Operating Systems

(نظم التشغيل)

المرحلة الرابعة

قسم علوم الحاسوب

كلية التربية للعلوم الصرفة

جامعة الموصل

2023-2022

Operating System

Operating System

An ***operating system*** is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing environments.

A fundamental responsibility of an operating system is to allocate the resources such as CPU, memory, and I/O devices, as well as storage to programs.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

نظام التشغيل هو برنامج يدير أجهزة الكمبيوتر. كما أنه يوفر أساسًا للبرامج التطبيقية ويعمل كوسيط بين مستخدم الكمبيوتر وأجهزة الكمبيوتر. توجد أنظمة التشغيل في كل مكان ، من السيارات والأجهزة المنزلية التي تتضمن أجهزة "إنترنت الأشياء" ، إلى الهواتف الذكية وأجهزة الكمبيوتر الشخصية وأجهزة الكمبيوتر الخاصة بالمؤسسات وبيئات الحوسبة السحابية.

تتمثل إحدى المسؤوليات الأساسية لنظام التشغيل في تخصيص الموارد مثل وحدة المعالجة المركزية والذاكرة وأجهزة الإدخال / الإخراج ، بالإضافة إلى التخزين للبرامج.

ونظرًا لأن نظام التشغيل كبير ومعقد ، يجب إنشاؤه قطعة قطعة. ويجب أن تكون كل من هذه الأجزاء جزءًا محددًا جيدًا من النظام ، مع مدخلات ومخرجات ووظائف محددة بعناية.

What Operating Systems Do?

A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and a ***user*** (Figure 1). The operating system controls the hardware and coordinates its use among the various application programs for the various users.

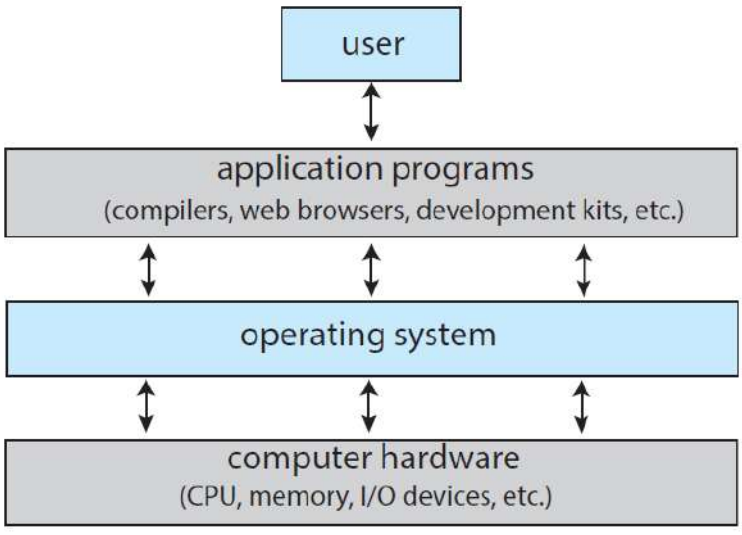


Figure 1: Abstract view of the components of a computer system

The operating system provides the means for proper use of these resources in the operation of the computer system. It simply provides an ***environment*** within which other programs can do useful work. We can explore operating system's role, From two viewpoints: that of the user and that of the system.

يمكن تقسيم نظام الكمبيوتر تقريبًا إلى أربعة مكونات: الجهاز ونظام التشغيل وبرامج التطبيق والمستخدم (الشكل 1). يتحكم نظام التشغيل في الأجهزة وينسق استخدامها بين برامج التطبيقات المختلفة لمختلف المستخدمين.وبذلك يوفر وسائل الاستخدام السليم لهذه الموارد في تشغيل نظام الكمبيوتر.وبالتالي يوفر البيئة التي يمكن للبرامج الأخرى من خلالها القيام بعمل مفيد.

يمكننا استكشاف دور نظام التشغيل من وجهتي نظرالمستخدم والنظام.

1. User View

The user's view of the computer varies according to the interface being used (monitor, keyboard, mouse, touch screen, and voice recognition).

In this case, the operating system is designed mostly for ***ease of use***, with some attention paid to performance and security and none paid to ***resource utilization***—how various hardware and software resources are shared.

تختلف نظرة المستخدم للكمبيوتر وفقًا للواجهة المستخدمة (الشاشة ولوحة المفاتيح والماوس وشاشة اللمس والتعرف على الصوت). في هذه الحالة ، تم تصميم نظام التشغيل في الغالب لسهولة الاستخدام ، مع منح بعض الاهتمام للأداء والأمان وعدم الاهتمام لاستخدام الموارد - كيفية مشاركة موارد الأجهزة والبرامج المختلفة.

2. System View

From the computer's point of view, an operating system can be viewed as a ***resource allocator***. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources by allocating them to specific programs and users so that it can operate the computer system efficiently and fairly. It also can be viewed as ***control program*** that manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

من وجهة نظر الكمبيوتر ، يمكن النظر إلى نظام التشغيل كمخصص للموارد (وقت وحدة المعالجة المركزية ومساحة الذاكرة ومساحة التخزين وأجهزة الإدخال / الإخراج وما إلى ذلك)المتوفرة في نظام الكومبيوتر والتي قد تكون مطلوبة لحل مشكلة ما. يعمل نظام التشغيل كمدير لهذه الموارد من خلال تخصيصها لبرامج ومستخدمين معينين حتى يتمكن من تشغيل نظام الكمبيوتر بكفاءة وعدالة. كما يمكن اعتباره برنامج تحكم يدير تنفيذ برامج المستخدم لمنع الأخطاء والاستخدام غير السليم للكمبيوتر. وفي النهاية فهو يهتم بشكل خاص بالتشغيل والتحكم في أجهزة الإدخال / الإخراج.

Operating-System Operations

For a computer to start running it needs to have an initial or bootstrap program to run. A bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the system, from CPU

registers to device controllers to memory contents. The bootstrap program must know where to locate the operating-system kernel and how to load it into memory.

لكي يبدأ الكمبيوتر في التشغيل ، يجب أن يكون لديه برنامج أولي أو برنامج(Bootstrap) ليتم تشغيله. يميل برنامج bootstrap إلى أن يكون بسيطًا وعادةً ما يتم تخزينه داخل أجهزة الكمبيوتر في (firmware). يقوم هذا البرنامج بتهيئة جميع جوانب النظام ، من سجلات وحدة المعالجة المركزية إلى وحدات التحكم في الجهاز إلى محتويات الذاكرة ويجب أن يعرف كذلك موقع نواة نظام التشغيل وكيفية تحميله في الذاكرة .

Once the kernel is loaded and executing, it can start providing services to the system and its users. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

وعند تحميل النواة وتنفيذها ، يمكنها البدء في تقديم الخدمات للنظام ومستخدميه. وباكتمال هذه المرحلة ،يكون قد تم تمهيد النظام بالكامل ، ويدخل النظام في انتظارحدوث الأحداث.

Multiprogramming and Multitasking

One of the most important aspects of operating systems is the ability to run multiple programs. Multiprogramming increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a process.

من أهم جوانب أنظمة التشغيل القدرة على تشغيل برامج متعددة. تزيد البرمجة المتعددة (Multiprogramming) من درجة استخدام وحدة المعالجة المركزية ، بالإضافة إلى إرضاء المستخدمين ، من خلال تنظيم البرامج بحيث يكون لدى وحدة المعالجة المركزية دائمًا برنامج لتنفيذه. في نظام متعدد البرامج (multiprogrammed system) ، يُطلق على البرنامج قيد التنفيذ اسم عملية.

The idea is as follows: The operating system keeps several processes in memory simultaneously (Figure 2). The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another process. When that process needs to wait, the CPU switches to another process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

وفكرة البرمجة المتعددة كما يلي: يحتفظ نظام التشغيل بعدة عمليات في الذاكرة في وقت واحد (الشكل 2). يختار نظام التشغيل إحدى هذه العمليات ويبدأ في تنفيذها. وفي لحظة ما قد تضطر العملية إلى الانتظار حتى تكتمل بعض المهام ، مثل عملية الإدخال / الإخراج. في نظام غير متعدد البرامج ، ستظل وحدة المعالجة المركزية في وضع الخمول اما في نظام متعدد البرامج فيقوم نظام التشغيل ببساطة بالتبديل إلى

عملية أخرى وتنفيذها بتحويل وحدة المعالجة المركزية إلى عملية أخرى ، وهكذا. في النهاية ستنتهي العملية الأولى من الانتظار وتستعيد وحدة المعالجة المركزية. وبالنتيجة فلن تكون وحدة المعالجة المركزية في وضع الخمول أبدًا طالما أن هناك عملية واحدة على الأقل تحتاج إلى التنفيذ..

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast response time.

تعدد المهام (Multitasking)هو امتداد منطقي للبرمجة المتعددة. ، تنفذ وحدة المعالجة المركزية في أنظمة تعدد المهام عمليات متعددة بالتبديل فيما بينها ، ولكون عمليات التحويل تحدث بشكل متكرر ، فان ذلك يوفر للمستخدم وقت استجابة سريعًا.

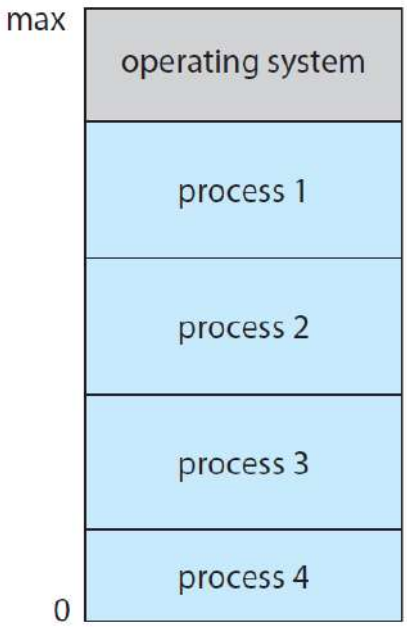


Figure 2: Memory layout for a multiprogramming system

When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. Since interactive I/O typically runs at “people speeds,” it may take a long time to complete. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

If several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is CPU scheduling. Finally, running multiple processes concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management.

عند تنفيذ عملية ، يتم تنفيذها عادةً لفترة قصيرة فقط قبل أن تنتهي أو تحتاج إلى إجراء عملية ادخال / إخراج. نظرًا لأن الإدخال / الإخراج التفاعلي يعمل عادةً "بسرعات الأشخاص" ، فقد يستغرق الأمر وقتًا

طويلاً حتى يكتمل ،وبدلاً من ترك وحدة المعالجة المركزية في وضع الخمول أثناء حدوث هذا الإدخال التفاعلي ، سيقوم نظام التشغيل بتحويل وحدة المعالجة المركزية بسرعة إلى عملية أخرى.

إذا كانت عدة عمليات جاهزة للتشغيل في نفس الوقت ، فيجب على النظام اختيار العملية التي سيتم تشغيلها بعد ذلك. واتخاذ هذا القرار هو جدولة وحدة المعالجة المركزية. يتطلب تشغيل عمليات متعددة بشكل متزامن أن يكون تأثير بعضها على البعض محدود في جميع مراحل نظام التشغيل ، بما في ذلك جدولة العمليات ، وتخزين القرص ، وإدارة الذاكرة.

Resource Management

As an operating system is a resource manager. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

كون نظام تشغيل هو مدير الموارد، فان وحدة المعالجة المركزية للنظام ومساحة الذاكرة ومساحة تخزين الملفات وأجهزة الإدخال / الإخراج من بين الموارد التي يجب على نظام التشغيل إدارتها.

1. Process Management

A program can do nothing unless its instructions are executed by a CPU. A program in execution, is a process. A programs such as a compiler, a word-processing program, and a social media app on a mobile device are processes. A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are typically allocated to the process while it is running. A single-threaded process has one program counter specifying the next instruction to execute. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code).

لا يمكن لاي برنامج ان يفعل شيئا ما لم يتم تنفيذ تعليماته بواسطة وحدة المعالجة المركزية(CPU).
البرنامج تحت التنفيذ هو process. برامج مثل المترجم ، وبرنامج معالجة الكلمات ، وتطبيق الوسائط الاجتماعية على جهاز محمول هي processes. تحتاج الـ process إلى موارد معينة - بما في ذلك وقت وحدة المعالجة المركزية والذاكرة والملفات وأجهزة الإدخال / الإخراج - لإنجاز مهمتها. وعادةً ما يتم تخصيص هذه الموارد للعملية أثناء تشغيلها. تحتوي العملية أحادية الخيوط single-threaded على عداد برنامج واحد يحدد التعليمات التالية للتنفيذ. وبالتالي ، على الرغم من أن عمليتين (two processes) قد ترتبطان ببرنامج واحد ، إلا أنهما تعتبران مع ذلك تسلسلي تنفيذ منفصلين. العملية (process) هي وحدة العمل في النظام. يتكون النظام من مجموعة من العمليات ، بعضها عبارة عن عمليات نظام تشغيل (تلك التي تنفذ كود النظام) والبقية منها عمليات مستخدم (تلك التي تنفذ كود المستخدم).
The operating system is responsible for the following activities in connection with process management:

• Creating and deleting both user and system processes

• Scheduling processes and threads on the CPUs

• Suspending and resuming processes

• Providing mechanisms for process synchronization

• Providing mechanisms for process communication

ان نظام التشغيل مسؤول عن الأنشطة التالية فيما يتعلق بإدارة العملية:

- إنشاء وحذف كل من عمليات المستخدم والنظام
- جدولة العمليات والخيوط على وحدات المعالجة المركزية
- تعليق واستئناف العمليات
- توفير آليات لمزامنة العملية
- توفير آليات لعملية الاتصال

2. Memory Management

Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. The main memory is generally the only large storage device that the CPU is able to address and access directly. To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the hardware design of the system. Each algorithm requires its own hardware support.

الذاكرة الرئيسية عبارة عن مجموعة كبيرة من البايتات ، يتراوح حجمها من مئات الآلاف إلى مليارات. فهي بشكل عام جهاز التخزين الكبير الوحيد الذي تستطيع وحدة المعالجة المركزية معالجته والوصول إليه مباشرة. ولتحسين استخدام وحدة المعالجة المركزية وسرعة استجابة الكمبيوتر لمستخدميه ، يجب أن تحتفظ أجهزة الكمبيوتر ذات الأغراض العامة بعدة برامج في الذاكرة ، مما يخلق الحاجة إلى إدارة الذاكرة. عند اختيار مخطط إدارة الذاكرة لنظام معين ، يجب أن نأخذ في الاعتبار العديد من العوامل - خاصة تصميم الأجهزة للنظام. تتطلب كل خوارزمية دعم الأجهزة الخاصة بها..

The operating system is responsible for the following activities in connection with memory management:

• Keeping track of which parts of memory are currently being used and

which process is using them

• Allocating and deallocating memory space as needed

• Deciding which processes (or parts of processes) and data to move into

and out of memory

ان مسؤولية نظام التشغيل فيما يتعلق بإدارة الذاكرة تكون عن الأنشطة التالية:

• تتبع أي أجزاء من الذاكرة يتم استخدامها حاليًا والعملية التي تستخدمها

• تخصيص مساحة الذاكرة وإلغاء تخصيصها حسب الحاجة

• تحديد العمليات (أو أجزاء العمليات) والبيانات المطلوب الانتقال إليها ونفاد الذاكرة

3. File-System Management

Computers can store information on several different types of physical media. Secondary storage is the most common, but tertiary storage is also possible. Each of these media has its own characteristics and physical organization. Most are controlled by a device, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random). A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. files are normally organized into directories to make them easier to use.

يمكن لأجهزة الكمبيوتر تخزين المعلومات على عدة أنواع مختلفة من الوسائط المادية. التخزين الثانوي (secondary storage) هو الأكثر شيوعًا ، لكن التخزين الثلاثي (territory storage) ممكن أيضًا. ان كل من هذه الوسائط لها خصائصها وتنظيمها المادي و يتم التحكم في معظمها بواسطة جهاز يمتلك خصائصه المعينة. تتضمن هذه الخصائص سرعة الوصول والسعة ومعدل نقل البيانات وطريقة الوصول (المتسلسلة أو العشوائية). ان الملف عبارة عن مجموعة من المعلومات ذات الصلة التي يحددها منشئها. بشكل عام ، تمثل الملفات البرامج (نماذج المصدر والعناصر) والبيانات. وعادة ما يتم تنظيم الملفات في أدلة لتسهيل استخدامها.

The operating system is responsible for the following activities in connection with file management:

• Creating and deleting files

• Creating and deleting directories to organize files

• Supporting primitives for manipulating files and directories

• Mapping files onto mass storage

• Backing up files on stable (nonvolatile) storage media

ويكون نظام التشغيل مسؤول عن الأنشطة التالية فيما يتعلق بإدارة الملفات:

• إنشاء وحذف الملفات

• إنشاء وحذف أدلة لتنظيم الملفات

• دعم الأساسيات لمعالجة الملفات والدلائل

• وضع وتعيين الملفات على مساحة التخزين الكبيرة

• النسخ الاحتياطي للملفات على وسائط تخزين مستقرة (غير متطايرة)

4. Mass-Storage Management

As we have already seen, the computer system must provide secondary storage to back up main memory. Most modern computer systems use HDDs and NVM devices as the principal on-line storage media for both programs and data.

كما لاحظنا سابقا ، يجب أن يوفر نظام الكمبيوتر تخزينًا ثانويًا لعمل نسخة احتياطية من الذاكرة الرئيسية. تستخدم معظم أنظمة الكمبيوتر الحديثة محركات الأقراص الثابتة وأجهزة NVM كوسائط تخزين أساسية على الإنترنت لكل من البرامج والبيانات.

The operating system is responsible for the following activities in connection with secondary storage management:

• Mounting and unmounting

• Free-space management

• Storage allocation

• Disk scheduling

• Partitioning

• Protection

ويكون نظام التشغيل مسؤول عن الأنشطة التالية فيما يتعلق بإدارة التخزين الثانوي:

• التركيب والفك

• إدارة المساحة الحرة

• تخصيص التخزين

• جدولة القرص

•التقسيم

• الحماية

At the same time, there are many uses for storage that is slower and lower in cost (and sometimes higher in capacity) than secondary storage. Examples of these storage devices are magnetic tape drives and their tapes and CD DVD and Blu-ray drives and platters.

في الوقت نفسه ، هناك العديد من الاستخدامات للتخزين التي تكون أبطأ وأقل تكلفة (وأحيانًا أعلى في السعة) من التخزين الثانوي. ومن أمثلة أجهزة التخزين هذه محركات الأشرطة الممغنطة وأشرطتها ومحركات الأقراص المضغوطة وDVD و Blu-ray والأطباق.

5. Cache Management

Caching is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon. Because caches have limited size, ***cache management*** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance.

التخزين المؤقت هو مبدأ مهم لأنظمة الكمبيوتر. ويعمل كالاتي، تحفظ المعلومات عادة في بعض أنظمة التخزين (مثل الذاكرة الرئيسية). وعن استخدامها ، يتم نسخها إلى نظام تخزين أسرع - ذاكرة التخزين المؤقت - على أساس مؤقت. عندما نحتاج إلى معلومة معينة ، يتم التحقق أولاً من وجودها في ذاكرة التخزين المؤقت. فان وجدت ، فانها تستخدم مباشرة من ذاكرة التخزين المؤقت. وان لم تكن موجودة ، فإن المعلومات تستخدم من المصدر ، وتوضع نسخة في ذاكرة التخزين المؤقت على افتراض أننا سنحتاجها مرة أخرى قريبًا. نظرًا لأن ذاكرات التخزين المؤقت ذات حجم محدود ، فإن إدارة ذاكرة التخزين المؤقت تعد مشكلة تصميمية مهمة. يمكن أن يؤدي التحديد الدقيق لحجم ذاكرة التخزين المؤقت وسياسة الاستبدال إلى زيادة الأداء بشكل كبير.

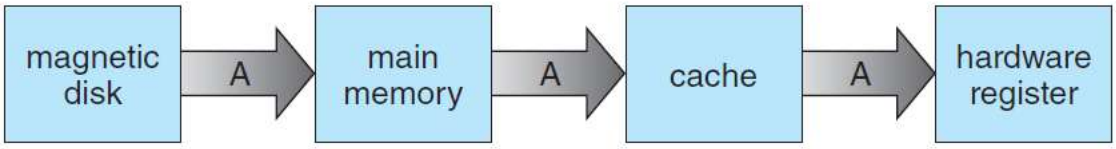


Figure 3: Migration of integer A form disk to register

Virtualization

Virtualization is a technology that allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer.

These environments can be viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other. A user of a ***virtual machine*** can switch among the various operating systems in the same way a user can switch among the various processes running concurrently in a single operating system. On laptops and desktops, a VMM allows the user to install multiple operating systems for

exploration or to run applications written for operating systems other than the native host.

المحاكاة الافتراضية هي تقنية تسمح لنا بالاستفادة من أجهزة كمبيوتر واحد (وحدة المعالجة المركزية ، والذاكرة ، ومحركات الأقراص ، وبطاقات واجهة الشبكة ، وما إلى ذلك) في انشاء العديد من بيئات التنفيذ المختلفة ، وبالتالي خلق الوهم بأن كل بيئة منفصلة تعمل على جهازها. الكمبيوتر الخاص.

يمكن النظر إلى هذه البيئات على أنها أنظمة تشغيل فردية مختلفة (على سبيل المثال ، Windowsو UNIX)والتي قد تعمل في نفس الوقت وقد تتفاعل مع بعضها البعض. يمكن لمستخدم الجهاز الظاهري التبديل بين أنظمة التشغيل المختلفة بنفس الطريقة التي يمكن بها للمستخدم التبديل بين العمليات المختلفة التي تعمل بشكل متزامن في نظام تشغيل واحد. يسمح VMM للمستخدم بتثبيت أنظمة تشغيل متعددة على أجهزة الكمبيوتر المحمولة وأجهزة سطح المكتب ، للاستكشاف أو لتشغيل التطبيقات المكتوبة لأنظمة تشغيل غير المضيف الأصلي.

Distributed Systems

A ***distributed system*** is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver.

A ***network operating system*** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages.

الانظمة الموزعة

النظام الموزع عبارة عن مجموعة من أنظمة الكمبيوتر المنفصلة ماديًا ، وربما غير المتجانسة ، والمتصلة بالشبكة لتزويد المستخدمين بإمكانية الوصول إلى الموارد المختلفة للنظام. يؤدي الوصول إلى مورد مشترك إلى زيادة سرعة الحساب ورفع الاداء الوظيفي وتوافر البيانات والموثوقية. تعامل بعض أنظمة التشغيل الوصول إلى الشبكة كشكل من أشكال الوصول إلى الملفات ، مع استخدام التفاصيل المتعلقة بالشبكات المحتواة في برنامج تشغيل جهاز واجهة الشبكة.

نظام تشغيل الشبكة هو نظام تشغيل يوفر ميزات مثل مشاركة الملفات عبر الشبكة ، إلى جانب اسلوب اتصال يسمح لعمليات مختلفة على أجهزة كمبيوتر مختلفة بتبادل الرسائل.

Linkers and Loaders

Usually, a program resides on disk as a binary executable file—for example, a.out or prog.exe. To run on a CPU, the program must be brought into memory and placed in the context of a process. Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an ***relocatable object file*** . Next, the ***linker*** combines these relocatable object files into a single binary ***executable file***. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library (specifiedwith the flag -lm).

عادةً ما يوجد برنامج على القرص كملف ثنائي قابل للتنفيذ executable file - على سبيل المثال ، a.out أو prog.exe. للتشغيل على وحدة المعالجة المركزية ، يجب إحضار البرنامج إلى الذاكرة ووضعه في سياق العملية. تتم ترجمة ملفات source files الى ملفات object files مصممة ليتم تحميلها في أي موقع في الذاكرة ، بتنسيق يُعرف باسم relocatable object file. بعد ذلك ، يجمع الـ Linker ملفات relocatable object files في ملف Executable واحد أثناء مرحلة الربط ، يمكن أيضًا تضمين ملفات أو مكتبات كائنات أخرى ، مثل مكتبة C القياسية أو مكتبة الرياضيات.

A ***loader*** is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is ***relocation***, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses. When the icon associated with the executable file is double-clicked, the loader is invoked and then it loads the specified program into memory using the address space of the newly created process.

يستخدم Loader لتحميل الملف الثنائي القابل للتنفيذ binary executable file في الذاكرة ، حيث يكون مؤهلاً للتشغيل على نواة وحدة المعالجة المركزية. ان الـ relocation هونشاط مرتبط بالربط والتحميل وهو الذي يخصص العناوين النهائية لأجزاء البرنامج ويضبط الكود والبيانات في البرنامج لمطابقة تلك العناوين. يتم استدعاء المحمل عندما يتم النقر نقرًا مزدوجًا فوق الرمز المرتبط بالملف القابل للتنفيذ ، ليقوم بتحميل البرنامج المحدد في الذاكرة باستخدام مساحة العنوان للعملية التي تم إنشاؤها حديثًا.

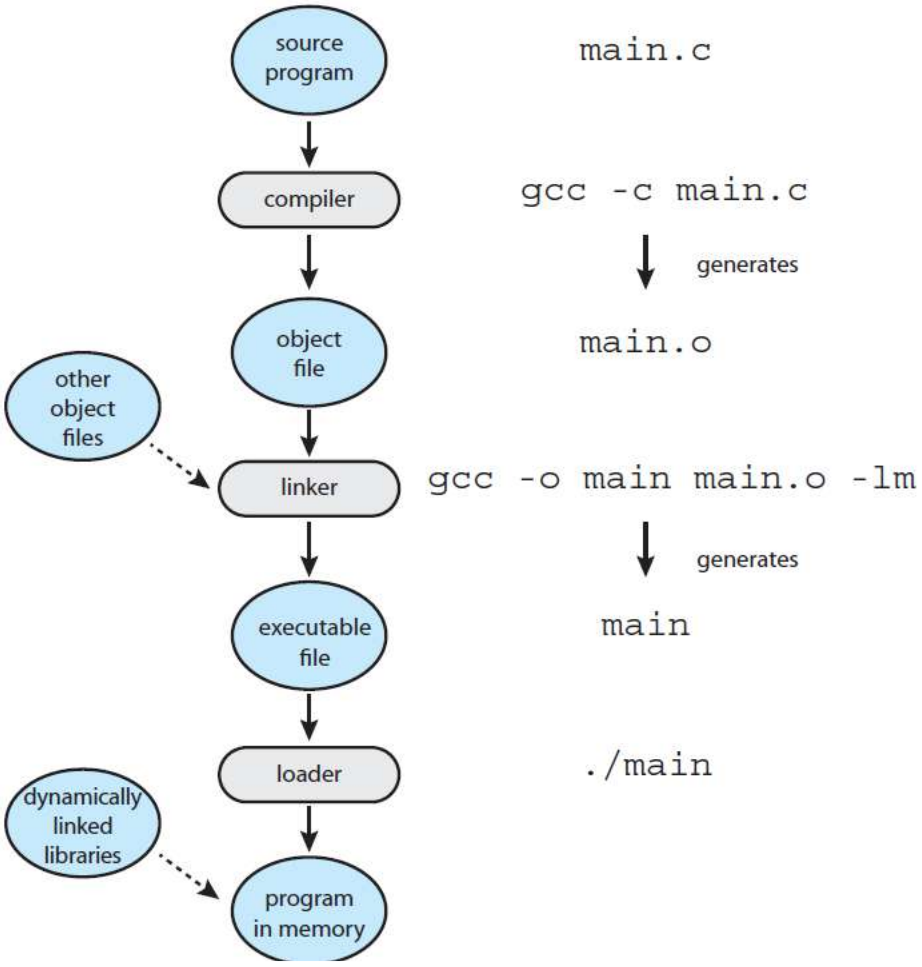


Figure 4: The role of linker and loader

The process described thus far assumes that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to dynamically link libraries as the program is loaded. Windows, for instance, supports dynamically linked libraries (***DLLs***). In this approach the library is conditionally linked and is loaded if it is required during program run time.
Object files and executable files typically have standard formats that include the compiled machine code and a symbol table containing metadata about functions and variables that are referenced in the program. For UNIX and Linux systems, this standard format is known as ELF (for ***Executable and Linkable Format***). Windows systems use the ***Portable Executable*** (PE) format, and macOS uses the Mach-O format.

تفترض العملية الموصوفة اعلاه أن جميع المكتبات تربط بملف executable وتحمل في الذاكرة. في الواقع ، تسمح معظم الأنظمة للبرنامج بربط المكتبات ديناميكيًا أثناء تحميل البرنامج. فـ Windowsعلى سبيل المثال ، يدعم المكتبات المرتبطة ديناميكيًا (DLLs) .في هذا النهج ،يتم ربط المكتبة بشكل مشروط ويتم تحميلها إذا كانت مطلوبة أثناء وقت تشغيل البرنامج.

عادةً ما يكون لملفات object files والملفات executable files تنسيقات قياسية والتي تشمل كود الآلة المترجم وجدول رموز يحتوي على بيانات أولية حول الوظائف والمتغيرات المشار إليها في البرنامج. بالنسبة لأنظمة UNIX و Linux ، يُعرف هذا التنسيق القياسي باسم ELF للتنسيق القابل للتنفيذ والقابل للربط). تستخدم أنظمة Windows تنسيق Portable Executable (PE) ، ويستخدم macOS تنسيق Mach-O.

Processes

The Process

As mentioned earlier, a process is a program in execution. The status of the current activity of a process is represented by the value of the ***program counter*** and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 1. These sections include:

- ***Text section***—the executable code
- ***Data section***—global variables
- ***Heap section***—memory that is dynamically allocated during program run time
- ***Stack section***—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

Although the stack and heap sections grow toward one another, the operating system must ensure they do not overlap one another.

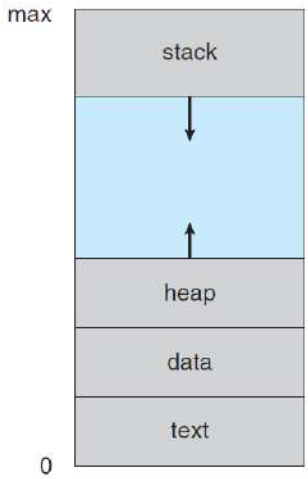


Figure 1: Layout of a process in memory

كما ذكر سابقًا ،الـ process عبارة عن برنامج قيد التنفيذ. يتم تمثيل حالة النشاط الحالي للعملية بقيمة عداد البرنامج ومحتويات سجلات المعالج. عادةً ما يتم تقسيم حيزالذاكرة لعملية ما إلى أقسام متعددة ، كما هو موضح في الشكل 1. وتشمل هذه الأقسام:

- Text section - الكود القابل للتنفيذ
- Data section – global variables
- Heap section - الذاكرة التي يتم تخصيصها ديناميكيًا أثناء وقت تشغيل البرنامج
- Stack section - تخزين مؤقت للبيانات عند استدعاء وظائف (مثل معلمات الوظيفة وعناوين الإرجاع والمتغيرات المحلية)

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate

process; and although the text sections are equivalent, the data, heap, and stack sections vary.

على الرغم من أنه قد ترتبط اثنتان من الـ processes بالبرنامج نفسه ، إلا أنهما يعتبران تسلسلي تنفيذ منفصلين. على سبيل المثال ، قد يقوم العديد من المستخدمين بتشغيل نسخ مختلفة من برنامج البريد ، أو قد يقوم نفس المستخدم باستدعاء نسخ عديدة من برنامج مستعرض الويب. كل من هذه هي process منفصلة ؛ وعلى الرغم من أن أقسام النص هي نفسها ، الا انه تختلف أقسام الـ data والـ heap والـ stack.

Process State

As a process executes, it changes ***state***. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- ***New***. The process is being created.
- ***Running***. Instructions are being executed.
- ***Waiting***. The process is waiting for some event to occur (such as an I/O
- completion or reception of a signal).
- ***Ready***. The process is waiting to be assigned to a processor.

اثناء تنفيذ ال process ، فان حالتها تتغير من حالة الى حالة. يتم تحديد حالة ال process جزئيًا من خلال النشاط الحالي لتلك الprocess. قد تكون الـ process في إحدى الحالات التالية:

- New. يتم إنشاء الـ process .
- Running. يتم تنفيذ التعليمات.
- Waiting. تنتظر الـ process حدوث بعض الأحداث (مثل إدخال / إخراج استكمال أو استقبال إشارة.)
- Ready. تنتظر الـ process أن يتم تخصيصها للمعالج.

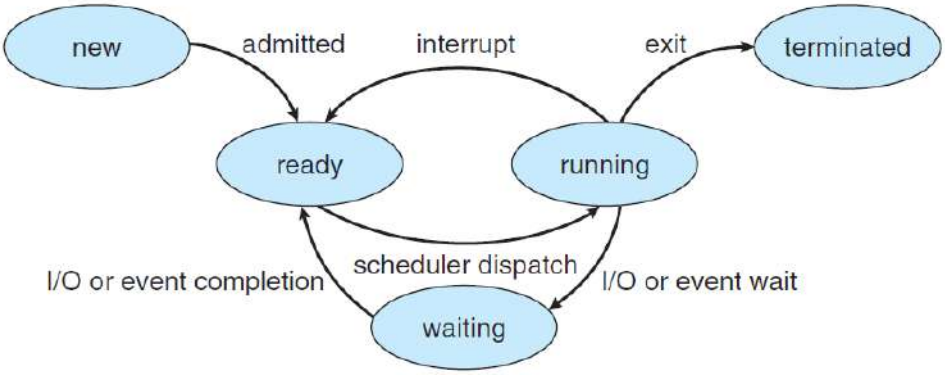


Figure 2: Diagram of process state

Process Control Block

Each process is represented in the operating system by a ***process control block (PCB)***—also called a ***task control block***. A PCB is shown in Figure 3. It contains many pieces of information associated with a specific process, including these:

يتم تمثيل كل عملية في نظام التشغيل بواسطة كتلة التحكم في العملية (PCB)تسمى أيضًا كتلة التحكم في المهام. تظهر الـ PCB في الشكل 3. ويحتوي على العديد من المعلومات المرتبطة بـprocess معينة ، بما في ذلك:

- ***Process state***. The state may be new, ready, running, waiting, halted, and so on.
- ***Program counter***. The counter indicates the address of the next instruction to be executed for this process.

Figure 3: Process Control Block(PCB)

CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

- ***CPU-scheduling information***. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- ***Memory-management information***. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- ***Accounting information***. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- ***I/O status information***. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

- Process state. قد تكون الحالة جديدة ، وجاهزة ، وعاملة ، ومنتظرة ، ومتوقفة ، وهكذا.
- Program counter يشير العداد إلى عنوان التعليمات التالية التي سيتم تنفيذها لهذه الـ process.
- سجلات وحدة المعالجة المركزية. تختلف السجلات من حيث العدد والنوع ، اعتمادًا على بنية الكمبيوتر. وهي تشمل المُجمِّعات ، وسجلات الفهرس ، ومؤشرات المكدس ، وسجلات الأغراض العامة. إلى جانب عداد البرنامج ، يجب حفظ معلومات الحالة هذه عند حدوث مقاطعة ، للسماح للعملية بالاستمرار بشكل صحيح بعد ذلك عند إعادة جدولتها للتشغيل.
- معلومات جدولة وحدة المعالجة المركزية. تتضمن هذه المعلومات أولوية العملية ، ومؤشرات لجدولة قوائم الانتظار ، وأي معلمات جدولة أخرى.
- معلومات إدارة الذاكرة. قد تتضمن هذه المعلومات عناصر مثل قيمة السجلات الأساسية والحدود وجداول الصفحات ، أو جداول المقطع ، اعتمادًا على نظام الذاكرة المستخدم بواسطة نظام التشغيل).
- المعلومات المحاسبية. تتضمن هذه المعلومات مقدار وحدة المعالجة المركزية والوقت الفعلي المستخدم ، والحدود الزمنية للاستخدام ، وأرقام الحسابات ، وأرقام العمليات ، وما إلى ذلك.
- معلومات حالة الإدخال / الإخراج. تتضمن هذه المعلومات قائمة بأجهزة الإدخال / الإخراج المخصصة للعملية ، وقائمة بالملفات المفتوحة ، وما إلى ذلك.

باختصار ، يعمل PCB ببساطة كمستودع لجميع البيانات اللازمة لبدء أو إعادة تشغيل عملية ما، إلى جانب بعض البيانات المحاسبية.

Process Scheduling

To meet the objectives of multiprogramming and multitasking, the ***process scheduler*** selects an available process (possibly from a set of several available processes) for program execution on a core. Each CPU core can run one process at a time. If there are more processes than cores, excess processes will have to wait until a core is free and can be rescheduled. The number of processes currently in memory is known as the ***degree of multiprogramming***.

لتحقيق أهداف البرمجة المتعددة والمهام المتعددة ، يختار مجدول العملية العملية المتاحة (ربما من مجموعة من العمليات المتاحة المتعددة) لتنفيذ البرنامج على النواة. يمكن لكل نواة من نوى الـ CPU تشغيل عملية واحدة في الوقت الواحد. إذا كانت هناك عمليات أكثر من النوى ، فعلى العمليات الزائدة الانتظار حتى يتم تحرير النواة و عندها يمكن إعادة جدولتها. يُعرف عدد العمليات الموجودة في ان واحد في الذاكرة بدرجة البرمجة المتعددة.

Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. In general, most processes can be described as either I/O bound or CPU bound. An ***I/O-bound process*** is one that spends more of its time doing I/O than it spends doing computations. A ***CPU-bound process***, in contrast, generates I/O requests infrequently, using more of its time doing computations.

يتطلب تحقيق التوازن بين أهداف البرمجة المتعددة ومشاركة الوقت أيضًا مراعاة السلوك العام للعملية. بشكل عام ، يمكن وصف معظم العمليات بأنها إما مرتبطة بالإدخال / الإخراج أو مرتبطة بوحدة المعالجة المركزية. I/O-bound process هي العملية التي تقضي وقتها في إجراء I / O أكثر مما تقضيه في إجراء العمليات الحسابية. على النقيض من ذلك ، فإن CPU-bound process تولد طلبات إدخال / إخراج بشكل غير متكرر ، وذلك باستخدام المزيد من وقتها في إجراء العمليات الحسابية.

Scheduling Queues

As processes enter the system, they are put into a ***ready queue***, where they are ready and waiting to execute on a CPU's core. Processes that are waiting for a certain event to occur — such as completion of I/O — are placed in a ***wait queue*** (Figure 4).

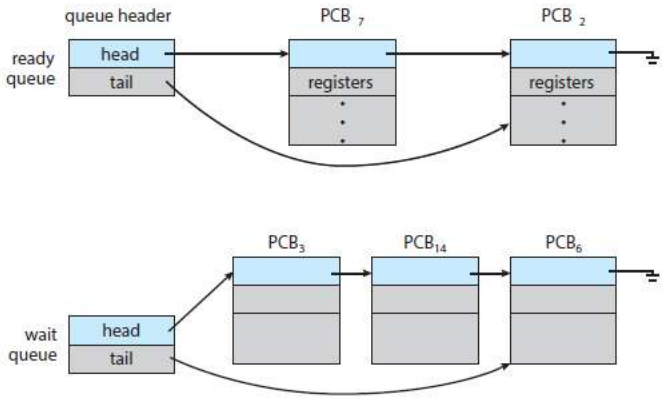


Figure 4: The ready queue and wait queues

A common representation of process scheduling is a ***queueing diagram***, such as that in Figure 5.

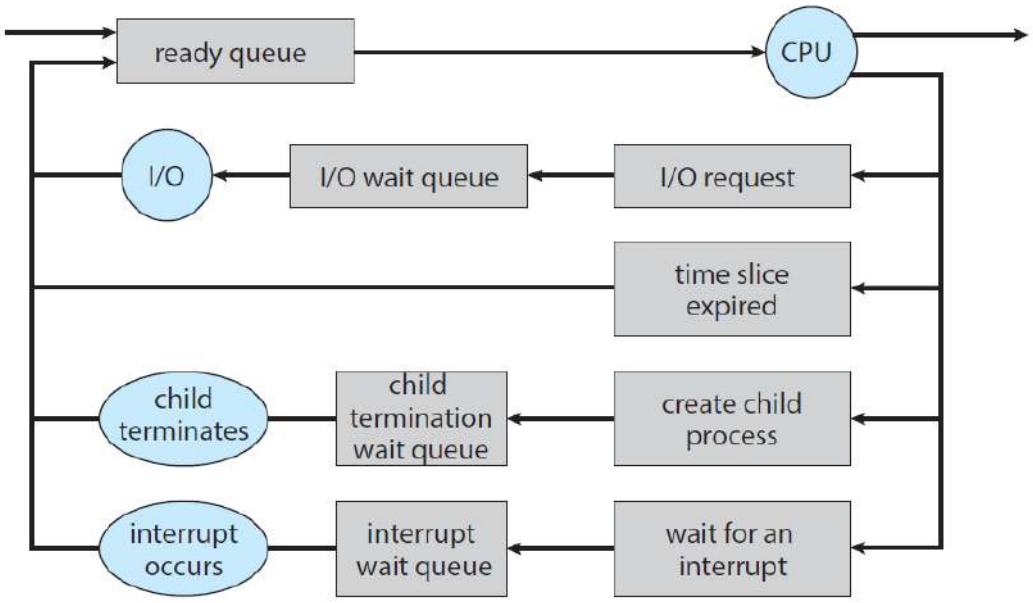


Figure 5: Queuing-diagram representation of process scheduling

عندما تدخل العمليات إلى النظام ، يتم وضعها في قائمة انتظار العمليات الجاهزة، حيث تكون جاهزة وتنتظر التنفيذ على قلب وحدة المعالجة المركزية. يتم وضع العمليات التي تنتظر حدوث حدث معين - مثل إكمال الإدخال / الإخراج - في قائمة انتظار (الشكل 4). التمثيل الشائع لجدولة العملية هو queuing diagram ، مثل الرسم البياني 5.

CPU Scheduling

A process migrates among the ready queue and various wait queues throughout its lifetime. The role of the ***CPU scheduler*** is to select from among the processes that are in the ready queue and allocate a CPU core to one of them. The CPU scheduler must select a new process for the CPU frequently.
Some operating systems have an intermediate form of scheduling, known as ***swapping***, whose key idea is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off.

يتم ترحيل العملية بين قائمة الانتظارللعمليات الجاهزة وقوائم الانتظار المختلفة طوال فترة وجودها في النظام. يتمثل دور برنامج جدولة وحدة المعالجة المركزية في الاختيار من بين العمليات الموجودة في قائمة الانتظار للعمليات الجاهزة وتخصيص نواة وحدة المعالجة المركزية إلى واحدة منها. وكذلك يجب أن يقوم برنامج جدولة وحدة المعالجة المركزية بتحديد عملية جديدة لوحدة المعالجة المركزية بشكل متكرر.
تحتوي بعض أنظمة التشغيل على شكل وسيط من الجدولة ، يُعرف باسم swapping ، وتتمثل فكرته الرئيسية في أنه في بعض الأحيان قد يكون من المفيد إزالة عملية من الذاكرة (ومن التنافس النشط لوحدة المعالجة المركزية) وبالتالي تقليل درجة البرمجة المتعددة. في وقت لاحق ، يمكن إعادة تقديم العملية في الذاكرة ، ويمكن متابعة تنفيذها من حيث توقفت.

Context Switch

When an interrupt occurs, the system needs to save the current ***context*** of the process running on the CPU and restore the saved context of a different process in a task called context switch. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state (see Figure 2), and memory-management information. Generically, we perform a ***state save*** of the current state of the CPU core, be it in kernel or user mode, and then a ***state restore*** to resume operations(Figure 6). Context switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).

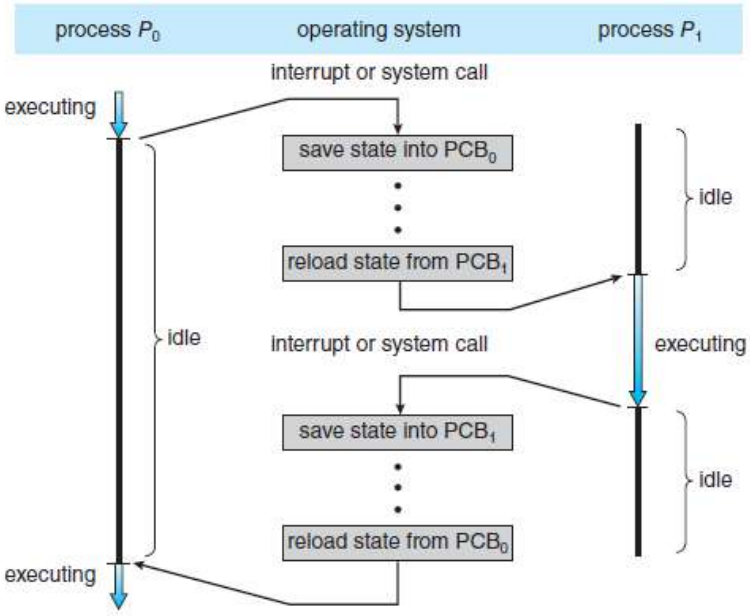


Figure 6: Diagram showing context switch from process to process

عند حدوث مقاطعة ، يحتاج النظام إلى حفظ السياق الحالي للعملية التي تعمل على وحدة المعالجة المركزية واستعادة السياق المحفوظ لعملية مختلفة في مهمة تسمى تبديل السياق. يتم تمثيل السياق في PCB للعملية. ويتضمن قيمة مسجلات وحدة المعالجة المركزية وحالة العملية (انظر الشكل 2) ومعلومات إدارة الذاكرة. بشكل عام ، يتم إجراء حالة حفظ للحالة الحالية لنواة وحدة المعالجة المركزية ، سواء كان ذلك في وضع الـ kernel أو المستخدم ، ثم استعادة الحالة لاستئناف العمليات. ان وقت تبديل السياق هو عبء خالص ، لأن النظام لا يقوم بعمل مفيد أثناء التبديل. كما تختلف سرعة التبديل من آلة إلى أخرى ، اعتمادًا على سرعة الذاكرة ، وعدد السجلات التي يجب نسخها ، ووجود تعليمات خاصة (مثل تعليمات واحدة لتحميل أو تخزين جميع السجلات).

A typical speed is a several microseconds. Context-switch times are highly dependent on hardware support. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch.

السرعة النموذجية هي عدة ميكروثانية. وتعتمد أوقات تبديل السياق بشكل كبير على دعم الأجهزة. وأيضًا ، كلما كان نظام التشغيل أكثر تعقيدًا ، زاد مقدار العمل الذي يجب القيام به أثناء تبديل السياق.

CPU SCHEDULING

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

جدولة وحدة المعالجة المركزية هي أساس أنظمة التشغيل متعددة البرامج. من خلال تبديل وحدة المعالجة المركزية بين العمليات ، يمكن لنظام التشغيل أن يجعل الكمبيوتر أكثر إنتاجية.

Basic Concepts

In ***multiprogramming***, when one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. On a ***multicore system***, this concept of keeping the CPU busy is extended to all processing cores on the system. Scheduling of this kind is a fundamental operating-system function.

في البرمجة المتعددة ، عندما تضطر إحدى العمليات إلى الانتظار ، يأخذ نظام التشغيل وحدة المعالجة المركزية من تلك العملية ويعطيها لعملية أخرى. في نظام متعدد النواة ، ينطبق مفهوم إبقاء وحدة المعالجة المركزية مشغولة على جميع نوى المعالجة في النظام. وتعتبر الجدولة من هذا النوع وظيفة أساسية في نظام التشغيل.

CPU–I/O Burst Cycle

Process execution consists of a ***cycle*** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a ***CPU burst***. That is followed by an ***I/O burst***, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 1).

An ***I/O-bound program*** typically has many short CPU bursts. A CPU- bound Program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.

يتكون تنفيذ العملية من دورة تنفيذ وحدة المعالجة المركزية وانتظار الإدخال / الإخراج. تتناوب العمليات بين هاتين الحالتين. تبدأ وحدة المعالجة المركزية بتنفيذ العملية. يتبع ذلك تنفيذ إدخال / إخراج ، يتبعه تنفيذ آخر لوحدة المعالجة المركزية ، ثم تنفيذ إدخال / إخراج آخر ، وهكذا. في النهاية ، ينتهي التنفيذ النهائي لوحدة المعالجة المركزية بطلب النظام لإنهاء التنفيذ (الشكل 1).

عادةً ما يحتوي البرنامج المرتبط بالإدخال / الإخراج على العديد من فترات تنفيذ وحدة المعالجة المركزية القصيرة. قد يحتوي البرنامج المرتبط بوحدة المعالجة المركزية على بضع دفعات طويلة من وحدة المعالجة المركزية. يمكن أن يكون هذا التوزيع مهمًا عند تنفيذ خوارزمية جدولة وحدة المعالجة المركزية.

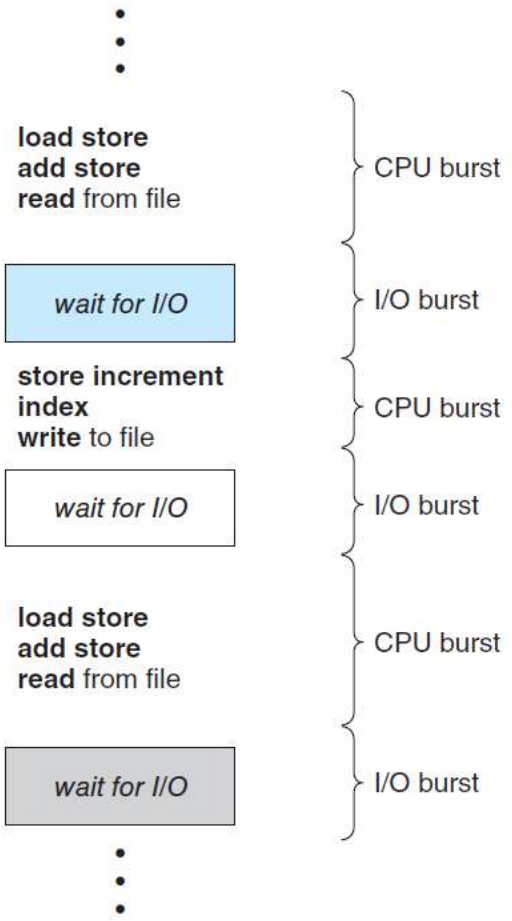


Figure 1 Alternating sequence of CPU and I/O bursts.

CPU Scheduler

When the CPU becomes ***idle***, the CPU scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process. ***Ready queue*** is not necessarily a first-in, first-out (FIFO) queue. A ready queue can be implemented as a ***FIFO queue, a priority queue, a tree, or simply an unordered linked list***. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

عندما تصبح وحدة المعالجة المركزية خاملة ، يقوم برنامج جدولة وحدة المعالجة المركزية بتحديد عملية من العمليات الموجودة في الذاكرة والجاهزة للتنفيذ وتخصيص وحدة المعالجة المركزية لتلك العملية. قائمة الانتظار الجاهزة ليست بالضرورة قائمة انتظار (FIFO) . يمكن تنفيذ قائمة انتظار العمليات الجاهزة كقائمة انتظار FIFO أو قائمة انتظار أولوية أو شجرة أو مجرد قائمة مرتبطة غير مرتبة. وايا كان الهيكل البياني للـ ready queue ، فان جميع العمليات في قائمة الانتظار الجاهزة تصطف في انتظار فرصة للتشغيل على وحدة المعالجة المركزية. ان العناصر في قوائم الانتظار هي بشكل عام كتل التحكم في العمليات (PCBs) للعمليات.

Preemptive and Nonpreemptive Scheduling

Under ***nonpreemptive scheduling***, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by ***terminating*** or by ***switching to the waiting state***. Virtually all modern operating systems including

Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms. Under ***preemptive scheduling***, the CPU is taken from the process abandomly and given to other process.

في الجدولة الـ nonpreemptive ، بمجرد تخصيص وحدة المعالجة المركزية لعملية ما ، فان العملية تحتفظ بوحدة المعالجة المركزية حتى تحررها إما عن طريق الإنهاء أو بالتبديل إلى حالة الانتظار. تستخدم جميع أنظمة التشغيل الحديثة تقريبًا بما في ذلك Windows و macOS و Linux و UNIX خوارزميات جدولة preemptive. في ظل هذه الجدولة ، يتم انتزاع وحدة المعالجة المركزية من العملية ويتم إعطاؤها لعملية أخرى.

Dispatcher

Another component involved in the CPU-scheduling function is the ***dispatcher***. The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. This function involves the following:

• Switching context from one process to another
• Switching to user mode
• Jumping to the proper location in the user program to resume that program

The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the ***dispatch latency*** and is illustrated in Figure 2.

عنصر آخر يشارك في وظيفة جدولة وحدة المعالجة المركزية هو المرسل. المرسل هو البرنامج الذي يمنح التحكم بوحدة المعالجة المركزية للعملية التي يتم اختيارها من قبل مجدول وحدة المعالجة المركزية. تتضمن هذه الوظيفة ما يلي:

•تبديل السياق من عملية إلى أخرى
•التبديل إلى وضع المستخدم
•القفز إلى المكان المناسب في برنامج المستخدم لاستئناف ذلك البرنامج

يجب أن يكون المرسل أسرع ما يمكن ، حيث يتم استدعاؤه أثناء كل تبديل سياق. يُعرف الوقت الذي يستغرقه المرسل لإيقاف إحدى العمليات وبدء تشغيل آخر باسم زمن تاخيرالإرسال وهو موضح في الشكل 2 .

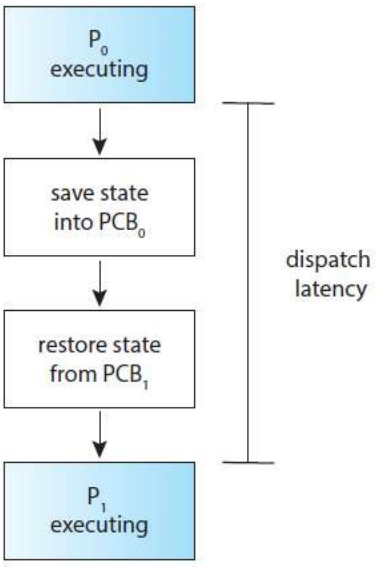


Figure 2 The role of the dispatcher

Scheduling Criteria

Different CPU-scheduling algorithms have different properties. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU-scheduling algorithms.

Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

تتميز خوارزميات جدولة وحدة المعالجة المركزية المختلفة بخصائص مختلفة. عند اختيار الخوارزمية التي يجب استخدامها في موقف معين ، يجب أن يؤخذ في الاعتبار خصائص الخوارزميات المختلفة.لقد تم اقتراح العديد من المعايير لمقارنة خوارزميات جدولة وحدة المعالجة المركزية .ان الخصائص التي تستخدم للمقارنة يمكن أن يكون لها الاثر الكبير في الحكم على خوارزمية ما انها الافضل. وتشمل المعايير التالي:

• ***CPU utilization***. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). (CPU utilization can be obtained by using the top command on Linux, macOS, and UNIX systems.)

نريد إبقاء وحدة المعالجة المركزية مشغولة قدر الإمكان. من الناحية المفاهيمية ، يمكن أن يتراوح استخدام وحدة المعالجة المركزية من 0 إلى 100 بالمائة. في النظام الحقيقي ، يجب أن تتراوح من 40 بالمائة (لنظام محمّل بشكل خفيف) إلى 90 بالمائة (لنظام محمّل بشكل كبير) .(يمكن الحصول على استخدام وحدة المعالجة المركزية باستخدام الأمر top في أنظمة Linux و macOS وUNIX)

• ***Throughput***. One measure of work is the number of processes that are completed per time unit, called ***throughput***. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

•أحد مقاييس العمل هو عدد العمليات التي يتم إكمالها لكل وحدة زمنية ، تسمى throughput. بالنسبة للعمليات الطويلة ، قد يكون هذا المعدل عملية واحدة على مدى عدة ثوان ؛ بينما للعمليات القصيرة ، قد تكون عشرات العمليات في الثانية.

• ***Turnaround time***. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.

•الفاصل الزمني من وقت إرسال العملية إلى وقت الانتهاء هو turnaround time. وهو مجموع الفترات التي تم قضاؤها في الانتظار في ready queue و فترات التنفيذ على وحدة المعالجة المركزية وفترات تنفيذ عمليات الإدخال / الإخراج.

• ***Waiting time***. The CPU-scheduling algorithm affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

•تؤثر خوارزمية جدولة وحدة المعالجة المركزية فقط على مقدار الوقت الذي تقضيه العملية في الانتظار في الـ ready queue. وقت الانتظار هو مجموع الفترات التي يتم قضاؤها في الانتظار في الـ ready queue

• ***Response time***. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, which is the time it takes to start responding, not the time it takes to output the response.
It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

•في النظام التفاعلي ، قد لا يكون وقت الاستجابة أفضل معيار. ففي كثير من الأحيان ، قد تنتج العملية بعض المخرجات في وقت مبكر إلى حد ما وتستمر في حساب نتائج جديدة أثناء اخراج لنتائج السابقة للمستخدم. وبالتالي ، هناك مقياس آخر هو الوقت من التقديم لطلب حتى يتم إنتاج الاستجابة الأولى. هذا المقياس يسمى response time ، وهو الوقت المستغرق لبدء الاستجابة ، وليس الوقت المستغرق لإخراج الاستجابة .

ان من المستحسن تعظيم CPU utilization والـ throughput وتقليل الـ response time والـ waiting time والـ turnaround time.

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPU scheduling algorithms. we describe these scheduling algorithms in the context of only one processing core available.

تتعامل جدولة وحدة المعالجة المركزية مع مشكلة تحديد أي من العمليات في الـ ready queue سيتم تخصيصها لنواة وحدة المعالجة المركزية. هناك العديد من خوارزميات جدولة وحدة المعالجة المركزية المختلفة. والخوارزميات التالية هي خوارزميات جدولة في سياق معالجة أساسية واحدة فقط متاحة.

1. First-Come, First-Served Scheduling

- The process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- The code for FCFS scheduling is simple to write and understand.
- the average waiting time under the FCFS policy is often quite long.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

• يتم تخصيص وحدة المعالجة المركزية أولاً للعملية التي تطلب وحدة المعالجة المركزية أولاً.
• تتم إدارة تنفيذ سياسة FCFS بسهولة من خلال قائمة انتظار FIFO.
• برمجة جدولة FCFS سهل الكتابة والفهم.
• متوسط وقت الانتظار بموجب سياسة FCFS غالبًا ما يكون طويلًا جدًا.
افترض لديك مجموعة العمليات التالية التي تصل في الوقت 0 ، مع تحديد زمن تنفيذ وحدة المعالجة المركزية بالمللي ثانية:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart**

- Waiting time for P1(WT_{P1}) is start time(p1) – arrival time(p1):

WT_{P1}=0 - 0= 0
WT_{P2}= 24 – 0 =24
WT_{P3} = 27 – 0 =27

- Average waiting time(AWT) is (0+ 24 + 27)/3 = 17 millisecondes.
- Turnaround time (TAT_P) for each process is calculated as end time(p) – Arrival time(p)
- TAT_{P1}= 24 – 0 = 24
- TAT_{P2}= 27 – 0 = 27
- TAT_{P3}= 30 – 0 = 30
- Average turnaround time(ATAT) is (24 + 27 +30) / 3 = 27 milliseconds

If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart

The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds.
Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

- The FCFS scheduling algorithm is nonpreemptive.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems.

- وبالتالي ، فإن متوسط وقت الانتظار بموجب سياسة FCFS ليس بالحد الأدنى بشكل عام وقد يختلف بشكل كبير إذا كانت أوقات تنفيذ وحدة المعالجة المركزية للعمليات تختلف اختلافًا كبيرًا.
- •خوارزمية جدولة FCFS هي nonpreemptive.
- •وبالتالي فإن خوارزمية FCFS ليست مناسبة بشكل خاص لأنظمة مشاركة الوقت.

Example: Consider the following set of processes

Process	Burst time	Arrival time
P1	10	0
P2	6	6
P3	3	7
P4	7	22

1. Draw the Gantt chart for these processes applying FCFS algorithm.
2. What is the waiting time for each process?
3. What is the Average waiting time?
4. What is the Turnaround time for each process?
5. What is the Average turnaround time?

1.ارسم مخطط Gantt لهذه العمليات التي تطبق خوارزمية FCFS.

2. ما هو وقت الانتظار لكل عملية؟
3. ما هو متوسط وقت الانتظار؟
4. ما هو الوقت المستغرق لكل عملية؟
5. ما هو متوسط الوقت المستغرق؟

2. Shortest Job First (SJF)

It is a ***nonpreemptive scheduling*** discipline in which the waiting job/process with the smallest estimated run time to completion is run next. If two jobs have the same run-time, FCFS is used. SJF reduce average waiting time over FCFS. The waiting times, however, have a larger ***variance*** (i.e. are more unpredictable) than FCFS, especially for large jobs. SJF selects jobs for service in a manner that ensures the next job will complete and leave the system as soon as possible.

إنه نظام جدولة ***nonpreemptive*** يتم فيه اختيار وتشغيل المهمة / العملية ذات أقل وقت تشغيل مقدر من بين المهام االمنتظرة. إذا وجدت وظيفتان او عمليتان لهما نفس وقت التشغيل ، فسيتم استخدام FCFS. يقلل SJF من متوسط وقت الانتظار مقارنة بـ FCFS .مع ذلك ، فإن فترات الانتظار لها تباين أكبر (أي لا يمكن التنبؤ به أكثر) من FCFS ، خاصة بالنسبة للوظائف الكبيرة. يختار SJF العمليات بطريقة تضمن إكمال المهمة التالية وترك النظام في أقرب وقت ممكن.

The obvious problem with SJF is that it requires precise knowledge of how long job/process will run, and this information is not usually available. The best SJF can do is to rely on user estimates of run times.

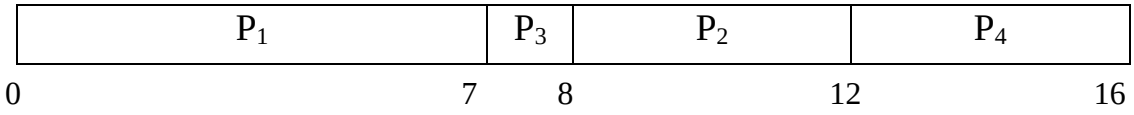
SJF scheduling is used frequently for ***long-term scheduling*** (i.e., batch jobs). Since predicting future CPU bursts is difficult, SJF scheduling is seldom used for short-term scheduling.

المشكلة الواضحة في SJF هي أنها تتطلب معرفة دقيقة بمدة العمل / العملية ، وهذه المعلومات ليست متاحة عادة. أفضل ما يمكن أن يفعله SJF هو الاعتماد على تقديرات المستخدم لأوقات التشغيل.

يتم استخدام جدولة SJF عادة للجدولة طويلة الأجل (أي وظائف الدُفعات). وحيث ان التنبؤ بازمان التنفيذالمستقبلية لوحدة المعالجة المركزية أمر صعب ، فنادراً ما تستخدم جدولة SJF للجدولة قصيرة المدى.

Example:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Average waiting time = (0 + 6 + 3 + 7)/4 = 4
Turnaround time for P_1**= 7**
Turnaround time for P_2= (8-2)+4=10
Turnaround time for P_3= (7-4)+1=4
Turnaround time for P_4=(12-5)+ 4=11
The average =(7+10+4+11)/4=8

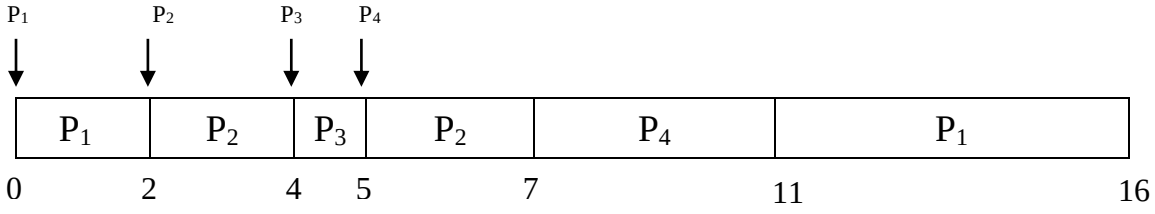
3. Shortest Remaining Time First(SRTF)

The SJF algorithm can be either preemptive or non-preemptive. The difference between them that when a new process arrives the ready queue, a non-preemptive SJF algorithm will allow the currently running process running to finish its CPU burst (SJF algorithm we discussed above), while in a preemptive SJF the algorithm will preempt the currently executing process when the newly arrived process is shorter than what is left of the currently executing process. Preemptive SJF scheduling is sometimes called ***shortest-remaining-time-first*** scheduling. SRTF has a ***higher overhead*** than SJF. It must keep track of the elapsed service time of the running job and must handle occasional preemptions. Arriving small processes will run almost immediately. SRT, however, has an even larger mean waiting time and variance of waiting time than SJF.

يمكن أن تكون خوارزمية SJF من نوع preemptive أو nonpreemptive. الفرق بينهما هو أنه عندما تصل عملية جديدة إلى قائمة الانتظار الجاهزة ، فإن خوارزمية SJF الـ nonpreemptive ستسمح للعملية الجارية قيد التشغيل حاليًا لإنهاء فترة تنفيذها على وحدة المعالجة المركزية (خوارزمية SJF التي ناقشناها أعلاه) ، بينما في الـ preemptive SJF ، ستعمل الخوارزمية على انتزاع الـ CPU من العملية قيد التنفيذ حاليًا عندما تكون العملية التي وصلت حديثًا أقصر مما تبقى من عملية التنفيذ الحالية. يُطلق أحيانًا على جدولة الـ preemptive SJF اسم جدولة Shortest-remaining-Time-First. خوارزمية SRTF تشكل عبئا على النظام اعلى من الـ SJF وذلك لانه يجب أن يتتبع وقت الخدمة المنقضي للوظيفة الجارية ، ويجب أن يتعامل مع المقاطعات اثناء التنفيذ. ان العمليات الصغيرة القادمة سيتم تشغيلها على الفور تقريبًا. ومع ذلك ، فإن SRT لديها متوسط وقت انتظار وتباين في وقت الانتظار أكبر من SJF.

Example 1:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



Process1 starts at time 0, since it is the only job in the queue. Process2 arrives at time 2. The remaining time for process1 (5 time units) is larger than the time required by process2 (4 time units), so process1 is preempted, and process2 is scheduled. The average turnaround time for this example is:

((16-0) +(7-2)+(5-4)+(11-5))/4 =7 time units

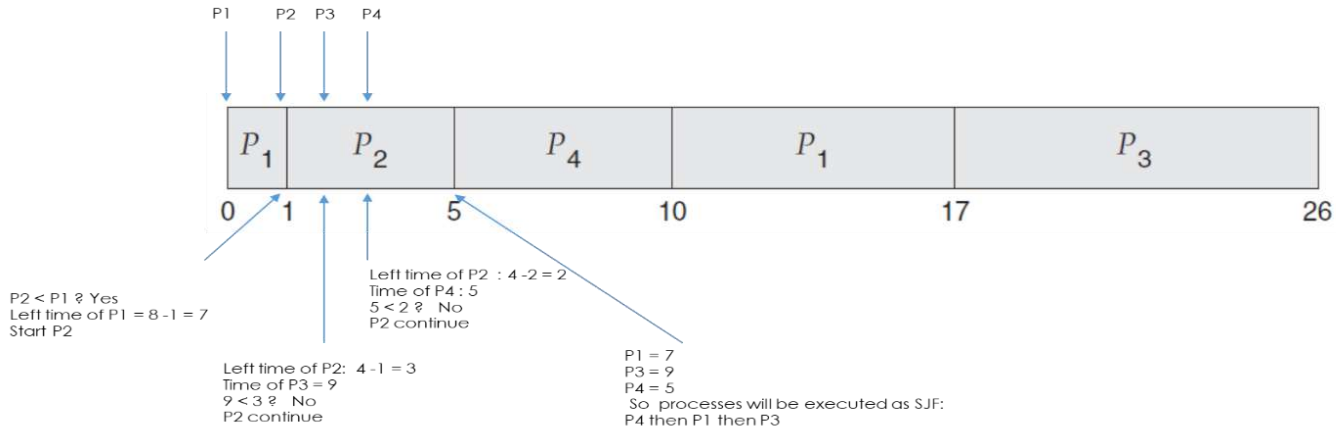
تبدأ العملية 1 في الوقت 0 ، لأنها الوظيفة الوحيدة في قائمة الانتظار. تصل العملية 2 في الوقت 2. الوقت المتبقي للعملية 1 (5 وحدات زمنية) أكبر من الوقت الذي تتطلبه العملية 2 (4 وحدات زمنية) ، لذلك يتم استباق العملية 1 ، ويتم جدولة العملية 2. متوسط الوقت المستغرق لهذا المثال هو:

((16-0) + (7-2) + (5-4) + (11-5)) / 4 = 7 وحدات زمنية

Example 2:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Waiting time for process P is the summation of waiting times for it in the ready queue
$WT_{P1} = (0 – 0) + (10 – 1) = 9$
$WT_{P2} = 1 – 1 = 0$
$WT_{P3} = 17 – 2 = 15$
$WT_{P4} = 5 – 3 = 2$
Average waiting time(AWT) is (0+ 0 + 15+2) / 4 = 4.75 millisecondes.
Turnaround time (TAT_P) for each process is calculated as end time(p) – Arrival time(p)
OR summation of waiting times(P) + Burst time(P)
$TAT_{P1} = 9 + 8 = 17$
$TAT_{P2} = 0 + 4 = 4$
$TAT_{P3} = 15 + 9 = 24$
$TAT_{P4} = 2 + 0 = 2$
Average turnaround time(ATAT) is (17 + 4 +24+ 2) / 4 = 11.75 milliseconds

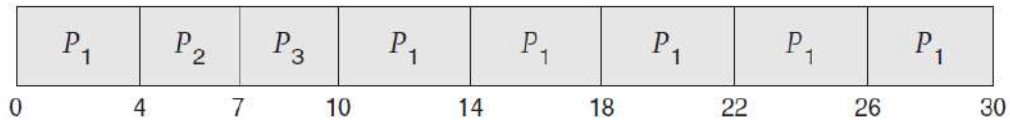
4. Round-Robin Scheduling

The ***round-robin (RR) scheduling algorithm*** is similar to FCFS scheduling, but ***preemption*** is added to enable the system to switch between processes. A small unit of time called a ***time quantum or time slice***, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum. The ***average waiting time*** under the RR policy is often long.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds

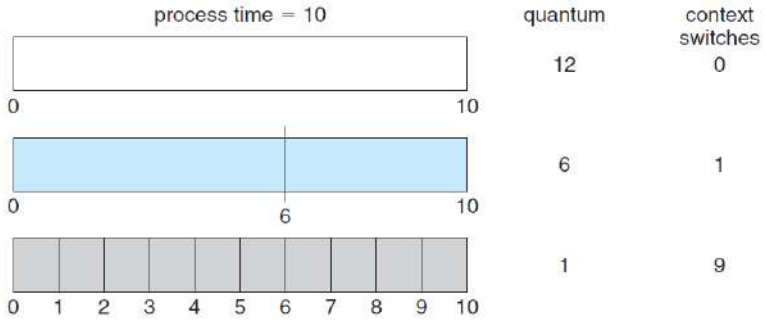
تشبه خوارزمية جدولة round-robin (RR) جدولة FCFS ، ولكن الـ preemption اضيف لتمكين النظام من التبديل بين العمليات. تعرف هنا وحدة صغيرة من الوقت ، تسمى time quantum او time slice. يبلغ طول الـ time quantum بشكل عام من 10 إلى 100 ميلي ثانية. يتم التعامل مع قائمة الانتظار الجاهزة على أنها قائمة انتظار دائرية. ويتنقل برنامج جدولة وحدة المعالجة المركزية حول قائمة الانتظار الجاهزة ، ويخصص وحدة المعالجة المركزية لكل عملية لفترة زمنية تصل إلى time quantum واحد.غالبًا ما يكون متوسط وقت الانتظار بموجب سياسة RR طويلًا.
افترض مجموعة العمليات التالية التي تصل في الوقت 0 ، مع تحديد طول التنفيذ على وحدة المعالجة المركزية بالمللي ثانية

Process	Burst Time
P_1	24
P_2	3
P_3	3



- Waiting time for P1 = (0-0) +(10-4) = 6 milliseconds
  P2 =4 – 0 =4 milliseconds
  P3 = 7 – 0 =7 milliseconds
- The average waiting time = (6 + 4 + 7) / 3 =5.66 milliseconds

The performance of the RR algorithm depends heavily on the size of the time quantum. Suppose the examples below:
يعتمد أداء خوارزمية RR بشكل كبير على حجم time quantumوكما في الامثلة الاتية:



To improve performance, the time quantum must be large with respect to the context switch time. We note that turnaround time depends on the size of the time quantum, and it can be improved if most processes finish their next CPU burst in a single time quantum.
ان اداء خوارزمية RR يعتمد بشكل كبير على الـ time quantum . ولتحسين الاداء فيجب ان يكون الـ time quantum كبيرا نسبة الى زمن الـ context switch . يلاحظ ان الزمن المستغرق (turnaround time) يعتمد على مقدار الـ time quantum ، ويمكن تحسينه ان تمكنت اغلب العمليات من انهاء زمن تنفيذها في time quantum واحد.

5. Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. Some systems use low numbers to represent low priority; others use low numbers for high priority. We will assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order *P*1, *P*2, · · ·, *P*5, with the length of the CPU burst given in milliseconds:

ترتبط الأولوية بكل عملية ، ويتم تخصيص وحدة المعالجة المركزية للعملية ذات الأولوية القصوى. تتم جدولة العمليات ذات الأولوية المتساوية بترتيب FCFS .تتم الإشارة إلى الأولويات بشكل عام من خلال مجموعة ثابتة من الأرقام ، مثل 0 إلى 7 أو من 0 إلى 4095. وتستخدم بعض الأنظمة أرقامًا منخفضة لتمثيل أولوية منخفضة ؛ في حين يستخدم البعض الآخر الارقام المنخفضة للحصول على أولوية عالية. وفي دراستنا سنفترض أن الأرقام المنخفضة تمثل أولوية عالية.

فعلى سبيل المثال ، افترض مجموعة العمليات التالية ، التي يُفترض أنها وصلت في الوقت 0 بالترتيب P1 ، P2 ، ٠٠ ، P5 ، مع تحديد طول نوبة تنفيذ وحدة المعالجة المركزية بالمللي ثانية:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:

وباستخدام جدولة الاولوية، ستجدول هذه العمليات كما في الاتي:

The average waiting time is 8.2 milliseconds.

Priorities can be defined either **internally or externally**. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political factors.

متوسط وقت الانتظار هو 8.2 مللي ثانية.

يمكن تحديد الأولويات داخليًا أو خارجيًا. تستخدم الأولويات المحددة داخليًا بعض الكميات أو الكميات القابلة للقياس لحساب أولوية العملية. فعلى سبيل المثال ، يتم استخدام الحدود الزمنية ومتطلبات الذاكرة وعدد

الملفات المفتوحة ونسبة متوسط نوبة تنفيذ الإدخال / الإخراج إلى متوسط نوبة تنفيذ وحدة المعالجة المركزية في الحوسبة على اساس الاولوية المحددة داخليا، بينما يتم تحديد الأولويات الخارجية وفقًا لمعايير خارج نظام التشغيل ، مثل أهمية العملية ونوع ومقدار الأموال التي يتم دفعها لاستخدام الكمبيوتر والقسم الراعي للعمل وعوامل أخرى ، غالبًا ما تتعلق بسياسة نظام الكومبيوتر.

Priority scheduling can be either preemptive or nonpreemptive. A major problem with priority scheduling algorithms is ***indefinite blocking*** or ***starvation***. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. A solution to the problem of indefinite blockage of low-priority processes is ***aging***. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of awaiting process by 1.

Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling. Let's illustrate with an example using the following set of processes, with the burst time in milliseconds:

يمكن أن تكون جدولة الأولوية preemptive أو nonpreemptive . المشكلة الرئيسية في خوارزميات جدولة الأولوية هي الحجب غير المحدد ، او الـ starvation .

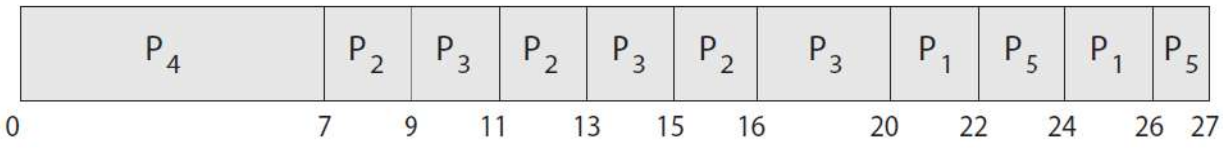
يمكن أن تترك خوارزمية جدولة الأولوية بعض العمليات ذات الأولوية المنخفضة في الانتظار إلى أجل غير مسمى . ان الحل لمشكلة الحجب غير المحدد للعمليات ذات الأولوية المنخفضة هو الـ aging. يتضمن الـ aging زيادة تدريجية في أولوية العمليات التي تنتظر في النظام لفترة طويلة. على سبيل المثال ، إذا كانت الأولويات تتراوح بين 127 (منخفض) إلى 0 (مرتفع) ، يمكننا بشكل دوري (على سبيل المثال ، كل ثانية) زيادة أولوية في انتظار العملية بنسبة 1.

والخيار الآخر لحل المشكلة هو الجمع بين جدولة round robin وجدولة الأولوية، حيث يقوم النظام بتنفيذ العمليات ذات الأولوية القصوى ومن ثم تشغيل العمليات متساوية الأولوية باستخدام جدولة round robin. المثال التالي لمجموعة من العمليات يوضح ذلك ، مع وقت نوبة التنفيذ بالملي ثانية:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:

باستخدام جدولة الأولوية مع round-robin للعمليات ذات الأولوية المتساوية ، ستتم جدولة هذه العمليات وفقًا لمخطط جانت التالي باستخدام time quantum قدره 2 مللي ثانية:



In this example, process *P*4 has the highest priority, so it will run to completion. Processes *P*2 and *P*3 have the next-highest priority, and they will execute in a round-robin fashion. Notice that when process *P*2 finishes at time 16, process *P*3 is the highest-priority process, so it will run until it completes execution.
Now, only processes *P*1 and *P*5 remain, and as they have equal priority, they will execute in round-robin order until they complete.

في هذا المثال ، تكون الأولوية القصوى للعملية P4 ، لذلك ستعمل حتى الاكتمال. العمليات P2 و P3 لها الأولوية القصوى التالية ، وسوف يتم تنفيذها بأسلوب round robin .لاحظ أنه عند انتهاء العملية P2 في الوقت 16 ، تكون العملية P3 هي العملية ذات الأولوية القصوى ، لذلك سيتم تشغيلها حتى تكتمل التنفيذ. الآن تبقى العمليات P1 و P5 فقط ، وبما أن لهما أولوية متساوية ، فسيتم تنفيذهما بترتيب جدولة round robin حتى تكتمل.

6. Multilevel Queue Scheduling

With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run. In practice, it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue. This is illustrated in Figure 5.7. This approach—known as ***multilevel queue***— also works well when priority scheduling is combined with round-robin: if there are multiple processes in the highest-priority queue, they are executed in round-robin order. In the most generalized form of this approach, a priority is assigned statically to each process, and a process remains in the same queue for the duration of its runtime.

مع كل من جدولة الـ priority وجدولة round robin ، قد توضع جميع العمليات في قائمة انتظار واحدة ، ثم يختار المجدول العملية ذات الأعلى الأولوية للتشغيل. غالبًا ما يكون من الأسهل أن تكون هناك قوائم انتظار منفصلة لكل أولوية ،والجدولة على اساس الاولية priority scheduling تقوم ببساطة بجدولة العملية في قائمة الانتظار ذات الأولوية القصوى. وهذا موضح في الشكل 1. هذا النهج - المعروف باسم قائمة الانتظار متعددة المستويات- يعمل أيضًا بشكل جيد عندما يتم دمج جدولة الأولوية مع round-robin :بحيث انه لو كانت هناك عمليات متعددة في قائمة الانتظار ذات الأولوية القصوى ، فانه يتم تنفيذها في ترتيب round robin. في الشكل الأكثر عمومية لهذا النهج ، تعين الاولوية بشكل ثابت لكل عملية ، وتبقى العملية في نفس قائمة الانتظار طوال مدة تشغيلها.

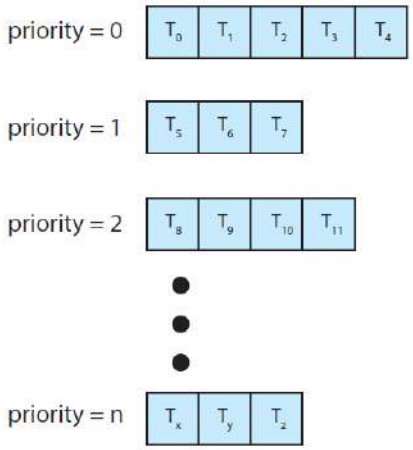


Figure 1 Separate queues for each priority.

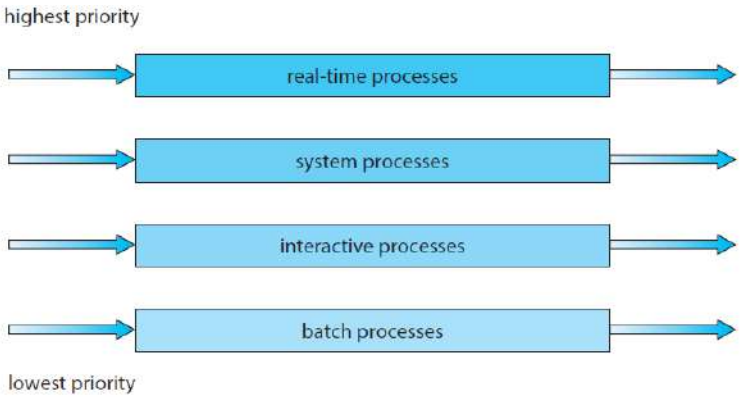


Figure 2 Multilevel queue scheduling.

A multilevel queue scheduling algorithm can also be used to partition processes into several separate queues based on the process type (Figure 2). For example, a division between ***foreground (interactive***) processes and *background (batch)* processes. Each type may have different scheduling needs.

Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

Let's look at an example of a multilevel queue scheduling algorithm with four queues, listed below in order of priority:

1. Real-time processes
2. System processes
3. Interactive processes
4. Batch processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for real-time processes, system processes, and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch

process would be preempted. Another possibility in this algorithm is to time-slice among the queues.

يمكن أيضًا استخدام خوارزمية جدولة قائمة انتظار متعددة المستويات لتقسيم العمليات في عدة قوائم انتظار منفصلة بناءً على نوع العملية (الشكل 2) .فعلى سبيل المثال ، تقسيم العمليات الى foreground(interactive) و background(batch) وقد يكون لكل منها نوع مختلف من احتياجات او متطلبات الجدولة.

يمكن استخدام قوائم انتظار منفصلة لـعمليات الـ foregroundوالـ background ، وقد يكون لكل قائمة انتظار خوارزمية جدولة خاصة بها. بالإضافة إلى ذلك ، يجب أن يكون هناك جدولة بين قوائم الانتظار ، والذي غالبا ما يتم تنفيذه كجدولة preemptive ذات أولوية ثابتة.

,ونبين ادناه بمثال خوارزمية جدولة قائمة انتظار متعددة المستويات مع أربعة قوائم انتظار مدرجة بترتيب الأولوية:

1. عمليات في الوقت الحقيقي
2. عمليات النظام
3. العمليات التفاعلية
4. عمليات الدُفعات

كل قائمة انتظار لها أولوية مطلقة على قوائم الانتظار ذات الأولوية المنخفضة. فعلى سبيل المثال، لا يمكن تنفيذ عملية من قائمة انتظار الـ batch (الدفعات) منخفضة الاولوية ما لم تكن قوائم الانتظار لعمليات الوقت الفعلي ، عمليات النظام والعمليات التفاعلية، الاعلى اولوية كلها فارغة. إذا دخلت عملية تفاعلية إلى قائمة الانتظار الجاهزة أثناء تشغيل عملية الدُفعة ، فان الـ CPU سوف ينتزع من عملية الدفعة. الاحتمال الآخر في هذه الخوارزمية هو تقسيم الوقت بين قوائم الانتظار.

7. Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. This setup has the advantage of low scheduling overhead, but it is inflexible. The ***multilevel feedback queue*** scheduling algorithm, in contrast, allows a process to move between queues.

If a process uses too much CPU time, it will be moved to a lower-priority queue. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

عادةً ، عند استخدام خوارزمية جدولة قائمة الانتظار متعددة المستويات ، يتم احلال العمليات وبقائها بشكل دائم في قائمة انتظار محددة عند دخولها النظام. هذا الإعداد يتميز بعبء منخفض على النظام ، لكنه غير مرن. في المقابل ، تسمح خوارزمية multilevel feedback queue scheduling للعملية بالتنقل بين قوائم الانتظار.

إذا كانت العملية تستهلك الكثير من وقت وحدة المعالجة المركزية ، سيتم نقلها إلى قائمة انتظار ذات أولوية أقل. بالإضافة إلى ذلك ، فإن العملية التي تنتظر وقتًا طويلاً في قائمة انتظار ذات اولوية واطئة قد يتم نقلها إلى قائمة انتظار ذات أولوية أعلى. هذا الشكل من aging يمنع الـ starvation .

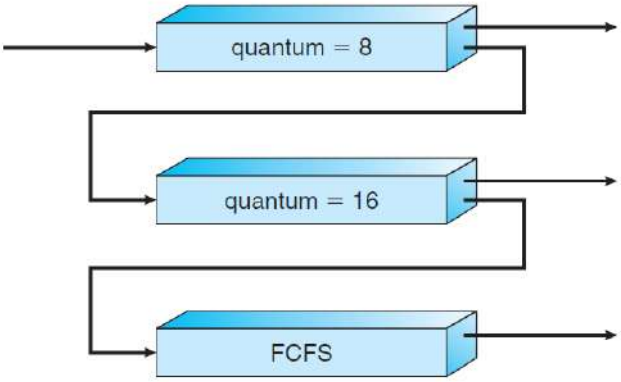


Figure 3 Multilevel feedback queues.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

• The number of queues
• The scheduling algorithm for each queue
• The method used to determine when to upgrade a process to a higher priority queue
• The method used to determine when to demote a process to a lower priority queue
• The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

بشكل عام ، يتم تحديد multilevel feedback queue scheduling من خلال المعلمات الاتية:

•عدد قوائم الانتظار
•خوارزمية الجدولة لكل قائمة انتظار
•الطريقة المستخدمة لتحديد وقت ترقية العملية إلى أولوية أعلى طابور
•الطريقة المستخدمة لتحديد متى يتم تخفيض عملية ما إلى أولوية أقل طابور
•الطريقة المستخدمة لتحديد قائمة الانتظار التي ستدخلها العملية عندما تحتاج تلك العملية إلى خدمة.

إن تعريف multilevel feedback queue scheduling يجعلها الخوارزمية الاعم من بين الخوارزميات. ولسوء الحظ ، إنها أيضًا الخوارزمية الأكثر تعقيدًا ، لأن تحديد أفضل برنامج جدولة يتطلب بعض الوسائل التي تحدد القيم لجميع المعلمات.

Deadlocks

In a multiprogramming environment, several threads may compete for a finite number of resources. A thread requests resources; if the resources are not available at that time, the thread enters a waiting state. Sometimes, a waiting thread can never again change state, because the resources it has requested are held by other waiting threads. This situation is called a ***deadlock***. deadlock is a situation in which *every process in a set of processes is waiting for an event that can be caused only by another process in the set.*

في بيئة البرمجة المتعددة ، قد تتنافس عدة threads على عدد محدود من الموارد. الـ thread تطلب الموارد. فان لم تكن الموارد متاحة في ذلك الوقت ، تدخل الـ thread في حالة انتظار. لا يمكن في بعض الأحيان ، للـ thread المنتظرة تغيير حالتها أبدًا مرة أخرى ، لأن الموارد التي طلبتها محتفظ بها من قبل threads منتظرة أخرى. هذا الوضع يسمى الجمود. الجمود هو الموقف الذي تنتظر فيه كل عملية في مجموعة من العمليات حدثًا يمكن أن يكون ناتجًا فقط عن عملية أخرى في المجموعة.

A system consists of a finite number of resources to be distributed among a number of competing threads. The resources may be partitioned into several ***types*** (or classes), each consisting of some number of identical ***instances***. If a thread requests an instance of a resource type, the allocation of ***any*** instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

يتكون النظام من عدد محدود من الموارد ليتم توزيعها على عدد من threads المتنافسة. قد يتم تقسيم الموارد إلى عدة أنواع (أو فئات) ، يتكون كل منها من عدد من العناصر المتطابقة. إذا طلب thread عنصرمن فئة مورد معين، فيجب أن يفي تخصيص أي عنصر من النوع بالطلب. إذا لم يحدث ذلك ، فلن تكون العناصر متطابقة ، ولم يتم تحديد فئات نوع المورد بشكل صحيح.

A thread must request a resource before using it and must release the resource after using it. A thread may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. Under the normal mode of operation, a thread may utilize a resource in only the **Request**, **Use, and Release** sequence. The request and release of resources may be system calls, Examples are the request() and release() of a device, open() and close() of a file, and allocate() and free() memory system calls.

يجب أن يطلب الـ thread المورد قبل استخدامه ويجب أن يحرره بعد استخدامه. قد يطلب ال thread جميع الموارد التي يحتاجها لينفذ مهمته المحددة. وقد لا يتجاوز عدد الموارد المطلوبة إجمالي عدد الموارد المتاحة في النظام. ، قد يستخدم الـ thread وفي ظل الوضع العادي للعملية موردًا في تسلسل ***Request, Use, and Release*** فقط. قد يكون طلب الموارد و تحريرها عبارة عن استدعاءات للنظام ، ومن الأمثلة على ذلك استدعاءات النظام request() و release() للجهاز، open() و close() للملف، و allocate() و free() للذاكرة.

A system table records whether each resource is free or allocated. For each resource that is allocated, the table also records the thread to which it is allocated. If a thread requests a resource that is currently allocated to another thread, it can be added to a queue of threads waiting for this resource.

يسجل جدول النظام ما إذا كان كل مورد حرا أم مخصصًا. ، يسجل الجدول ايضا الـ thread الذي تم تخصيص المورد له. إذا طلب الـ thread موردًا مخصصا في ذلك الحين لـ thread آخر ، فعندها يمكن إضافة الـ thread إلى قائمة انتظار الـ threads التي تنتظر هذا المورد.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion. At least one resource must be held in a nonsharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.

2. Hold and wait. A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.

3. No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.

4. Circular wait. A set *{T*0, *T*1, ..., *Tn }* of waiting threads must exist such that *T*0 is waiting for a resource held by *T*1, *T*1 is waiting for a resource held by *T*2, ..., *Tn*−1 is waiting for a resource held by *Tn*, and *Tn* is waiting for a resource held by *T*0.

يمكن أن تنشأ حالة Adeadlock إذا توفرت الشروط الأربعة التالية في وقت واحد في نظام:

1. ***mutual exclusion*** يجب الاحتفاظ بمورد واحد على الأقل في وضع غير قابل للمشاركة ؛ أي ، يمكن لـ thread واحد فقط في كل مرة استخدام المورد. إذا thread آخر هذا المورد ، فيجب تأجيل الطلب حتى يتم تحرير المورد.

2. ***Hold and Wait*** يجب أن تحتفظ الـ thread بمورد واحد على الأقل وتنتظر الحصول على موارد إضافية يتم الاحتفاظ بها حاليًا من قبل threads أخرى.

3. ***No Preemption*** لا يمكن انتزاع الموارد ؛ أي ، يمكن تحرير المورد طواعية فقط من قبل الـ thread الذي يحمله ، بعد أن يكمل هذا الـ thread مهمته.

4. ***Circular Wait*** . يجب أن توجد المجموعة {T0, T1, …, Tn} للـ threads المنتظرة بحيث تنتظر T0 موردًا تحتفظ بهT1 ، وتنتظر T1 موردًا تحتفظ به T2 ، ...، Tn − 1 تنتظر موردًا تحتفظ به Tn ، وتنتظر Tn موردًا تحتفظ به T0.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a ***system resource-allocation graph***. This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices *V* is partitioned into two different types of nodes: *T = {T*1, *T*2, ..., *Tn }*, the set consisting of all the active threads in the system, and *R = {R*1, *R*2, ..., *Rm }*, the set consisting of all resource types in the system. A directed edge *Ti* → *Rj* is called a ***request edge***; a directed edge *Rj* → *Ti* is called an ***assignment edge***. Pictorially, we represent each thread *Ti* as a circle and each resource type *Rj* as a rectangle.

The resource-allocation graph shown in Figure 8.4 depicts the following situation.
• The sets *T*, *R*, and *E*:
◦ *T* = {*T*1, *T*2, *T*3 }
◦ *R* = {*R*1, *R*2, *R*3, *R*4 }
◦ *E* = {*T*1 → *R*1, *T*2 → *R*3, *R*1 → *T*2, *R*2 → *T*2, *R*2 → *T*1, *R*3 → *T*3 }
• Resource instances:
◦ One instance of resource type *R*1
◦ Two instances of resource type *R*2
◦ One instance of resource type *R*3
◦ Three instances of resource type *R*4
• Thread states:
◦ Thread *T*1 is holding an instance of resource type *R*2 and is waiting for an instance of resource type *R*1.
◦ Thread *T*2 is holding an instance of *R*1 and an instance of *R*2 and is waiting for an instance of *R*3.
◦ Thread *T*3 is holding an instance of *R*3.

يمكن وصف حالات الـ deadlock بشكل أكثر دقة بواسطة الرسم البياني الموجه المسمى system resource-allocation graph. يتكون هذا الرسم البياني من مجموعة من الرؤوس V ومجموعة من الحواف E. مجموعة الرؤوس V مقسمة إلى نوعين مختلفين من العقد: T={ T1, T2, …,Tn} وهيالمجموعة التي تتكون من كل الـ threads النشطة في النظام ، و R={R1, R2, …,Rm} ، وهي المجموعة التي تتكون من جميع أنواع الموارد في النظام. تسمى الحافة الموجهة Ti → Rj حافة الطلب (***request edge***) ؛ تسمى الحافة الموجهة Rj → Ti حافة التخصيص (***assignment edge***). من الناحية التصويرية ، نمثل كل خيط Ti كدائرة وكل نوع مورد Rj كمستطيل.
يوضح الـ resource-allocation graph الموضح في الشكل 1 الحالة التالية.
• *E: المجموعات T و R و*
◦ *T = {T1 ،T2 ،T3}*
◦ *R = {R1 ،R2 ،R3 ،R4}*
◦ *E = {T1 → R1 ،T2 → R3 ،R1 → T2 ،R2 → T2 ،R2 → T1 ،R3 → T3}*

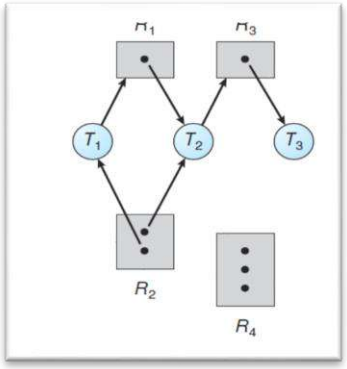


Figure 1: Resource-allocation graph

عناصرالموارد:
◦عنصر واحد لنوع الموردR1
عنصران من نوع المورد R2
◦عنصر واحد لنوع المورد R3
ثلاث عناصر من نوع المورد R4
•عناصر الـ thread
◦الـ T1 thread مستحوذاعلى عنصر لنوع المورد R2 وينتظر عنصرا لنوع المورد R1 .
◦الـ T2 thread مستحوذاعلى عنصر R1 وعنصر R2 وهو في انتظار عنصر R3
◦الـ T3 thread مستحوذا على عنصر R3 .

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

بالنظر إلى تعريف الـ resource allocation graph، يمكن إظهار أنه إذا كان الرسم البياني لا يحتوي على دورات ، فلن يكون هناك thread في النظام في حالة deadlock. اما إذا كان الرسم البياني يحتوي على دورة ، فقد يكون هناك deadlock .

If each resource type has exactly one instance, then a ***cycle*** implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each thread involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

إذا كان لكل نوع مورد عنصر واحد بالضبط ، فإن الحلقة تشير إلى حدوث deadlock. إذا كانت الحلقة تتضمن مجموعة من أنواع الموارد ، ولكل منها عنصر واحد فقط ، فقد حدث deadlock. وكل thread مشارك في الحلقة دخل حالة الـ deadlock. في هذه الحالة ، تعتبر الحلقة في الرسم البياني شرطًا ضروريًا وكافيًا لوجود deadlock .

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 1.

إذا كان لكل نوع من أنواع الموارد عدة عناصر ، فإن الحلقة لا تعني بالضرورة حدوث deadlock. في هذه الحالة ، الحلقة في الرسم البياني ضرورة ولكنها ليست شرطًا كافيًا لوجود الـ deadlock .لتوضيح هذا المفهوم ، نعود إلى الـ resource-allocation graph الموضح في الشكل 1.

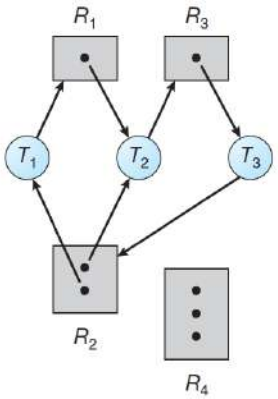


Figure 2: Resource-allocation graph with a deadlock

Now consider the resource-allocation graph in Figure 2. In this example, we also have a cycle:

$T1 \rightarrow R1 \rightarrow T3 \rightarrow R2 \rightarrow T1$

الآن افترض الـ resource-allocation graph في الشكل 3 في هذا المثال ، لدينا أيضًا حلقة:

T1 → R1 → T3 → R2 → T1

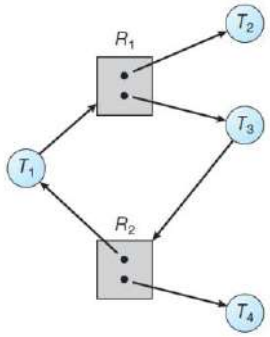


Figure 3 Resource-allocation graph with a cycle but no deadlock.

Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

• We can ignore the problem altogether and pretend that deadlocks never occur in the system.

• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.

• We can allow the system to enter a deadlocked state, detect it, and recover.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. ***Deadlock prevention*** provides a set of methods to ensure that at least one of the necessary conditions cannot hold. ***Deadlock avoidance*** requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime.

طرق التعامل مع الـ deadlock

بشكل عام ، يمكننا التعامل مع مشكلة الـ deadlock في واحدة من ثلاثة طرق:

•يمكننا أن نتجاهل المشكلة تمامًا ونتظاهر بأن الـ deadlock لا يحدث أبدًا في النظام.

•يمكننا استخدام بروتوكول لمنع أو تجنب المآزق ، وضمان أن لن يدخل النظام أبدًا في حالة deadlock.
•يمكننا السماح للنظام بالدخول في حالة الـ deadlock واكتشافه واستعادته.

لضمان عدم حدوث الـ deadlock أبدًا ، يمكن للنظام استخدام وسيلة منع الوصول إلى deadlock أو مخطط تجنب الـ deadlock. يوفر الـ ***deadlock prevention*** (منع الوصول الى الـ deadlock) مجموعة من الطرق للتأكد من أن واحدًا على الأقل من الشروط الضرورية لا يمكن أن تعقد. يتطلب تجنب الجمود (***deadlock avoidance***) إعطاء نظام التشغيل المزيد من المعلومات مقدمًا بشأن أي الموارد سيطلبها الـ thread ويستخدمها خلال حياته.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

من أجل حدوث الـ deadlock ، فان من الضروري ان تتحقق وتصمد كل من الشروط الأربعة. من خلال كسر واحد على الأقل من هذه الشروط ، فانه يمنع حدوث الـ deadlock .

1.Mutual Exclusion

The mutual-exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource.

يجب أن يستمر شرط الاستبعاد المتبادل. ذلك يعني انه يجب على مورد واحد على الأقل ان يكون غير قابل للمشاركة. لا تتطلب الموارد القابلة للمشاركة وصولا حصريا اليها وبالتالي لا تكون مشاركة في الـ deadlock. تعد ملفات (للقراءة-فقط) مثالًا جيدًا للموارد القابلة للمشاركة.

2.Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, **we must guarantee that, whenever a thread requests a resource, it does not hold any other resources.** One protocol that *we can use requires each thread to request and be allocated all its resources before it begins execution*.

An alternative protocol allows a thread *to request resources only when it has none.* Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, starvation is possible. A thread that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other thread.

لضمان عدم حدوث شرط hold and wait أبدًا في النظام ، يجب ضمان أنه عندما تطلب process ما موردًا ، فإنه يجب ان لا يكون بحوزتها مصادر أخرى. احد البروتوكولات التي يمكن استخدامها يتطلب ان كل process تطلب ,وتخصص لها جميع مواردها قبل أن يبدأ التنفيذ.

يسمح بروتوكول اخر للـ process بطلب الموارد فقط عندما لا يكون بحوزتها اي منها. كلا هذين البروتوكولين لهما عيبان رئيسيان. أولا ، استخدام الموارد قد يكون منخفضا ، حيث يمكن تخصيص الموارد ولكنها غير مستخدمة لفترة طويلة. ثانيًا ، امكانية حصول الـ starvation. فالـ process الذي يحتاج إلى العديد من الموارد الشائعة قد يضطر إلى الانتظار إلى أجل غير مسمى ، لأنه سيتم تخصيص واحدا على الأقل من الموارد التي يحتاجها دائمًا لـ process اخرى.

3.No Preemption

To ensure that this condition does not hold, we can use the following protocol. *If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted.*

The preempted resources are added to the list of resources for which the thread is waiting. The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions.

للتأكد من أن هذا الشرط لا يصمد ، يمكننا استخدام البروتوكول التالي. إذا كان بحوزة الـ process بعض الموارد ويطلب موردًا آخر لا يمكن تخصيصه على الفور (أي ، يجب أن ينتظر) ، فعندها، جميع الموارد التي بحوزة الـ process في ذلك الحين تنتزع منها.

تتم إضافة الموارد التي انتزعت إلى قائمة الموارد التي من أجلها قد وضعت الـ process في حالة انتظار. سيتم إعادة تشغيل ال process فقط عندما يمكنها استعادة مواردها القديمة ، وكذلك الجديدة التي تطلبها. غالبًا ما يتم تطبيق هذا البروتوكول على الموارد التي يمكن حفظ حالتها بسهولة واستعادتها لاحقًا ، مثل سجلات وحدة المعالجة المركزية واجراءات قاعدة البيانات.

3.Circular Wait

One way to ensure that this condition never holds *is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.* To illustrate, we let *R = {R*1, *R*2, ..., *Rm }* be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

احد الطرائق للتأكد من أن هذا الشرط لا ينطبق أبدًا هو فرض ترتيب إجمالي لـجميع أنواع الموارد والمطالبة بأن يطلب كل process الموارد بترتيب تصاعدي. للتوضيح ، افترض *R = {R*1, *R*2, ..., *Rm }* هي مجموعة أنواع الموارد. يعين عدد صحيح فريد لكل نوع مورد ، مما يسمح بمقارنة مصدرين وتحديد ايهما يجب ان يسبق يسبق الآخر في الطلب.

Deadlock Avoidance

Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.

إن الآثار الجانبية المحتملة لمنع حالات الجمود بطريقة الـ deadlock prevention هي انخفاض استخدام الجهاز وانخفاض إنتاجية النظام.

طريقة بديلة لتجنب الـ deadlock هي طلب معلومات إضافية حول كيفية طلب الموارد. يتطلب كل طلب عند اتخاذ قرار اجابة الطلب(منح الموارد) أن يأخذ النظام في الاعتبار الموارد المتاحة حاليًا ، والموارد المخصصة حاليًا لكل process ، والطلبات وعمليات التحرير المستقبلية لكل process .وبالنظر إلى هذه المعلومات

المسبقة ، من الممكن بناء خوارزمية تضمن أن النظام لن يدخل في حالة deadlock. تقوم خوارزمية تجنب الـ deadlock بشكل ديناميكي مستمر بفحص حالة تخصيص الموارد لضمان عدم وجود حالة انتظار دائرية.

A state is *safe* if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of threads <*T*1, *T*2, ..., *Tn*> is a safe sequence for the current allocation state if, for each *Ti*, the resource requests that *Ti* can still make can be satisfied by the currently available resources plus the resources held by all *Tj*, with $j < i$.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.

تكون الحالة آمنة إذا كان بإمكان النظام تخصيص الموارد لكل process (حتى الحد الأقصى) بترتيب ما مع الاستمرار في تجنب الوصول إلى deadlock. وبشكل اخر، يكون النظام في حالة آمنة فقط إذا كان هناك تسلسل آمن.السلسلة <T1,T2, …, Tn) هو تسلسل آمن لحالة التخصيص الحالية إذا كان من الممكن تلبية طلبات الموارد التي لا يزال بإمكان Ti القيام بها ، لكل Ti ، من خلال الموارد المتاحة حاليًا بالإضافة إلى الموارد المحتفظ بها من قبل جميع Tj ، مع j <i

الحالة الآمنة ليست حالة deadlock. على العكس من ذلك ، فإن حالة الـ deadlock هي حالة غير آمنة.

ليست كل الحالات غير الآمنة هي deadlock . الشكل 1

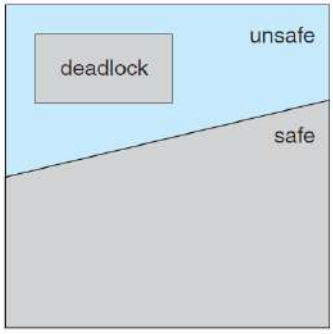


Figure 1 Safe, unsafe, and deadlocked state spaces.

Example:

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	2

Resource-Allocation-Graph Algorithm

In resource-allocation graph,we introduce a new type of edge, called a ***claim edge***. A claim edge *Ti→Rj* indicates that thread *Ti* may request resource *Rj* at some time in the future. When thread *Ti* requests resource *Rj*, the claim edge *Ti → Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Ti*, the assignment edge *Rj →Ti* is reconverted to a claim edge *Ti →Rj*. The request can be granted only if converting the request edge *Ti → Rj* to an assignment edge *Rj → Ti* does not result in the formation of a cycle in the resource-allocation graph.

الـ resource-allocation graph algorithm تقدم نوعًا جديدًا من الحافات ، يسمى claim edge. تشير حافة المطالبة Rj → Ti إلى أن Ti process قد يطلب مورد Rj في وقت ما في المستقبل. عندما يطلب الـ Ti process المورد Rj ، يتم تحويل Ti → Rj claim edge إلى request edge. وبالمثل ، عندما يتم تحرير مورد R بواسطة Ti ، يتم إعادة تحويل حافة التخصيص Rj → Ti إلى حافة مطالبة .Ti →Rj . لا يمكن منح الطلب إلا إذا كان تحويل حافة الطلب Rj → Ti إلى حافة تخصيص Rj → Ti لا يؤدي إلى تكوين دورة في الرسم البياني لتخصيص الموارد.الشكلين 2 و 3

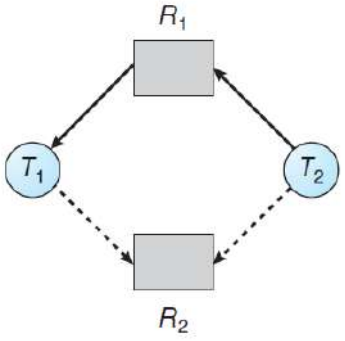


Figure 2 Resource-allocation graph for deadlock avoidance

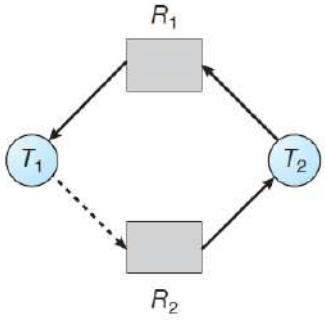


Figure 3 An unsafe state in a resource-allocation graph.

.

Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. Several data structures must be maintained to implement the banker's algorithm.

We need the following data structures, where *n* is the number of threads in the system and *m* is the number of resource types:

لا تطبق خوارزمية resource-allocation graph لتخصيص الموارد على نظام تخصيص الموارد بوجود مثيلات (عناصر) متعددة لكل نوع مورد .يجب تخصيص العديد من هياكل البيانات لتنفيذ الـ Bankers algorithm. تحتاج الخوارزمية إلى هياكل البيانات التالية ، حيث n هو عدد الـ processes في النظام و mهو عدد أنواع الموارد:

• **Available**.Avector of length *m* indicates the number of available resources of each type. If ***Available***[*j*] equals *k*, then *k* instances of resource type *Rj* are available.

متجه بطول m يشير إلى عدد الموارد المتاحة من كل نوع

Max. An $n \times m$ matrix defines the maximum demand of each thread. If ***Max***[*i*][*j*] equals *k*, then thread *Ti* may request at most *k* instances of resource type *Rj*.

مصفوفة n × m تحدد الحد الأقصى للطلب لكل process .

• **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each thread. If ***Allocation***[*i*][*j*] equals *k*, then thread *Ti* is currently allocated *k* instances of resource type *Rj*.

مصفوفة n × m تحدد عدد الموارد من كل نوع المخصصة لكل process .

• **Need**. An $n \times m$ matrix indicates the remaining resource need of each thread. If ***Need***[*i*][*j*] equals *k*, then thread *Ti* may need *k* more instances of resource type *Rj* to complete its task. Note that ***Need***[*i*][*j*] equals ***Max***[*i*][*j*] − ***Allocation***[*i*][*j*].

مصفوفة n × m تمثل الحاجة المتبقية للموارد من كل نوع لكل process

a.Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

يمكن الآن تقديم خوارزمية لمعرفة ما إذا كان النظام في حالة آمنة ام لا. يمكن وصف هذه الخوارزمية على النحو التالي:

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available*** and ***Finish***[*i*] = ***false*** for *i* = 0, 1, ..., *n* − 1.

2. Find an index *i* such that both

a. ***Finish***[*i*] == ***false***

b. ***Need****i* ≤ ***Work***

If no such *i* exists, go to step 4.

3. ***Work*** = ***Work*** + ***Allocation****i*

Finish[*i*] = ***true***

Go to step 2.

4. If ***Finish***[*i*] == ***true*** for all *i*, then the system is in a safe state.

b.Resource-Request Algorithm

بعد ذلك ، نصف الخوارزمية لتحديد ما إذا كانت الطلبات يمكن أن تكون كذلك ممنوحة بأمان.

Let ***Request****i* be the request vector for thread *Ti*. If ***Request****i* [*j*] == *k*, then thread *Ti* wants *k* instances of resource type *Rj*. When a request for resources is made by thread *Ti*, the following actions are taken :

1. If ***Request****i* ≤ ***Need****i*, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.

2. If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Ti* must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to thread *Ti* by modifying the state as follows:
Available = ***Available***–***Request****i*
Allocation*i* = ***Allocation****i* + ***Request****i*
Need*i* = ***Need****i* –***Request****i*
If the resulting resource-allocation state is safe, the transaction is completed, and thread *Ti* is allocated its resources. However, if the new state is unsafe, then *Ti* must wait for ***Requesti,*** and the old resource-allocation state is restored.

An Illustrative Example
To illustrate the use of the banker's algorithm, consider a system with five threads *T*0 through *T*4 and three resource types *A, B,* and *C.* Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances. Suppose that the following snapshot represents the current state of the system:
لتوضيح استخدام Bankers algorithm، ضع في اعتبارك نظامًا به خمسة processes من T0 إلى T4 وثلاثة أنواع من الموارد A و B و C .افترض أن اللقطة التالية تمثل الحالة الحالية للنظام:
مثال توضيحي:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

المصفوفة Need تمثل بالشكل التالي:

	Need
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

Suppose now that *T*1 request is ***Request***1 = (1,0,2). To decide whether this request can be immediately granted, we first check that Request1 ≤ Available—that is, that (1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:
افترض الآن أن طلب T1 هو(1 ،0 ،Request1 = (2.لتحديد ما إذا كان يمكن منح هذا الطلب على الفور ، نتحقق أولاً من أن الطلب 1 متوفر - أي أن (1،0،2) ≤ (3،3،2) ، وهذا صحيح. ثم نتظاهر بأن هذا الطلب قد تم الوفاء به ، ونصل إلى الحالة الجديدة التالية:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm to decide if we can immediately grant the request of thread *T*1 or to postponed granting it. A request for (3,3,0) by *T*4 cannot be granted. Why? A request for (0,2,0) by *T*0 cannot be granted, even though the resources are available. Why?

يجب أن نحدد ما إذا كانت حالة النظام الجديدة هذه آمنة. للقيام بذلك ، نقوم بتنفيذ safety algorithm الخاصة بنا لتحديد ما إذا كان بإمكاننا الموافقة على طلب الـ T1 process على الفور أو تأجيل منحه. لا يمكن منح طلب (3،3،0) من قبل .T4لماذا ا؟ لا يمكن منح طلب لـ (0،2،0) بواسطة T0 ، على الرغم من توفر الموارد. لماذا ا؟

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock.

Next, we discuss algorithms pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type.

إذا كان النظام لا يستخدم خوارزمية منع الـ deadlock أو تجنبه، فقد تحدث حالة الـ deadlock. في هذه البيئة ، قد يوفر النظام:

- خوارزمية تقوم بفحص حالة النظام لتحديد ما إذا كان قد حدث deadlock
- خوارزمية للتعافي من الـ deadlock.

فيما ياتي نناقش الخوارزميات المتعلقة بالأنظمة ذات عنصر واحد فقط من كل نوع من الموارد ، وكذلك الأنظمة التي تحتوي على عدة عناصر لكل نوع من الموارد.

1. *Single Instance of Each Resource Type*

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a ***wait-for*** graph.

إذا كانت جميع الموارد تحتوي على عنصر واحد فقط ، فيمكننا تحديد خوارزمية الكشف عن حالة الـ deadlock التي تستخدم شكلا من الـ resource-allocation graph ، يسمى الـ wait-for graph .

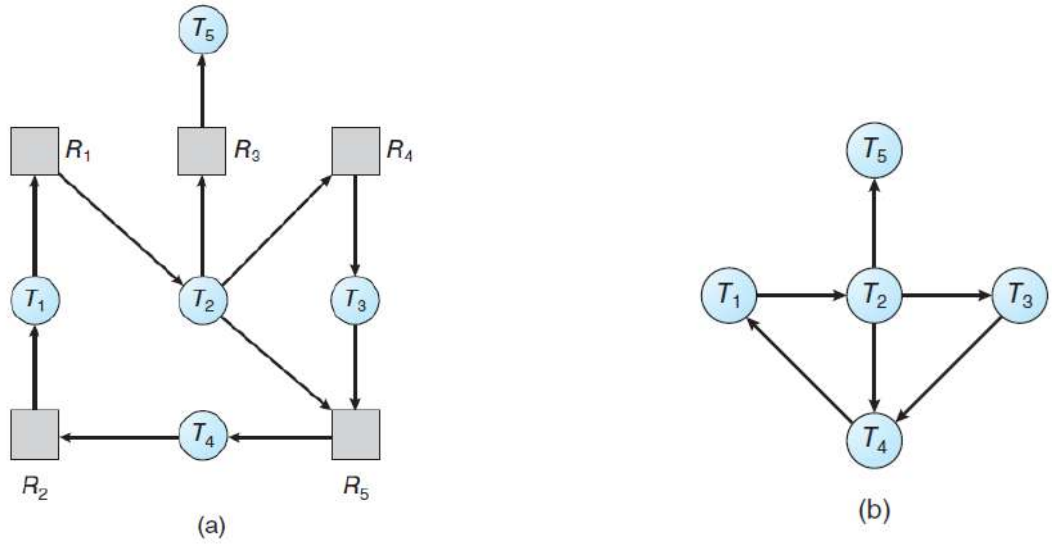


Figure 1 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to ***maintain*** the *wait for* graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph.

كما ذكر سابقا ، يوجد deadlock في النظام إذا وفقط إذا احتوى الـ wait-for graph على حلقة. لاكتشاف حالات الـ deadlock ، يحتاج النظام إلى الحفاظ على الـ wait-for graph واستدعاء خوارزمية تبحث بشكل دوري عن حلقة فيها.

2. Several Instances of a Resource Type

with multiple instances of each resource type, we turn to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.

مع عدة عناصر من كل نوع من الموارد ، ننتقل إلى خوارزمية الكشف عن حالة الـ deadlock التي تنطبق على مثل هذا النظام. تستخدم هذه الخوارزمية العديد من هياكل البيانات المتغيرة بمرور الوقت والتي تشبه تلك المستخدمة في خوارزمية المصرفي.

مثال:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*T*0, *T*2, *T*3, *T*1, *T*4> results in ***Finish***[*i*] == *true* for all *i.*

Suppose now that thread *T*2 makes one additional request for an instance of type *C.* The ***Request*** matrix is modified as follows:

النظام الان ليس في حالة deadlock. في الواقع ، إذا قمنا بتنفيذ الخوارزمية الخاصة بنا ، فسنجد أن التسلسل <T0, T2, T3, T1, T4> ينتج عنه

Finish[*i*] == *true* for all *i.*

افترض الآن أن الـ T2 process يقدم طلبًا إضافيًا لعنصر من النوع C. عندها سيتم تعديل مصفوفة الطلب على النحو التالي:

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by thread *T*0, the number of available resources is not sufficient to fulfill the requests of the other threads. Thus, a deadlock exists, consisting of threads *T*1, *T*2, *T*3, and *T*4.

نجد أن النظام أصبح الآن في حالة deadlock. وعلى الرغم من أنه يمكننا استعادة الموارد التي يحتفظ بها process T0 ، إلا أن عدد الموارد المتاحة غير كافٍ لتلبية طلبات سلاسل الرسائل الأخرى. وبالتالي ، يوجد deadlock ، يتكون من سلسلة العمليات T1 و T2 و T3 و T4.

Detection-Algorithm Usage

When to invoke the detection algorithm depends on two factors:

1. How ***often*** is a deadlock likely to occur?

2. How ***many*** threads will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Deadlocks occur only when some thread makes a request that cannot be granted immediately. In the extreme, then, we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. Of course, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent.

يعتمد وقت استدعاء خوارزمية الكشف على عاملين:

1. كم مرة يحتمل حدوث مأزق؟
2. كم عدد المواضيع التي ستتأثر بالمأزق عند حدوثه؟

في حالة حدوث حالات deadlock بشكل متكرر ، يجب استدعاء خوارزمية الاكتشاف بشكل متكرر. يحدث deadlocks فقط عندما تقدم بعض الـ processes طلبًا لا يمكن منحه على الفور. في أقصى الحدود ، إذن ، يمكننا استدعاء خوارزمية اكتشاف الـ deadlock في كل مرة يتعذر فيها منح طلب التخصيص على الفور. وعليه ، فإن استدعاء خوارزمية الكشف عن حالة الdeadlock لكل طلب مورد سيؤدي إلى زيادة كبيرة في وقت الحساب. لذلك فان البديل الأقل تكلفة هو ببساطة استدعاء الخوارزمية على فترات زمنية محددة - على سبيل المثال ، مرة واحدة في الساعة أو عندما ينخفض استخدام وحدة المعالجة المركزية إلى أقل من 40 بالمائة.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, then it can be recovered manually or automatically. There are two options for breaking a deadlock. One is simply to abort one or more threads to break the circular wait. The other is to preempt some resources from one or more of the deadlocked threads.

To eliminate deadlocks by aborting a process or thread, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

عندما تحدد خوارزمية الكشف وجود deadlock ، فيمكن ازالته يدويًا أو تلقائيًا. هناك خياران لكسر الـ deadlock. احدهما ببساطة هو انهاء واحد أو أكثر من الـ processes لكسر الانتظار الدائري. والآخر هو انتزاع بعض الموارد من واحدة أو أكثر من الـ processes التي وصلت إلى deadlock .

للتخلص من الـ deadlock عن طريق انهاء عملية أو سلسلة عمليات ، نستخدم إحدى طريقتين. في كلتا الطريقتين ، يستعيد النظام جميع الموارد المخصصة للعمليات المنتهية.

A• Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a

long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

- ***انهاء جميع الـ processes التي وصلت حالة الـ deadlock***

من الواضح أن هذه الطريقة ستكسر حلقة الـ deadlock ، لكن بتكلفة كبيرة. فربما تكون الـ processes التي وصلت إلى deadlock قد قامت بحسابات استغرقت فترة طويلة ، وحينها يجب التخلص من نتائج هذه الحسابات الجزئية وقد يتعين إعادة حسابها لاحقًا.

B• ***Abort one process at a time until the deadlock cycle is eliminated.*** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it may leave that file in an incorrect state.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated.

We should abort those processes whose termination will incur the minimum cost. Unfortunately, the term ***minimum cost*** is not a precise one.

• ***انهاء عملية واحدة في كل مرة حتى يتم التخلص من حلقة الـ deadlock***. تشكل هذه الطريقة عبئًا كبيرًا ، حيث أنه بعد انهاء كل عملية ، يجب استدعاء خوارزمية الكشف عن الـ deadlock لتحديد ما إذا كانت أي عمليات لا تزال في الـ deadlock .

قد لا يكون انهاء العملية سهلاً. إذا كانت العملية في منتصف تحديث ملف ، قد يؤدي إنهاءها إلى ترك هذا الملف في حالة غير صحيحة. إذا تم استخدام طريقة الإنهاء الجزئي ، فيجب علينا تحديد أي من العمليات التي وصلت الـ deadlock يجب إنهاءها. يجب انهاء تلك العمليات التي سيتكبد إنهاءها أقل تكلفة. لسوء الحظ ، مصطلح الحد الأدنى للتكلفة قد لا يمكن تحديده بدقة .

Many factors may affect which process is chosen, including:

1. What the priority of the process is

2. How long the process has computed and how much longer the process will compute before completing its designated task

3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)

4. How many more resources the process needs in order to complete

5. How many processes will need to be terminated.

قد تؤثر العديد من العوامل في اختيار الـ process ، بما في ذلك:

1. ما هي أولوية العملية
2. كم من الوقت قد قضت الـ process في الحوسبة وكم من الوقت اقد تبقى لها قبل إتمام المهمة الموكلة اليها
3. عدد وأنواع الموارد التي استخدمتها العملية (على سبيل المثال ، ما إذا كان من السهل انتزاع الموارد)
4. عدد الموارد التي تحتاجها العملية لإكمالها
5. كم عدد العمليات التي ستحتاج إلى إنهاء

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

للتخلص من الـ deadlocks باستخدام انتزاع الموارد ، نقوم على التوالي بانتزاع بعض الموارد من العمليات ومنح هذه الموارد لعمليات أخرى حتى يتم كسر دورة الـ deadlock .

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1.Selecting a victim.

It must be determined the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2.Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep
more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.

The most common solution is to include the number of rollbacks in the
cost factor.

إذا كان انتزاع الموارد مطلوبًا للتعامل مع الـ deadlock ، فيجب معالجة ثلاث قضايا:

1. اختيار الضحية.

يجب تحديد ترتيب لانتزاع الموارد لتقليل التكلفة. قد تتضمن عوامل التكلفة معلمات مثل عدد الموارد التي تحتفظ بها العملية المتوقفة ومقدار الوقت الذي استغرقته العملية حتى تلك اللحظة.

2. Rollback : إذا انتزعنا موردًا من عملية ما فمن الواضح أنه لا يمكنه الاستمرار في تنفيذه الطبيعي ؛ لافتقاده إلى بعض الموارد المطلوبة. في هذه الحالة يجب إعادة العملية إلى حالة آمنة وإعادة تشغيلها من تلك الحالة.

نظرًا لأنه ، بشكل عام ، من الصعب تحديد ماهية الحالة الآمنة ، فإن أبسط حل هو التراجع الكامل: اي إيقاف العملية ثم إعادة التشغيل. على الرغم من أنه من الأكثر فاعلية التراجع عن العملية فقط بقدر ما هو ضروري لكسر الـ deadlock ، فان هذه الطريقة تتطلب من النظام الحفاظ على مزيد من المعلومات حول حالة جميع الـ processes الكائنة حاليا تحت التنفيذ

3. Starvation : كيف نضمن عدم حدوث الـ starvation؟ بمعنى ، كيف يمكننا ضمان عدم استبعاد الموارد دائمًا من نفس العملية؟

في نظام يعتمد اختيار الضحايا في المقام الأول على عوامل التكلفة ، قد يحدث أن يتم اختيار نفس العملية دائمًا كضحية. ونتيجة لذلك ، لا تكمل هذه العملية مهمتها المحددة .من الواضح أننا يجب أن نضمن ان العملية يمكن اختيارها كضحية فقط لعدد محدود (صغير) من المرات . الحل الأكثر شيوعًا هو تضمين عدد مرات التراجع في عامل التكلفة.

Threads and Concurrency

The process model introduced previously assumed that a process was an executing program with a single thread of control. Virtually all modern operating systems, however, provide features enabling a process to contain multiple threads of control. Identifying opportunities for parallelism through the use of threads is becoming increasingly important for modern multicore systems that provide multiple CPUs.

يفترض نموذج العملية الذي تم تقديمه مسبقًا أن العملية عبارة عن برنامج تنفيذي ذي مسار تحكم واحد. ومع ذلك ، فإن جميع أنظمة التشغيل الحديثة تقريبًا توفر ميزات تمكّن العملية من احتواء سلاسل تحكم متعددة. لقد أصبحت فرص استخدام التوازي من خلال استخدام الـ threads ذات أهمية متزايدة للأنظمة الحديثة متعددة النواة التي توفر وحدات معالجة مركزية متعددة.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

الـ thread هو وحدة أساسية لاستخدام وحدة المعالجة المركزية ؛ وهو يتألف من معرف الـ thread، وعداد برنامج (PC) ، ومجموعة registers ، ومكدس. تشترك الـ thread مع threads أخرى تنتمي إلى نفس العملية في قسم الكود وقسم البيانات وموارد نظام التشغيل الأخرى ، مثل الملفات المفتوحة والإشارات. العملية التقليدية لها خيط (thread) تحكم واحد. إذا كانت العملية تحتوي على عدة خيوط (threads) للتحكم ، فيمكنها أداء أكثر من مهمة واحدة في كل مرة. يوضح الشكل 1 الفرق بين العملية التقليدية أحادية الخيط والعملية متعددة الخيوط.

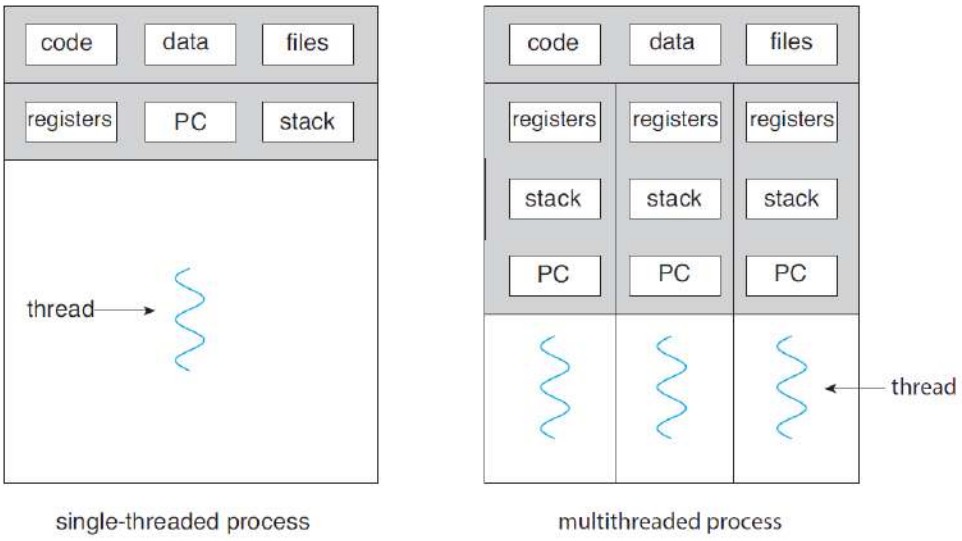


Figure 1 Single-threaded and multithreaded processes

1. **Motivation**

Most software applications that run on modern computers and mobile devices are ***multithreaded.*** An application typically is implemented as a separate process with several threads of control. Below we highlight a few examples of multithreaded applications:

• An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.

• A web browser might have one thread to display images or text while another thread retrieves data from the network.

• A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

1. الدافع

معظم تطبيقات البرامج التي تعمل على أجهزة الكمبيوتر والأجهزة المحمولة الحديثة متعددة ***الخيوط*** (***multithreaded***) . عادة ما يتم تنفيذ التطبيق كعملية منفصلة مع العديد من خيوط التحكم. فيما يلي نسلط الضوء على بعض الأمثلة على التطبيقات متعددة الخيوط(***multithreaded applications***) :

•التطبيق الذي يقوم بإنشاء صور مصغرة من مجموعة صور قد يستخدم thread منفصلة لإنشاء صورة مصغرة من كل صورة منفصلة.

•قد يحتوي مستعرض الويب على thread واحد يعرض صورًا أو نصًا بينما يقوم thread آخر باسترداد البيانات من الشبكة.

•قد يحتوي معالج الكلمات على thread لعرض الرسومات و thread آخر للرد على ضغطات المفاتيح من المستخدم وthread ثالث لإجراء التدقيق الإملائي والنحوي في الخلفية.

Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

In certain situations, a single application may be required to perform several similar tasks. For example, a busy web server may accept thousands of ***client requests*** for web pages, images, sound, and so forth. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

يمكن أيضًا تصميم التطبيقات للاستفادة من إمكانات المعالجة على الأنظمة متعددة النواة. حيث يمكن لمثل هذه التطبيقات أداء العديد من المهام كثيفة الاستخدام لوحدة المعالجة المركزية، يمكنها اداؤها بالتوازي عبر نوى الحوسبة المتعددة.

في حالات معينة ، قد يكون المطلوب من تطبيق واحد اداء عدة مهام مماثلة. على سبيل المثال ، قد يقبل خادم ويب الآلاف من ***طلبات العميل*** لصفحات الويب والصور والصوت وما إلى ذلك. إذا كان خادم الويب يعمل كعملية تقليدية ***أحادية الخيط*** (***single-threaded***) ، فسيكون قادرًا على خدمة عميل واحد فقط في كل مرة ، وقد يضطر العميل إلى الانتظار وقتًا طويلاً حتى تتم خدمة طلبه.

One solution is to have the server run as a single process that accepts requests, then it creates a separate process to service that request. Problem with this solution is that process creation is time consuming and resource intensive. It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests, then rather than creating another process, the server creates a new thread to service the request and resumes listening for additional requests. This is illustrated in Figure 2.

يتمثل أحد الحلول في تشغيل الخادم كعملية واحدة تقبل الطلبات ، ثم يقوم بإنشاء عملية منفصلة لخدمة هذا الطلب. تكمن مشكلة هذا الحل في أن إنشاء العملية يستغرق وقتًا طويلاً ويستهلك الكثير من الموارد. ان من الأفضل عمومًا استخدام عملية واحدة تحتوي على مجموعة threads. فإذا كانت عملية خادم الويب متعددة الخيوط (multithreaded) ، فسيقوم الخادم بإنشاء thread منفصلة تستمع إلى طلبات العميل ، ثم بدلاً من إنشاء عملية أخرى ، يقوم الخادم بإنشاء thread جديدة لخدمة الطلب ويستأنف الاستماع للطلبات الإضافية. هذا موضح في الشكل 2.

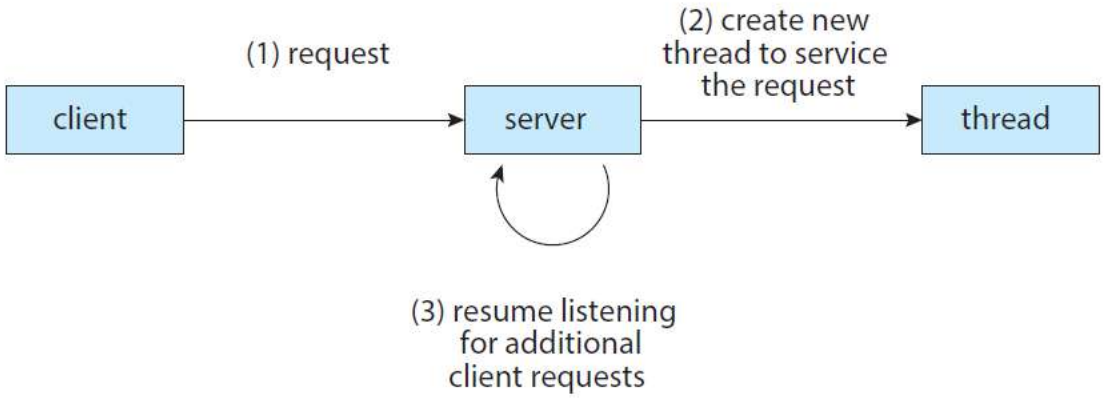


Figure 2 Multithreaded server architecture.

Most operating system kernels are also typically multithreaded. As an example, during system boot time on Linux systems, several kernel threads are created. Each thread performs a specific task, such as managing devices, memory management, or interrupt handling.
Many applications can also take advantage of multiple threads, including basic sorting, trees, and graph algorithms. In addition, programmers who must solve contemporary CPU-intensive problems in data mining, graphics, and artificial intelligence can leverage the power of modern multicore systems by designing solutions that run in parallel.
معظم نوى نظام التشغيل عادةً ما تكون متعددة الخيوط (multithreaded).فعلى سبيل المثال ، أثناء وقت تمهيد النظام على أنظمة Linux ، يتم إنشاء العديد من خيوط (threads) الـ kernel.
والتي يؤدي كل منها مهمة محددة ، مثل إدارة الأجهزة أو إدارة الذاكرة أو معالجة المقاطعة.
كما يمكن للعديد من التطبيقات أيضًا الاستفادة من الخيوط المتعددة ، بما في ذلك الفرز الأساسي والأشجار وخوارزميات الرسم البياني. بالإضافة إلى ذلك ، يمكن للمبرمجين الذين يقومون بحل المشكلات المعاصرة -التي تعتمد على الاستخدام المكثف لوحدة المعالجة المركزية في استخراج البيانات والرسومات والذكاء الاصطناعي -الاستفادة من قوة الأنظمة الحديثة متعددة النواة من خلال تصميم حلول تعمل بالتوازي.

2 Benefits

The benefits of ***multithreaded programming*** can be broken down into four major categories:

1. ***Responsiveness***. Multithreading an interactive application may allow a program to continue running even if part of it is blocked

or is performing a lengthy operation, thereby increasing responsiveness to the user.

2. ***Resource sharing***. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. ***Economy.*** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. In general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. ***Scalability***. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless of how many are available.

الفوائد

يمكن تقسيم فوائد البرمجة متعددة الـ threads إلى أربع فئات رئيسية:

1. ***الاستجابة*** (Responsiveness). قد يسمح انشاء التطبيق التفاعلي كمتعدد الـ threads له بالاستمرار في العمل حتى إذا تم توقف جزء منه أو قام بإجراء عملية طويلة ، ويؤدي ذلك بالتالي الى زيادة الاستجابة للمستخدم.
2. ***تقاسم الموارد*** (Resource sharing) . تتمثل فائدة مشاركة التعليمات البرمجية والبيانات في أنها تتيح للتطبيق الحصول على العديد من خيوط التنفيذ المختلفة داخل نفس مساحة العنوان .
3. ***الاقتصاد*** (Economy). يعد تخصيص الذاكرة والموارد لإنشاء العملية أمرًا مكلفًا. ونظرًا لأن الخيوط تشترك في موارد العملية التي تنتمي إليها ، فمن الأكثر اقتصادا إنشاء الـ threadsواستخدامها في تبديل السياق. وبشكل عام ، فإن إنشاء thread يستهلك وقتًا وذاكرة أقل من إنشاء العملية. بالإضافة إلى ذلك ، يكون تبديل السياق عادةً أسرع بين الـ threads مما هو منه بين العمليات.
4. ***قابلية التوسع*** (Scalability). يمكن أن تكون فوائد الـ multithreading أكبر في بنية متعددة المعالجات ، حيث قد تعمل الخيوط (threads) بالتوازي على نوى معالجة مختلفة. يمكن تشغيل العملية ذات الخيط الواحد على معالج واحد فقط ، بغض النظر عن عدد المعالجات المتاحة.

Multicore Programming

Earlier in the history of computer design, single-CPU systems evolved into multi-CPU systems. A later trend in system design is to place multiple computing cores on a single processing chip where each core appears as a separate CPU to the operating system. We refer to such systems as ***multicore, and multithreaded programming***. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. On a system with multiple cores, however, **concurrency** means that some threads can run in parallel because the system can assign a separate thread to each core

البرمجة متعددة النواة

في وقت سابق من تاريخ تصميم الكمبيوتر ، تطورت أنظمة وحدة المعالجة المركزية الواحدة إلى أنظمة متعددة وحدات المعالجة المركزية. ظهر بعدها اتجاه في تصميم النظام وهو وضع نوى حوسبة متعددة على شريحة معالجة واحدة حيث تظهر كل نواة كوحدة معالجة مركزية منفصلة لنظام التشغيل. يشار الى هذه الانظمة كانظمة برمجة متعددة النواة . توفر البرمجة متعددة الخيوط (***multithreaded programming***) آلية لاستخدام أكثر كفاءة لهذه النوى الحاسوبية المتعددة وللتزامن المحسن. وفي نظام متعدد النوى ، يعني التزامن أن بعض الخيوط يمكن أن تعمل بالتوازي ، لأن النظام يمكنه تعيين thread منفصلة لكل نواة

Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. Consider, for example, summing the contents of an array of size *N*. On a single-core system, one thread would simply sum the elements [0] . . . [$N - 1$]. On a dual-core system, however, thread *A*, running on core 0, could sum the elements [0] . . . [$N/2 - 1$] while thread *B*, running on core 1, could sum the elements [$N/2$] . . . [$N - 1$]. The two threads would be running in parallel on separate computing cores. **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data. Consider again our example above. In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

بشكل عام ، هناك نوعان من التوازي: توازي البيانات **Data parallelism** وتوازي المهام **Task parallelism**. يركز توازي البيانات على توزيع مجموعات فرعية من نفس البيانات عبر نوى حوسبة متعددة وتنفيذ نفس العملية على كل نواة. فلناخذ على سبيل المثال ، جمع محتويات مصفوفة من الحجم N. في نظام أحادي النواة ، يقوم thread واحد بجمع العناصر [0]. . . [N - 1]. ومع ذلك ، في نظام ثنائي النواة ، يمكن للـ A thread ، الذي يعمل على النواة 0 ، جمع العناصر [0]. . . [N/2 - 1]بينما الـ thread B ، الذي يعمل على النواة 1 ، يمكنه جمع العناصر [N-1] . . .[N/2]. سيتم تشغيل الخيطين بالتوازي على نوى حوسبة منفصلة. يتضمن توازي المهام توزيع المهام (الخيوط) عبر نوى الحوسبة المتعددة. كل thread يقوم بعملية فريدة. قد تعمل الـ threads المختلفة على نفس البيانات ، أو قد تعمل على بيانات مختلفة. وعلى عكس المثال في توازي البيانات ، قد يشتمل مثال توازي المهام على خيطين ، كل منهما يؤدي عملية إحصائية معينةعلى مجموعة العناصر. وهنا تعمل الthread مرة أخرى بالتوازي على انوية حوسبة منفصلة ، لكن كل منها يؤدي عملية فريدة.

Fundamentally, then, data parallelism involves the distribution of data across multiple cores, and task parallelism involves the distribution of tasks across multiple cores, as shown in Figure 1. However, data and task parallelism are not mutually exclusive, and an application may in fact use a hybrid of these two strategies.

بشكل أساسي ، إذن ، يتضمن توازي البيانات توزيع البيانات عبر نوى متعددة ، وتوازي المهام يتضمن توزيع المهام عبر نوى متعددة ، كما هو موضح في الشكل 1. ومع ذلك ، فإن توازي البيانات والمهام لا يستبعد أحدهما الآخر ، وقد يكون التطبيق في الواقع استخدام مزيج من هاتين الاستراتيجيتين.

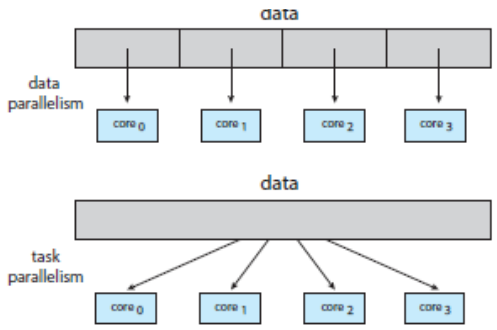


Figure 1. Data and task parallesim

Multithreading Models

Support for threads may be provided either at the user level, for ***user threads***, or by the kernel, for ***kernel threads***. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, and mac OS— support kernel threads.

قد يتم توفير الدعم للـ thread إما على مستوى المستخدم ، للـ user threads ، أو عن طريق النواة ، للـ kernel threads. يتم دعم الـ user threads فوق kernel وتتم إدارتها بدون دعم kernel ، في حين تدعم kernel threads وتدار مباشرةً بواسطة نظام التشغيل. تدعم جميع أنظمة التشغيل المعاصرة تقريبًا - بما في ذلك Windows و Linux و Mac OS – الـKernel threads

Ultimately, a relationship must exist between user threads and kernel threads. We look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

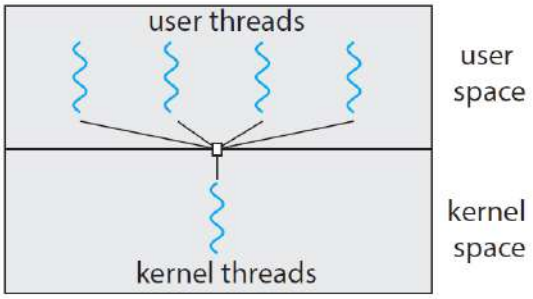
في النهاية ، يجب أن توجد علاقة بين الـ user threads والـ kernel threads. نلقي نظرة على ثلاث طرق شائعة لتأسيس مثل هذه العلاقة: نموذج many-to-one ونموذج the one-to-one ونموذج many-to-many

1. Many-to-One Model

The many-to-one model (Figure 2) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

يقوم نموذج many-to-one (الشكل 2) بتعيين العديد من الـ user threads إلى kernel thread واحد. تتم إدارة الـ thread بواسطة مكتبة الـ threads في مساحة المستخدم ، لذا فهي فعالة .ومع ذلك ، سيتم حظر العملية بأكملها إذا قام thread بإجراء استدعاء نظام مسببا للحظر. و نظرًا لأن thread واحد فقط يمكنه الوصول إلى النواة في وقت واحد ، فإن الـ threads المتعددة غير قادرة على العمل بالتوازي على أنظمة متعددة النواة.

Figure 2. Many to one model



2. One-to-One Model

The one-to-one model (Figure 3) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system.

يقوم النموذج one-to-one (الشكل 3) بتعيين كل user thread إلى kernel thread .يوفر النموذج تزامنًا أكثر من نموذج many-to-one من خلال السماح لـ thread آخر بالتشغيل عندما يقوم thread بإجراء استدعاء نظام مسببا للحظر. كما يسمح أيضًا بتشغيل threads متعددة بالتوازي على معالجات متعددة. العيب الوحيد لهذا النموذج هو أن إنشاء user thread يتطلب إنشاء kernel thread المقابل ، وقد يثقل العدد الكبير من الـ kernel threads أداء النظام.

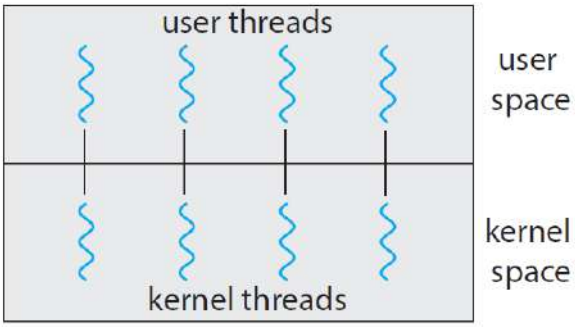
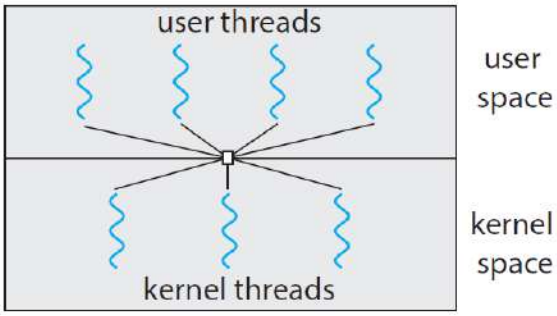


Figure 3. One to one model

3. Many-to-Many Model

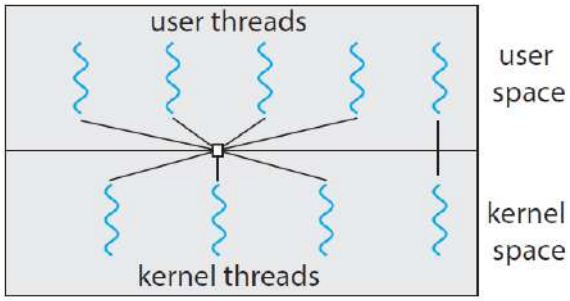
The many-to-many model (Figure 4) multiplexes many user-level threads to a smaller or equal number of kernel threads.

Figure 4. Many to many model



يعمل نموذج many-to-many (الشكل 4) على تعدد إرسال العديد من user-level threads إلى عدد أصغر أو متساومن الـ kernel thread .

One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to



be bound to a kernel thread. This variation is sometimes referred to as the ***two-level model*** (Figure 5).

Figure 5. Two level model

شكل اخر من نموذج many-to-many يقوم على تعدد إرسال العديد من user-level threads إلى عدد أصغر أو متساوٍ من الـ kernel threads ولكنه يسمح أيضًا بربط user-level thread بـال kernel thread. يشار إلى هذا الاختلاف أحيانًا بالنموذج ذي المستويين (الشكل 5).

Memory Management

Introduction

Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory; they must be moved there before the CPU can operate on them.

تتكون الذاكرة من مجموعة كبيرة من البايتات ، ولكل منها عنوانها الخاص. تقوم وحدة المعالجة المركزية بجلب الايعازات من الذاكرة وفقًا لقيمة عداد البرنامج. الذاكرة الرئيسية والسجلات المضمنة في كل CPU هي محل التخزين -للأغراض العامة - الوحيد الذي يمكن للـ CPU الوصول إليه مباشرة. لذلك ، فان أي ايعازات تنفيذية ، وأي بيانات يتم استخدامها في الايعازات ، يجب أن تكون في أحد أجهزة التخزين ذات الوصول المباشر. وإذا لم تكن البيانات في الذاكرة ؛ يجب نقلها إلى هناك قبل أن تتمكن وحدة المعالجة المركزية من العمل عليها.

To lessen the frequency of accessing the memory and the time needed to access it, fast memory between the CPU and main memory called *cache* is added on the CPU chip. For proper system operation, the operating system must be protected from access by user processes, as well as protect user processes from one another. Hardware implements this protection in several ways. Here, we outline one possible implementation.

Protection can be provided by using two registers, usually a base and a limit, as illustrated in Figure 1.

لتقليل مرات الوصول إلى الذاكرة والوقت اللازم للوصول إليها ، تتم إضافة ذاكرة سريعة بين وحدة المعالجة المركزية والذاكرة الرئيسية تسمى ذاكرة التخزين المؤقت (cache) على شريحة وحدة المعالجة المركزية. وليعمل النظام بشكل صحيح ، يجب حماية نظام التشغيل من الوصول اليه من قبل عمليات المستخدم ، وكذلك حماية عمليات المستخدم من بعضها البعض. تنفذ الأجهزة هذه الحماية بعدة طرق مختلفة ، وسنوضح هنا طريقة واحدة منها.

يمكن توفير الحماية باستخدام سجلين ، هما الـ base والـ limit ، كما هو موضح في الشكل 1.

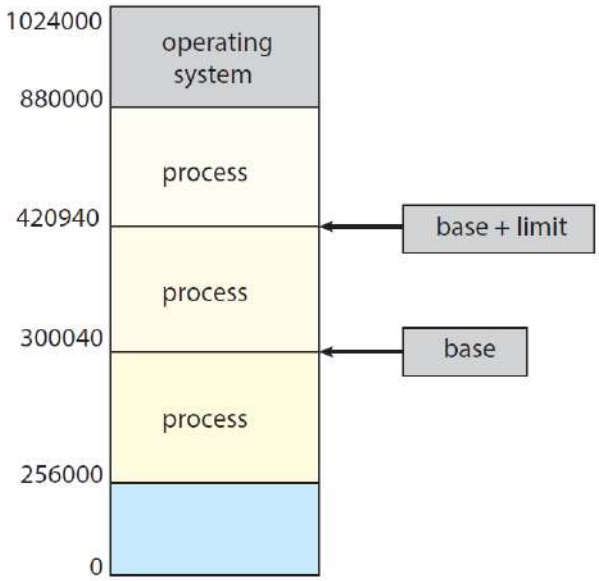


Figure 1 A base and a limit register define a logical address space.

The ***base register*** holds the smallest legal physical memory address; the ***limit register*** specifies the size of the range. Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers as illustrated in Figure 2.
يحتوي سجل الـ base على أصغر عنوان ذاكرة فعلي للـ process ؛ يحدد سجل الـ limit حجم النطاق (البرنامج). يتم تحقيق حماية مساحة الذاكرة من خلال جعل أجهزة وحدة المعالجة المركزية تقارن كل عنوان تم إنشاؤه في وضع المستخدم مع السجلات كما هو موضح في الشكل 2.

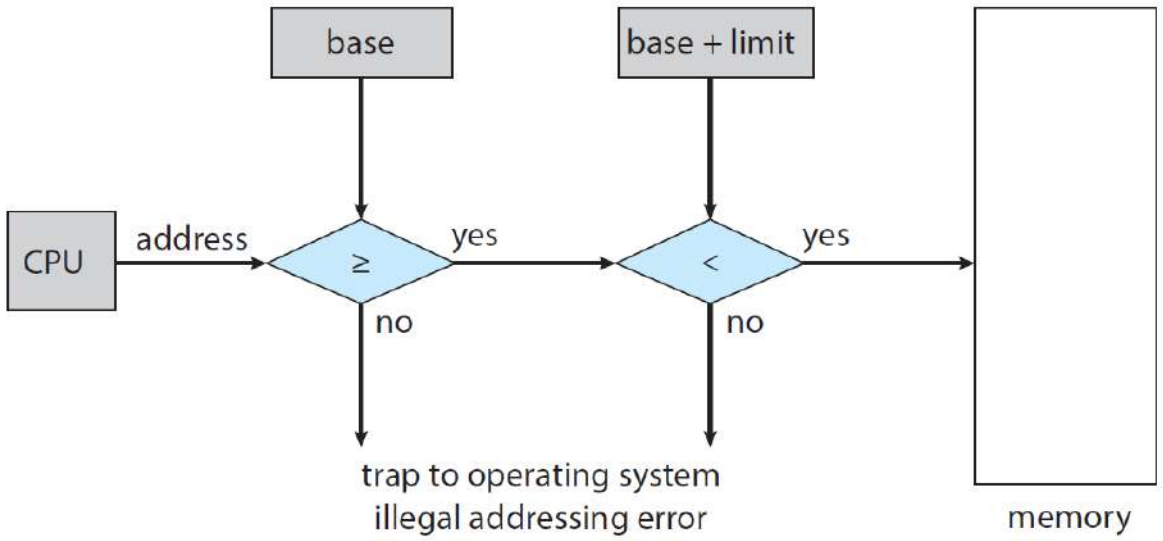


Figure 2 Hardware address protection with base and limit registers.

2 Address Binding

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000.

Addresses in the source program are generally symbolic (such as the variable *count*). A compiler typically ***binds*** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linker or loader in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way (figure 3):

تسمح معظم الأنظمة لـ process المستخدم بالبقاء في أي جزء من الذاكرة الرئيسية. وبالتالي ، على الرغم من أن مساحة عنوان الكمبيوتر قد تبدأ عند 00000 ، لا يلزم أن يكون العنوان الأول لـ process المستخدم هو 00000.

تكون العناوين في البرنامج المصدر رمزية بشكل عام (مثل عدد المتغيرات). عادةً ما يقوم المترجم بربط هذه العناوين الرمزية بالعناوين القابلة لإعادة تحديد موضعها (relocatable addresses)(مثل "14 بايت من بداية هذه الوحدة"). يقوم الرابط linker أوالمحمل (loader) بدوره بربط الـ relocatable addresses بالعناوين المطلقة (absolute addresses) (مثل 74014). كل binding هو تغيير تمثيل العنوان من مساحة عنوان إلى أخرى. تقليديًا ، يمكن ربط الايعازات والبيانات بعناوين الذاكرة في أي خطوة على طول الطريق (الشكل 3):

- *Compile time*. If you know at compile time where the process will reside in memory, then ***absolute code*** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

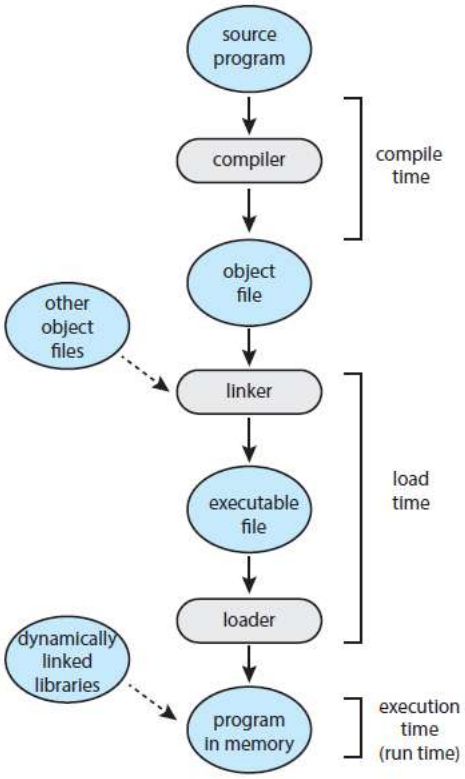


Figure 3 Multistep processing of a user program.

لو علم في وقت الترجمة المكان الذي ستقيم فيه الـ process في الذاكرة ، فيمكن عندئذٍ إنشاء absolute code. على سبيل المثال ، إذا علم أن عملية المستخدم ستقيم بدءًا من الموقع R ، فستبدأ الشفرة المكونة من قبل الـ compiler في هذا الموقع وتمتد من هناك. إذا تغير موقع البداية في وقت لاحق ، فسيكون من الضروري إعادة ترجمة هذاه الشفرة.

- *Load time*. If it is not known at compile time where the process will reside in memory, then the compiler must generate ***relocatable code***. In this case, the final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

- وقت التحميل. إذا لم يكن معروفًا في وقت الترجمة مكان وجود الـ process في الذاكرة ، فيجب على الـ compiler إنشاء relocatable code . في هذه الحالة ، يتم تأخير الربط النهائي للعناوين حتى وقت التحميل. إذا تغير عنوان البداية ، فسنحتاج فقط إلى إعادة تحميل شفرة المستخدم لدمج هذه القيمة المتغيرة.

- *Execution time*. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work.
- وقت التنفيذ. إذا كان من الممكن نقل الـ process أثناء تنفيذها من مقطع ذاكرة إلى آخر ، فيجب تأخير الربط حتى وقت التشغيل. يجب أن تتوفر أجهزة خاصة لكي تنفذ عملية الربط هذه.

3 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a *logical address*, whereas an address seen by the memory unit that is, the one loaded into the *memory-address register* of the memory is commonly referred to as a *physical address*.

عادةً ما يُشار إلى العنوان الذي يتم إنشاؤه بواسطة وحدة المعالجة المركزية بالعنوان المنطقي (logical address) ، في حين يُشار إلى العنوان الذي تراه وحدة الذاكرة ، وهو العنوان الذي تم تحميله في سجل عناوين الذاكرة (memory-address register) للذاكرة ، بالـ physical address .

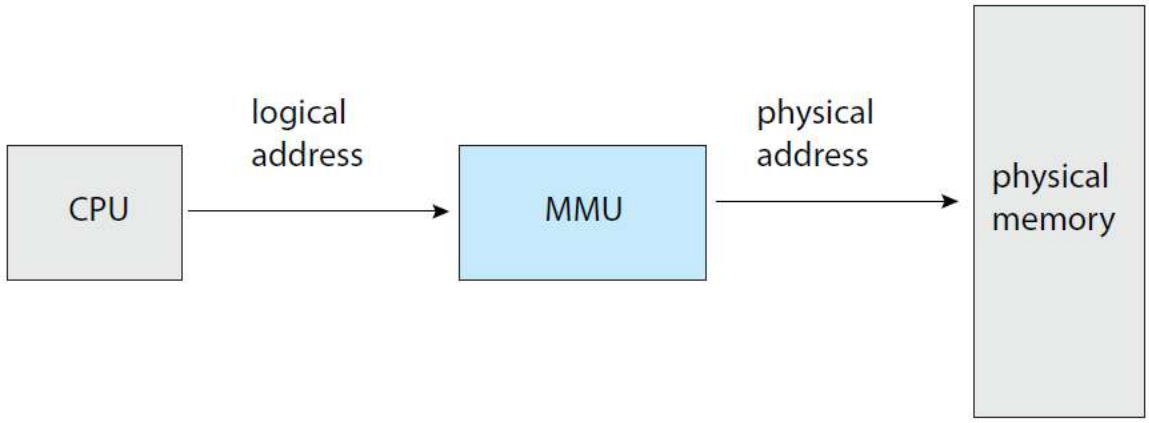


Figure 4 Memory management unit (MMU).

Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a ***virtual address***. The set

of all logical addresses generated by a program is a ***logical address space***. The set of all physical addresses corresponding to these logical addresses is a ***physical address space***. The run-time mapping from virtual to physical addresses is done by a hardware device called the ***memory-management unit (MMU***) (Figure 4).

يؤدي ربط العناوين في وقت التجميع أو التحميل إلى إنشاء عناوين منطقية logical ومادية physical متطابقة. ومع ذلك ، بينما ينتج ربط العنوان في وقت التنفيذ عناوين منطقية ومادية مختلفة. في هذه الحالة ، يشارعادةً إلى العنوان المنطقي كعنوان افتراضي virtual address. ان مجموعة جميع العناوين المنطقية التي يتم إنشاؤها بواسطة البرنامج هي مساحة عنوان منطقي (logical address space). اما مجموعة جميع العناوين الفعلية المقابلة لهذه العناوين المنطقية هي مساحة العنوان الفعلية (physical address space).ان تغيير تمثيل العناوين في وقت التنفيذ من virtual addresses إلى logical addresses يتم بواسطة جهاز يسمى وحدة إدارة الذاكرة (MMU) الشكل 4.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range *R* + 0 to *R* + *max* for a base value *R*). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max.* However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management (figure 5).

لدينا الآن نوعان مختلفان من العناوين: العناوين المنطقية logica addresses (في النطاق من 0 إلى max) والعناوين الفعلية physical addresses (في النطاق R + 0 إلى R + max للقيمة الأساسية R).يقوم برنامج المستخدم بإنشاء logical addresses فقط ويفترض أن العملية تعمل في مواقع الذاكرة من 0 إلى الحد الأقصى. ومع ذلك ، يجب تغيير تمثيل هذه الـ logical addresses إلى الـ physical addresse قبل استخدامها للوصول الى الذاكرة (الشكل 5).

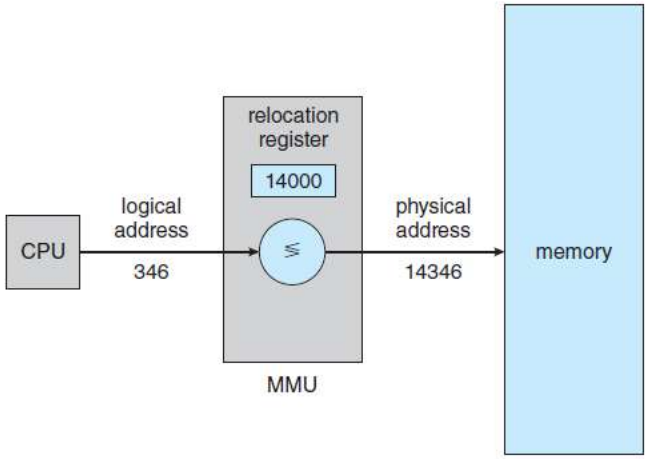


Figure 5 Dynamic relocation using a relocation register.

4 Dynamic Loading

If the entire program and all data of a process are to be in physical memory for the process to execute, then the size of a process has been limited to the size of physical memory. To obtain better memory-space utilization, we can use ***dynamic loading***. With dynamic loading, a routine is not loaded until it is called. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, then the desired routine is loaded into memory and the program's address tables are updated to reflect this change. Then control is passed to the newly loaded routine.

إذا كان البرنامج بأكمله وجميع بيانات الـ process موجودة في الذاكرة الرئيسية لاجل تنفيذ العملية , فعندها سيتحدد حجم الـ process بحجم الـ physical memory. وللحصول على استخدام أفضل لمساحة الذاكرة ، يمكننا استخدام التحميل الديناميكي (dynamic loading). مع التحميل الديناميكي ، لا يتم تحميل روتين حتى يتم استدعاؤه. يتم تحميل البرنامج الرئيسي في الذاكرة ويتم تنفيذه. عندما يحتاج روتين إلى استدعاء روتين آخر ، يتحقق روتين الاستدعاء أولاً لمعرفة ما إذا كان قد تم تحميل الإجراء الآخر. فاذا لم يكن محملا ، فسيتم تحميل الروتين المطلوب في الذاكرة ويتم تحديث جداول عناوين البرنامج لتعكس هذا التغيير. ثم يتم تمرير التحكم إلى الروتين الذي تم تحميله حديثًا.

The advantage of dynamic loading is that a routine is loaded only when it is needed. In such a situation, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system.

تتمثل ميزة الـ dynamic loading في أنه يتم تحميل الروتين فقط عند الحاجة إليه. في مثل هذه الحالة ، على الرغم من أن الحجم الإجمالي للبرنامج قد يكون كبيرًا ، إلا أن الجزء المستخدم (ومن ثم تحميله) قد يكون أصغر بكثير.

لا يتطلب التحميل الديناميكي دعمًا خاصًا من نظام التشغيل.

5 Dynamic Linking and Shared Libraries

Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run (refer back to Figure 3). Some operating systems support only ***static linking***, in which system libraries are treated like any other object module and are combined by the loader into the binary program image. In dynamic linking, linking, rather than loading, is postponed until execution time. Dynamic linking decreases the size of an executable image and may save main memory. A second advantage of DLLs is that these libraries can be shared among multiple processes so that only one instance of the DLL is in main memory.

المكتبات المرتبطة ديناميكيًا (DLLs) هي مكتبات نظام يتم ربطها ببرامج المستخدم عند تنفيذ هذه البرامج (راجع الشكل 3). تدعم بعض أنظمة التشغيل الـ static linking فقط ، والذي يتم فيه التعامل مع مكتبات النظام مثل أي object module أخر ويتم دمجها بواسطة الـ loader في شفرة البرنامج الثنائي. يتم في الـ dynamic linking تأجيل الارتباط فضلا عن التحميل حتى وقت التنفيذ. يقلل الـ dynamic linking من حجم الصورة القابلة للتنفيذ وقد يوفر من مساحة الذاكرة الرئيسية. الميزة الثانية لـ DLL هي أنه يمكن مشاركة هذه المكتبات بين processes متعددة ، بحيث تكون نسخة واحدة فقط من DLL في الذاكرة الرئيسية.

When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary. It then adjusts addresses that reference functions in the dynamic library to the location in memory where the DLL is stored.

عندما يشير أحد البرامج إلى روتين موجود في مكتبة ديناميكية ، يقوم الـ loader بتحديد موقع DLL ، وتحميلها في الذاكرة إذا لزم الأمر. ثم يقوم بضبط العناوين التي تشير إلى الوظائف في المكتبة الديناميكية إلى الموقع الموجود في الذاكرة حيث يتم تخزين .DLL

Dynamically linked libraries can be extended to library updates (such as bug fixes). In addition, a library may be replaced by a new version, and

all programs that reference the library will automatically use the new version.

Dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

توفر الـ DDL امكانية التحديث (مثل إصلاحات الأخطاء). بالإضافة إلى ذلك ، قد يتم استبدال المكتبة بإصدار جديد ، وستستخدم جميع البرامج التي تشير إلى المكتبة الإصدار الجديد تلقائيًا.

يتطلب الربط الديناميكي (dynamic linking) والمكتبات المشتركة (shared libraries) بشكل عام المساعدة من نظام التشغيل. إذا كانت الـ processes في الذاكرة محمية من بعضها البعض ، فإن نظام التشغيل هو الكيان الوحيد الذي يمكنه التحقق لمعرفة ما إذا كان الروتين المطلوب موجودًا في مساحة ذاكرة process أخرى أو يمكنه السماح لعدة processes بالوصول إلى عناوين الذاكرة نفسها.

Contiguous Memory Allocation

The memory is usually divided into two partitions: one for the operating system and one for the user processes. We usually want several user processes to reside in memory at the same time. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

تنقسم الذاكرة عادةً إلى قسمين: أحدهما لنظام التشغيل والآخر لعمليات المستخدم. ينبغي عادةً أن تبقى العديد من عمليات المستخدم في الذاكرة في نفس الوقت. في contiguous memory allocation ، يتم احتواء كل process في مقطع من الذاكرة مجاور للمقطع الذي يحتوي على process التالية.

Memory Protection

A process can be prevented from accessing memory that it does not own by combining the idea of relocation register with the idea of limit register.

Because every address a CPU generates is checked against these registers, we can protect the operating system and the other users' programs and data from being modified by this running process.

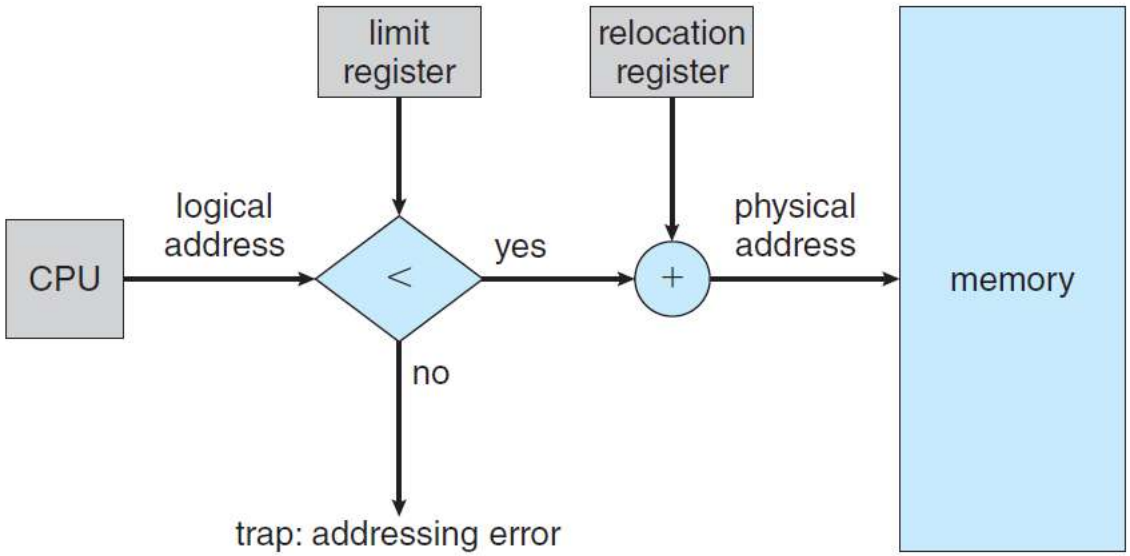


Figure 1 Hardware support for relocation and limit registers.

يمكن منع الـ process من الوصول إلى الذاكرة التي لا تمتلكها من خلال الجمع بين فكرة الـ relocation register مع فكرة الـ limit register .

ولأن كل عنوان تنشئه وحدة المعالجة المركزية يتم فحصه مقابل هذه السجلات ، لذلك يمكن حماية نظام التشغيل وبرامج المستخدمين الآخرين وبياناتهم من التعديل من خلال ال process قيد التشغيل.

2 Memory Allocation

One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process.

In this *variable partition* scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Fig 2 depicts this scheme.

تتمثل إحدى أبسط طرق تخصيص الذاكرة في وضع الـ processes في اقسام متغيرة الحجم في الذاكرة ، حيث قد يحتوي كل قسم على process واحدة بالضبط. يحتفظ نظام التشغيل في نظام الـ *variable partition* هذا ، بجدول يشير إلى أجزاء الذاكرة المتوفرة والأجزاء المشغولة. الشكل 2 يصور هذا المخطط.

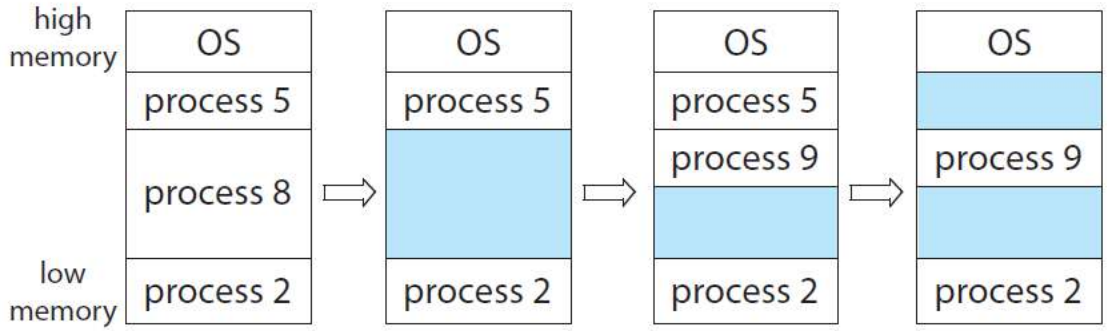


Figure 2 Variable partition.

As processes enter the system, the operating system allocates and loads it into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process. The memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. The first-fit, best-fit, and worst-fi strategies are the ones most commonly used to select a free hole from the set of available holes.

عندما تدخل الـ processes إلى النظام ، يقوم نظام التشغيل بتخصيص ذاكرة لها وتحميلها فيها، حيث يمكنها بعد ذلك التنافس على وقت وحدة المعالجة المركزية. عندما تنتهي process ما ، فإنها تحرر ذاكرتها ، والتي قد يوفرها نظام التشغيل بعد ذلك ل process أخرى. تتكون مساحات الذاكرة المتاحة من مجموعة من الفجوات ذات الأحجام المختلفة المنتشرة في جميع أنحاء الذاكرة. عندما تصل process ما وتحتاج إلى ذاكرة ، يبحث النظام في مجموعة الفجوات عن فجوة كبيرة بما يكفي لهذه الـ process. ان الإستراتيجيات الأكثر استخدامًا لتحديد فجوة حرة من مجموعة الفجوات المتاحة هي first-fit، best-fit ، و worst-fit .

- **First fit.** Allocate the first hole that is big enough.
- **Best fit**. Allocate the smallest hole that is big enough. This strategy produces the smallest leftover hole.
- **Worst fit.** Allocate the largest hole. This strategy produces the largest leftover hole.

- First fit. ويتم فيها تخصيص الفجوة الأولى الكبيرة بما يكفيلاستيعاب ال process
- Best fit. ويتم فيها تخصيص أصغر فجوة كبيرة بما يكفي لاستيعاب ال process. تنتج هذه الاستراتيجية أصغر فجوة متبقية.
- Worst fit. ويتم فيها تخصيص أكبر فجوة. تنتج هذه الاستراتيجية أكبر فجوة متبقية .

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.
أظهرت عمليات المحاكاة أن كلاً من first fit و best fit أفضل من worst fit من حيث تقليل الوقت واستخدام التخزين. من الواضح أنه ليس لكل من الـ first fit ولا الـ best fit أفضلية على الآخرى من حيث استخدام التخزين ، ولكن الـ first fit تكون أسرع عمومًا.

3 Fragmentation

A: External fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from ***external fragmentation***. The free memory space is broken into little pieces as processes are loaded and removed from memory. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous.

B: Internal fragmentation

Memory fragmentation can be internal. Consider a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If the requested block is allocated exactly, a hole of 2 bytes will be left. The difference between these two numbers is internal fragmentation —unused memory that is internal to a partition.

One solution to the problem of external fragmentation is ***compaction***. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however.

3التجزئة

ا: التجزئة الخارجية

تعاني كل من استراتيجيتي الـ first fit والـ best fit لتخصيص الذاكرة من التجزئة الخارجية. ان تحميل العمليات وإزالتها من الذاكرة يتسبب في تقسيم مساحة الذاكرة الخالية إلى أجزاء صغيرة. توجد التجزئة الخارجية عندما يكون مجموع مساحات الذاكرة الإجمالية كاف لتلبية الطلب ولكن المساحات هذ ليست متجاورة.

ب: التجزئة الداخلية

يمكن أن يكون تجزئة الذاكرة داخليًا. افترض فجوة تبلغ مساحتها 18464 بايت. افترض أن العملية التالية تتطلب 18462 بايت. إذا تم تخصيص المساحة المطلوبة بالضبط ، فسيتم ترك مساحة صغيرة بحجم 2 بايت. الفرق بين هذين الرقمين هو التجزئة الداخلية - ذاكرة غير مستخدمة داخلية لقسم.

يعتبر الـ compaction أحد الحلول لمشكلة التجزئة الخارجية. الهدف هو وصل محتويات الذاكرة بحيث يتم وضع كل الذاكرة الخالية معًا في كتلة واحدة كبيرة. ومع ذلك ، فإن الـ compaction ليس ممكنًا دائمًا.

Paging

Paging is a memory management scheme that permits a process's physical address space to be noncontiguous. Paging avoids external fragmentation and the associated need for compaction. Paging in its various forms is used in most operating systems. Paging is implemented through cooperation between the operating system and the computer hardware (figure 3).

paging

الـ paging عبارة عن اسلوب لإدارة الذاكرة يسمح بأن يكون physical address space للعملية غير مستمر او غيرمتعاقب. يتجنب الـ paging التجزئة الخارجية. يُستخدم الـ paging بأشكاله المختلفة في معظم أنظمة التشغيل. يتم تنفيذ الـ paging من خلال التعاون بين نظام التشغيل وأجهزة الكمبيوتر (الشكل 3).

1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called ***frames*** and breaking logical memory into blocks of the same size called ***pages*****.** When a process is to be executed, its pages are loaded into any available memory frames from their source. The logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 264 bytes of physical memory.

Every address generated by the CPU is divided into two parts: a ***page number (p)*** and a ***page offset (d):***

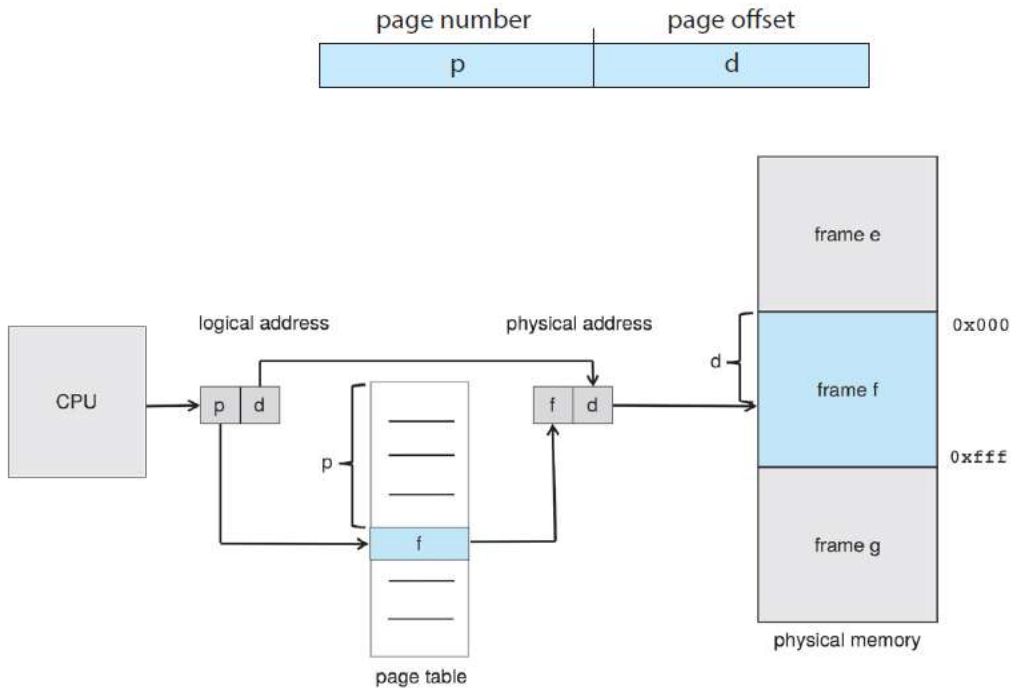


Figure 3 Paging hardware.

1الطريقة الأساسية

تتضمن الطريقة الأساسية لتنفيذ الـ paging تقسيم الذاكرة الرئيسية إلى مساحات ثابتة الحجم تسمى frames وتقسيم الذاكرة المنطقية إلى مساحات من نفس الحجم تسمى الصفحات pages. عندما يتم تنفيذ process ما ، يتم تحميل الـ pages لها من مصدرها في أي frames متاحة من الذاكرة. وبذلك تصبح مساحة العنوان

المنطقية الآن منفصلة تمامًا عن مساحة العنوان الفعلية ، لذلك يمكن أن تحتوي الـ process على مساحة عنوان منطقية 64 بت على الرغم من أن النظام يحتوي على أقل من 264 بايت من الذاكرة الرئيسية.
ينقسم كل عنوان يتم إنشاؤه بواسطة الـ CPU إلى جزأين: رقم الصفحة (p) وإزاحة الصفحة (d):
The page number is used as an index into a per-process ***page table*** *(figure 4)*. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture Figure 5,6).
يتم استخدام رقم الصفحة كفهرس في الـ page table الخاص بكل process (الشكل 4). حجم الصفحة عادة ما يتراوح بين 4 كيلوبايت و 1 غيغابايت لكل صفحة ، اعتمادًا على معمارية جهاز الحاسوب الشكل 5 ، 6.

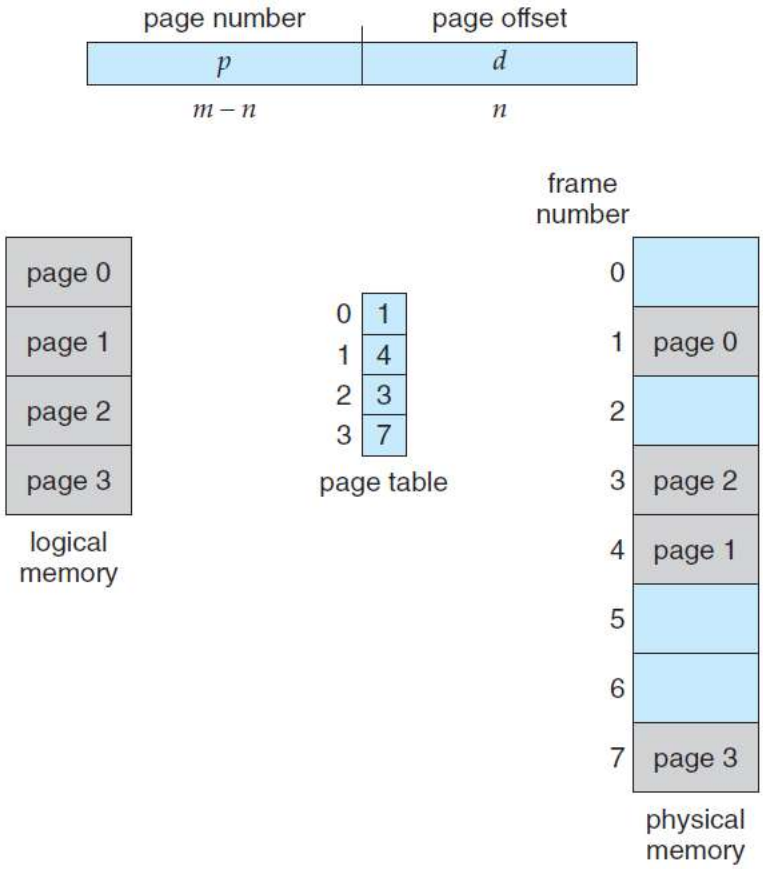


Figure 4 Paging model of logical and physical memory.

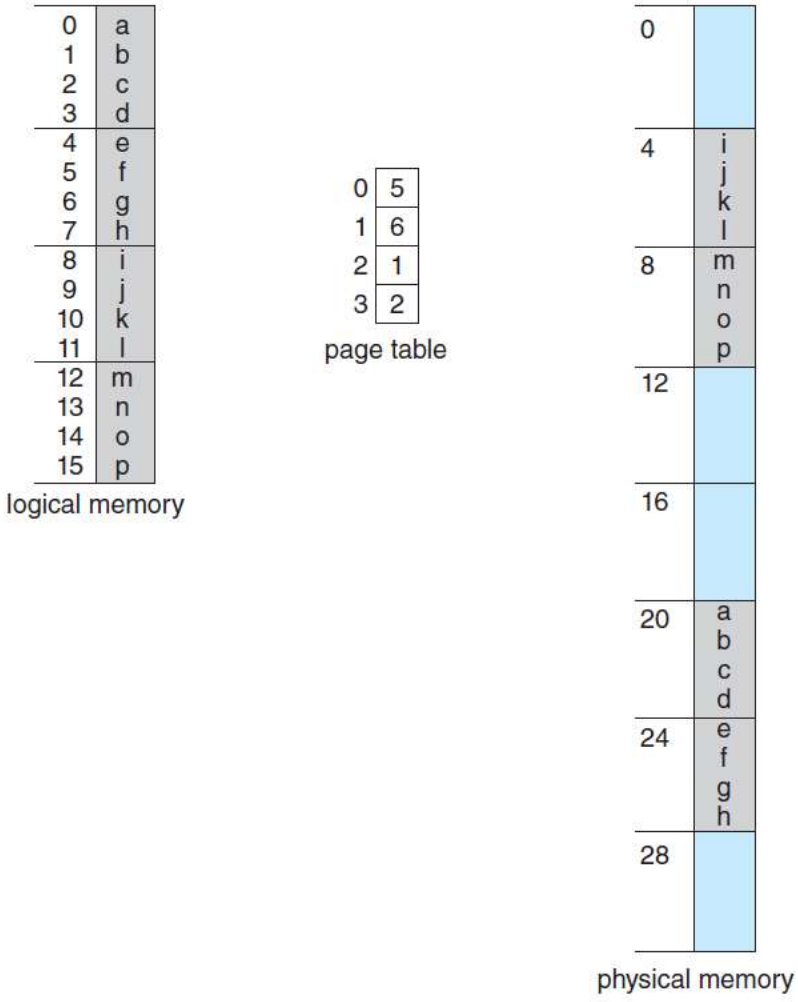


Figure 5 Paging example for a 32-byte memory with 4-byte pages.

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.

أحد الجوانب المهمة للـ paging هو الفصل الواضح بين رؤية المبرمج للذاكرة والذاكرة الرئيسية الفعلية. يرى المبرمج الذاكرة على أنها مساحة واحدة ، تحتوي فقط على هذا البرنامج الواحد. في الواقع ، فإن برنامج المستخدم مبعثر في جميع أنحاء الذاكرة الرئيسية ، والتي تحتوي أيضًا على برامج أخرى.

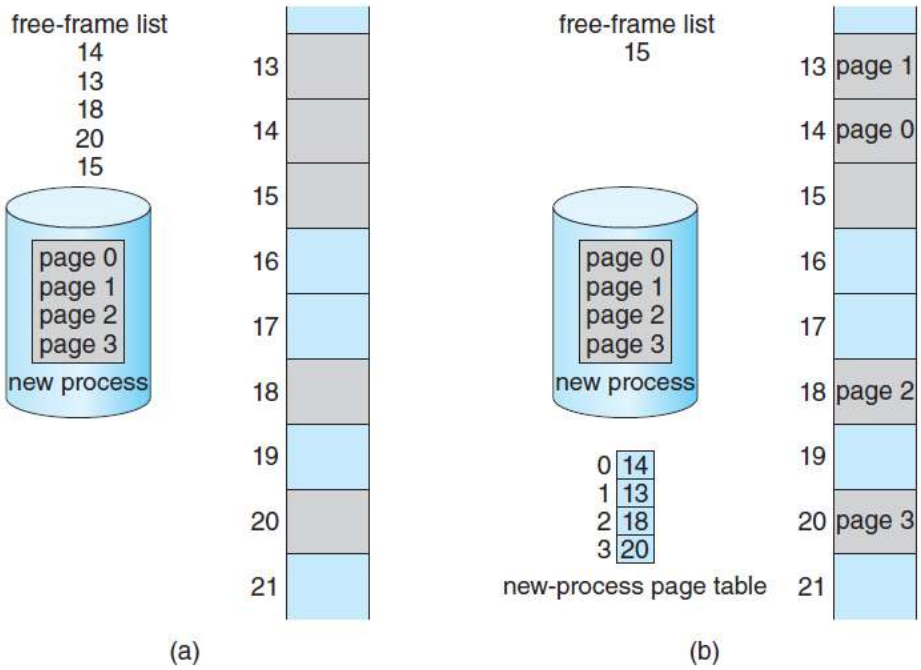


Figure 6 Free frames (a) before allocation and (b) after allocation.

Allocation details of physical memory are kept in a single, system-wide data structure called a ***frame table***. It contains information such as, which frames are allocated, which frames are available, how many total frames there are, and so on. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes). The operating system maintains a copy of the page table for each process.

يتم الاحتفاظ بتفاصيل تخصيص الذاكرة الرئيسية في هيكل بيانات واحد على مستوى النظام يسمى frame table. يحتوي على معلومات مثل ، الـ frames التي تم تخصيصها ، والـ frames المتاحة ، وعدد الـ frames الإجمالية الموجودة ، وما إلى ذلك. يحتوي الـ frame table على سطر او سجل واحد لكل frame في الذاكرة الرئيسية ، يشير إلى ما إذا كان هذا الأخير خاليا أم مخصصًا ، وإذا تم تخصيصه ، فلاي صفحة من أي عملية (أو عمليات). يحتفظ نظام التشغيل بنسخة من جدول الصفحات لكل عملية.

3 Protection

Memory protection in a paged environment is accomplished by ***protection bits*** associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read–write or read-only.

One additional bit is generally attached to each entry in the page table: a ***valid –invalid*** bit. When this bit is set to *valid,* the associated page is in the process's logical address space a legal (or valid) page. When the bit is set to *invalid,* the page is not in the process's logical address space.

يتم تحقيق حماية الذاكرة في البيئة المقسمة إلى pages عن طريق بتات الحماية المرتبطة بكل frame. عادة تكون هذه البتات في الـ page table. بت واحد يمكن أن يحدد صفحة للقراءة والكتابة أو للقراءة فقط.

يتم إرفاق بت واحد إضافي بشكل عام بكل إدخال في جدول الصفحة هو valid-invalid bit .عند تعيين هذا البت كـ valid ، تكون الصفحة موجودة في مساحة العنوان المنطقي للعملية ، وبالتالي هي صفحة قانونية (أو صالحة). عندما يتم تعيين البت كـ invalid ، فلن تكون الصفحة موجودة في مساحة العنوان المنطقي للعملية.

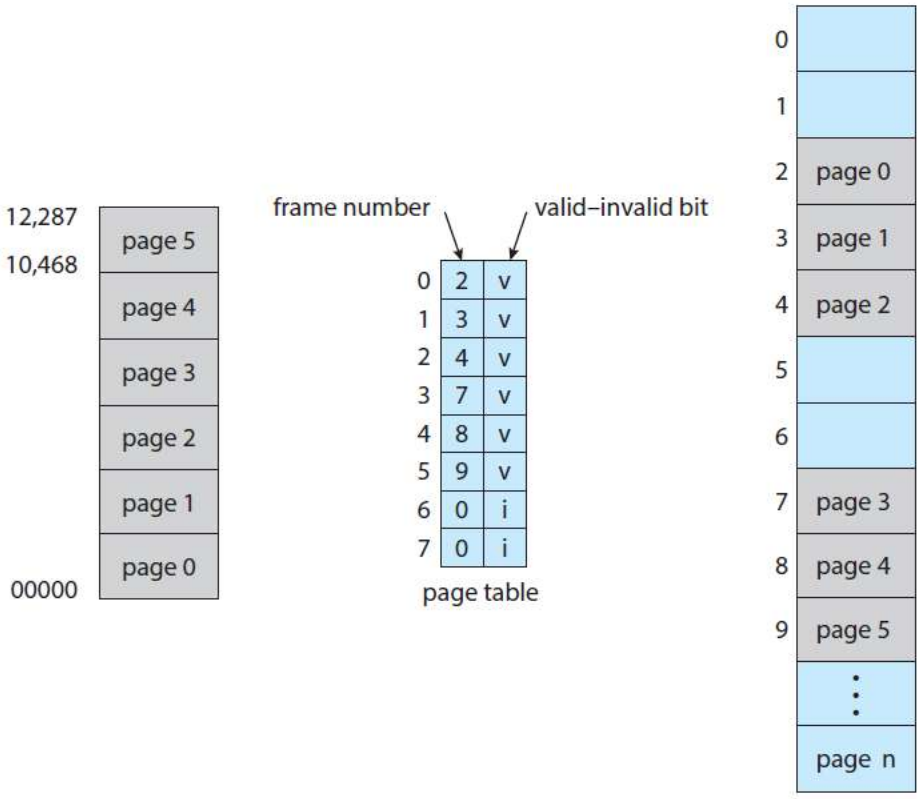


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Rarely does a process use all its address range. It would be wasteful in these cases to create a page table with entries for every page in the address range. Some systems provide hardware, in the form of a **page-table length register** (**PTLR**), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap in the operating system.

نادرًا ما تستخدم العملية كل نطاق عناوينها. وفي هذه الحالات سيكون من الضياع إنشاء page table بإدخالات لكل صفحة في نطاق العنوان. توفر بعض الأنظمة دائرة الكترونية هي page-table length register (PTLR)، للإشارة إلى حجم الـ page table. يتم فحص ومقارنة كل عنوان منطقي مقابل هذه القيمة للتحقق من أن العنوان موجود ضمن نطاق العملية. فشل هذا الاختبار يسبب خطأ في نظام التشغيل.

Contiguous Memory Allocation

The memory is usually divided into two partitions: one for the operating system and one for the user processes. We usually want several user processes to reside in memory at the same time. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

تنقسم الذاكرة عادةً إلى قسمين: أحدهما لنظام التشغيل والآخر لعمليات المستخدم. ينبغي عادةً أن تبقى العديد من عمليات المستخدم في الذاكرة في نفس الوقت. في contiguous memory allocation ، يتم احتواء كل process في مقطع من الذاكرة مجاور للمقطع الذي يحتوي على process التالية.

Memory Protection

A process can be prevented from accessing memory that it does not own by combining the idea of relocation register with the idea of limit register.

Because every address a CPU generates is checked against these registers, we can protect the operating system and the other users' programs and data from being modified by this running process.

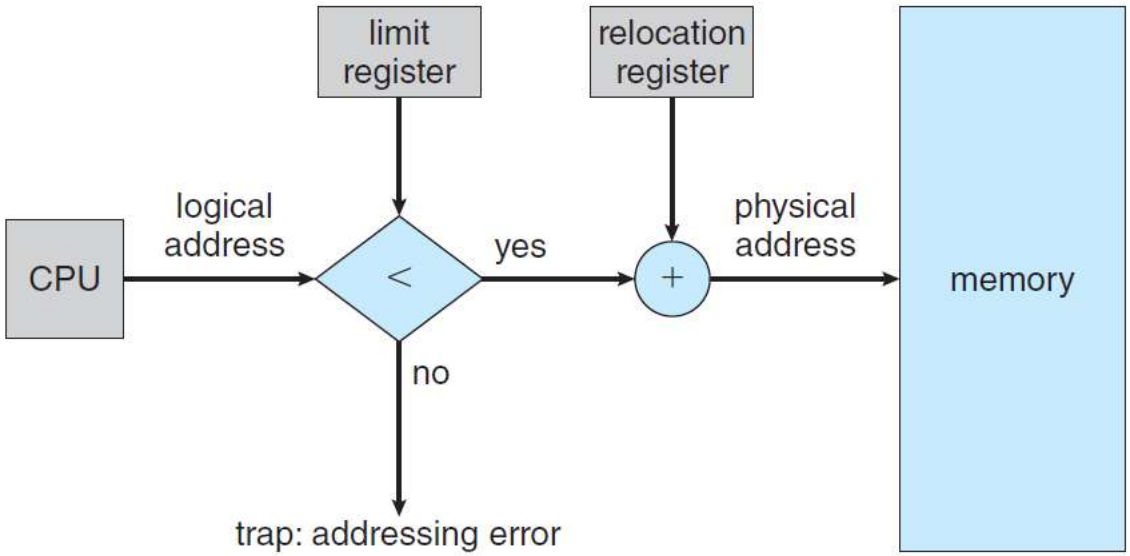


Figure 1 Hardware support for relocation and limit registers.

يمكن منع الـ process من الوصول إلى الذاكرة التي لا تمتلكها من خلال الجمع بين فكرة الـ relocation register مع فكرة الـ limit register .

ولأن كل عنوان تنشئه وحدة المعالجة المركزية يتم فحصه مقابل هذه السجلات ، لذلك يمكن حماية نظام التشغيل وبرامج المستخدمين الآخرين وبياناتهم من التعديل من خلال ال process قيد التشغيل.

2 Memory Allocation

One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process.

In this ***variable partition*** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Fig 2 depicts this scheme.

تتمثل إحدى أبسط طرق تخصيص الذاكرة في وضع الـ processes في اقسام متغيرة الحجم في الذاكرة ، حيث قد يحتوي كل قسم على process واحدة بالضبط. يحتفظ نظام التشغيل في نظام الـ ***variable partition*** هذا ، بجدول يشير إلى أجزاء الذاكرة المتوفرة والأجزاء المشغولة. الشكل 2 يصور هذا المخطط.

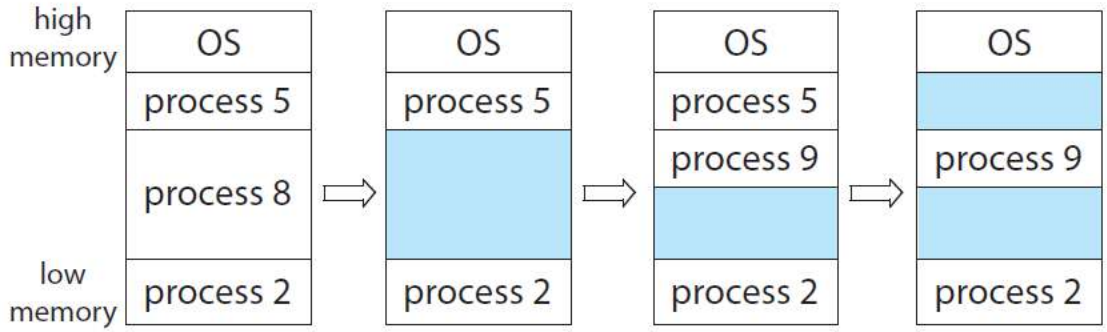


Figure 2 Variable partition.

As processes enter the system, the operating system allocates and loads it into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process. The memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. The first-fit, best-fit, and worst-fi strategies are the ones most commonly used to select a free hole from the set of available holes.

عندما تدخل الـ processes إلى النظام ، يقوم نظام التشغيل بتخصيص ذاكرة لها وتحميلها فيها، حيث يمكنها بعد ذلك التنافس على وقت وحدة المعالجة المركزية. عندما تنتهي process ما ، فإنها تحرر ذاكرتها ، والتي قد يوفرها نظام التشغيل بعد ذلك ل process أخرى. تتكون مساحات الذاكرة المتاحة من مجموعة من الفجوات ذات الأحجام المختلفة المنتشرة في جميع أنحاء الذاكرة. عندما تصل process ما وتحتاج إلى ذاكرة ، يبحث النظام في مجموعة الفجوات عن فجوة كبيرة بما يكفي لهذه الـ process. ان الإستراتيجيات الأكثر استخدامًا لتحديد فجوة حرة من مجموعة الفجوات المتاحة هي first-fit، best-fit ، و worst-fit .

- **First fit.** Allocate the first hole that is big enough.
- **Best fit**. Allocate the smallest hole that is big enough. This strategy produces the smallest leftover hole.
- **Worst fit.** Allocate the largest hole. This strategy produces the largest leftover hole.

- First fit. ويتم فيها تخصيص الفجوة الأولى الكبيرة بما يكفيلاستبعاب ال process
- Best fit. ويتم فيها تخصيص أصغر فجوة كبيرة بما يكفي لاستيعاب ال process. تنتج هذه الاستراتيجية أصغر فجوة متبقية.
- Worst fit. ويتم فيها تخصيص أكبر فجوة. تنتج هذه الاستراتيجية أكبر فجوة متبقية .

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.
أظهرت عمليات المحاكاة أن كلاً من first fit و best fit أفضل من worst fit من حيث تقليل الوقت واستخدام التخزين. من الواضح أنه ليس لكل من الـ first fit ولا الـ best fit أفضلية على الآخرى من حيث استخدام التخزين ، ولكن الـ first fit تكون أسرع عمومًا.

3 Fragmentation

A: External fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from ***external fragmentation***. The free memory space is broken into little pieces as processes are loaded and removed from memory. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous.

B: Internal fragmentation

Memory fragmentation can be internal. Consider a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If the requested block is allocated exactly, a hole of 2 bytes will be left. The difference between these two numbers is internal fragmentation —unused memory that is internal to a partition.

One solution to the problem of external fragmentation is ***compaction***. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however.

3التجزئة

ا: التجزئة الخارجية

تعاني كل من استراتيجيتي الـ first fit والـ best fit لتخصيص الذاكرة من التجزئة الخارجية. ان تحميل العمليات وإزالتها من الذاكرة يتسبب في تقسيم مساحة الذاكرة الخالية إلى أجزاء صغيرة. توجد التجزئة الخارجية عندما يكون مجموع مساحات الذاكرة الإجمالية كاف لتلبية الطلب ولكن المساحات هذ ليست متجاورة.

ب: التجزئة الداخلية

يمكن أن يكون تجزئة الذاكرة داخليًا. افترض فجوة تبلغ مساحتها 18464 بايت. افترض أن العملية التالية تتطلب 18462 بايت. إذا تم تخصيص المساحة المطلوبة بالضبط ، فسيتم ترك مساحة صغيرة بحجم 2 بايت. الفرق بين هذين الرقمين هو التجزئة الداخلية - ذاكرة غير مستخدمة داخلية لقسم.

يعتبر الـ compaction أحد الحلول لمشكلة التجزئة الخارجية. الهدف هو وصل محتويات الذاكرة بحيث يتم وضع كل الذاكرة الخالية معًا في كتلة واحدة كبيرة. ومع ذلك ، فإن الـ compaction ليس ممكنًا دائمًا.

Paging

Paging is a memory management scheme that permits a process's physical address space to be noncontiguous. Paging avoids external fragmentation and the associated need for compaction. Paging in its various forms is used in most operating systems. Paging is implemented through cooperation between the operating system and the computer hardware (figure 3).

paging

الـ paging عبارة عن اسلوب لإدارة الذاكرة يسمح بأن يكون physical address space للعملية غير مستمر او غيرمتعاقب. يتجنب الـ paging التجزئة الخارجية. يُستخدم الـ paging بأشكاله المختلفة في معظم أنظمة التشغيل. يتم تنفيذ الـ paging من خلال التعاون بين نظام التشغيل وأجهزة الكمبيوتر (الشكل 3).

1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called ***frames*** and breaking logical memory into blocks of the same size called ***pages*****.** When a process is to be executed, its pages are loaded into any available memory frames from their source. The logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 264 bytes of physical memory.

Every address generated by the CPU is divided into two parts: a ***page number (p)*** and a ***page offset (d):***

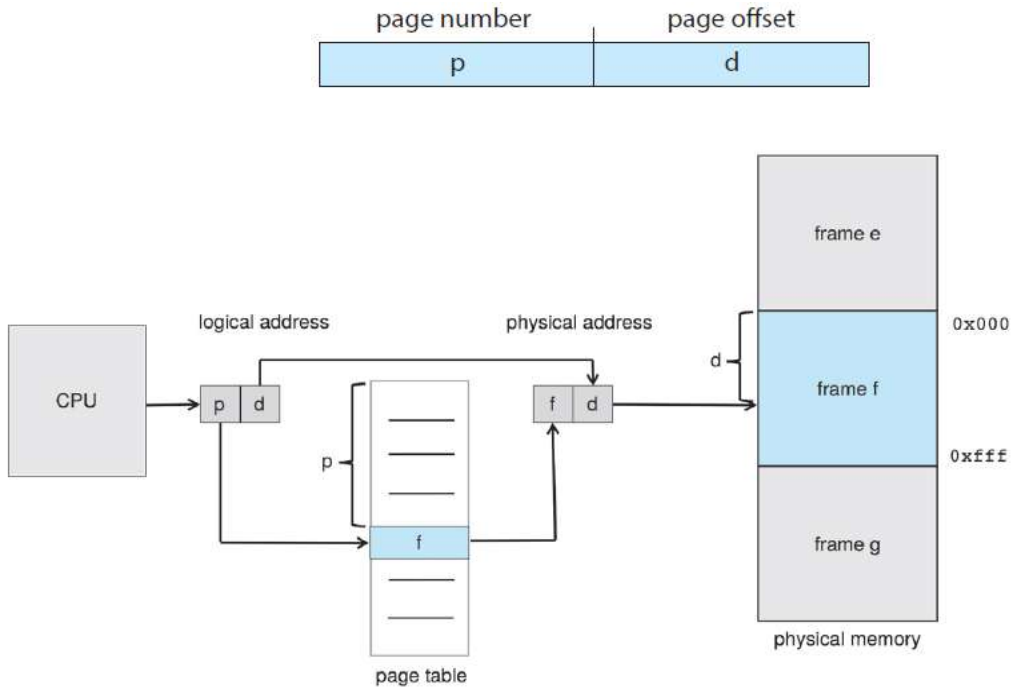


Figure 3 Paging hardware.

1الطريقة الأساسية

تتضمن الطريقة الأساسية لتنفيذ الـ paging تقسيم الذاكرة الرئيسية إلى مساحات ثابتة الحجم تسمى frames وتقسيم الذاكرة المنطقية إلى مساحات من نفس الحجم تسمى الصفحات pages. عندما يتم تنفيذ process ما ، يتم تحميل الـ pages لها من مصدرها في أي frames متاحة من الذاكرة. وبذلك تصبح مساحة العنوان

المنطقية الآن منفصلة تمامًا عن مساحة العنوان الفعلية ، لذلك يمكن أن تحتوي الـ process على مساحة عنوان منطقية 64 بت على الرغم من أن النظام يحتوي على أقل من 264 بايت من الذاكرة الرئيسية.
ينقسم كل عنوان يتم إنشاؤه بواسطة الـ CPU إلى جزأين: رقم الصفحة (p) وإزاحة الصفحة (d):
The page number is used as an index into a per-process ***page table*** *(figure 4)*. The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture Figure 5,6).
يتم استخدام رقم الصفحة كفهرس في الـ page table الخاص بكل process (الشكل 4). حجم الصفحة عادة ما يتراوح بين 4 كيلوبايت و 1 غيغابايت لكل صفحة ، اعتمادًا على معمارية جهاز الحاسوبز الشكل 5 ، 6.

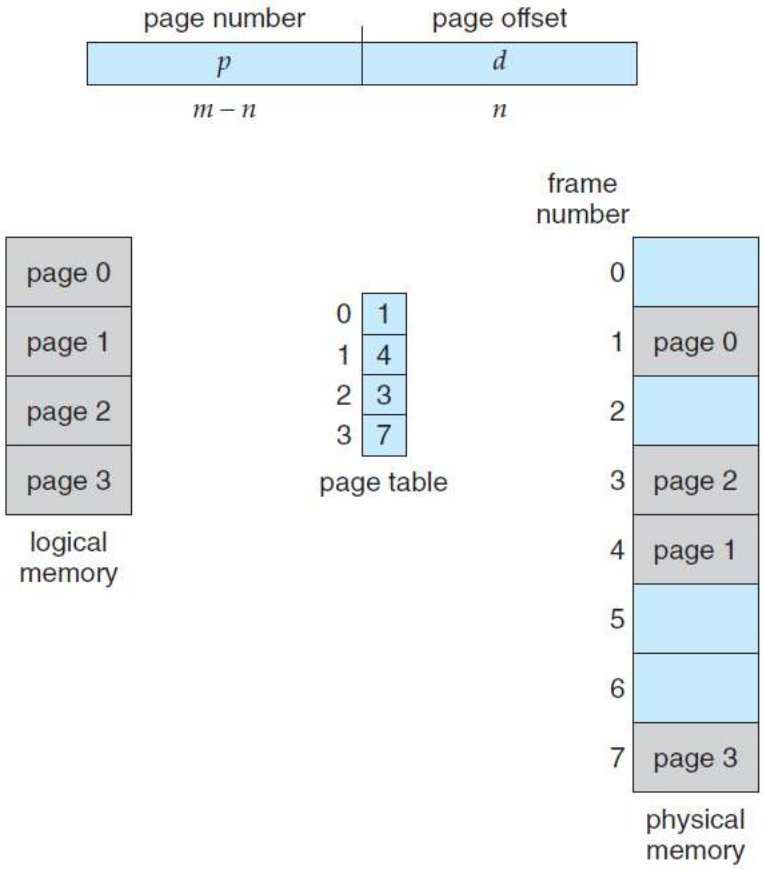


Figure 4 Paging model of logical and physical memory.

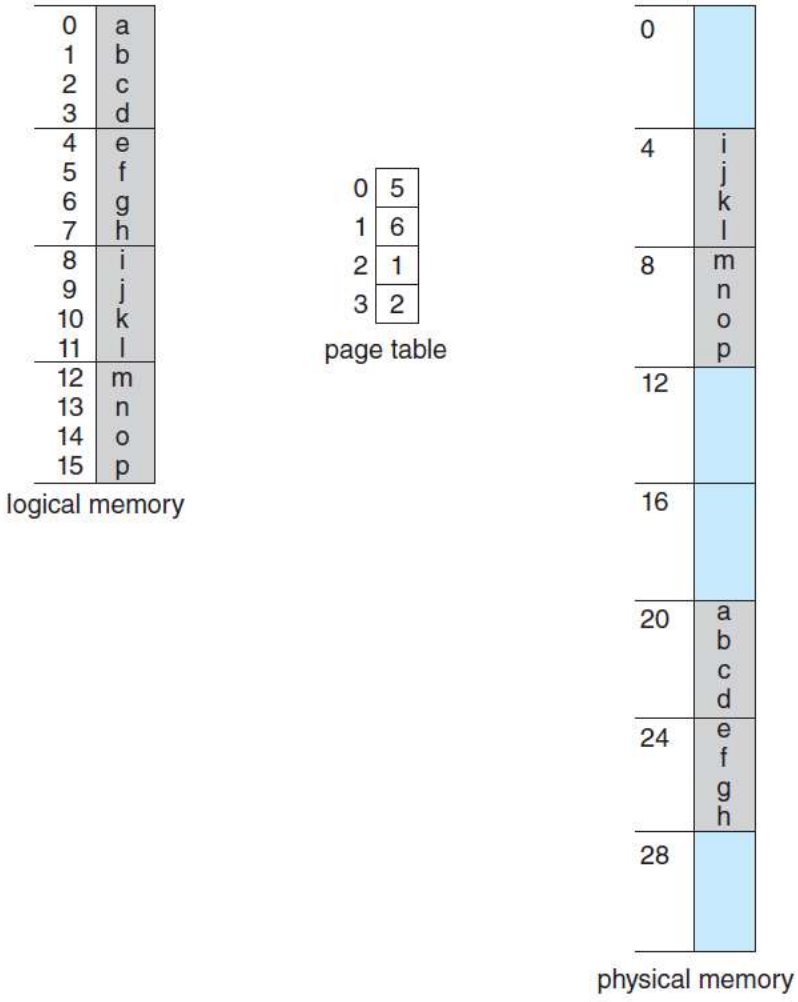


Figure 5 Paging example for a 32-byte memory with 4-byte pages.

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.

أحد الجوانب المهمة للـ paging هو الفصل الواضح بين رؤية المبرمج للذاكرة والذاكرة الرئيسية الفعلية. يرى المبرمج الذاكرة على أنها مساحة واحدة ، تحتوي فقط على هذا البرنامج الواحد. في الواقع ، فإن برنامج المستخدم مبعثر في جميع أنحاء الذاكرة الرئيسية ، والتي تحتوي أيضًا على برامج أخرى.

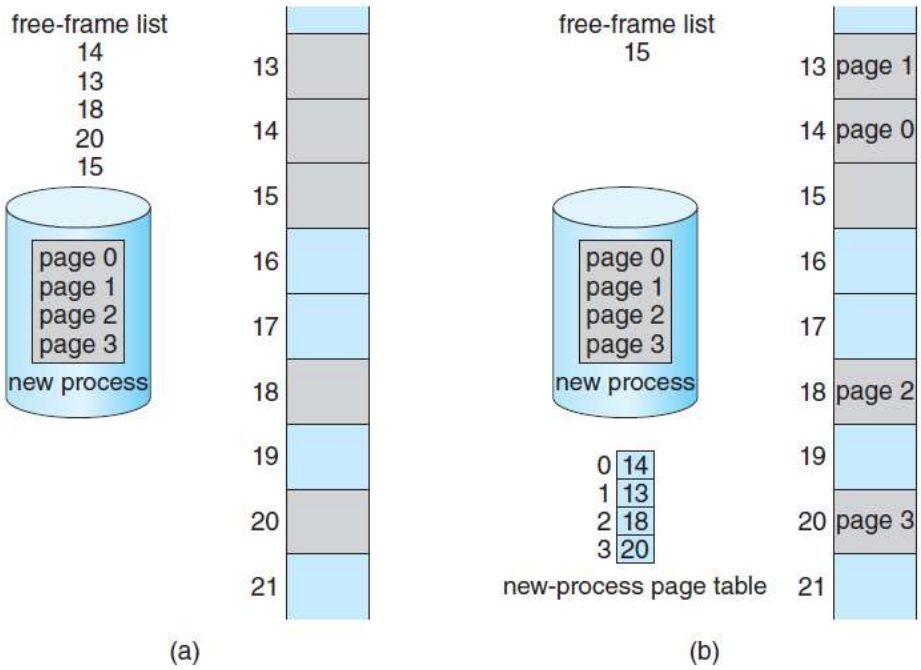


Figure 6 Free frames (a) before allocation and (b) after allocation.

Allocation details of physical memory are kept in a single, system-wide data structure called a ***frame table***. It contains information such as, which frames are allocated, which frames are available, how many total frames there are, and so on. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes). The operating system maintains a copy of the page table for each process.
يتم الاحتفاظ بتفاصيل تخصيص الذاكرة الرئيسية في هيكل بيانات واحد على مستوى النظام يسمى frame table. يحتوي على معلومات مثل ، الـ frames التي تم تخصيصها ، والـ frames المتاحة ، وعدد الـ frames الإجمالية الموجودة ، وما إلى ذلك. يحتوي الـ frame table على سطر او سجل واحد لكل frame في الذاكرة الرئيسية ، يشير إلى ما إذا كان هذا الأخير خاليا أم مخصصًا ، وإذا تم تخصيصه ، فلاي صفحة من أي عملية (أو عمليات). يحتفظ نظام التشغيل بنسخة من جدول الصفحات لكل عملية.

3 Protection

Memory protection in a paged environment is accomplished by ***protection bits*** associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read–write or read-only.

One additional bit is generally attached to each entry in the page table: a ***valid –invalid*** bit. When this bit is set to *valid,* the associated page is in the process's logical address space a legal (or valid) page. When the bit is set to *invalid,* the page is not in the process's logical address space.

يتم تحقيق حماية الذاكرة في البيئة المقسمة إلى pages عن طريق بتات الحماية المرتبطة بكل frame. عادة تكون هذه البتات في الـ page table. بت واحد يمكن أن يحدد صفحة للقراءة والكتابة أو للقراءة فقط.

يتم إرفاق بت واحد إضافي بشكل عام بكل إدخال في جدول الصفحة هو valid-invalid bit .عند تعيين هذا البت كـ valid ، تكون الصفحة موجودة في مساحة العنوان المنطقي للعملية ، وبالتالي هي صفحة قانونية (أو صالحة). عندما يتم تعيين البت كـ invalid ، فلن تكون الصفحة موجودة في مساحة العنوان المنطقي للعملية.

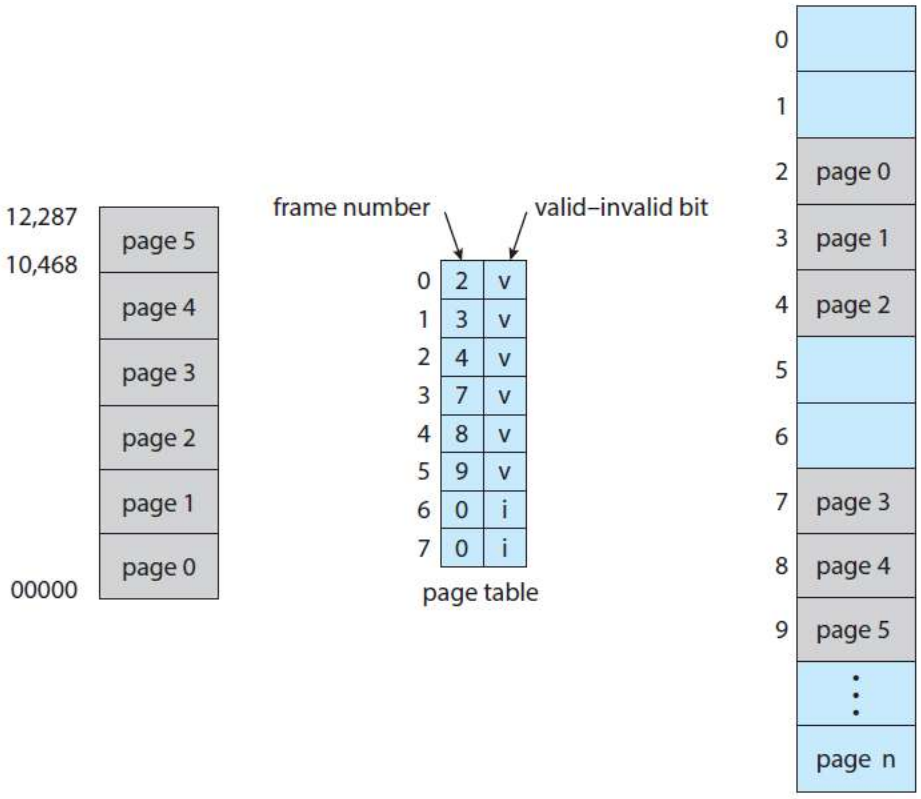


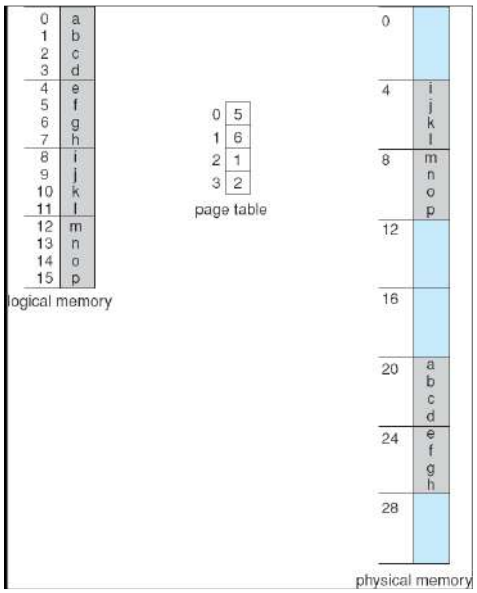
Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Rarely does a process use all its address range. It would be wasteful in these cases to create a page table with entries for every page in the address range. Some systems provide hardware, in the form of a **page-table length register** (**PTLR**), to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap in the operating system.

نادرًا ما تستخدم العملية كل نطاق عناوينها. وفي هذه الحالات سيكون من الضياع إنشاء page table بإدخالات لكل صفحة في نطاق العنوان. توفر بعض الأنظمة دائرة الكترونية هي page-table length register (PTLR)، للإشارة إلى حجم الـ page table. يتم فحص ومقارنة كل عنوان منطقي مقابل هذه القيمة للتحقق من أن العنوان موجود ضمن نطاق العملية. فشل هذا الاختبار يسبب خطأ في نظام التشغيل.

Consider a paging system of a page size of 4 bytes, with a physical address space of 1024 byte. And with a logical address space of 64 bytes.

1.What is the page number and page offset of logical address 0?
2.What is the physical address of logical address 0?
3.What is the logical address corresponding to the physical address 27.
4.How many bits are needed to represent the logical address.
5.How many bits are needed to represent the physical address.
6. Represent the logical address 18 in binary.
7. Represent the physical address 13 in binary.



Sol:

1. page_number = logical_address *Div* page_size
= 0 Div 4 = 0
page_offset= logical_address Mod page_size
= 0 Mod 4 = 0

2. physical_address = (frame_number* page_size) + page_offset
= (5 * 4) + 0 = 20.

3. frame_numbe r= physical_address Div frame_size
=27 Div 4 =6
As we look for this frame number in the page table, we find it corresponds to page 1.
frame_offset = physical_address Mod frame_size
= 27 Mod 4 = 3 (page_offset)
Logical_address= (page_number* page_size) + page_offset
= (1 * 4) + 3 = 7

4. Logical address space = 64 byte =2^6byte
Number of bits in logical address = 6 bits

5. Physical address space= 1024 bytes=2^{10} Bytes
Number of bits in physical address = 10 bits

6.We represent the value 18 in binary
10010 and complete the number of bits to 6. So the final result will be 010010.

7. We represent the value 13 in binary:
1101 and complete the number of bits to 10. So the final result will be 0000001101