحة بُدِّلت swapped out.
4. يمكن إعادة تخصيص صفحة العملية B للعمليـة A وذلـك بعـد نقـل بيانـات العمليـة B، حيث يجب أن تصفّر الصفحة قبل إعادة تخصيصها لمنع العملية A من قراءة بيانات العملية B.
5. عندها يستطيع استدعاء الدالة `sbrk` إرجاع نتيجة وهي إعطاء الدالة `malloc` حيّزًا إضافيًا في الكومـة، ثم تخصص الدالة`malloc` قطعة الذاكرة المطلوبة وتستأنف العملية A عملها.
6. قـد يسـمح مجـدول نظـام التشـغيل scheduler اسـتئناف العمليـة B عنـد اكتمـال العمليـة A أو عنـد مقاطعتها. تلاحظ وحدة إدارة الذاكرة أن الصفحة التي بُدّلت والتي تحاول العملية B الوصـول إليهـا غـير صالحة invalid ثم تسبب حدوث مقاطعة.
7. يرى نظام التشغيل أن الصفحة بُدّلت عند استلامه للمقاطعة فيقوم بنقل الصفحة من القـرص الصـلب إلى الذاكرة.
8. ثم تستطيع العملية B استئناف عملها عندما تُبدَّل الصفحة.

يحسّن تبديل الصفحات من استخدامية utilization الذاكرة الحقيقية كثيرًا عندما يعمل جيدًا وبذلك يسـمح لعمليات أكثر أن تُشغَّل في حيّز أصغر والسبب هو:

- لا تستخدم معظم العمليات كامل ذاكرتها المخصصة ولا تنفَّذ أقسامٌ كثيرة من جزء نص البرنامج أبـدًا أو قد تنفَّذ مرة واحدة ولا تنفَّذ مرة أخرى، ولكن يمكن تبديل هذه الصفحات بدون أن تسبب مشاكلًا.
- إذا سرّب البرنامج ذاكرةً فقد يترك حيّزًا مخصصًا وراءه ولا يصل إليه أبدًا مرةً أخرى ولكن يسـتطيع نظـام التشغيل إيقاف هذا التسرب بفعالية عن طريق تبديل هذه الصفحات.

- يوجد على معظم أنظمة التشغيل عملياتٍ تشبه العفاريت daemons الـتي تبقى خاملـة معظم الـوقت وتتنبّه أحيانًا لتستجيب للأحداث ويمكن تبديل هذه العمليات عندما تكون خاملة.
- قد يفتح المستخدم نوافذ متعددة ولكن يكون عـدد قليـل منهـا فـاعلًا في نفس الـوقت وبالتـالي يمكن تبديل هذه العمليات غير الفاعلة.
- يمكن وجود عدة عملياتٍ مشغِّلةٍ لنفس البرنامج بحيث تتشارك هـذه العمليـات في جـزء نص البرنـامج والجزء الساكن لتجنب الحاجة إلى إبقاء نسخ متعددة في الذاكرة الحقيقية.

إذا أضفت مجمل الذاكرة المخصصة إلى كل العمليات فهذا سيزيد من حجم الذاكرة الحقيقـة بصـورة كبـيرة ومع ذلك لا يزال بإمكان النظام العمل جيدًا.

يجب على العملية التي تحاول الوصـول إلى صـفحة مبدَّلـة أن تعيـد البيانـات من القـرص الصـلب والـذي يستغرق عدة ميلي ثواني، ويكون هذا التأخير ملحوظًا غالبًا، فإذا تركت نافذةً خاملة لمـدة طويلـة ثم عـدت إليهـا فستكون بطيئةً في البداية وقد تسمع القرص الصلب يعمل ريثما تُبدّل الصفحات.

إن مثل هذه التأخيرات العرضية مقبولة ولكن إذا كان لديك عدة عمليات تستخدم حيّزًا كبيرًا فستتواجه هذه العمليات مع بعضها بعضًا، حيث تطرد العملية A عند تشـغيلها الصـفحات الـتي تحتاجهـا العمليـة B، ثم تطـرد العملية B عند تشغيلها الصفحات التي تحتاج إليها العملية A، وبالتـالي تصـبح كلا العمليـتين بطيئـتين إلى حـدٍ كبير ويصبح النظام غير مستجيب، يدعى هذا السيناريو بالتأزّم thrashing.

يمكن أن يتجنّب نظام التشغيل هذا التأزُّم من خلال اكتشاف زيادة في تبديل الصـفحات ثم ينهي أو يوقـف عملياتٍ حتى يستجيب النظام مرة أخرى، ولكن يمكن القول أن نظـام التشـغيل لا يقـوم بـذلك أو لا يقـوم بـذلك بصورة جيدة وإنما يترك الأمر أحيانًا للمسـتخدمين من خلال الحـد من اسـتخدامهم للـذاكرة الحقيقيـة أو محاولـة استرجاع النظام عند ظهور التأزّم.

8. تعدد المهام Multitasking

يتضمن المعالج نوى متعددة في العديد من الحواسيب الحالية وهذا يعني أنه يستطيع تشغيل عدة عمليات في نفس الوقت، وكل نواةٍ لديها القدرة على القيام بتعدد المهام multitasking أي يمكنها التبديل من عملية لعمليةٍ أخرى بسرعة، وبذلك تخلق وهمًا بوجود عدة عمليات مُشغَّلة في الوقت ذاته. يسمى جزء نظام التشغيل الذي يطبّق تعدد المهام بالنواة kernel وهي الجزء الأعمق في نظام التشغيل وتكون محاطةً بالصدفة shell سواء كان نظام تشغيل يشبه الجوزة nut أو البذرة seed، فالنواة kernel هي المستوى الأدنى من البرمجيات software في نظام التشغيل وتكون هذه النواة محاطةً بطبقات أخرى متعددة، وإحدى هذه الطبقات واجهةٌ interface تسمى صدفة shell حيث تلاحظ أن الاختصاصيين في علوم الحاسوب يحبون الاستعارات metaphors.

عمل النواة الأساسي هو معالجة المقاطعات، والمقاطعة هي الحدث الذي يوقِف دورة التعليمة instruction cycle القياسية ويسبب قفز تدفق التنفيذ execution flow إلى جزءٍ خاص من الشيفرة يدعى معالج المقاطعة interrupt handler.

للمقاطعة نوعان هما: مقاطعة عتادية hardware interrupt ومقاطعة برمجية software interrupt، حيث تحدث المقاطعة العتادية عندما يرسل جهازٌ ما إشارات إلى المعالج مثل تسبُّب واجهة الشبكة network interface بحدوث مقاطعة عند وصول حزمة بيانات packet of data أو مثل المقاطعة التي يسببها القرص الصلب disk drive عند اكتمال عملية نقل البيانات، وتحوي معظم الأنظمة مؤقتات timers تسبِّب مقاطعات عند الفواصل الزمنية المنتظمة regular intervals أو بعد انتهاء الوقت المستغرَق elapsed time.

تحدث المقاطعة البرمجية بسبب برنامجٍ قيد التشغيل مثـل عـدم اكتمـال تعليمـة لسـببٍ مـا، فتُنبّـه هـذه التعليمة مقاطعةً وبالتالي يعالج نظام التشغيل الشرط الخاص بالعمليـة المُقاطَعـة، حيث تُعـالَج أخطـاء الأعـداد العشرية floating-point errors مثل خطأ القسمة على صفر باستخدام المقاطعات.

ينشئ برنامج استدعاء نظام system call عندما يريد هذا البرنامج الوصول إلى جهاز عتادي، ويشبه استدعاءُ النظام استدعاءَ دالةٍ ولكن بدلًا من القفز إلى بداية الدالة ينفّذ استدعاءُ النظام تعليمة خاصة، وتنبّهُ هـذه التعليمـة مقاطعةً مسببةً قفز تدفق التنفيذ إلى النواة، ثم تقرأ النواة معامِلات استدعاء النظام وتجري العمليـات المطلوبـة ثم تستأنف العملية المُقاطَعة.

8.1 حالة العتاد Hardware state

تتطلب معالجة المقاطعات تعاونًا بين العتاد والبرمجيات، حيث من الممكن وجـود تعليمـات متعـددة قيـد التشغيل ضمن المعالج CPU وبيانات مُخزَّنة في المسجلات بالإضـافة إلى حالـة عتـادٍ hardware state أخـرى عند حدوث مقاطعة.

يكون العتاد عادةً مسؤولًا عن وصول المعالج إلى حالة الاسـتقرار consistent state فيجب أن تكتمـل كـل تعليمة أو أن تتصرف كأنها لم تبدأ من الأساس أي لا وجود لتعليمة نصفها مكتمـلٌ على سـبيل المثـال، والعتـاد مسؤولٌ أيضًا عن حفظ عدّاد البرنامج program counter ويختصر إلى PC الذي تستخدمه النواة لتعرف من أين ستستأنف تنفيذ التعليمات، ثم يستلم معالج المقاطعة interrupt handler مسؤولية حفـظ بقيـة حالـة العتـاد قبل أن يقوم بأي شيء آخر يعدّل حالة العتاد هذه ثم يستعيد حالة العتاد المحفوظة سابقًا قبل استئناف العمليـة المُقاطعَة، حيث يمكن اختصار سلسلة الأحداث السابقة كما يلي:

1. يحفظ العتاد عدّاد البرنامج في مسجّلٍ خاص عند حدوث المقاطعة ثم يقفز العتاد إلى معـالج المقاطعـة المناسب.
2. ثم يخزّن معالجُ المقاطعة عدادَ البرامج وحالة المسجل status register في الذاكرة إلى جانب محتويات مسجلات البيانات التي من المُخطَّط استخدامها.
3. ثم يُشغّل معالج المقاطعة الشيفرة المطلوبة لمعالجة هذه المقاطعة.
4. يستعيد معالج المقاطعة محتويات المسجلات التي خزّنها سابقًا ثم أخيرًا يستعيد عداد البرنامج للعملية المقاطعة وهذا يؤدّي إلى العودة إلى التعليمة المُقاطَعة.

إذا استخدِمت هذه الآلية بصورة صحيحة فلا يمكن أن تعلم العملية المقاطَعة بحدوث المقاطعـة أبـدًا إلّا إذا اكتشفت تغيّرًا في الوقت الفاصل بين التعليمات.

8.2 تبديل السياق Context switching

يمكن أن تكون معالجات المقاطعة سريعةً لأنها غير ملزمةٍ بحفظ كامل حالة العتاد وإنما تحفـظ المسـجلات التي من المخطط استخدامها فقط، ولكن لا تستأنف النـواةُ العمليـةَ المقاطعـة دائمًـا عنـد حـدوث مقاطعـةٍ مـا وبالتالي يكون للنواة حرية التبديل إلى عملية أخرى، وتدعى هذه الآلية بتبديل السياق context switch.

لا تعلم النواة أيّ مسجلات ستستخدمها العملية لذلك يجب أن تحفـظ كـل المسـجلات، ويجب على النـواة تصفير البيانات المخزنة في وحدة إدارة الذاكرة memory management unit عند التبديل إلى عمليـة جديـدة، حيث يمكن أن يستغرق تحميل بيانات العملية الجديدة إلى الذاكرة المخبئية بعض الـوقت بعـد تبـديل السـياق إليها لذلك يكون تبديل السياق بطيئًا نسبيًا فقد يستغرق آلاف الدورات أو عدة ميكرو ثانية.

يُسمَح لكل عملية في نظام متعدد المهام أن تُشغَّل لفترة زمنية قصيرة تدعى بشريحة زمنيـة time slice أو حصّة quantum، وتضبط النواة مـؤقت العتـاد hardware timer خلال عمليـة تبـديل السـياق، وهـذا يسـبب حدوث مقاطعة عند نهاية الشريحة الزمنية، وبالتالي تستطيع النواة عند حدوث مقاطعةٍ التبديلَ إلى عملية أخـرى أو السماح للعملية المقاطَعة أن تستأنف عملها، وجزء نظام التشغيل الذي يقرّر اختيار أحـد هـذين الخيـارين هـو المجدول scheduler.

8.3 دورة حياة العملية

يخصص نظام التشغيل للعملية عند إنشائها بنية بيانات تتضمن معلومـات عن هـذه العمليـة وتـدعى بينـة البيانـات هـذه بكتلـة تحكم العمليـة process control block وتختصـر إلى PCB الـتي تتتبّـع حالة العملية process state، ويكون للعملية أربع حالات هي:

- التنفيذ Running: عند تنفيذ العملية ضمن النواة core.
- الاستعداد Ready: عندما تكون العمليـة جـاهزة للتنفيـذ ولكنهـا لا تُنفَّذ ويجب عليهـا الانتظـار لأن عـدد العمليات القابلة للتنفيذ أكبر من عدد الأنوية cores.
- الإيقاف Blocked: إذا كان غير ممكن أن تُنفَّذ العملية لأنها تنتظر حدثًا مستقبليًا مثـل اتصـال شـبكة أو قراءة من القرص الصلب.
- الاكتمـال Done: إذا اكتمـل تنفيـذ العمليـة ولكنهـا تملـك معلومـات حالـة المغـادرة exit status information التي لم تُقرَأ بعد.

الأحداث التي تسبب انتقال العملية من حالة إلى أخرى هي:

- تُنشَأ العملية عندما ينفّذ البرنامج المُشغَّل استدعاء نظام مثل `fork`، حيث تصبح العملية المنشَأة أو الجديدة في نهاية استدعاء النظام في حالة الاستعداد ثم قد يستأنف المجدول العملية الأصلية التي تسمى العملية الأب parent أو يبتدئ المجدول العملية الجديدة التي تسمى العملية الابن child.
- تتغير حالة العملية من حالة الاستعداد إلى حالة التنفيذ عندما يبتدئها المجدول أو يستأنفها.
- تتغير حالة العملية من حالة التنفيذ إلى الاستعداد عندما تُقاطَع العملية ويختار المجدول ألا يستأنفها.
- إذا نفّذت العملية استدعاء النظام الذي لا يكتمل على الفور وإنما يحتاج وقتًا مثل الطلب من القرص الصلب فتصبح العملية بحالة الإيقاف وعندها يختار المجدول عمليةً أخرى لتنفيذها.
- إذا اكتملت عمليةٌ ما مثل عملية طلب من القرص الصلب فإنها تسبب مقاطعة، ويحدد معالج المقاطعة العملية المنتظرة لعملية الطلب هذه ويبدّل حالتها من حالة الإيقاف إلى الاستعداد ثم يختار المجدول أن يستأنفها أم لا.
- إذا استدعت العملية الدالة `exit` فإن معالج المقاطعة يخزّن شيفرة المغادرة exit code في كتلة تحكم العملية PCB ثم يغير حالة العملية إلى حالة الاكتمال.

8.4 الجدولة Scheduling

من الممكن وجود مئات العمليات على الحاسوب ولكن معظمها في حالة إيقاف blocked ويكون عدد قليل منها في حالة استعداد أو تنفيذ، والمجدول هو الذي يقرر أية عمليةٍ تبدأ التنفيذ أو تستأنف عملها عند حدوث مقاطعة. هدف المجدول الرئيسي هو تقليل وقت الاستجابة response time قدر الإمكان على الحاسوب المحمول laptop أو على محطة العمل workstation، حيث يجب أن يستجيب الحاسوب بسرعة لإجراءات المستخدم. وقت الاستجابة مهمٌ أيضًا في المخدمات servers بالإضافة إلى أنه يجب على المجدول زيادة الإنتاجية throughput والتي هي عدد الطلبات المنجزة خلال واحدة الزمن، وفي الحقيقة لا يملك المجدول معلومات كثيرة عمّا تفعله العمليات لذلك تعتمد قراراته في اختيار العملية على عدة استنتاجات نذكرها تاليًا.

أولًا، يمكن أن تكون العمليات محدودةً بموارد مختلفة، فالعملية التي تقوم بعمليات حسابية كثيرة محدودةٌ بالمعالج CPU-bound أي أن وقت تشغيل هذه العملية يعتمد على كمية الوقت الذي تأخذه من وقت المعالج، أما العملية التي تقرأ بيانات من الشبكة أو من القرص الصلب فتكون محدودةً بعمليات الإدخال والإخراج I/O-bound أي تكون هذه العملية أسرع إذا كان إدخال أو إخراج البيانات أسرع، ولكنها لن تنفّذ أسرع إذا كان وقت المعالج الخاص بها أكبر، ويمكن أن تكون العملية التي تتفاعل مع المستخدم في حالة الإيقاف حيث ستبقى منتظرةً إجراءات المستخدم معظم الوقت. يصنّف نظام التشغيل العمليات أحيانًا تبعًا لسلوكها السابق ويجدولها بناءً على ذلك، فمن المحتمل أن تنفّذ العملية التفاعلية interactive process مباشرةً عندما تنتهي من حالة الإيقاف لأن المستخدم ينتظر ردًا منها، بينما تكون العملية المحدودة بالمعالج CPU-bound والتي ما زالت تنفّذ منذ مدة طويلة أقلَّ حساسيةً لعامل الوقت.

ثانيًا، إذا كان من المحتمل أن تُشغَّل العملية لفترة قصيرة ثم تطلب شيئًا يجعلها في حالة إيقاف، فيمكن أن تُشغَّل على الفور لسببين هما: (1) إذا استغرق الطلب بعض الوقت لإكماله فيجب أن يبدأ في أقرب وقت ممكن، (2) من الأفضل أن تنتظر عملية ذات وقت تنفيذ طويل لفترةٍ قصيرة وليس العكس، بصورة مشابهة افترض أنك تصنع فطيرة تفاح، حيث يستغرق تحضير الطبقة الخارجية للفطيرة 5 دقائق ولكن يجب تركها لتبرد لمدة نصف ساعة ويستغرق تحضير حشوة الفطيرة 20 دقيقة، فإذا حضّرت الطبقة الخارجية أولًا فيمكنك تحضير الحشوة ريثما تبرد الطبقة الخارجية وبالتالي تنهي تحضير الفطيرة خلال 35 دقيقة، أما إذا حضّرت الحشوة أولًا فيستغرق تحضير الفطيرة 55 دقيقة.

تستخدم معظم المجدولات بعض نماذج الجدولة المعتمدة على الأولوية priority-based scheduling، حيث يكون لكل عملية أولوية تزيد أو تنقص خلال الوقت ويختار المجدول العملية القابلة للتنفيذ ذات الأولوية العليا، وهناك عدة عوامل لتحديد أولوية العملية هي:

- تبدأ العملية عادةً برقم أولوية عالٍ نسبيًا لذلك تبدأ التنفيذ بسرعة.
- إذا طلبت العملية شيئًا ما جعلها في حالة إيقاف قبل انتهاء شريحتها الزمنية ضمن المعالج فمن المحتمل أن تكون عملية تفاعلية مع المستخدم interactive أو عملية محدودة بعمليات الإدخال والإخراج I/O-bound لذلك يجب أن تصبح أولويتها أعلى.
- إذا انتهت الشريحة الزمنية الخاصة بالعملية ضمن المعالج ولم ينتهِ تنفيذ هذه العملية فمن المحتمل أن تكون عملية ذات وقت تنفيذ طويل long-running ومحدودة بالمعالج CPU-bound لذلك يجب أن تصبح أولويتها أقل.
- إذا توقفت مهمةٌ لمدة طويلة ثم أصبحت بحالة استعداد فيجب أن تحصل على زيادة في الأولوية لتتمكن من الاستجابة على الشيء الذي انتظرته.
- إذا توقفت العملية A بسبب انتظارها للعملية B وهاتان العمليتان مرتبطتان عن طريق أنبوب pipe مثلًا فيجب أن تصبح أولوية العملية B أعلى.
- يسمح استدعاء النظام `nice` للعملية بتقليل أولويتها (ولا تسمح بزيادتها) مما يسمح للمبرمجين بتمرير معلومات محددة إلى المجدول.

لا تؤثر خوارزميات الجدولة scheduling algorithms كثيرًا على أداء معظم أنظمة التشغيل التي تعمل بأحمال workloads عادية فسياسات الجدولة scheduling policies البسيطة جيدة كفاية لهذه الأنظمة.

8.5 الجدولة في الوقت الحقيقي Real-time scheduling

الجدولة مهمةٌ جدًا بالنسبة للبرامج التي تتفاعل مع العالم الحقيقي، فقد يضطر البرنامج الذي يقرأ بيانات من الحسّاسات sensors والذي يتحكم بالمحركات إلى إكمال المهام المتكررة بالحد الأدنى من التكرار وأن

يتفاعل مع الأحداث الخارجية بالحد الأقصى من وقت الاستجابة، ويُعبّر عن هذه المتطلبات بالمهـام الـتي يجب إكمالها قبل المواعيد النهائية deadlines.

تـــــدعى جدولـــــة المهـــــام من أجـــــل الوفـــــاء بالمواعيـــــد النهائيـــــة بالجدولـــــة في الـــــوقت الحقيقي real-time scheduling، ويمكن تعـديل أنظمـة التشـغيل الـتي تسـتخدم للأغـراض العامـة مثل لينكس Linux لتتعامـل مـع الجدولـة في الـوقت الحقيقي بالنسـبة لبعض التطبيقـات، وقـد تشـمل هـذه التعديلات ما يلي:

- توفير واجهات برمجة تطبيقات APIs أثرى للتحكم في أولويات المهام.
- تعديل المجدول لضمان تشغيل العملية ذات الأولوية الأعلى خلال مدة زمنية محددة.
- إعادة تنظيم معالجات المقاطعة لضمان أكبر وقت لاكتمال العمليات.
- تعديل الأقفال locks وآليات المزامنة الأخرى synchronization mechanisms سـنتطرق إليهـا لاحقًـا للسماح لمهمة ذات أولوية عالية أن تسبق مهمة ذات أولوية أقل.
- اختيار تطبيق تخصيص الذاكرة الديناميكي الذي يضمن أكبر وقت لاكتمال العمليات.

توفر أنظمة التشغيل في الوقت الحقيقي real-time operating systems إمكانيات متخصصـة بالنسـبة للتطبيقات الأكثر طلبًا وخاصة في المجـالات الـتي تمثـل فيهـا الاسـتجابة في الـوقت الحقيقي مسـألةَ حيـاة أو موت، وتكون هذه الأنظمة ذات تصميم أبسط بكثير من أنظمة التشغيل ذات الأغراض العامة.

9. مفهوم الخيوط Threads

الخيط Thread هو نوع معين أو خاص من العمليات، حيث ينشئ نظام التشغيل حيز عناوين جديدًا عند إنشاء عملية، ويتضمن هذا الحيز جزء الشيفرة أو نص البرنامج text segment والجزء الساكن static segment وجزء الكومة heap، وينشئ نظام التشغيل أيضًا خيط تنفيذ thread of execution جديدًا يتضمن عداد البرنامج program counter وحالة عتاد أخرى واستدعاء المكدس.

العمليات التي رأيتها لحد الآن هي عمليات ذات خيط وحيد single-threaded أي يوجد خيط تنفيذ واحد فقط يعمل في كل حيز عناوين، وستتعرف على العمليات ذات الخيوط المتعددة multi-threaded، أي التي تملك خيوطًا متعددة تعمل في نفس حيز العناوين.

تتشارك كل الخيوط بنفس جزء الشيفرة ضمن العملية الواحدة أي أنها تشغّل نفس الشيفرة، ولكن تشغّل هذه الخيوط المختلفة أجزاءً مختلفة من تلك الشيفرة، وتتشارك الخيوط ضمن العملية الواحدة بنفس الجزء الساكن static segment، لذلك إذا غيّر أحد الخيوط متغيرًا عامًا global variable فإن بقية الخيوط ترى هذا التغيير، ويتشاركون أيضًا بالكومة heap لذلك تستطيع الخيوط التشارك بقطع الذاكرة المخصصة ديناميكيًا dynamically-allocated chunks، ولكن يكون لكل خيطٍ جزء المكدس الخاص به لذلك تستطيع الخيوط استدعاء دوالٍ دون التداخل مع بعضها البعض، ولا تصل الخيوط عادةً إلى المتغيرات المحلية لخيطٍ آخر، حيث لا تستطيع الوصول إليها في بعض الأحيان.

9.1 الخيوط القياسية

الخيـوط القياسـية الأكـثر شـيوعًا والمسـتخدمة مـع C هي خيـوط POSIX أو اختصـارًا Pthreads. تعـرّف خيوط POSIX القياسية نمـوذج خيـط thread model وواجهـةً interface لإنشـاء الخيـوط والتحكم بهـا، وتـوفّر معظم نسـخ UNIX تطبيقًـا للصـنف Pthreads. ويشـبه اسـتخدامُ Pthreads اسـتخدامَ معظم مكتبـات لغة C حيث:

- تضمّن ملفات الترويسات headers files في بداية برنامجك.
- تكتب الشيفرة التي تستدعي دوالًا معرّفة باستخدام Pthreads.
- تربط link البرنامج عند تصريفه compile مع مكتبة Pthread.

يضمِّن البرنامج ملفات الترويسات التالية:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

أول اثنين من ملفات الترويسات السابقة همـا تضـمين لمكتبـات قياسـية، أمـا ملـف الترويسـات الثـالث فيُسـتخدم من أجـل Pthreads، ويُسـتخدم ملـف الترويسـات الرابـع من أجـل متغـيرات تقييد الوصول semaphores. يمكنك استخدام الخيار -l في سطر الأوامـر لتصـريف البرنـامج مـع مكتبـة Pthread باستخدام الأداة gcc كما يلي:

```
gcc -g -O2 -o array array.c -lpthread
```

يصـرِّف الأمـر السـابق ملفًـا مصـدريًا يـدعى array.c مـع معلومـات تنقيح الأخطـاء debugging info والتحسين optimization ويربطه مع مكتبة Pthread ثم يولّد ملفًا تنفيذيًا يدعى array.

9.2 إنشاء الخيوط

تدعى دالة Pthread التي تنشئ خيوطًا pthread_create، وتُظهر الدالة التالية كيفية استخدامها:

```
pthread_t make_thread(void *(*entry)(void *), Shared *shared)
{
    int n;
    pthread_t thread;
```

```
    n = pthread_create(&thread, NULL, entry, (void *)shared);
    if (n != 0)
    {
        perror("pthread_create failed");
        exit(-1);
    }
    return thread;
}
```

الدالة make_thread هي دالة مغلّفة وكُتبت لجعل الدالة pthread_create سهلة الاستخدام ولتوفير التحقق من الأخطاء error-checking. نوع القيمة المعادة من الدالة pthread_create هو pthread_t والذي يمكنك التفكير به كمعرّف id أو مِقبض handle للخيط الجديد. إذا نجح تنفيذ الدالة pthread_create فستعيد القيمة 0 وتعيد الدالة make_thread مقبض الخيط الجديد، وإذا ظهر خطأ فتعيد الدالة pthread_create شيفرة الخطأ وتطبع الدالة make_thread رسالة خطأ وتنتهي. Shared المعامل الثاني للدالة make_thread هو عبارة عن بنية structure عُرِّفت لتتضمن القيم المشتركة بين الخيوط، حيث يمكنك تعريف نوع جديد من خلال استخدام عبارة typedef كما يلي:

```
typedef struct
{
    int counter;
} Shared;
```

والمتغير المشترك الوحيد في هذه الحالة هو counter، وتخصص الدالة make_shared حيّزًا للبنية Shared وتهيئ محتوياتها كما يلي:

```
Shared *make_shared()
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->counter = 0;
    return shared;
}
```

لديك الآن بنية بيانات مشتركة وإذا عدتَ إلى الدالة make_thread وتحديدًا المعامل الأول الذي هو عبارة عن مؤشر pointer إلى دالة، وتأخذ هذه الدالة مؤشر void وتعيد مؤشر void أيضًا. إذا أصبح نظرك مشوشًا بسبب صيغة تصريح هذا النوع فلست الوحيد في ذلك، على كل حال إن الهدف الأساسي من هذا المعامل هو أن يحدد للدالة مكان بدء تنفيذ الخيط الجديد، وتدعى هذه الدالة entry:

```
void *entry(void *arg)
{
    Shared *shared = (Shared *)arg;
    child_code(shared);
    pthread_exit(NULL);
}
```

يجب أن يُصرَّح عن معامل الدالة entry كمؤشر void، ولكنه في هذا البرنامج مؤشرٌ إلى بنية Shared لذلك يمكن تبديل نوعه typecast ثم تمريره إلى الدالة child_code التي تقوم بالعمل الحقيقي، حيث تطبع الدالة child_code قيمة المتغير المشترك counter ثم تزيد قيمته كما يلي:

```
void child_code(Shared *shared)
{
    printf("counter = %d\n", shared->counter);
    shared->counter++;
}
```

تستدعي الدالةُ entry الدالةَ pthread_exit بعد أن تنتهي الدالة child code وتعيد قيمةً، حيث يمكن أن تُستخدم الدالة pthread_exit لتمرير قيمة إلى الخيط الذي يُضم مع الخيط الحالي، وبالتالي في هذه الحالة لا يبقى شيء للخيط الابن لعمله فتُمرَّر القيمة الخالية NULL، وأخيرًا تنشئ الشيفرة التالية الخيوط الأبناء child threads كما يلي:

```
int i;
pthread_t child[NUM_CHILDREN];

Shared *shared = make_shared(1000000);

for (i = 0; i < NUM_CHILDREN; i++)
{
    child[i] = make_thread(entry, shared);
}
```

NUM_CHILDREN هو ثابت وقت التصريف constant compile-time الذي يحدد عدد الخيوط الأبناء، و child هي مصفوفة مقابض الخيوط thread handles.

9.3 ضم الخيوط

إذا أراد خيطٌ انتظار خيطٍ آخر ليكتمل فإنه يستدعي الدالة pthread_join، وتجد فيما يلي الدالة المغلّفة للدالة pthread_join:

```
void join_thread(pthread_t thread)
{
    int ret = pthread_join(thread, NULL);
    if (ret == -1)
    {
        perror("pthread_join failed");
        exit(-1);
    }
}
```

معامل الدالة المغلّفة هو مقبض الخيط الذي تنتظره ليكتمل، وعمل الدالة المغلّفة هو فقط استدعاء الدالة pthread_join والتحقق من النتيجة. يستطيع أي خيط أن يضم أي خيطٍ آخر، ولكن في النماذج الأكثر شيوعًا ينشئ الخيط الأب parent thread كل الخيوط الأبناء ويضمها join. تجد فيما يلي الشيفرة التي تنتظر الخيوط الأبناء بها:

```
for (i = 0; i < NUM_CHILDREN; i++)
{
    join_thread(child[i]);
}
```

تنتظر هذه الحلقات أحد الخيوط الأبناء في كل مرة وذلك حسب ترتيب إنشائها، ولا يوجد ضمان أن تكتمل الخيوط الأبناء في هذا الترتيب ولكن تعمل هذه الحلقة بصورة صحيحة حتى في حال لم يحدث ذلك، فإذا تأخر أحد الخيوط الأبناء فيجب أن تنتظر الحلقة، ويمكن أن تكتمل الخيوط الأبناء الأخرى خلال وقت الانتظار هذا، حيث لا يمكن أن تنتهي هذه الحلقة إلا في حال اكتمال جميع الخيوط الأبناء. يمكنك الاطلاع على المثال ضمن counter/counter.c ثم تصريفه وتشغيله كما يلي:

```
$ make counter
gcc -Wall counter.c -o counter -lpthread
$ ./counter
```

فعند تشغيله مع 5 خيوط أبناء سينتج الخرج التالي:

```
counter = 0
counter = 0
counter = 1
counter = 0
counter = 3
```

وسينتج خرج آخر عندما تشغله على حاسوبك، وإذا شغلته مرة أخرى سينتج خرج مختلف أيضًا، فماذا يحدث؟

9.4 الأخطاء المتزامنة Synchronization errors

مشكلة البرنامج السابق أن الخيوط الأبناء تستطيع الوصول إلى المتغير المشترك `counter` بدون تزامن، لذلك تستطيع عدة خيوط قراءة نفس قيمة المتغير `counter` قبل أن يزيد أي خيطٍ قيمته. يمكن أن تشرح سلسلة الأحداث التالية الخرج الذي حصلت عليه سابقًا:

```
Child A reads 0
Child B reads 0
Child C reads 0
Child A prints 0
Child B prints 0
Child A sets counter=1
Child D reads 1
Child D prints 1
Child C prints 0
Child A sets counter=1
Child B sets counter=2
Child C sets counter=3
Child E reads 3
Child E prints 3
Child D sets counter=4
Child E sets counter=5
```

يمكن أن تُقاطَع الخيوط في أماكن مختلفة في كل مرة تشغّل فيها البرنامج، أو قد يختار المجدول خيوطًا مختلفة ليشغّلها، لذلك ستكون سلسلة الأحداث والنتائج مختلفة.

افترض أنك تريد فرض بعض الترتيب، أي مثلًا تريد أن يقرأ كل خيط قيمةً مختلفة للمتغير `counter` ثم يزيدها، وبالتالي تُظهر قيمة المتغير `counter` عدد الخيوط التي نفّذت الدالة `child_code`، ويمكنك استخدام كائن المزامنة mutex لتطبيق ذلك، حيث كائن المزامنة mutex هو عبارة عن كائن object يضمن

حدوث إقصاء متبادل mutual exclusion لكتلة من الشيفرة، أي ينفّذ خيطٌ واحد فقط كتلة الشيفرة في نفس الوقت.

كتبتُ، يقول الكاتب، نموذجًا يدعى `mutex.c` يوفر كائنات المزامنة، ستجد فيما يلي نسخةً من الدالة `child_code` التي تستخدم كائن المزامنة لتأمين تزامن الخيوط:

```
void child_code(Shared *shared)
{
    mutex_lock(shared->mutex);
    printf("counter = %d\n", shared->counter);
    shared->counter++;
    mutex_unlock(shared->mutex);
}
```

حيث يجب على كل خيط أن يقفل lock كائن المزامنة قبل أن يصل أي خيطٍ آخر إلى المتغير المشترك `counter`، وهذا يؤدي إلى حظر كل الخيوط الأخرى من الوصول إلى هذا المتغير.

افترض أن الخيط A قفل كائن المزامنة وهو في منتصف الدالة `child_code`، فإذا وصل الخيط B ونفّذ الدالة `mutex_lock` يتوقف تنفيذ الخيط B.

ينفّذ الخيط A الدالة `mutex_unlock` عندما ينتهي، وبالتالي يسمح للخيط B متابعة تنفيذه، أي تنفّذ الخيوط الدالة `child_code` على التوالي بحيث ينفّذها خيطٌ واحدٌ فقط في نفس الوقت، وبالتالي لا يتعارض أي خيط مع الخيوط الأخرى، وإذا شغّلت الشيفرة مع 5 خيوط أبناء سينتج:

```
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
```

وهذا هو المطلوب. يجب إضافة كائن المزامنة `Mutex` إلى البنية Shared لكي يعمل هذا الحل بالصورة الصحيحة:

```
typedef struct
{
    int counter;
    Mutex *mutex;
} Shared;
```

وتهيئته في الدالة make_shared:

```
Shared *make_shared(int end)
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->counter = 0;
    shared->mutex = make_mutex(); //-- هذا السطر جديد
    return shared;
}
```

9.5 كائن المزامنة Mutex

تعريفي، يقول الكاتب، للكائن Mutex هو مغلّف لنوعٍ يدعى pthread_mutex_t وهو معرّفٌ في واجهة برمجة التطبيقات للخيوط POSIX، ولإنشاء كائن مزامنة POSIX يجب تخصيص حيز للنوع pthread_mutex_t ثم استدعاء الدالة pthread_mutex_init.

إحدى مشاكل واجهة برمجة التطبيقات هذه أن النوع pthread_mutex_t يتصرف كبنية، لذلك إذا مررته كوسيط سينشئ نسخةً تجعل كائن المزامنة يتصرف بصورة غير صحيحة، ويمكنك تجنب ذلك من خلال تمرير النوع pthread_mutex_t باستخدام عنوانه.

تجعل الشيفرة التي كتبتها، يقول الكاتب، الأمور أسهل من خلال تعريف نوعٍ هو النوع Mutex الذي هو عبارة عن اسم للنوع pthread_mutex_t يمكن قراءته بطريقة أسهل:

```
#include <pthread.h>

typedef pthread_mutex_t Mutex;
```

ثم تعريف دالةٍ هي الدالة make_mutex التي تخصص حيّزًا لكائن المزامنة وتهيئته:

```
Mutex *make_mutex()
{
    Mutex *mutex = check_malloc(sizeof(Mutex));
    int n = pthread_mutex_init(mutex, NULL);
    if (n != 0)
        perror_exit("make_lock failed");
    return mutex;
}
```

القيمة المعادة هي مؤشر يمكن أن تمرره كوسيط دون أن يسبب نسخًا غير مرغوبة. الـدوال الـتي تسـتخدم لقفل وفك قفل كائن المزامنة هي دوالٌ مغلّفة بسيطة لدوال POSIX:

```
void mutex_lock(Mutex *mutex)
{
    int n = pthread_mutex_lock(mutex);
    if (n != 0)
        perror_exit("lock failed");
}

void mutex_unlock(Mutex *mutex)
{
    int n = pthread_mutex_unlock(mutex);
    if (n != 0)
        perror_exit("unlock failed");
}
```

10. المتغيرات الشرطية والتزامن بين العمليات

يمكن حل العديد من مشاكل التزامن synchronization البسيطة باستخدام كائنات المزامنة mutexes، ولكن يوجد مشكلة أكبر هي مشكلة منتج-مستهلك Producer-Consumer problem التي تُحل باستخدام أداة جديدة هي المتغير الشرطي condition variable.

10.1 طابور العمل work queue

تُنظَّم الخيوط في بعض البرامج ذات الخيوط المتعددة لتجري عدة مهام، وتتواصل هذه الخيوط مع بعضها البعض غالبًا عن طريق طابور queue، حيث تدعى الخيوط التي تضع بيانات في الطابور بالخيوط المنتجة producers، وتدعى الخيوط التي تأخذ بيانات من الطابور بالخيوط المستهلكة consumers.

يمكن أن يوجد خيطٌ يشغّل واجهة المستخدم الرسومية graphical user interface -وتختصر إلى GUI- للاستجابة لأحداث المستخدم في التطبيقات التي لديها واجهة مستخدم رسومية على سبيل المثال، ويمكن أن يوجد خيطٌ آخر يعالج طلبات المستخدم، حيث يمكن أن يضع خيطُ واجهة المستخدم الرسومية الطلبات في طابورٍ ثم يأخذ الخيط المقابل هذه الطلبات ويعالجها.

تحتاج تطبيق طابور لدعم هذا التنظيم، بحيث يحافظ تطبيق الطابور على الخيوط thread safe، وهذا يعني أنه يستطيع كلا الخيطين (أو أكثر من خيطين في بعض الأحيان) الوصول إلى الطابور في نفس الوقت، وتحتاج أيضًا أن تعالج الحالات الخاصة مثل أن يكون الطابور فارغًا وأن يكون حجم الطابور منتهٍ عندما يمتلئ.

سأبدأ، يقول الكاتب، بطابورٍ بسيط لا يحافظ على الخيوط ثم ترى كيف يكون ذلك خاطئًا وكيف يُصحَّح ذلك الخطأ. شيفرة هذا المثال موجودة في المجلد queue حيث يتضمن الملف `queue.c` التطبيق الأساسي للمخزَن الدائري circular buffer. تجد تعريف البنية `Queue` فيما يلي:

```
typedef struct
{
    int *array;
    int length;
    int next_in;
    int next_out;
} Queue;
```

`array` هو المصفوفة التي تتضمن عناصر الطابور وهي أعداد صحيحة ints في هذا المثال، ولكنها يمكن أن تكون بنى structures تتضمن أحداث المستخدم وعناصر العمل وغير ذلك. `length` هو طول المصفوفة، و `next_in` هو دليل index المصفوفة الذي يحدد مكان إضافة العنصر التالي في الطابور، أما `next_out` هو دليل العنصر التالي الذي يجب حذفه من الطابور. تخصص الدالة `make_queue` حيزًا للبنية `Queue` وتهيئ حقولها كما يلي:

```
Queue *make_queue(int length)
{
    Queue *queue = (Queue *)malloc(sizeof(Queue));
    queue->length = length + 1;
    queue->array = (int *)malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    return queue;
}
```

تحتاج القيمة الابتدائية للمتغير `next_out` بعض الشرح، فبما أن الطابور فارغ مبدئيًا فلا وجود لعنصر تالٍ لحذفه، لذلك يكون المتغير `next_out` غير صالح invalid، وضبط `next_out == next_in` هي حالة خاصة تحدد أن الطابور فارغ، فيمكن كتابة ما يلي:

```
int queue_empty(Queue *queue)
{
    return (queue->next_in == queue->next_out);
}
```

يمكنك الآن إضافة عناصر إلى الطابور باستخدام الدالة `queue_push`:

```
void queue_push(Queue *queue, int item)
{
```

```
    if (queue_full(queue))
    {
        perror_exit("queue is full");
    }

    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
}
```

إذا كان الطابور ممتلئًا فإن الدالة queue_push تطبع رسالة خطأ وتغادر، أما إذا كان الطابور غير ممتلئ فتدخِل الدالة queue_push عنصرًا جديدًا ثم تزيد قيمة المتغير next_in باستخدام الدالة queue_incr كما يلي:

```
int queue_incr(Queue *queue, int i)
{
    return (i + 1) % queue->length;
}
```

تعود قيمة الدليل i إلى الصفر عندما يصل الدليل إلى نهاية المصفوفة، وهذا هو المكان الذي نواجه فيه الجزء الصعب، فإذا واصلنا إضافة عناصر إلى الطابور فسيعود المتغير next_in ويلحق بالمتغير next_out، وإذا كان next_in == next_out ستستنتج بصورة غير صحيحة أن الطابور فارغ، لذلك يجب تعريف حالة خاصة أخرى لتحديد أن الطابور ممتلئ لتجنب ذلك:

```
int queue_full(Queue *queue)
{
    return (queue_incr(queue, queue->next_in) == queue->next_out);
}
```

حيث إذا واصلت زيادة المتغير next_in ليصل إلى قيمة المتغير next_out فهذا يعني أنك لا تستطيع إضافة عنصر آخر إلى الطابور بدون جعل الطابور يبدو فارغًا، لذلك يجب التوقف عن إضافة عناصر أخرى قبل نهاية الطابور بعنصر واحد (يجب أن تعرف أن نهاية الطابور يمكن أن تكون في أي مكان وليس بالضرورة عند نهاية المصفوفة).

يمكن الآن كتابة الدالة queue_pop التي تحذف وتعيد العنصر التالي من الطابور كما يلي:

```
int queue_pop(Queue *queue)
{
    if (queue_empty(queue))
    {
        perror_exit("queue is empty");
    }

    int item = queue->array[queue->next_out];
    queue->next_out = queue_incr(queue, queue->next_out);
    return item;
}
```

وإذا جربت سحب pop عنصر من طابور فارغ فستطبع الدالة queue_pop رسالة خطأ وتغادر.

10.2 المستهلكون والمنتجون Producers-consumers

تنشئ الآن بعض الخيوط لتصل إلى هذا الطابور، حيث شيفرة المنتج producer هي كما يلي:

```
void *producer_entry(void *arg)
{
    Shared *shared = (Shared *)arg;

    for (int i = 0; i < QUEUE_LENGTH - 1; i++)
    {
        printf("adding item %d\n", i);
        queue_push(shared->queue, i);
    }
    pthread_exit(NULL);
}
```

أما شيفرة المستهلك consumer هي:

```
void *consumer_entry(void *arg)
{
    int item;
    Shared *shared = (Shared *)arg;

    for (int i = 0; i < QUEUE_LENGTH - 1; i++)
```

```
    {
        item = queue_pop(shared->queue);
        printf("consuming item %d\n", item);
    }
    pthread_exit(NULL);
}
```

وشيفرة الخيط الأب الذي يبدأ الخيوط وينتظرها لتنتهي هي:

```
pthread_t child[NUM_CHILDREN];

Shared *shared = make_shared();

child[0] = make_thread(producer_entry, shared);
child[1] = make_thread(consumer_entry, shared);

for (int i = 0; i < NUM_CHILDREN; i++)
{
    join_thread(child[i]);
}
```

والبنية المشتركة التي تتضمن الطابور هي:

```
typedef struct
{
    Queue *queue;
} Shared;

Shared *make_shared()
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->queue = make_queue(QUEUE_LENGTH);
    return shared;
}
```

تمثل الشيفرة السابقة التي حصلت عليها حتى الآن بدايةً جيدة ولكن لديها بعض المشاكل هي:

- لا يحافظ الوصول إلى الطابور على الخيوط، حيث يمكن أن تصل خيوط متعددة إلى المتغيرات `array` و`next_in` و`next_out` في نفس الوقت، وهذا يترك الطابور تالفًا وفي حالة غير مستقرة.

- إذا جُدوِل الخيط المستهلك أولًا فسيجد الطابور فارغًا، وبالتالي يطبع رسالة خطأ وينتهي، لذلك من الأفضل أن يتوقف المستهلك حتى يصبح الطابور غير فارغ. وبالمثل يجب إيقاف المنتج إذا كان الطابور ممتلئًا.

ستُحل المشكلة الأولى في الفقرة القادمة باستخدام Mutex، وستحل المشكلة الثانية في الفقرة التي بعدها باستخدام المتغيرات الشرطية.

10.3 الإقصاء المتبادل Mutual exclusion

يحافظ الطابور على الخيوط باستخدام كائن المزامنة (mutex)، حيث تضيف أولًا المؤشر Mutex إلى بنية الطابور:

```
typedef struct
{
    int *array;
    int length;
    int next_in;
    int next_out;
    Mutex *mutex; //-- هذا السطر جديد
} Queue;
```

ثم تهيئه في الدالة make_queue:

```
Queue *make_queue(int length)
{
    Queue *queue = (Queue *)malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *)malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_mutex(); //-- جديد
    return queue;
}
```

ثم تضيف شيفرة التزامن إلى الدالة queue_push:

```
void queue_push(Queue *queue, int item)
{
```

```
    mutex_lock(queue->mutex); //-- جديد
    if (queue_full(queue))
    {
        mutex_unlock(queue->mutex); //-- جديد
        perror_exit("queue is full");
    }

    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
    mutex_unlock(queue->mutex); //-- جديد
}
```

يجب قفل Mutex قبل التحقق إذا كان الطابور ممتلئًا أم لا، فإذا كان ممتلئًا يجب فك قفل Mutex قبل المغادرة، وإلا سيتركه الخيط مقفلًا فلا يستطيع أي خيطٍ آخر أن يستأنف عمله. شيفرة التزامن للدالة queue_pop هي:

```
int queue_pop(Queue *queue)
{
    mutex_lock(queue->mutex);
    if (queue_empty(queue))
    {
        mutex_unlock(queue->mutex);
        perror_exit("queue is empty");
    }

    int item = queue->array[queue->next_out];
    queue->next_out = queue_incr(queue, queue->next_out);
    mutex_unlock(queue->mutex);
    return item;
}
```

لاحظ أن دوال Queue الأخرى والتي هي queue_full وqueue_empty وqueue_incr لا تحاول قفل كائن المزامنة، فيجب على كل خيط يستدعي هذه الدوال أن يقفل كائن المزامنة أولًا. أصبح الطابور محافظًا على الخيوط باستخدام الشيفرة الجديدة التي أضيفت، ولا يجب أن ترى أخطاء تزامن إذا شغلت هذه الشيفرة، ولكنك سترى أن الخيط المستهلك يغادر أحيانًا لأن الطابور فارغ، أو قد ترى الخيط المنتج يغادر بسبب أن الطابور ممتلئ أو كلا الأمرين معًا، وبالتالي الخطوة القادمة هي إضافة المتغيرات الشرطية.

10.4 المتغيرات الشرطية Condition variables

المتغير الشرطي هو عبارة عن بينة بيانات مرتبطة بشرط، ويسمح المتغير الشرطي بإيقاف الخيوط حتى يتحقق الشرط أو تصبح قيمته true، فقد تتحقق الدالة thread_pop على سبيل المثال فيما إذا كان الطابور فارغًا أم لا، فإذا كان فارغًا تنتظر شرطًا هو (الطابور غير فارغ). وقد تتحقق الدالة thread_push أيضًا فيما إذا كان الطابور ممتلئًا، فإذا كان ممتلئًا تتوقف حتى يصبح غير ممتلئ. تعالج الشيفرة التالية الشرط الأول، حيث تضيف أولًا متغيرًا شرطيًا إلى البنية Queue:

```
typedef struct
{
    int *array;
    int length;
    int next_in;
    int next_out;
    Mutex *mutex;
    Cond *nonempty; //-- جديد
} Queue;
```

ثم تهيئه في الدالة make_queue:

```
Queue *make_queue(int length)
{
    Queue *queue = (Queue *)malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *)malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_mutex();
    queue->nonempty = make_cond(); //-- جديد
    return queue;
}
```

إذا وجدت الطابور فارغًا في الدالة queue_pop لا تغادر بل استخدم المتغير الشرطي لتوقف التنفيذ:

```
int queue_pop(Queue *queue)
{
    mutex_lock(queue->mutex);
    while (queue_empty(queue))
```

```
    {
        cond_wait(queue->nonempty, queue->mutex); //-- جديد
    }
    int item = queue->array[queue->next_out];
    queue->next_out = queue_incr(queue, queue->next_out);
    mutex_unlock(queue->mutex);
    cond_signal(queue->nonfull); //-- جديد
    return item;
}
```

الدالة `cond_wait` معقدة، فوسيطها الأول هـو المتغـير الشـرطي والشـرط الـذي يجب انتظـاره في هـذه الحالة هو (الطابور غير فارغ)، أما وسيطها الثاني هو كائن المزامنة الذي يحمي الطـابور. يفـك الخيـط قفـل كـائن المزامنة ثم يتوقف عندما يستدعي الخيط الذي قفل كائن المزامنة الدالةَ `cond_wait`، وهـذا شـيء مهم جـدًا. إذا لم تقفل الدالة `cond_wait` كائن المزامنة قبل التوقف فلن يستطيع أي خيطٍ آخر أن يصل إلى الطـابور ولن تضاف أي عناصر أخرى إلى الطابور، وبالتالي قد يبقى الطابور فارغًا دائمًا، فيمكن أن يشـغّل المنتج بينمـا يكـون المستهلك متوقفًا عند `nonempty`.

تبين الشيفرة التالية ما يحدث عنما يشغّل المنتج الدالة `queue_push`:

```
void queue_push(Queue *queue, int item)
{
    mutex_lock(queue->mutex);
    if (queue_full(queue))
    {
        mutex_unlock(queue->mutex);
        perror_exit("queue is full");
    }
    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
    mutex_unlock(queue->mutex);
    cond_signal(queue->nonempty); //-- جديد
}
```

تقفل الدالـة `queue_push` المتغـير `Mutex` وتتحقـق فيمـا إذا كـان الطـابور ممتلئًا أم لا، وعلى فـرض أن الطـابور ليس ممتلئًا حيث تضـيف الدالـة `queue_push` عنصـرًا جديـدًا إلى الطـابور ثم تفـك قفـل المتغـير `Mutex`، ولكن تقوم هذه الدالة بشيء آخر قبل أن تعيد شيئًا، حيث تنبّه signals المتغير الشرطي `nonempty`،

ويحدد تنبيه Signalling المتغير الشرطي أن الشرط صحيح true، وليس لإشارة التنبيه أي تأثير إذا لم يوجد خيوطٌ تنتظر المتغير الشرطي.

إذا وجد خيوط تنتظر المتغير الشرطي فيعود أحد هذه الخيوط إلى العمل ويستأنف تنفيذ الدالة `cond_wait`، ولكن يجب على الخيط الذي استأنف عمله أن ينتظر المتغير `Mutex` ويقفله مرة أخرى قبل أن ينهي تنفيذ الدالة `cond_wait`.

عُد الآن إلى الدالة `queue_pop` وشاهد ما يحدث عندما ينهي الخيط تنفيذ الدالة `cond_wait`، حيث يعود الخيط مرة أخرى إلى بداية حلقة while ويتحقق من الشرط مرة أخرى. افترض أنه تحقق الشرط أي أن الطابور غير فارغ، فعندما يغادر الخيط المستهلك حلقة while، هذا يؤدي إلى شيئين:

1. تحقق الشرط أي يوجد عنصر واحد على الأقل في الطابور.
2. قُفِل المتغير `Mutex` أي أن الوصول إلى الطابور آمن.

تفك الدالة `queue_pop` قفل كائن المزامنة وتنتهي بعد حذف عنصر من الطابور، سأبين، يقول الكاتب، كيفية عمل شيفرة `Cond` ولكن يجب أولًا الإجابة عن سؤالين مهمين هما:

لماذا توجد الدالة `cond_wait` ضمن حلقة `while` بدلًا من وجودها ضمن عبارة `if`، أي لماذا يجب التحقق من الشرط مرة أخرى بعد انتهاء تنفيذ الدالة `cond_wait`؟

السبب الرئيسي لإعادة التحقق من الشرط هو إمكانية اعتراض إشارة تنبيه، حيث افترض أن الخيط A ينتظر `nonempty`، ويضيف الخيط B عنصرًا إلى الطابور ثم ينبه `nonempty`، فيستيقظ الخيط A ويحاول قفل كائن المزامنة، ولكن قبل أن يقوم الخيط A بذلك، يأتي الخيط الشرير C ويقفل كائن المزامنة ويسحب عنصرًا من الطابور ثم يفك قفل كائن المزامنة، وبالتالي أصبح الطابور فارغًا الآن مرة أخرى، ولكن لا يتوقف الخيط A مرة أخرى، ويستطيع الخيط A قفل كائن المزامنة وينهي تنفيذ الدالة `cond_wait`، فإذا لم يتحقق الخيط A من الشرط مرة أخرى فقد يحاول سحب عنصر من طابورٍ فارغ، وقد يسبب ذلك خطأ.

أما السؤال الثاني الذي يظهر عندما يتعلم الناس المتغيرات الشرطية هو:

كيف يعرف المتغير الشرطي الشرطَ الذي يتعلق به؟

هذا السؤال مفهوم لأنه لا يوجد اتصال صريح بين بنية `Cond` والشرط المتعلق بها، حيث يكون الاتصال مضمّنًا في طريقة استخدامه، وهذه إحدى الطرق للتفكير في ذلك: فالشرط المتعلق ب `Cond` هو الشيء الذي تكون قيمته خاطئة false عندما تستدعي الدالة `cond_wait`، وتكون قيمته صحيحة true عندما تستدعي الدالة `cond_signal`.

ليس من الضروري تمامًا استدعاء الدالة cond_signal فقط عندما يكون الشرط صحيحًا، نظرًا لأن الخيوط يجب أن تتحقق من الشرط عند انتهاء تنفيذها للدالة cond_wait. إذا كان لديك سبب للاعتقاد بأن الشرط قد يكون صحيحًا فيمكنك استدعاء الدالة cond_signal كاقتراحٍ للتحقق من ذلك.

10.5 تنفيذ المتغير الشرطي

البنية Cond التي استخدمت في الفقرة السابقة هي مغلّف لنوعٍ يدعى pthread_cond_t المعرّف في واجهة برمجة التطبيقات للخيوط POSIX. البنية Cond شبيهة جدًا بالبنية Mutex والتي هي مغلّفة للنوع pthread_mutex_t، حيث تعريف النوع Cond كما يلي:

```
typedef pthread_cond_t Cond;
```

تخصص الدالة make_cond حيزًا وتهيئ المتغير الشرطي وتعيد مؤشرًا:

```
Cond *make_cond()
{
    Cond *cond = check_malloc(sizeof(Cond));
    int n = pthread_cond_init(cond, NULL);
    if (n != 0)
        perror_exit("make_cond failed");

    return cond;
}
```

أما الدالتان المغلّفتان للدالتين cond_wait وcond_signal:

```
void cond_wait(Cond *cond, Mutex *mutex)
{
    int n = pthread_cond_wait(cond, mutex);
    if (n != 0)
        perror_exit("cond_wait failed");
}
void cond_signal(Cond *cond)
{
    int n = pthread_cond_signal(cond);
    if (n != 0)
        perror_exit("cond_signal failed");
}
```

11. متغيرات تقييد الوصول Semaphores

متغيرات تقييد الوصول Semaphores طريقةٌ جيدة للتعرف على التزامن، ولكنها ليست مستخدمة على نطاق واسع من الناحية العملية كاستخدام كائنات المزامنة mutexes والمتغيرات الشرطية، ومع ذلك توجد بعض مشاكل المزامنة التي يمكن حلها ببساطة باستخدام متغيرات تقييد الوصول، مما يؤدي إلى الوصول إلى حلول صحيحة ودقيقة.

يقدم هذا الفصل واجهة برمجة التطبيقات بلغة C للعمل مع متغيرات تقييد الوصول، وكتابة تطبيق لمتغير تقييد الوصول semaphore باستخدام كائنات المزامنة mutexes والمتغيرات الشرطية.

11.1 معيار POSIX لمتغيرات تقييد الوصول

متغير تقييد الوصول semaphore هو بنية بيانات تُستخدم لمساعدة الخيوط أن تعمل مع بعضها البعض دون تداخلٍ فيما بينها، يحدد POSIX القياسي واجهةً لمتغيرات تقييد الوصول، وهي ليست جزءًا من الخيوط Pthreads، ولكن توفّر معظم نظم التشغيل التي تعتمد على يونكس والتي تطبق Pthreads متغيرات تقييد الوصول أيضًا.

لمتغيرات تقييد الوصول POSIX نوع هو sem_t، ووضع مغلّف له لجعل استخدامه أسهل كالعادة:

```
typedef sem_t Semaphore;

Semaphore *make_semaphore(int value);
void semaphore_wait(Semaphore *sem);
void semaphore_signal(Semaphore *sem);
```

Semaphore هـو مـرادف للنـوع sem_t، ولكنـني، يقـول الكـاتب، وجـدت Semaphore أسـهل للقـراءة وذكّرني الحرف الكبير في أوله بمعاملته ككائن object وتمريره كمؤشر pointer:

```
Semaphore *make_semaphore(int value)
{
    Semaphore *sem = check_malloc(sizeof(Semaphore));
    int n = sem_init(sem, 0, value);
    if (n != 0)
        perror_exit("sem_init failed");
    return sem;
}
```

تأخذ الدالة make_semaphore القيمة الابتدائية لمتغير تقييد الوصول كمعاملٍ لهـا، وتخصـص حـيزًا لـه وتهيئه ثم تعيد مؤشرًا إلى Semaphore.

تعيد الدالة sem_init القيمة 0 إذا نجح تنفيذها وتعيد -1 إذا حدث خطأ ما. أحد الأمور الجيـدة لاسـتخدام الدوال المغلّفة هو أنك تستطيع تغليـف encapsulate شـيفرة التحقـق من الخطـأ، ممـا يجعـل الشـيفرة الـتي تستخدم هذه الدوال أسهل للقراءة. يمكن تطبيق الدالة semaphore_wait كما يلي:

```
void semaphore_wait(Semaphore *sem)
{
    int n = sem_wait(sem);
    if (n != 0)
        perror_exit("sem_wait failed");
}
```

والدالة semaphore_signal:

```
void semaphore_signal(Semaphore *sem)
{
    int n = sem_post(sem);
    if (n != 0)
        perror_exit("sem_post failed");
}
```

أفضّل، يقول الكاتب، أن أسمي عملية تنبيه الخيط المتوقف بالمصطلح signal على أن أسميها بالمصطلح post على الرغم أن كلا المصطلحين شائعي الاستخدام. يظهر المثال التالي كيفية استخدام متغير تقييد الوصول ككائن مزامنة:

```
Semaphore *mutex = make_semaphore(1);

semaphore_wait(mutex);
// protected code goes here
semaphore_signal(mutex);
```

يجب أن تهيئ متغير تقييد الوصول الذي تستخدمه ككائن مزامنة بالقيمة 1 لتحدد أن كائن المزامنة غير مقفل، أي يستطيع خيطٌ واحد تمرير متغير تقييد الوصول دون توقف. استخدم اسم المتغير `mutex` للدلالة على أن متغير تقييد الوصول استخدم ككائن مزامنة، ولكن تذكّر أن سلوك متغير تقييد الوصول مختلف عن كائن مزامنة الخيط `Pthread`.

11.2 المنتجون والمستهلكون مع متغيرات تقييد الوصول

يمكن كتابة حل لمشكلة منتج-مستهلك Producers-consumers باستخدام دوال مغلّفة لمتغير تقييد الوصول semaphores، حيث يصبح التعريف الجديد للبنية `Queue` باستبدال كائن المزامنة والمتغيرات الشرطية بمتغيرات تقييد الوصول كما يلي:

```
typedef struct
{
    int *array;
    int length;
    int next_in;
    int next_out;
    Semaphore *mutex;  //-- جديد
    Semaphore *items;  //-- جديد
    Semaphore *spaces; //-- جديد
} Queue;
```

والنسخة الجديدة من الدالة `make_queue` هي:

```
Queue *make_queue(int length)
{
    Queue *queue = (Queue *)malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *)malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_semaphore(1);
    queue->items = make_semaphore(0);
    queue->spaces = make_semaphore(length - 1);
    return queue;
}
```

يُستخدم المتغير `mutex` لضمان الوصول الحصري إلى الطابور، حيث قيمته الابتدائية هي 1 وبالتالي كـائن المزامنة غير مقفل مبدئيًا.

المتغير `items` هو عدد العناصر الموجودة في الطابور والذي هو أيضًا عدد الخيوط المسـتهلكة الـتي يمكن أن تنفذ الدالة `queue_pop` دون توقف، ولا يوجد أي عنصر في الطابور مبدئيًا.

أما المتغير `spaces` فهو عدد المساحات الفارغة في الطابور وهو أيضًا عدد الخيوط المنتجة الـتي يمكن أن تنفّذ الدالة `queue_push` دون توقف، ويمثل عدد المساحات مبدئيًا سعة الطابور وتساوي `length-1`.

النسخة الجديدة من الدالة `queue_push` التي تشغّلها الخيوط المنتجة هي كما يلي:

```
void queue_push(Queue *queue, int item)
{
    semaphore_wait(queue->spaces);
    semaphore_wait(queue->mutex);

    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);

    semaphore_signal(queue->mutex);
    semaphore_signal(queue->items);
}
```

لاحظ أنه لا ينبغي على الدالة queue_push استدعاء الدالة queue_full مرة أخرى، حيث بدلًا من ذلك يتتبّع متغير تقييد الوصول عدد المساحات المتاحة ويوقف الخيوط المنتجة إذا كان الطابور ممتلئًا.

النسخة الجديدة من من الدالة queue_pop هي:

```
int queue_pop(Queue *queue)
{
    semaphore_wait(queue->items);
    semaphore_wait(queue->mutex);

    int item = queue->array[queue->next_out];
    queue->next_out = queue_incr(queue, queue->next_out);

    semaphore_signal(queue->mutex);
    semaphore_signal(queue->spaces);

    return item;
}
```

شُرح هذا الحل باستخدام شيفرة عامة (pseudo-code) في الفصل الرابع من كتاب The Little Book of Semaphores.

11.3 صناعة متغيرات تقييد وصول خاصة

أية مشكلةٍ تُحَل باستخدام متغيرات تقييد الوصول تُحل أيضًا باستخدام المتغيرات الشرطية وكائنات المزامنة، ويمكن إثبات ذلك من خلال استخدام المتغيرات الشرطية وكائنات المزامنة لتطبيق متغير تقييد الوصول، حيث يمكن تعريف البنية Semaphore كما يلي:

```
typedef struct
{
    int value, wakeups;
    Mutex *mutex;
    Cond *cond;
} Semaphore;
```

المتغير value هو قيمة متغير تقييد الوصول، ويحصي المتغير wakeups عدد التنبيهات المعلقة pending signals، أي عدد الخيوط التي تنبّهت ولكنها لم تستأنف تنفيذها بعد، والسبب وراء استخدام wakeups هو التأكد من أن متغيرات تقييد الوصول الخاصة بك لديها الخاصية 3 المشروحة في كتاب The Little Book of Semaphores.

يوفر المتغير mutex الوصول الحصري إلى لمتغيرين value وwakeups، المتغير cond هو المتغير الشرطي الذي تنتظره الخيوط إذا كانت تنتظر متغير تقييد الوصول. تمثل الشيفرة التالية شيفرة التهيئة للبنية Semaphore:

```
Semaphore *make_semaphore(int value)
{
    Semaphore *semaphore = check_malloc(sizeof(Semaphore));
    semaphore->value = value;
    semaphore->wakeups = 0;
    semaphore->mutex = make_mutex();
    semaphore->cond = make_cond();
    return semaphore;
}
```

11.4 تنفيذ متغير تقييد الوصول

تطبيقي، يقول الكاتب، لمتغيرات تقييد الوصول باستخدام كائنات المزامنة POSIX والمتغيرات الشرطية كما يلي:

```
void semaphore_wait(Semaphore *semaphore)
{
    mutex_lock(semaphore->mutex);
    semaphore->value--;

    if (semaphore->value < 0)
    {
        do
        {
            cond_wait(semaphore->cond, semaphore->mutex);
        } while (semaphore->wakeups < 1);
        semaphore->wakeups--;
```

```
    }
    mutex_unlock(semaphore->mutex);
}
```

يجب على الخيـط الـذي ينتظـر متغـير تقييـد الوصـول أن يقفـل كـائن المزامنـة قبـل إنقـاص قيمة المتغير value، وإذا أصبحت قيمة متغير تقييد الوصول سالبة سـيتوقف الخيـط حـتى يصـبح التنبـه لإيقاظـه wakeup متاحًا، وطالما الخيط متوقف فإن كائن المزامنة غير مقفل، وبالتالي يمكن أن يتنبه خيطٌ آخر.

شيفرة الدالة semaphore_signal هي:

```
void semaphore_signal(Semaphore *semaphore)
{
    mutex_lock(semaphore->mutex);
    semaphore->value++;

    if (semaphore->value <= 0)
    {
        semaphore->wakeups++;
        cond_signal(semaphore->cond);
    }
    mutex_unlock(semaphore->mutex);
}
```

يجب على الخيط مرة أخرى أن يقفل كائن المزامنة قبل زيادة قيمة المتغير value، وإذا كانت قيمة متغـير تقييد الوصول سالبة فهذا يعني أن الخيوط منتظرة، وبالتالي يزيد تنبيه الخيط قيمة المتغـير wakeups ثم ينبـه المتغير الشرطي، وعند ذلك قد تتنبه أحد الخيوط المنتظـرة ولكن يبقى كـائن المزامنـة مقفلًا حـتى يفـك الخيـط المتنبه قفله، وعند ذلك أيضًا تعيد أحد الخيوط المنتظرة من الدالـة cond_wait ثم تتحقـق من أن التنبيـه مـا زال متاحًا، فإذا لم يكن متاحًا فإن الخيط يعود وينتظر المتغير الشرطي مرة أخرى، أما إذا كان التنبيـه متاحًـا فـإن الخيط ينقص قيمة المتغير wakeups ويفك قفل كائن المزامنة ثم يغادر.

قد يوجد شيء واحد ليس واضحًا في هذا الحل وهـو اسـتخدام حلقـة do...while، هـل يمكنـك معرفـة سبب عدم كونها حلقة while تقليدية؟ وما الخطأ الذي سيحدث؟

المشكلة مع حلقة while هي أنه قد لا يملك هـذا التطـبيق الخاصـية 3، فمن الممكن أن يتنبـه الخيـط ثم يُشغّل ويلتقط تنبيهه الخاص. من المضمون مع حلقة do...while أن يلتقط أحـد الخيـوط المنتظـرة التنبيـه

الذي أنشأه خيطٌ ما، حتى إذا شُغّل خيط التنبيه وحصل على كائن المزامنة قبل استئناف أحد الخيـوط المنتظـرة، ولكن اتضـح أنـه يمكن أن ينتهـك التنبيـه الزائـف في الـوقت المناسـب spurious wakeup well-timed هذا الضمان.

أحدث إصدارات أكاديمية حسوب