Operating System

-

Système d'exploitation

Professeur: Gregory Seront *Institut Paul Lambin* *2020*

Kevin Jullien

Étant donné le COVID19 et le confinement, certaines matières n'ont pas été vues en cette année scolaire 2019-2020.

Confinement dit cours à distance, ce qui a entraîné Gregory Seront à faire des vidéos sur la matière de la (quasi) seconde moitié du quadrimestre.

La première moitié de cette synthèse a donc été complétée de notes personnelles et de la synthèse de Bruno Loverius – merci! – (2018-2019) qui s'est documenté avec l'ouvrage de référence du cours "Andrew Tanenbaum, *Systèmes d'exploitation*, PEARSON Education", l'autre moitié en suivant les vidéos mises à notre disposition.

Les marques pages sont liés à tous les titres et non pas par catégories générales.

Notez que tous les mots en **gras** sont généralement à retenir par cœur, tant leur signification (si un acronyme) que la définition et le concept complet.
Il y a habituellement une question sur un acronyme lors de l'examen du type: "Que veulent dire les lettres HAL?".

Définition

Un système d'exploitation est un programme informatique servant à gérer les ressources hardware et software d'un ordinateur.

Ressources software:

- Pilotes (*drivers*);
- Fichiers;
- Interface graphique;
- Processus / tâches / threads;
- Connections;
- Droits;
- ...

Un OS est une "machine étendue", une abstraction du matériel (**HAL** => Hardware Abstraction Layer) qui nous cache tous les détails qu'on ne veut pas voir.

Abstractions:

- Fichier: sur disque ou sur clef USB -> même accès, hardware différent
- Mémoire:
  - Chaque programme se croit seul sur la machine;
  - Mémoire virtuelle: faire croire qu'on a plus de mémoire qu'on en a.
- Interface graphique: On ne communique pas directement avec la souris ou le clavier.

Ou trouve-t-on des OS?

- Ordinateurs et serveurs "classiques";
- Smartphones (iOS, Android, Linux);
- Lecteurs DVD (principalement Linux);
- Routeurs (Cisco OS, Linux, ...);
- Avions, satellites, voitures, machines à laver;
- Consoles de jeux.

Le premier ordinateur: ENIAC (Electronic Numerical Integrator And Computer)

Une machine colossale de 50 tonnes, 19000 tubes à vide, 1500 relais, 100000 résistances et une fréquence de 100kHZ. Le tout sans OS, et donc à programmer à la main.

L'origine du mot "bug" vient d'ailleurs d'insectes (*bug*) qui faisaient griller les lampes en s'y frottant de trop près.

<u>Batch OS</u>

Niveau programmation, on a ensuite évolué avec les cartes perforées. Beaucoup plus rapide mais on pouvait avoir plusieurs caisses de cartes pour un seul programme.

- Abstraction hardware/software;
- Pas d'interactivité;
- Plusieurs jobs pouvaient être faits à la fois;
- Encore utilisé aujourd'hui (dans les banques).

OS Time Sharing

La base de l'OS moderne: le temps est partagé entre les programmes pour éviter les pauses.

- Plusieurs utilisateurs;
- Partage des ressources (mémoire, disque, ...).

OS Personal Computer

Retour en arrière:

- 1 seul utilisateur;
- 1 seul programme à la fois;
- Uniquement une abstraction hardware (une boite à outils);
- Identique à MS-DOS.

OS Personal Computer: Windows 95

- OS multi-tâches;
- Un seul utilisateur;
- Plusieurs programmes en même temps.

OS Personal Computer: Windows NT/XP/7/10

Plusieurs utilisateurs.

Différents types d'OS

OS interactif: tels que cités précédemment

OS temps réel: Importance du respect des délais pour les tâches:

- Lecture capteur;
- Correction trajectoire;
- ...
- Hard deadline => Délai à respecter à tout prix (freinage pour une voiture, ...);
- Soft deadline => délai plus souple (mais le retard entraîne des conséquences).

Machine à programme fixe

Les premiers ordinateurs avaient donc un programme câblé en dur dans les machines. (ENIAC, Automate de Jacquard - métier à tisser-).

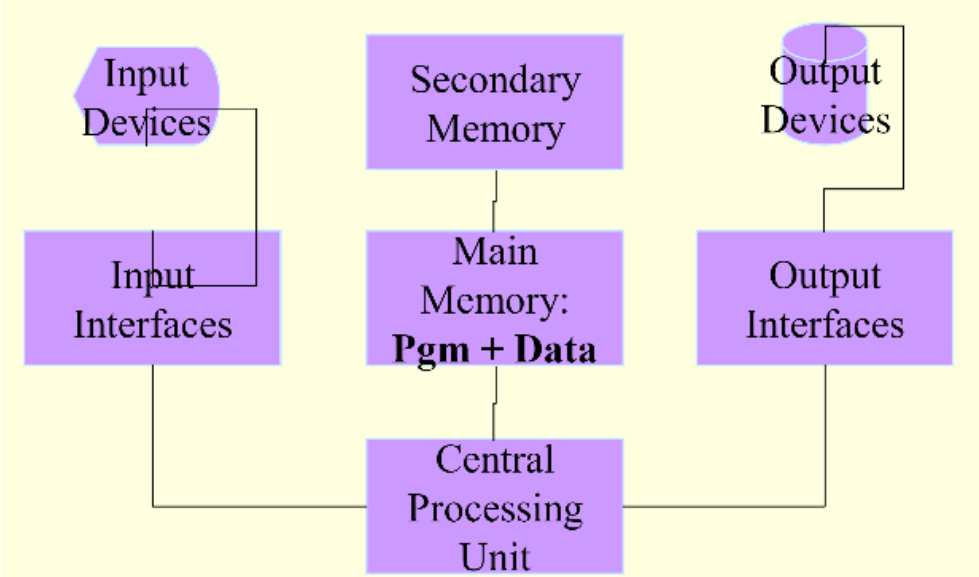
L'ennui est qu'il n'y avait aucune distinction entre le processeur et le programme.

➔ Aucune souplesse, changement de programme impossible sans reconfiguration complet de la machine.

Machine Von Neumann

Séparation des fonctionnalités de bases (calcul simple, lecture et enregistrement en mémoire) du programme en lui-même. Le programme et les données seront donc stockées dans la même mémoire.

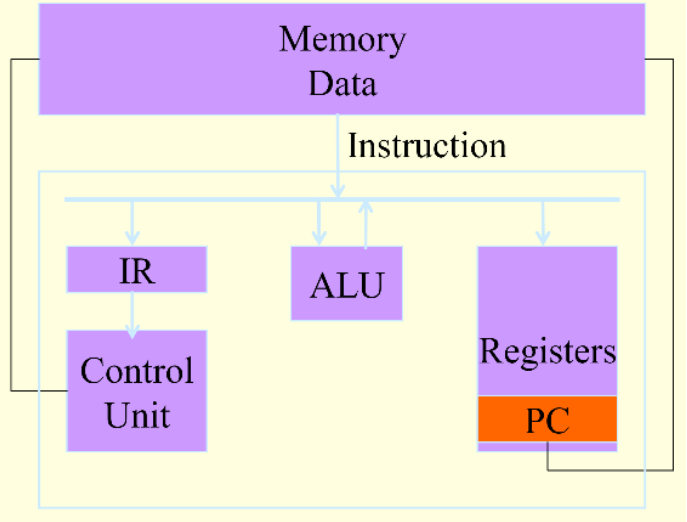
Schématisé:



CPU

IR: Registre d'Instruction (*Instruction Register*)

PC: Compteur Ordinal (*Program Counter*)

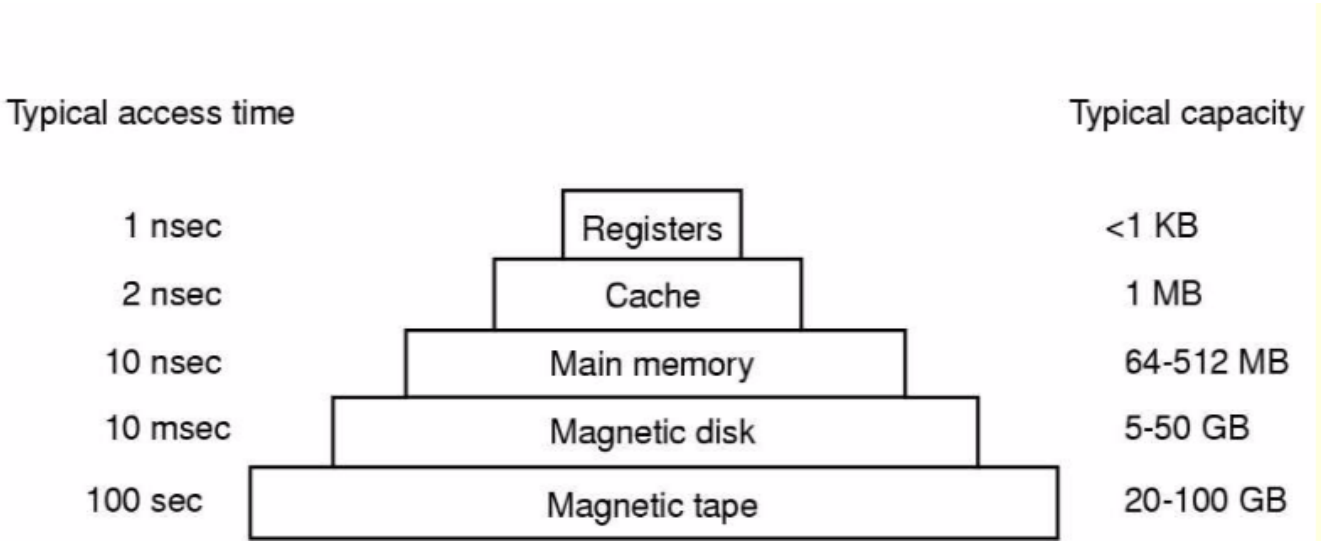


Mémoire

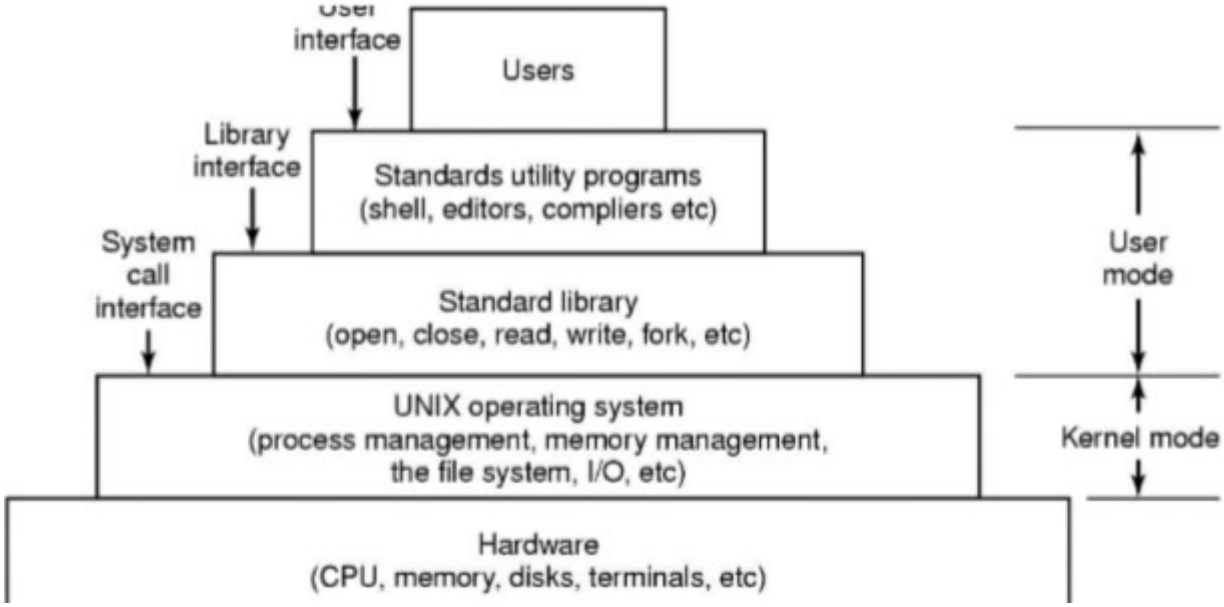
Le lien mémoire cache <-> mémoire cache est gérée par le hardware.

Le lien mémoire centrale <-> mémoire magnétique est gérée par l'OS.

La protection de la mémoire centrale est gérée par l'OS.



Structure d'un OS



Plus on remonte dans les couches, plus l'abstraction est importante.

Le mode noyau (*Kernel*), première couche d'abstraction, est l'endroit où tout est possible: gestion des périphériques, de la mémoire, ...
Seul l'OS a accès à ce mode.

Le mode utilisateur (*user*) contient 3 couches:

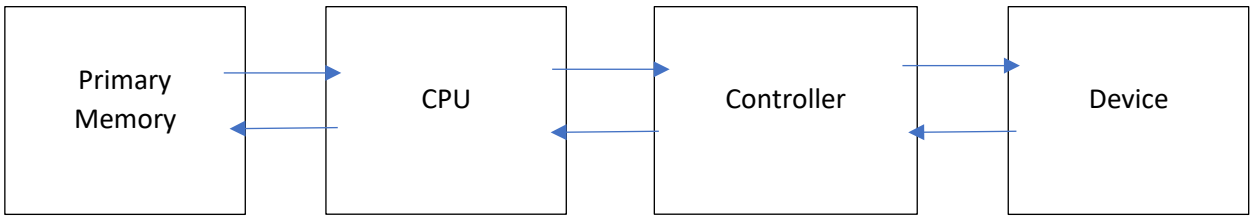
1. Librairie Standard: qui va s'occuper de faire les appels système pour demander à l'OS de faire les actions demandées;
2. Programmes utilisateurs standards;
3. Utilisateurs: C'est là que l'humain interagit.

Puisque seul l'OS a accès aux périphériques, tout ce qui se fait en mode user doit passer par l'OS.

Une requête à l'OS pour l'utilisation d'un périphérique est donc un appel système. Ce dernier va s'assurer que la demande soit bien encadrée (en s'assurant que celui qui l'a demandé a bien les autorisations de lire/écrire où il le demande -> protection).

Accès aux périphériques

On passe par le **contrôleur de périphériques** qui est un élément **hardware** et qui va servir à établir le contact entre les éléments.



L'accès à ce contrôleur se fait via mapping mémoire ou instructions spéciales d'accès aux ports (in, out).

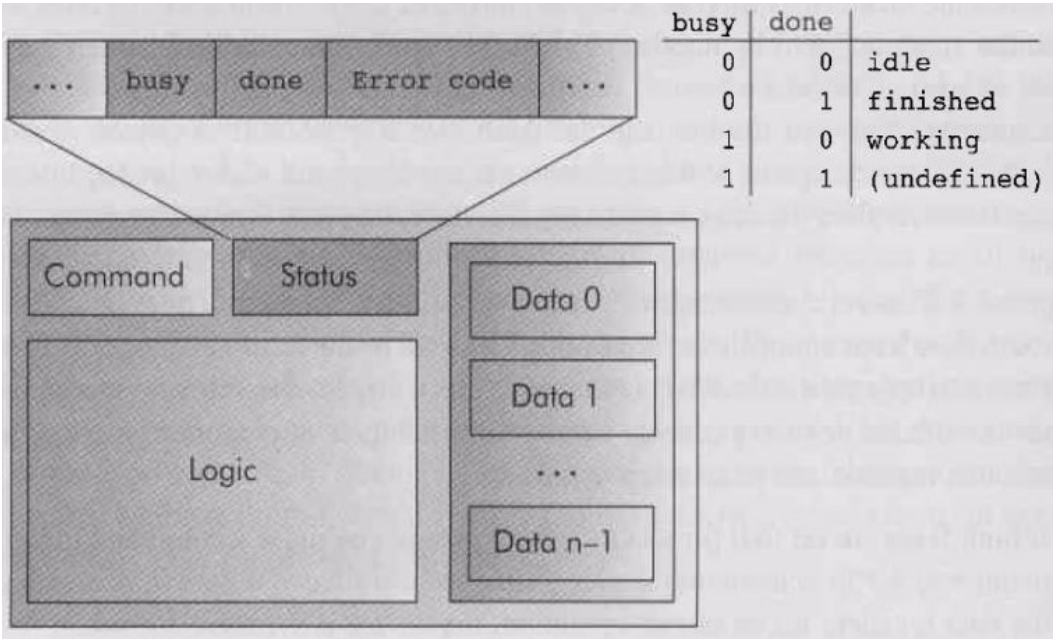
Les périphériques peuvent avoir des registres dédiés ou être mappés en mémoire.

Ces registres contiennent 2 zones, in et out, selon qu'on veuille lire ou écrire.

Le contrôleur a des registres propres dans lesquels le CPU va pouvoir lire et/ou écrire.

- Registre status: indique si le périphérique est occupé ou non, si la dernière opération s'est déroulée avec succès;
- Registre de commandes: indique la commande à exécuter par le contrôleur (envoyées par le CPU);
- Registre de données: indique les paramètres d'une commande.

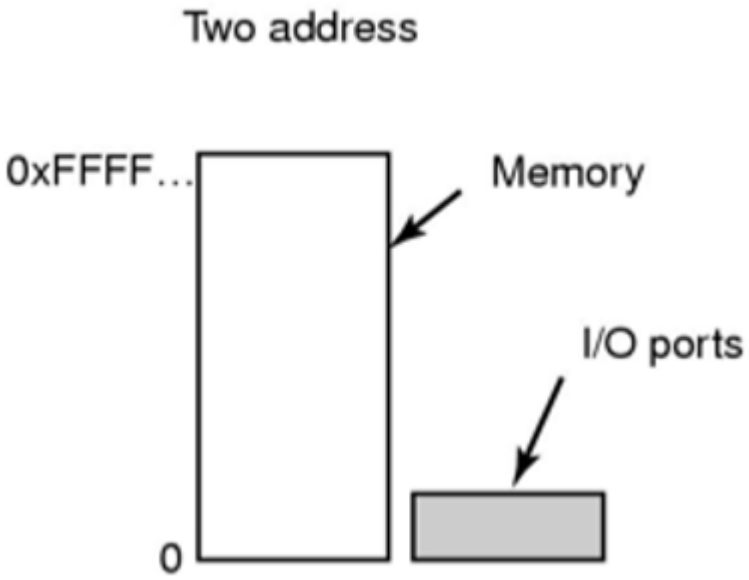
Contrôleur de périphériques:



Ports

Un port est un espace d'adresse hors mémoire dédié à un périphérique spécifique.

Ça implique l'utilisation d'un nouveau bus et des instructions particulières (IN et OUT).

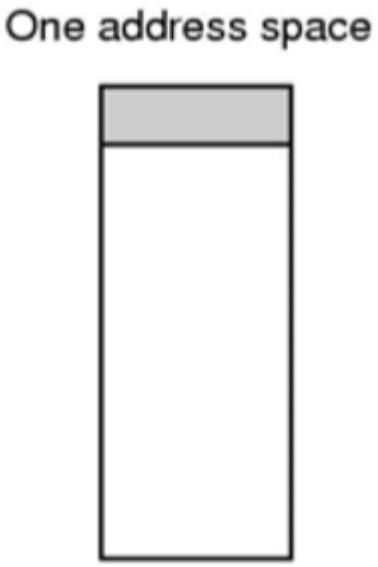


Mapping

Le mapping, c'est la simulation d'un registre périphérique en mémoire.

L'avantage est dans l'économie matérielle, l'inconvénient est que la mémoire utilisée pour ces registres ne peut pas être utilisée puisqu'elle est occupée.

Ici, les instructions IN et OUT ne sont pas nécessaires, un simple MOV suffit.



Protection du hardware

Pour assurer la protection de l'accès au hardware, un simple bit de Kernel est placé dans le processeur, dans le registre d'état(**PSW**: *program status word*). C'est lui qui dira si le mode Kernel est activé ou non.

Ce bit ne peut pas être modifié depuis la zone user mais uniquement par le biais d'appels système.

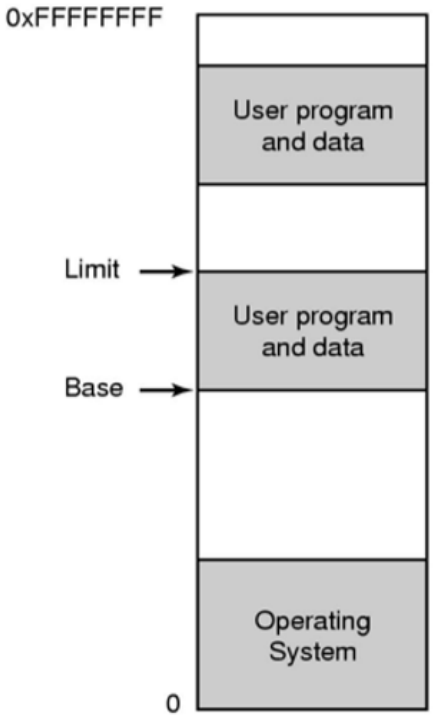
Certaines actions privilégiées le nécessitant:

- Instructions E/S (ports et mapping);
- Accès aux zones mémoire protégées;
- ...

Protection de la mémoire

Deux limites sont définies pour chaque programme dans des registres séparés.

Un programme est libre de faire ce que bon lui semble dans les limites de sa zone, en dehors cependant, il lui faut une autorisation super-user. Il devra donc passer par l'OS pour faire sa requête, ce dernier vérifiant si le programme y est autorisé (afficher une fenêtre, par exemple).



L'instruction TRAP

L'appel système se fait suite à l'exécution d'une instruction trap envoyée par le programme.
À la suite de cette instruction, l'OS sauvegardera tous les registres et passera en mode Kernel, va vérifier le code de l'appel grâce à la **table de déroutement** (**/!** qui est protégée en étant accessible en mode Kernel uniquement, rendant dont sa modification impossible par l'utilisateur), lancera l'instruction et rendra la main au programme.

Exemple d'un read:

1-3 empiler les paramètres (context switch)

4 placer le code de l'appel dans un registre

5 le registre

6 bascule en mode Kernel

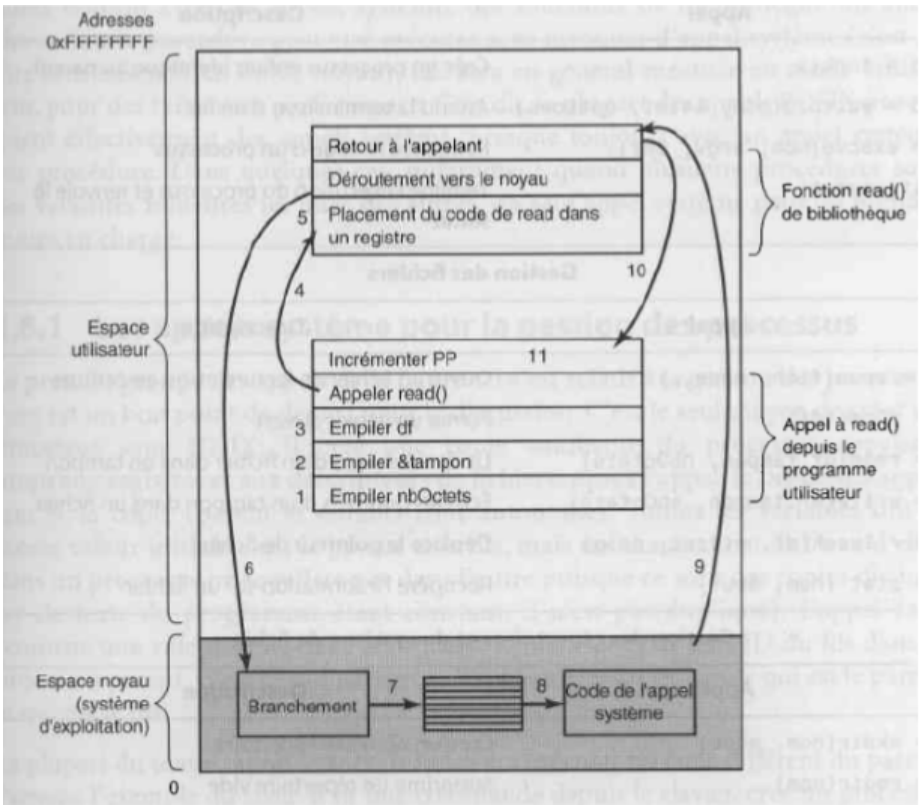
7 examen du code d'appel selon une table indicée

8 exécution du code d'appel

9 basculement du bit de kernel, déroutement vers le noyau. La fonction de bibliothèque reprend le contrôle

10 reprise du contrôle par le programme

11 nettoyage des piles.



PSW chez Intel

CPL (*Current Privilege Level*): les 2 derniers bits du registre CS

On passe du coup de 2 à 4 niveaux de privilèges.

3 : niveau le plus bas.

0 : niveau le plus haut.

Fatalement, cela entraîne que la modification de ce registre CS ne puisse être modifié que par appel système.

Context switch

C'est la sauvegarde des données du CPU lorsqu'il change de tâche. Cette opération peut se faire plusieurs centaines de fois par seconde, rendant l'action transparente aux yeux de l'utilisateur.

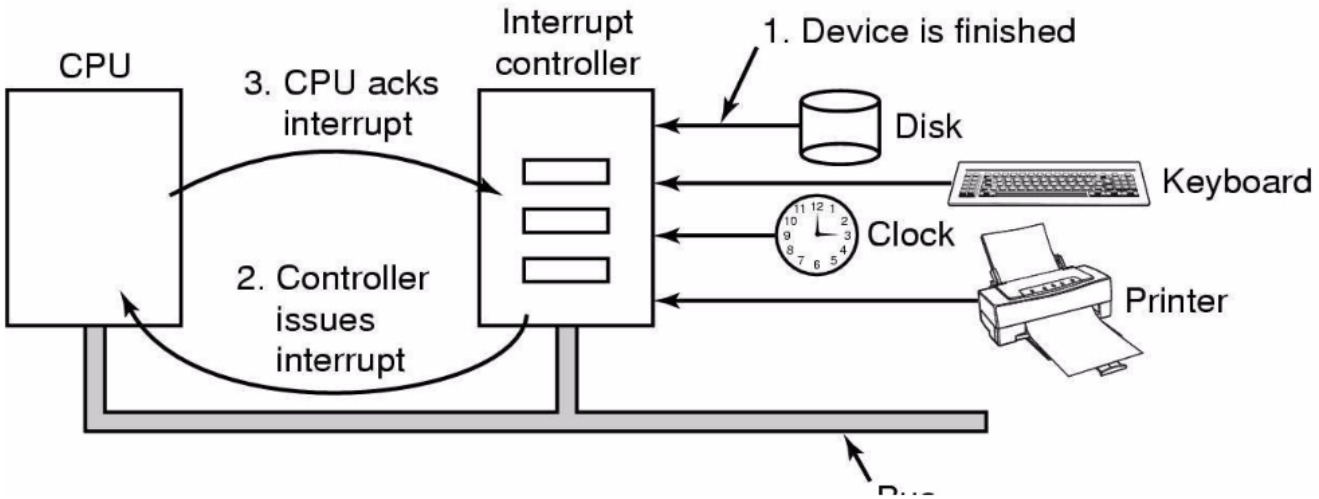
- Empilement des paramètres sur la pile
- Sauvegarde de la progression du processus en cours

Résumé

- Aucun accès direct au hardware en mode utilisateur;
- Le PSW contient le bit de Kernel, indiquant dans quel mode on se trouve;
- Aucune instruction, même privilégiée, n'accède au hardware
- Seuls les appels systèmes accèdent au hardware;
- Ces appels se font via une instruction trap et son context switch.

Les interruptions sont le moyen de communication des périphériques pour annoncer quelque chose tel que:

- L'entrée d'un caractère au clavier;
- La fin d'une lecture disque;
- Le niveau critique de la batterie;
- La fin d'un timer;
- ...



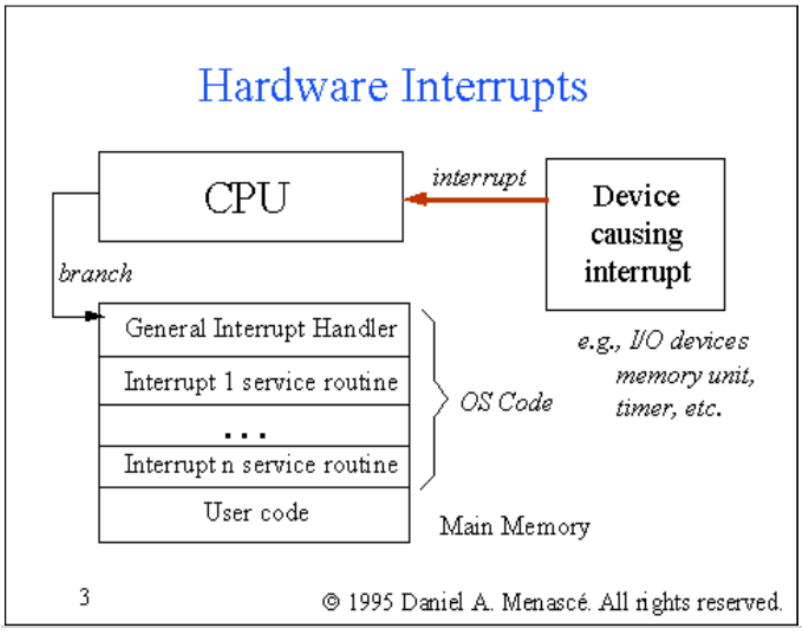
Toutes ces interruptions se font vers le **contrôleur d'interruption** via un signal électrique.

Lorsque le contrôleur reçoit une interruption hardware, il va placer le numéro du périphérique dans un de ses registres.

Sans contrôleur, le signal est directement envoyé au CPU.

Lorsque ce dernier décide de gérer l'interruption, il va voir dans le **gestionnaire d'interruption** (*General Interrupt Handler*) spécifique au périphérique pour savoir de quoi il s'agit.

Puisque cette interruption fait la demande d'aller voir dans une table qui se trouve en mode Kernel, il s'agit en fait d'un appel système, provoqué par le matériel cette fois-ci.



Le **gestionnaire d'interruption** est spécifique à tout périphérique et fait partie du driver. C'est donc un bout software, contrairement au contrôleur.

Interruptions simultanées

Si deux périphériques venaient à envoyer une interruption simultanément, on parlerait alors de "course".

Chaque interruption a un niveau d'importance et pourra donc, si son niveau est inférieur à l'autre, être masquée le temps de traiter celle avec le niveau le plus haut.
Toute interruption est envoyée en continu dans le cas où elle ne serait pas traitée immédiatement.

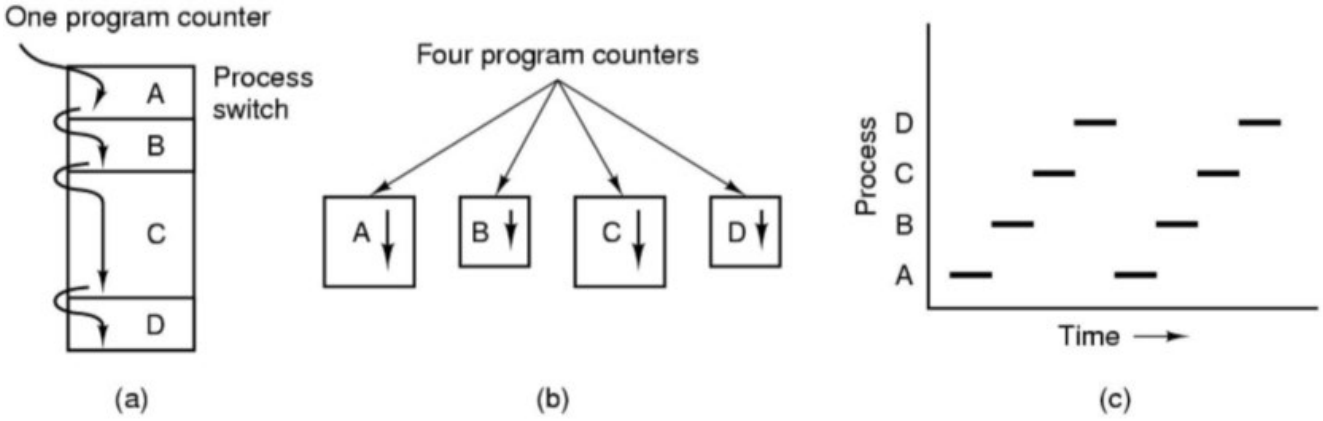
Processus

Tout ordinateur peut exécuter plusieurs programmes (processus) en même temps, pour l'utilisateur du moins.

La réalité est que tous ces processus sont alternés, les uns après les autres, à une vitesse imperceptible pour l'utilisateur.

A moins d'avoir un multicore, ce qui est commun aujourd'hui, le CPU ne pourra pas travailler sur plusieurs processus de manière simultanée.

On considérera donc un CPU monocore pour les explications à venir.



a: monocore b: multicore c: utilisation CPU par processus

Le changement d'un processus à un autre oblige à sauvegarder les registres pour pouvoir se retrouver, une fois revenu sur le processus qui a été suspendu, à un état identique que lors de sa suspension.

Il est intéressant de noter qu'un programme se croit seul:

- Sur le CPU;
- Dans la mémoire;
- Avec les périphériques.

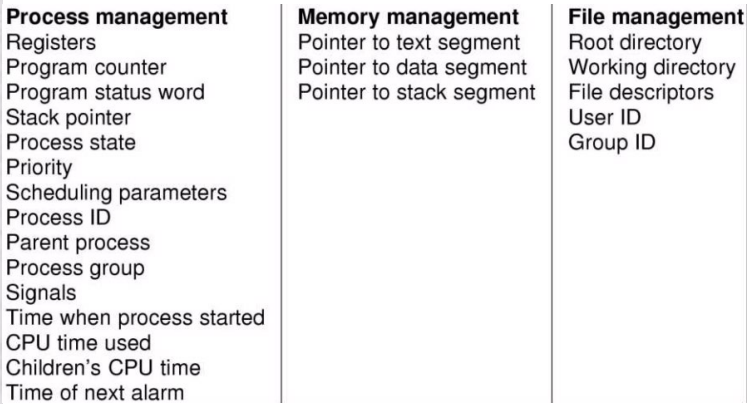
Définition du processus:

Pour l'utilisateur: Abstraction de la machine hardware pour permettre à un programme de "croire qu'il est seul".

Pour l'OS: Code à exécuté, ressources utilisées, état de la mémoire et du processeur (droits, propriétaire, ...).

Puisqu'il faut sauvegarder les données d'un processus avant d'en alterner, l'OS contient une table de processus qui contient toutes ces informations.
La table contient une entrée par processus: le **PCB** (*process control block*).

Gestion du processus	Gestion de la mémoire	Gestion de fichier
Registres	Pointeur vers un segment de texte	Répertoire racine
Compteur ordinal	Pointeur vers un segment de données	Répertoire de travail
Mot d'état du programme	Pointeur vers un segment de la pile	Descripteurs de fichiers
Pointeur de la pile		ID utilisateur
État du processus		ID du groupe
Priorité		
Paramètres d'ordonnancement		
ID du processus		
Processus parent		
Groupe du processus		
Signaux		
Heure de début du processus		
Temps de traitement utilisé		
Temps de traitement du fils		
Heure de la prochaine alerte		



Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Registres: L'endroit où tous les registres seront enregistrés lors d'un context switching.

Compteur ordinal: Instruction Pointer: l'adresse de l'instruction exécutée pour reprendre le processus.

Mot d'état: PSW.

État du processus: indique si le processus est actif, en attente d'une réponse ou en attente de son tour.

Paramètres d'ordonnancement: voir plus tard.

ID du processus: un ID unique permettant son identification.

Gestion de la mémoire: les données ici pourront varier selon le type de processeur utilisé.

Gestion de fichier: Les informations sur les fichiers actuellement utilisés par le processus.

Création des processus

1. Au démarrage de la machine (Task Manager);
2. Par un autre processus via appel système:
   a. Unix -> fork() (clone le processus courant);
   b. Windows -> createProcess().
3. Par l'utilisateur via le shell ou la GUI (le shell est un processus qui fera un appel système).

Fin des processus

1. Arrêt normal (fin de programme);
2. Arrêt sur erreur (plantage => interruption);
3. Arrêt par un autre processus:
   a. Unix -> Kill
   b. Windows -> TerminateProcess
4. Arrêt via shell ou task manager

Hiérarchie des processus

Unix

Hormis **init**, le processus créé au démarrage de la machine, tous les autres sont créés par un processus.

Chaque processus créé a ainsi un processus "parent" qui fait partie de ses caractéristiques propres.

Chaque processus peut avoir une quantité non limitée de processus enfants.

Si un processus parent se termine, tous ses enfants se termineront eux aussi.

Windows

Pas de parents/enfants sauf la création d'un jeton particulier à la création d'un processus, qui sera donné au créateur. Ce jeton permettra au processus créateur d'avoir le contrôle sur son engeance, sauf s'il décide de passer ce jeton à un autre processus, qui héritera du contrôle.

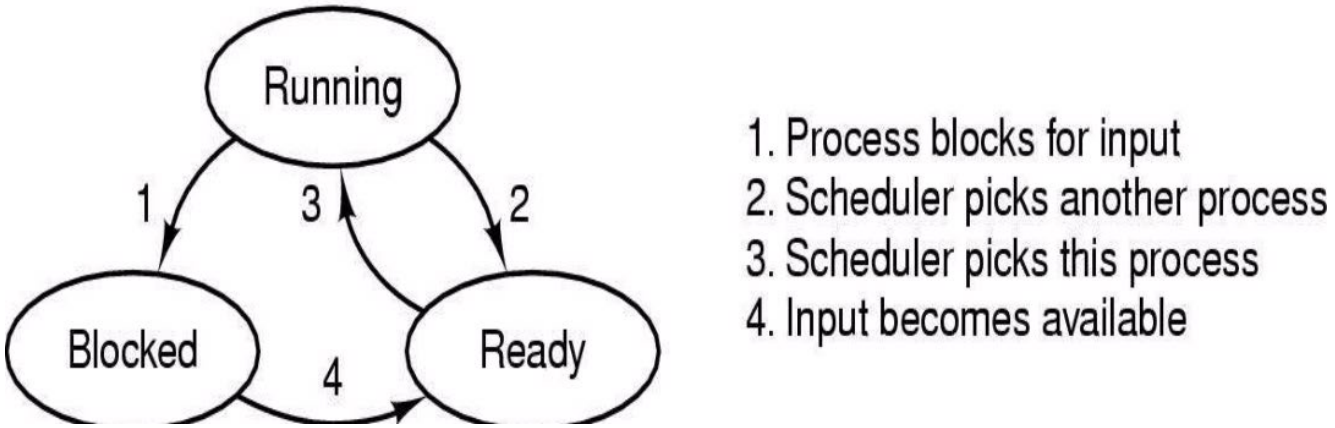
Les états des processus

Les processus ne sont donc pas toujours en activité. Il peut donc se trouver dans 3 états:

1. **Occupé** (*Running*): il est actif et utilise le processeur;
2. **Prêt** (*Ready*): il est prêt à être exécuté;
3. **Bloqué** (*Blocked*): Il n'est pas prêt à être exécuté et est en attente d'une ressource ou d'une réponse d'un périphérique.

Pour aider à la gestion de ce système, l'OS contient un **ordonnanceur**(*scheduler*).
Un timer se trouve aussi accessible au CPU pour éviter qu'un unique processus ne tourne sans arrêt.
S'il dépasse donc le quantum défini, il se mettre en "prêt".

a. Un processus est en train de s'exécuter et demande une ressource ou communique sur un périphérique (ex: lecture disque, attente entrée clavier, ...) Il passe alors en statut "bloqué" jusqu'à ce que sa demande soit assouvie;
b. Un processus est "prêt" et l'ordonnanceur estime que c'est à son tour d'être exécuté (à la suite de la suspension temporaire d'un autre programme);
c. Un processus est en train de s'exécuter mais depuis trop longtemps, il passe alors en statut "prêt";
d. Un processus était "bloqué" car en attente d'une ressource et la reçoit, il passe alors en statut "prêt".



Changement d'état des processus

A chaque changement d'état, un appel système se fait (ou une interruption) et donc, context switching

1. Le matériel place dans la pile le compteur ordinal, etc.;
2. Le matériel charge un nouveau compteur ordinal à partir du vecteur d'interruption;
3. La procédure en langage assembleur sauvegarde les registres;
4. La procédure en langage assembleur définit une nouvelle pile;
5. Le service d'interruption en C s'exécute généralement pour lire les entrées et les placer dans le tampon);
6. L'ordonnanceur décide du prochain processus à exécuter;
7. La procédure en C retourne au code assembleur;
8. La procédure en langage assembleur démarre le nouveau processus actif.

L'ordonnanceur

C'est donc lui qui va décider du processus qui va pouvoir utiliser le processeur selon un algorithme.

Le context switching étant couteux, il faut envisager un temps suffisamment long avant de changer de processus pour que la sauvegarde des registres ne prenne pas une proportion trop importante du temps de travail, mais un temps suffisamment court aussi pour que l'utilisateur ne sente pas qu'un programme attend qu'un autre ait terminé de faire ce qu'il a à faire.

Quand ordonnancer

- Création d'un nouveau processus: le parent et l'enfant sont "prêt", lequel utiliser ?;
- Fin d'un processus: choix du suivant (si pas de suivant, un processus d'inactivité se lance);
- Interruption E/S: choix parmi les processus "prêt";
- Lorsqu'un processus bloque car en attente E/S. S'il s'agit de l'attente d'une donnée provenant d'un autre processus, lancer ce processus est intéressant pour pouvoir permettre au premier de continuer son activité;
- Interruption d'horloge: si un processus utilise trop longtemps le processeur, il peut devoir être mis en suspens et un autre processus doit se lancer.
    - Algorithme non préemptif: sélectionne un processus et le laisse s'exécuter jusqu'à ce qu'il bloque (demande E/S, sur un autre processus, ...). Tant qu'il ne bloque pas, il tournera.
    - Algorithme préemptif: un quantum de temps est défini après lequel, si un même processus a occupé le processeur, il passera en statut "bloqué" pour permettre à d'autres processus de pouvoir fonctionner à leur tour.

A noter que la majorité des algorithmes d'ordonnancement sont préemptifs, principalement sans doute car la majorité des ordinateurs nécessitent de pouvoir faire du multi-tâche et qu'un utilisateur a besoin d'interactivité.

Objectifs de l'ordonnancement

L'ordonnanceur ne sera donc pas le même selon la nécessité de l'OS. Quelques exemples:

Tous les systèmes

- Équité: attribuer à chaque processus un temps processeur équitable;
- Application de la politique: faire en sorte que la politique définie soit bien appliquée;
- Équilibre: faire en sorte que toutes les parties du système soient occupées;

Systèmes de traitement par lots

- Capacité de traitement: optimiser le nombre de jobs à l'heure;
- Délai de rotation: réduire le délai entre la soumission et l'achèvement;
- Utilisation du processeur: faire en sorte que le processeur soit occupé en permanence.

Systèmes interactifs

- Temps de réponse: répondre rapidement aux requêtes;
- Proportionnalité: répondre aux attentes utilisateurs.

Système temps réel

- Respecter les délais: éviter de perdre des données;
- Prévisibilité: éviter la dégradation de la qualité dans les systèmes multimédias.

Ordonnancement des systèmes interactifs

Le **Round Robin**, ou tourniquet, est un système qui définit un quantum de temps pour chaque processus de manière à distribuer équitablement l'usage CPU. L'intérêt est que l'ordinateur semble, pour l'utilisateur, faire toutes ses demandes immédiatement. La réactivité est donc l'intérêt principal de ce système.

Le changement de processus prend 1ms. Si on réglait donc le quantum à 4ms, en 5 ms, on aurait 4ms de travail. 20% du temps du processeur serait donc utilisé pour alterner les processus.
Perte d'énergie, de temps de calcul,...

À l'inverse, si on choisissait 99ms de quantum, on ne perdrait que 1% du temps de travail du processus, mais l'interactivité prendrait cher car tourner sur 10 processus différents prendrait 1 seconde, temps que l'utilisateur ressentirait.

L'accord du quantum est donc entre 20 et 50ms

Windows 10 est réglé sur 20ms

Windows serveur sur 120ms

Ordonnancement par priorité

Tous les processus ne sont pas égaux.

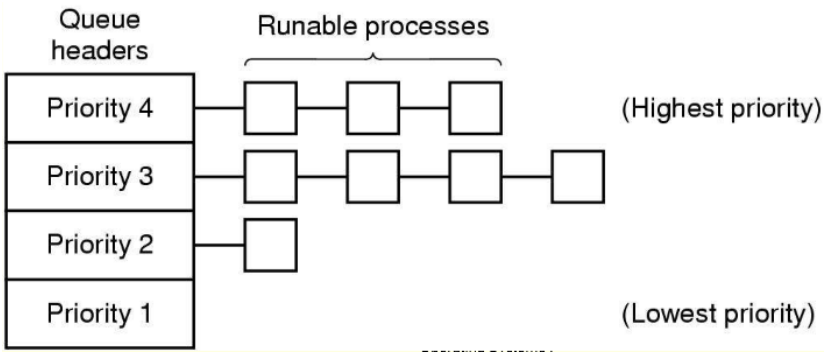
Pouvoir déplacer sa souris, rafraîchir l'écran, ... est largement prioritaire à l'envoi d'un mail ou le clignotement programmé d'une led décorative.

On peut donc, en plus du round robin, instaurer une priorité à chaque processus de manière à ce que, lorsque l'ordonnanceur doit faire son choix, il sélectionne avant tout les processus prioritaires.

Si on ne fait pas attention, on peut provoquer une **famine** (*starvation*). C'est ce qui se passe si un processus de très faible priorité n'est jamais lancé car d'autres processus à haute priorité sont toujours présents dans la liste.
Il existe donc deux solutions:

- Diminuer la priorité des processus qui s'exécutent trop (*aging*);
- Augmenter la priorité de ceux qui ne s'exécutent pas assez (C'est le choix de Windows).

On peut ainsi avoir un round Robin multiple, un par ordre de priorité.

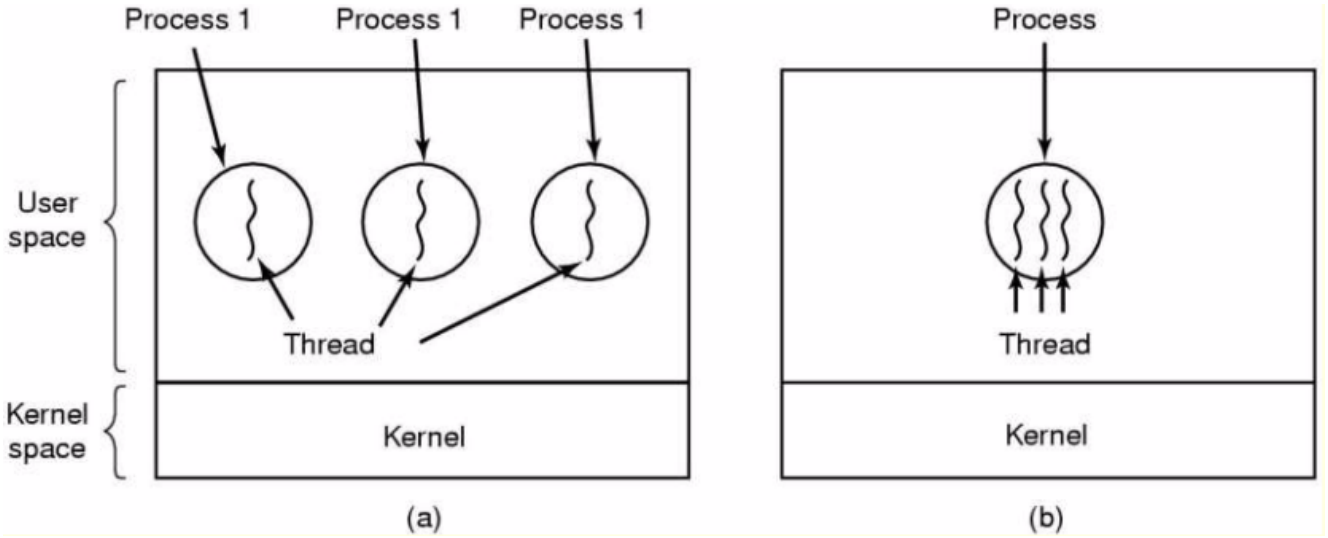


Processus léger (*thread*)

Un processus est concrètement constitué de deux choses:

- Les ressources (fichiers, mémoire, droits, ...);
- Le fil d'exécution (fil = thread).

Pour assurer la réactivité d'un processus, il est intéressant de le diviser en petits sous-processus. Ce sont les threads.



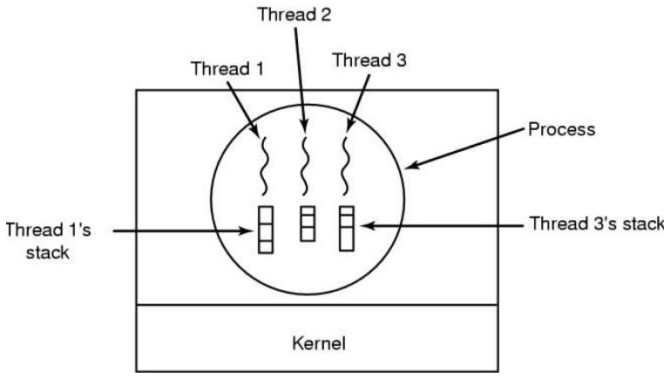
a) mono-threading b) multi-threading

Le multi-threading permet à plusieurs threads de partager les mêmes ressources:

- Espace d'adressage mémoire;
- Variables globales;
- Fichiers ouverts;
- Alertes en attente.

Il reste des éléments privés:

- Compteur ordinal (Program Counter -> IP);
- Registres;
- Pile;
- État (ready, blocked, running, terminated);



Création et fin des threads

Lors de la création d'un processus, il n'y a qu'un seul thread.
C'est lui qui va pouvoir en créer d'autres (*thread_create(function)* pour Unix).

Pour terminer, la commande *thread_exit()* peut être exécutée.

Dans certains cas, les threads ne sont pas préemptibles au sein du processus, ils doivent donc rendre la main eux-mêmes avec la commande *thread_yield()*.

Soucis des threads

Beaucoup de problèmes peuvent survenir lors de l'interaction des threads entre eux.

Puisqu'ils partagent les mêmes ressources, que se passe-t-il si un thread ferme un fichier qu'un autre utilise ou si l'un modifie une variable commune?

Lorsqu'un *fork()*, une copie identique du processus se créé. On copie donc tous les threads. Quid si un thread attendait le clavier? Qui va recevoir l'information?

Avantages des threads

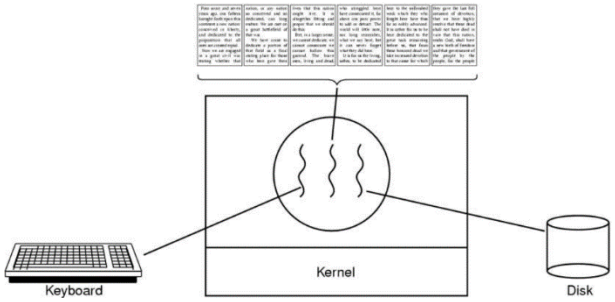
La création d'un thread est 100x plus rapide que celle d'un processus. C'est donc un outil qui, s'il est bien maitrisé, permet de gagner beaucoup de temps.

Avec les systèmes multicore, plusieurs cores peuvent utiliser un thread différent d'un même processus, accélérant énormément sa vitesse d'exécution.

L'architecture logicielle est simplifiée: On pense le traitement en unités indépendantes. On doit moins se soucier du temps de réponse pendant qu'on fait autre chose.

Architecture logicielle multi-threads

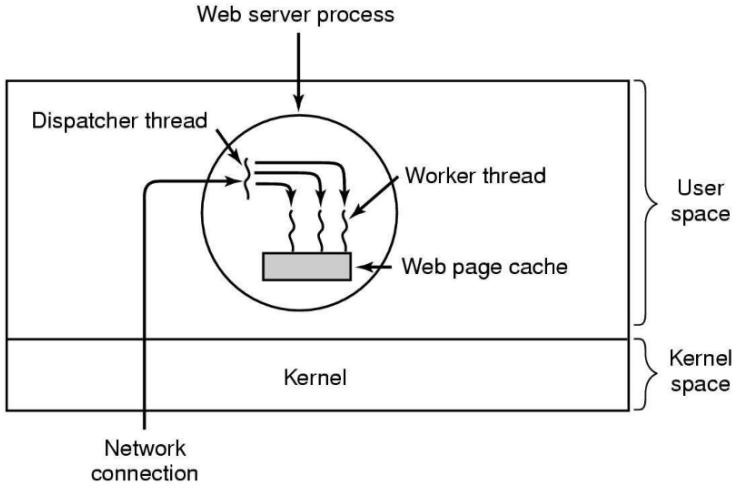
Traitement de texte



Un thread lit le clavier, un second s'occupe de mettre à jour l'affichage et un 3eme fait les sauvegardes des modifications sur le disque.

Serveur web

Ici, on a comme exemple un serveur web qui va donc devoir se charger d'envoyer les pages demandées par les utilisateurs. Puisque certaines pages sont plus demandées que d'autres, elles sont placées dans un cache pour permettre leur lecture plus rapide.

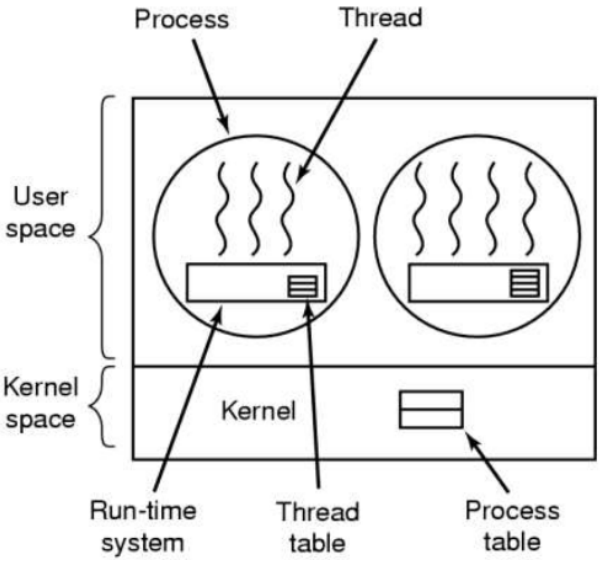


Le dispatcher thread va donc analyser la requête entrante et l'enverra à un thread "bloqué", appelé worker. Ce dernier va vérifier dans le cache si la page s'y trouve, sinon il ira la lire sur le disque avant de la renvoyer. L'avantage est que, pendant que le worker va chercher l'information (et donc est potentiellement bloqué si la page n'est pas dans le cache), le dispatcher peut continuer à traiter les demandes des autres clients et demander à d'autres workers d'aller chercher les pages demandées.

Implémentation des threads dans l'espace utilisateur

Ils peuvent être implémentés dans l'OS ou dans l'espace utilisateur.

Dans ce second cas, ce n'est pas l'OS qui gère les threads mais une librairie runtime (ex: Java Virtual Machine). L'OS n'a d'ailleurs aucune idée de l'existence de ces threads et si l'un d'eux fait un appel bloquant, c'est tout le processus qui passera à l'état "bloqué". Un thread qui attendrait une réaction d'un périphérique mettrait donc tout le processus en suspens, ce qui est très inefficace.



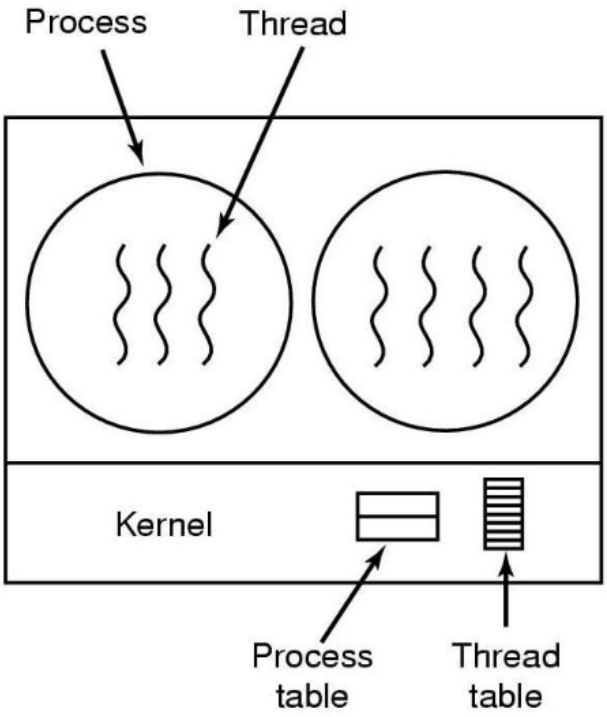
Il y a une table des threads par processus, très similaire à la table des processus de l'OS. L'avantage notable est que le processus peut avoir son propre algorithme d'ordonnancement. Puisque la gestion des threads dans ces conditions ne nécessite pas de faire des appels systèmes, tout est beaucoup plus léger.

Implémentation des threads dans l'OS

Dans ce cas-ci, c'est l'OS qui va gérer les threads. Tout va donc se passer dans le noyau et chaque demande de création/suppression de thread passera par le mode Kernel à coup d'appels systèmes.

La table des threads se trouve dans l'OS.

Si un thread est bloqué, aucun problème pour en faire tourner un autre.



Avantages et inconvénients de l'implémentation user

Avantages:

- Appels plus légers (pas d'appel système et context switching plus léger);
- Contrôle de la politique d'ordonnance;
- N'utilise pas de ressources OS (nombre de threads potentiellement limité dans l'OS).

Inconvénients:

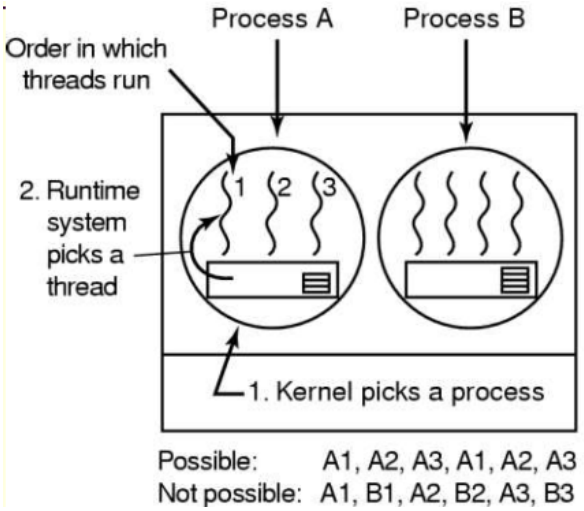
- Appels système bloquants: tout le processus est bloqué si un thread fait un appel système;
- Un thread mal programme qui ne rendrait jamais la main bloque tout le processus;
- La plupart des applications utilisant des threads font souvent des appels systèmes bloquants;
- Ordonnancement d'un thread n'alourdit pas tellement plus l'appel.

Ordonnancement des threads partie utilisateur

- L'OS alloue un quanta au processus;
- Le runtime du processus réparti le quanta entre les threads;
- Algorithmes d'ordonnancement au choix du processus.

On peut alterner les threads appartenant uniquement au processus en cours d'exécution.

ex: Quanta de 50ms et threads dépensent 5ms chacun

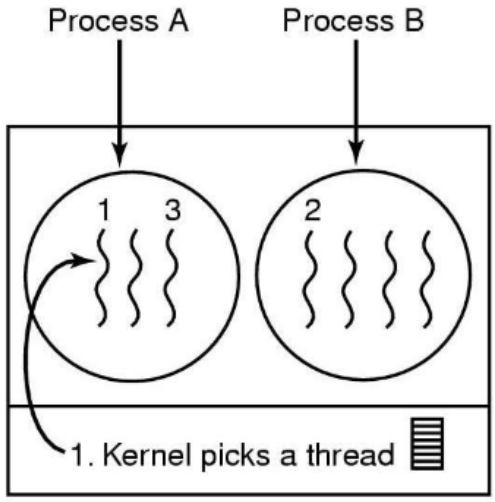


Ordonnancement des threads partie Kernel

- L'OS fait l'ordonnancement au niveau des threads

Les threads peuvent être alternés aisément, même d'un processus à l'autre, sans aucune limite puisque l'OS les reconnaît en tant que tels.

Le context switch est cependant plus lourd depuis le mode Kernel.

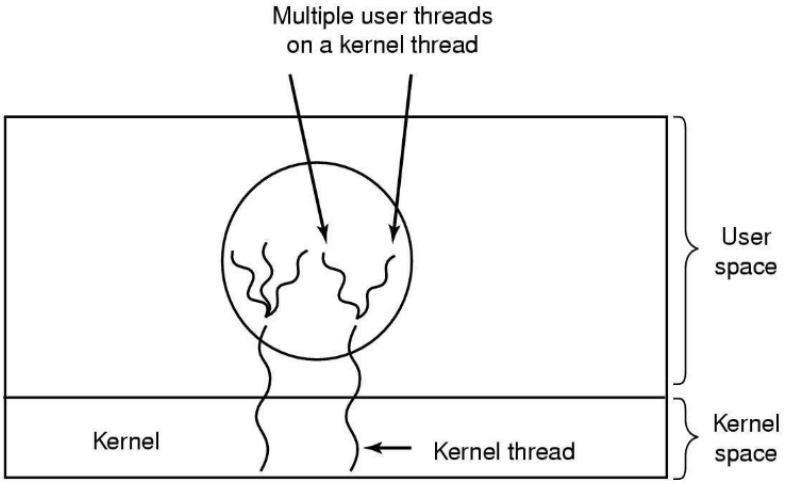


ex: Quanta de 50ms et threads dépensent 5ms chacun

Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

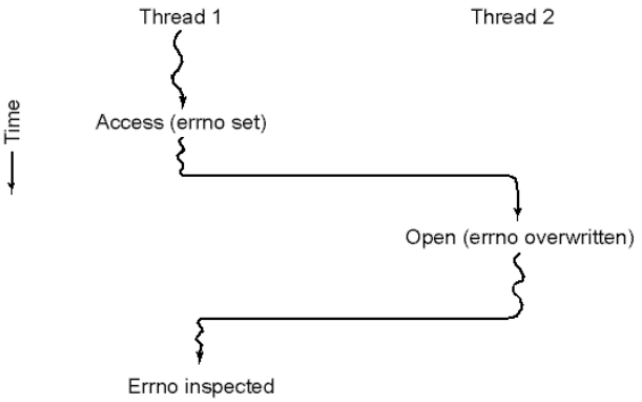
Implémentations hybrides

Des threads utilisateur sont multiplexés (liés) à des threads noyaux. Ces threads multiplexés sont créés, détruits, ordonnancés comme un thread utilisateur d'un processus qui s'exécute sur un OS dépourvu de fonctionnalité multi-threading.



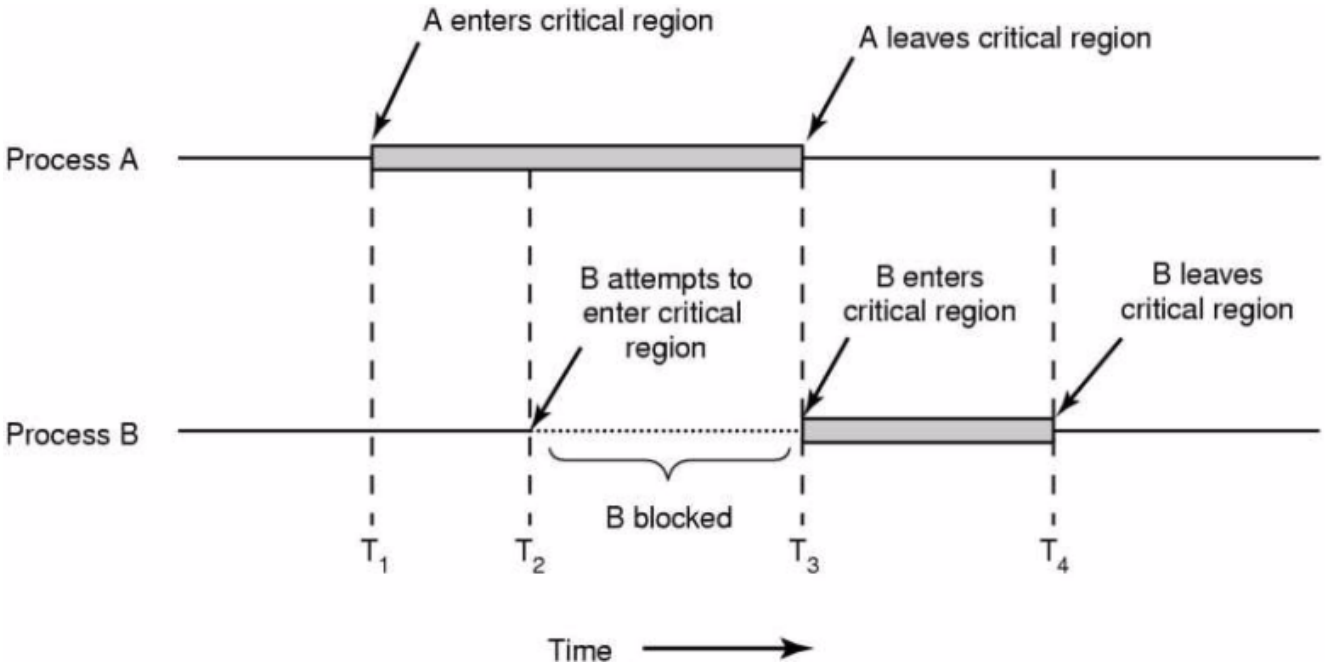
Problèmes possibles

- Accès concurrents à une même ressource: Si le thread 2 émet une erreur là où l'activité du thread 1 n'en avait pas émis et que le thread 1 fait la lecture de l'erreur après activité du thread 2, il croira qu'une erreur est survenue.
- ➔ La modification d'une variable qui devait être utilisée par un thread A dont le quantum est écoulé, le thread B modifie cette variable. Lorsque A va lire la variable, la donnée s'y trouvant aura été modifiée par B, posant fatalement un problème.



Sections critiques

Aussi appelé ***Lock***: permet de verrouiller une zone sur laquelle un thread travaille, pour éviter à tout autre thread d'y accéder, que ce soit en lecture ou écriture.
Les threads voulant accéder à une section critique passeront donc en état "bloqué" jusqu'à ce que la section soit à nouveau disponible, et pourra passer en "prêt" ensuite.



Deadlock

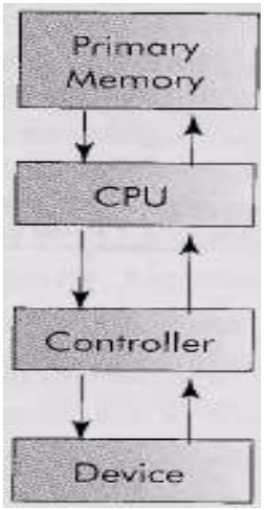
Les processus ne sont pas tout seul dans l'ordinateur, contrairement à leur système de fonctionnement qui fait comme si c'était le cas.
Ils partagent des ressources, ce qui est une source potentielle de blocage. (ex: graveur DVD, lecteur de cartouche, ...)

Par exemple, un ordinateur a un graveur A et un lecteur disque B.
Si, au même moment, deux programmes ont besoin d'accéder à ces lecteurs mais, pour une raison ou une autre, le programme 1 accède au graveur A et le programme 2 accède au disque B, et ce simultanément, ils vont tous les deux garder la main sur le périphérique auquel ils ont accès en attendant la libération du second.

Le programme 1 garde le graveur A en attendant le lecteur B
<-> ==> **Deadlock**
Le programme 2 garde le lecteur B en attendant le graveur A

S'il s'agissait d'un problème majeur au début des ordinateurs pouvant faire tourner plusieurs programmes en même temps, des solutions ont été trouvées:
- Définir un ordre d'accès prioritaire. (ex: toujours le graveur A, suivi du lecteur B);
- Relâchement d'une ressource si toutes les ressources ne sont pas libérées après un temps donné (**préemption**);
- ...

Rappels

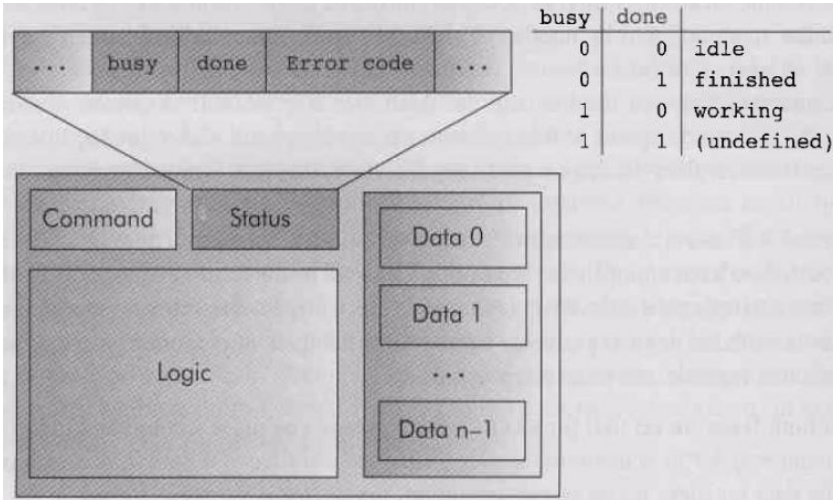


L'OS ne dirige pas directement les têtes de lecture, il passe par un **contrôleur**.

Le contrôleur offre la possibilité de piloter un périphérique physique au moyen d'instructions exécutées par le CPU.

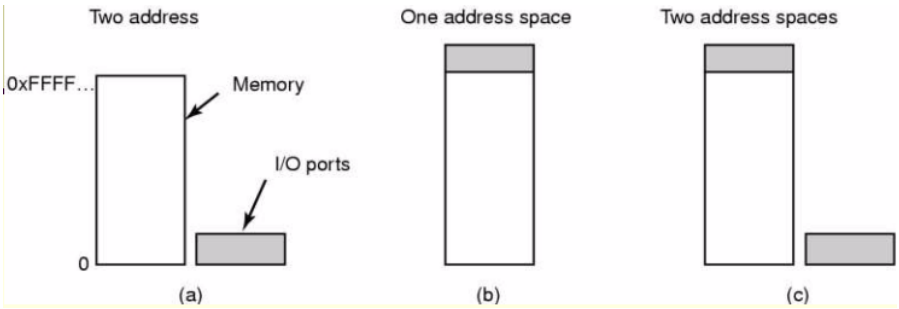
- Assure la transformation des signaux digitaux en signaux analogiques;
- Interprète les instructions en ordre de positionnement pour une tête de lecture, une cassette, ...

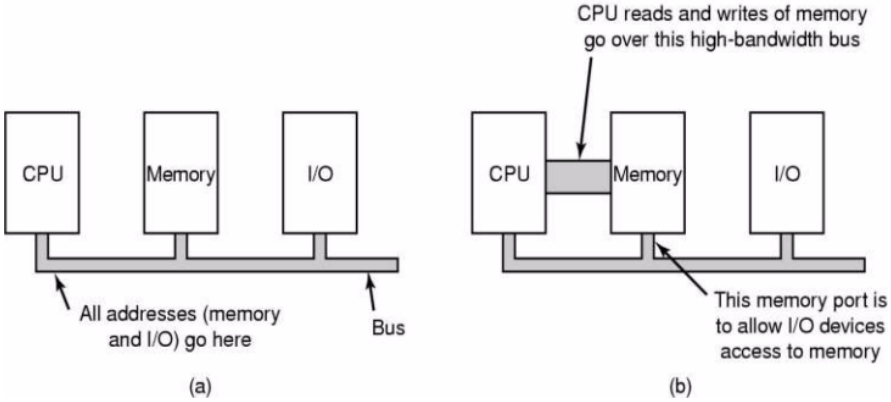
Il est accédé via des registres de contrôle.



a) Ces registres de contrôles sont accédés via des ports
   - In Reg, Port
   - Out Port, Reg

b) Ces ports peuvent être mappés en mémoire, un simple MOV suffit dans ce cas
   - Mov Reg, adresse





Abstraction du contrôleur

Les détails des opérations E/S de chaque contrôleur diffèrent.

C'est donc l'OS qui propose une interface pour masquer toutes ces différences au programmeur:

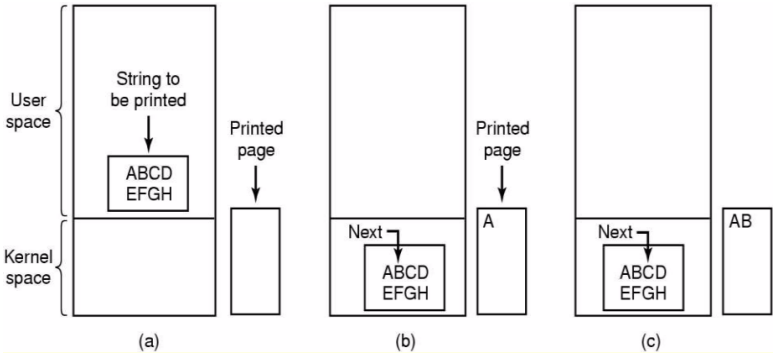
- Le programmeur emploie une interface E/S abstraite implémentée pour dialoguer avec un grand nombre de périphériques, et ce sans connaître les détails de chacune.
- Les opérations de haut niveau incluent allocate/deallocate et read/write.

Approche naïve

Imaginons la demande de copie d'un texte "ABCDEFGH".

Voici les différentes étapes:

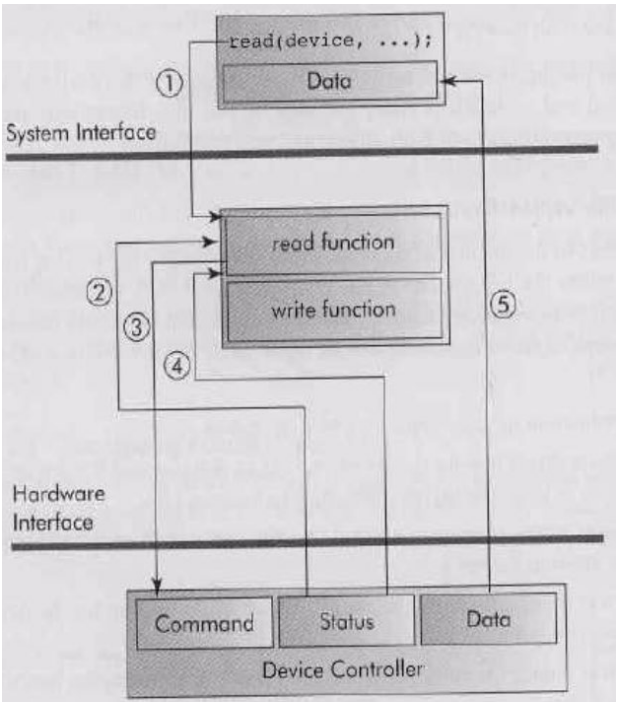
1. Demande de write par l'utilisateur;
2. le driver regarde le status du périphérique pour voir si busy => si oui, attente;
3. Mise en place de la commande dans le registre du contrôleur;
4. Attente status OK;
5. Copie du caractère suivant;
6. ...




```
copy_from_user(buffer, p, count);          /* p is the kernel bufer */
for (i = 0; i < count; i++) {              /* loop on every character */
      while (*printer_status_reg != READY) ;  /* loop until ready */
      *printer_data_register = p[i];       /* output one character */
}
return_to_user( );
```

Voici à quoi le code ressemblerait, avec une boucle while qui tournerait en attendant la libération du périphérique s'il ne l'était pas.

A noter que les données sont copiées de l'espace utilisateur vers l'espace Kernel pour assurer son accès et sa non-corruption.



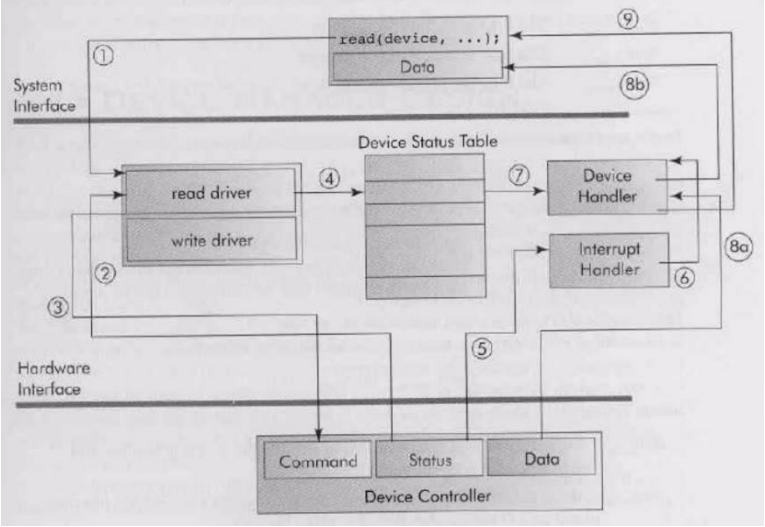
Si tout se passait comme ça, on pourrait occuper le processeur à 'attendre' (boucle while) inutilement, ce qui rendrait le tout très peu efficace.

Interruptions

Le périphérique va donc prévenir, à l'aide d'une interruption, lorsqu'il a terminé.

Au lieu d'une boucle while, à essayer sans cesse de savoir quand le périphérique est occupé, on attend plutôt qu'il prévienne, permettant donc de pouvoir travailler sur d'autres processus.

1. Demande de read par l'utilisateur;
2. Le driver regarde si le status du périphérique est busy, si oui => attente;
3. Mise en place de la commande dans le registre du contrôleur;
4. Stockage de l'info dans device status table. On rend le contrôle à un autre programme;
5. Lorsque le périphérique a terminé -> Interruption de sa part;
6. Le gestionnaire d'interruption détermine de qui vient cette interruption et branche sur le bon gestionnaire de périphérique;
7. Le gestionnaire de périphérique regarde le status du périphérique;
8. Il copie les données chez l'utilisateur;
9. Il rend la main à l'utilisateur

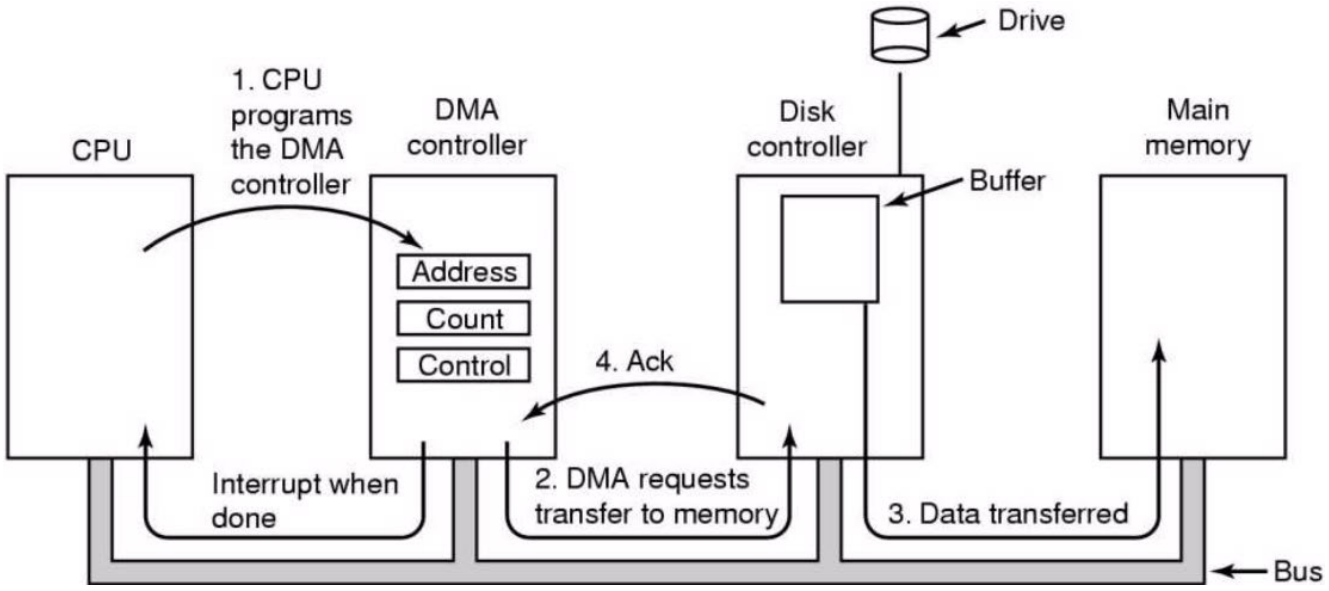


C'est donc en 4-5 que le processus pourra être suspendu par l'ordonnanceur; le temps que le périphérique se libère.

Direct Memory Access

Le **DMA** est un système permettant aux périphériques d'accéder à la mémoire sans passer par l'OS.

On aura donc un contrôleur DMA, qui fera des interruptions par bloc de données transférées plutôt qu'à chaque donnée individuelle. Le CPU ne sera donc pas interrompu inutilement et un nombre de fois important.

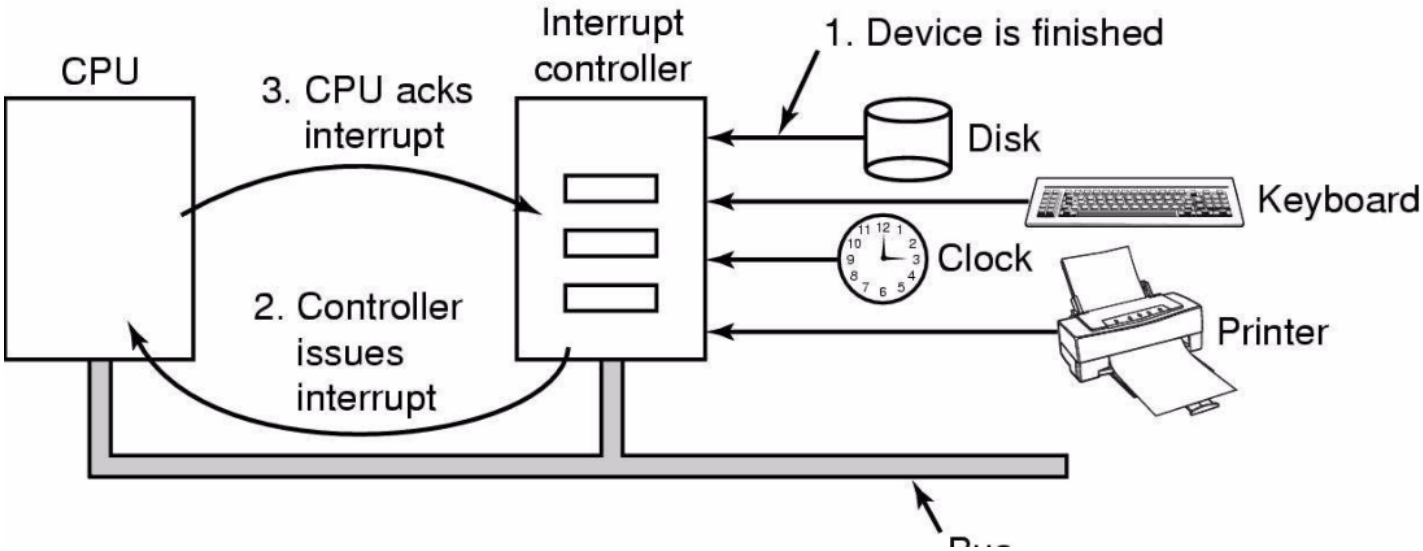


On retrouve souvent ce dernier sur la carte mère. Il a accès au bus de système. Il est conçu de plusieurs registres dans lesquels le CPU peut lire ou écrire. Dont un registre d'adresse mémoire, un pour le nombre d'octets, et un ou plusieurs registres de contrôle qui servent à spécifier le port d'E/S à employer,...

L'image ci-dessus simule une demande de lecture sur disque en passant par le DMA.

1. Programmation du DMA;
   a. Que doit-il transférer (quantité d'information dans le registre count)
   b. Où doit-il placer l'information (registre adresse)
2. Le DMA demande, au contrôleur de disque, la lecture des données, qu'il placera dans la mémoire tampon interne du contrôleur.
   Que cette demande provienne du DMA ou du CPU importe peu au contrôleur de disque;
3. Le contrôleur copie les données dans la mémoire;
4. Le contrôleur annonce la copie des données
5. Le registre Count est décrémenté, et s'il n'est pas à zéro, il reste des données à transférer.
   S'il reste des données à transférer, on recommence la boucle à partir de 2, ce jusqu'à ce que le registre Count soit à zéro.

Contrôleur d'interruptions

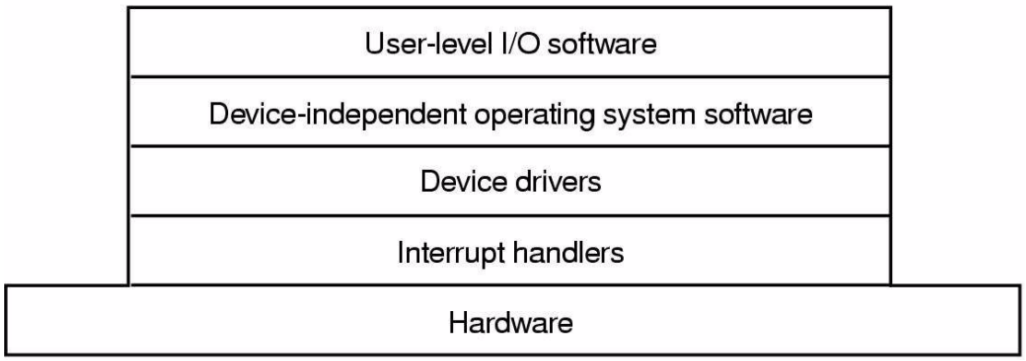


Lorsqu'un périphérique a terminé de faire ce qu'il avait à faire, il envoie une interruption au contrôleur d'interruptions.

Si cette interruption est seule, elle sera traitée immédiatement, dans le cas contraire, le niveau de priorité de chaque interruption reçue sera vérifié pour savoir laquelle traiter prioritairement.

L'interruption ayant été ignorée, le périphérique devra donc continuellement envoyer son interruption jusqu'à ce qu'elle soit traitée.

Structure de logiciel d'entrées/sorties



L'objectif est de cacher les interruptions le plus possible. On veut éviter au maximum que l'OS n'ait connaissance de leur existence. Le pilote du périphérique qui commence une opération d'E/S aura donc tendance à se bloquer tant que le périphérique fait son boulot jusqu'à ce que l'interruption survienne.

Une fois l'interruption survenue, la procédure d'interruption sait quoi faire et va gérer le tout.
Le pilote sera donc débloqué histoire qu'il puisse reprendre les opérations d'E/S. Ce modèle fonctionne mieux quand les pilotes sont structurés sous forme de processus du noyau possédant leurs propres états, piles et compteur ordinal.

Bien sûr, ce n'est pas aussi simple que ça.
En fait, le traitement d'une interruption représente une sacrée masse de travail pour le système.

En tout, 10 étapes sont comptées pour la gestion d'une interruption.

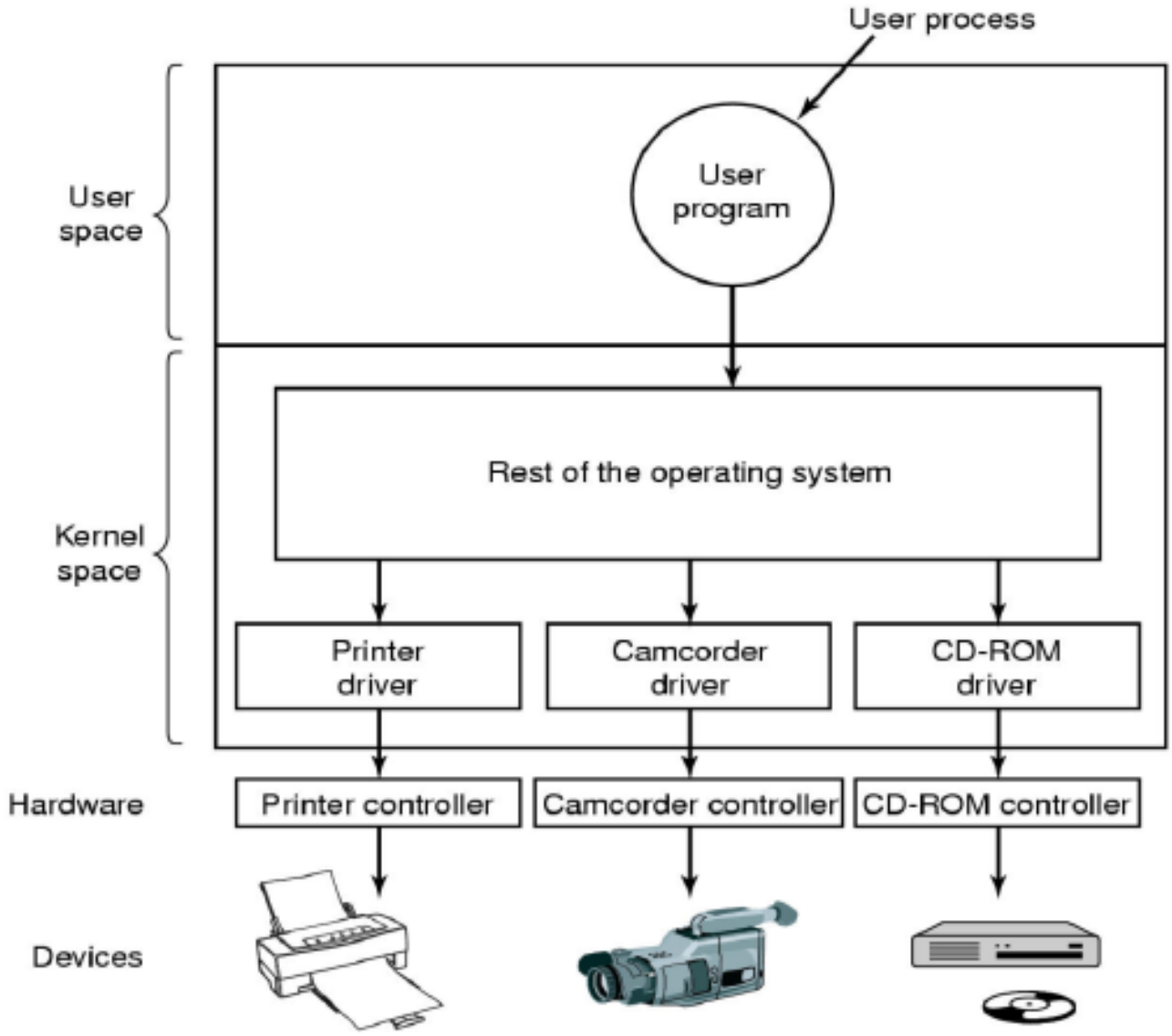
Le pilote de périphérique

Puisque tous les périphériques sont différents, l'OS ne peut pas savoir comment tous les gérer.
Certains claviers peuvent avoir des configurations particulières, des écrans faits de technologies différentes, etc.

Les pilotes(*drivers*) font partie du noyau, et doivent donc être 100% sûrs pour éviter les catastrophes.

Ils font d'ailleurs partie d'une grosse partie de l'espace disque qu'un OS occupe.

Le pilote est donc le traducteur entre l'OS et le périphérique.



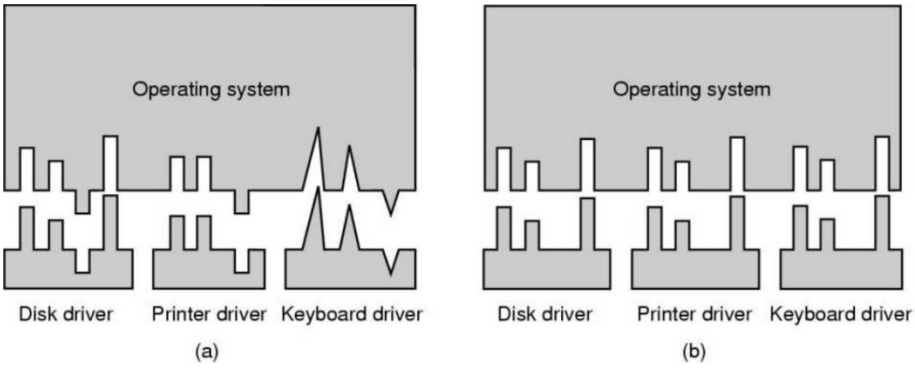
Logiciels d'entrées/sorties indépendant du hardware

L'intérêt est de permettre à l'OS de fonctionner de manière générique avec ses périphériques, sans se soucier des spécificités du matériel.

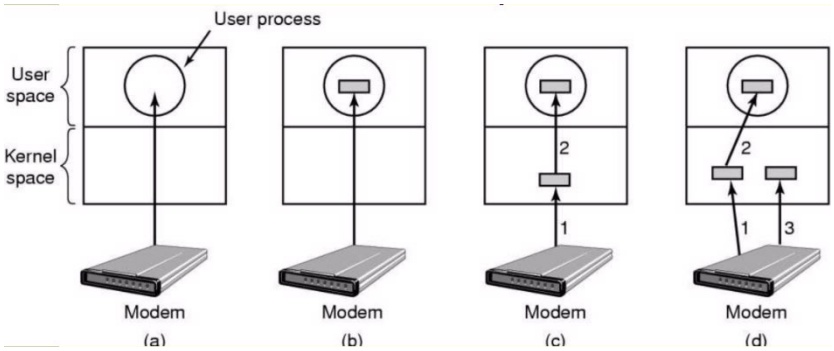
On tend donc vers des drivers façonnés selon des normes pour les rendre génériques.

C'est principalement utile pour les pilotes fabricant, car il est plus facile que ceux-ci conçoivent les pilotes de leurs propres périphériques.

Ici, on voit l'uniformisation des pilotes sur la figure **b**, contrairement à la figure **a** où l'OS devrait s'adapter.



Mise en mémoire tampon



a. Pas de **tampon**(*buffer*):
   Le processus va donc attendre un caractère, le lira quand il arrivera, et se remettra en pause en attendant un suivant. Chaque caractère reçu lancera donc une interruption, suivi d'un context switch pour réactiver le programme, ce qui est très lourd pour le système.
b. Type lecture sur disque:
   Un espace mémoire d'une taille définie permettra d'attendre d'avoir un certain nombre de caractères avant que le processus ne se bloque, limitant donc le nombre de context switch
c. Type lecture de donnée reçue via réseau, peut-être crypté ou non complété et doit donc être traité avant réception:
   On place un tampon en mode Kernel où les caractères qui arrivent sont immédiatement placés.
   Une fois cet espace plein, il est transféré à celui en mode utilisateur.
   - Si un caractère arrive pendant ce transfert, il risque d'être perdu.
d. Le double buffer permet de réceptionner des données x lorsque des données y se trouvent encore dans le premier buffer:
   On peut donc transférer les données d'un des buffers tout en remplissant le second. On intervertira ainsi celui dans lequel on écrit et celui qu'on transfère ou qui est en attente de transfert (car l'espace user est occupé).

La mise en mémoire tampon est aussi importante en sortie.

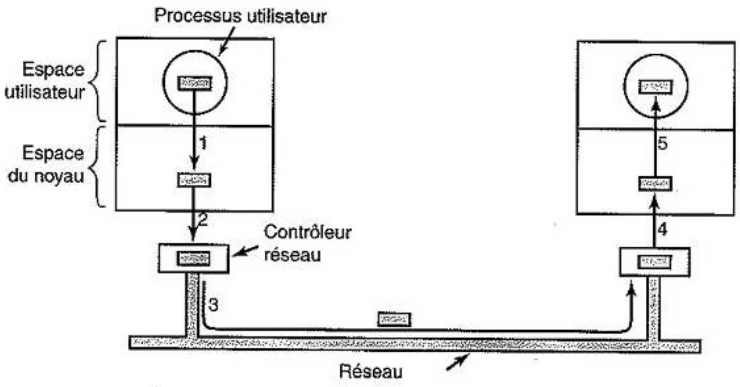


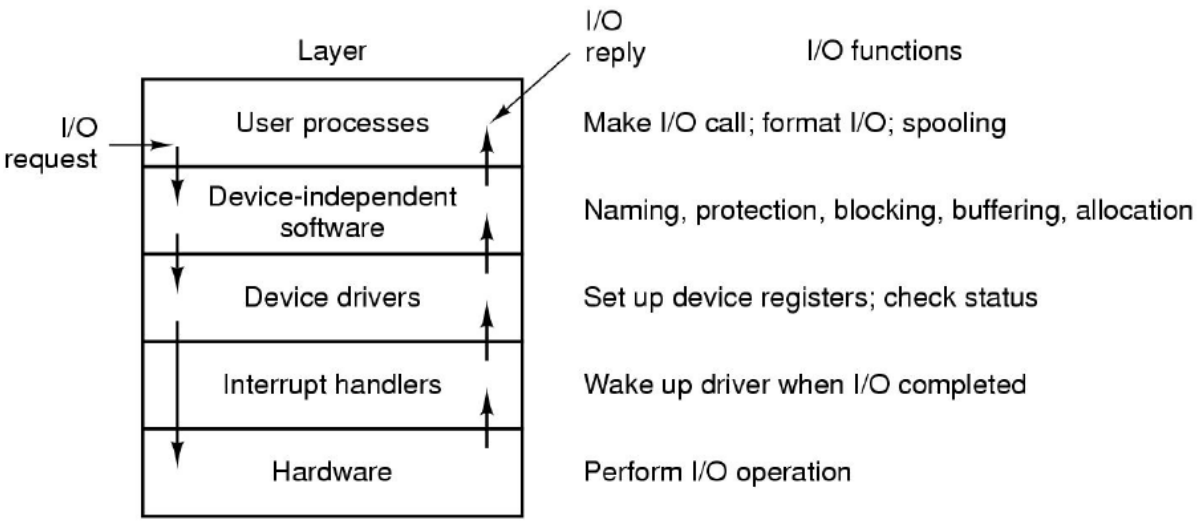
Figure 5.16 • Un réseau peut comprendre de nombreuses copies d'un même paquet.

Ci-dessus, un exemple de transfert de donnée via réseau.

1. Pour rapidement libérer le processus qui demande la copie, on copie les données à transférer dans le mode noyau;
2. Le pilote réseau copie les données dans le contrôleur réseau (la vitesse de transfert se doit d'être uniforme, ce que le pilote ne peut pas assurer car il pourrait être perturbé pendant ce temps);
3. Envoi des données;
4. Réception (et copie) des données dans le contrôleur réseau à la réception;
5. Copie des données dans un tampon noyau;
6. Copie finale des données dans l'espace utilisateur.

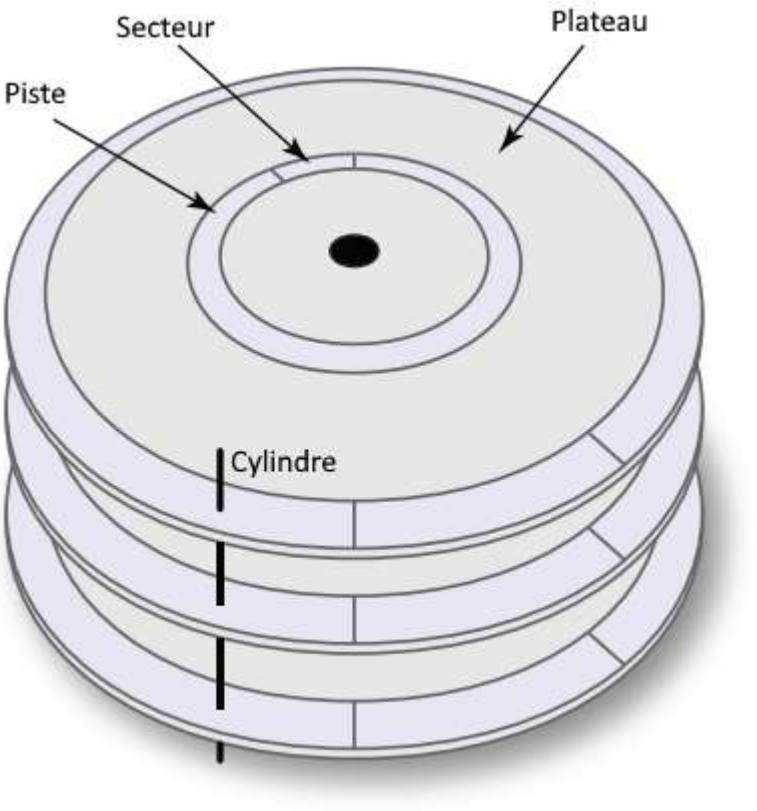
On a donc dû faire 5 copies, l'une après l'autre, puisque chaque copie dépend de la précédente et non pas de la première.

On voit assez clairement que, malgré les avantages apportés par ce système, un inconvénient majeur existe:
les performances sont réduites si le nombre de copies est trop important.



Disque dur

Technologie un peu dépassée mais les structures de fichiers se sont basées sur son fonctionnement, d'où l'intérêt de s'y pencher.



Un disque dur contient plusieurs **disques**.

Chaque disque est fait de 2 **faces**.

Chaque face contient des **pistes**.

Chaque piste est divisée en **secteurs**.

Croisement piste/secteur: **tracksector**.

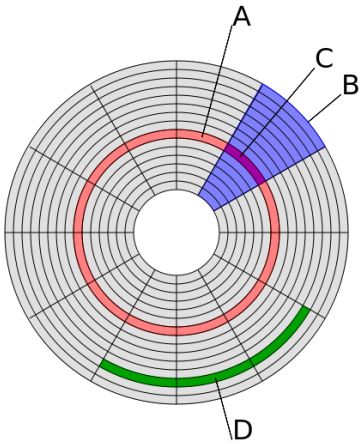
Rassemblement de tracksector: **bloc**

ou *cluster*.

/! Le **tracksector** est l'unité physique la plus petite auquel le disque peut accéder.

/! Le **bloc** est l'unité la plus petite auquel l'OS peut accéder.

La taille des blocs est variable mais est strictement identique dans chaque partition.



A.	Piste	*Track*
B.	Secteur	*sector*
C.	*Tracksector*	
D.	Bloc	*cluster*

Formatage et partitionnement

Il y a deux types de formatage:

1. Bas niveau:
   a. Fait d'usine;
   b. Consiste à magnétiser les repères pour guider la tête sur les pistes et secteurs.
2. Haut niveau
   a. Structuration du disque;
   b. Définition de la taille de l'unité d'allocation (bloc).

Le partitionnement consiste à diviser le disque en "zones" différentes (disques logiques)
On peut donc, avec un même disque dur, avoir un lecteur C :, D :, E :, ...

Le **MBR** (Master Boot Record) se trouve au tout début du disque physique et charge un petit programme qui reconnait la **table de partition** (se trouvant aussi sur le disque) et sait donc comment le disque est partitionné.

Exemple d'une partition UNIX:

1. Bootlock: Lorsqu'on a décidé sur quelle partition démarrer, c'est par lui qu'on va commencer.
   Il contient un morceau de l'OS à démarrer;
2. Super Block: information de gestion des fichiers:
   - Taille;
   - Taille bloc;
   - etc.
3. Free space Management: information sur les espaces libres;
4. I-node: information sur l'appartenance des blocs à leurs fichiers;
5. Files and directory: espace des fichiers et dossiers -> toutes les données;

Le MBR n'est plus utilisé aujourd'hui et est remplacé par le GPT/UEFI GUID.

Types de systèmes de fichiers:

FAT 16, 32	WINDOWS	Ancien système encore utilisé pour les clefs USB
NTFS	WINDOWS, LINUX	New Technology File System
EXT4	Linux	
ISO 9660	CD-ROM	
EXFAT		FAT grande taille

Fichier

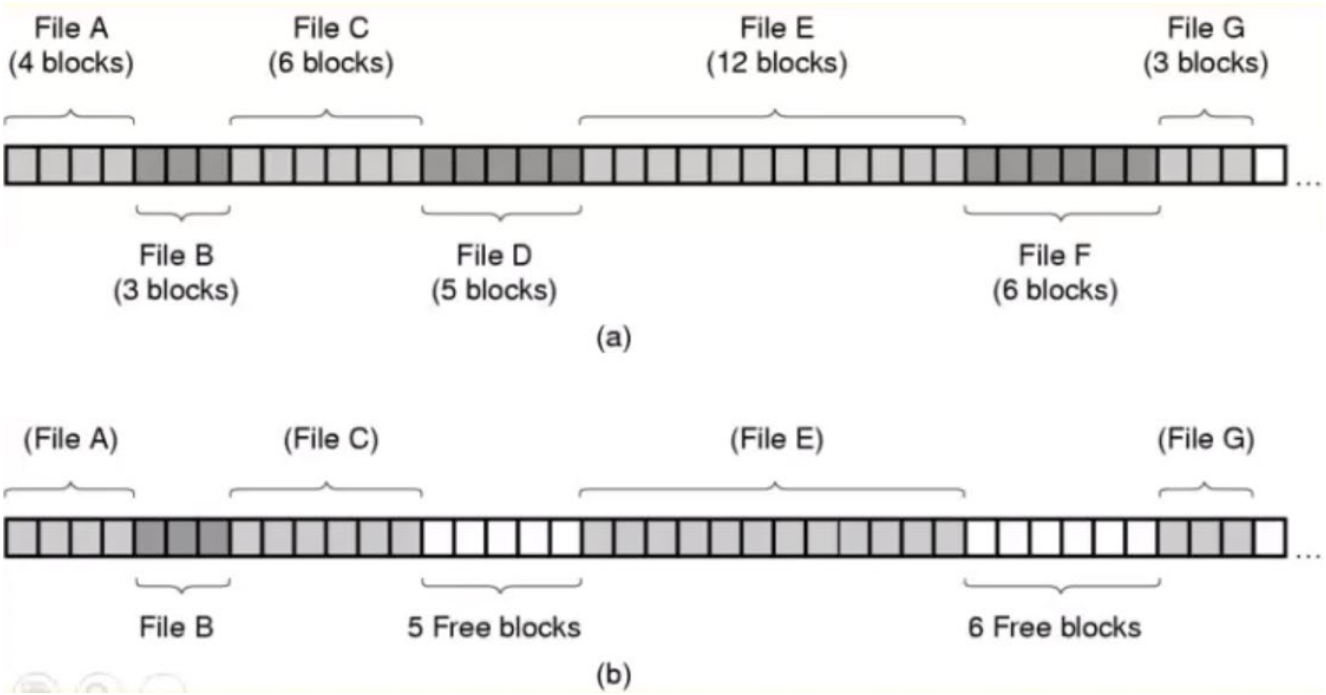
Un fichier n'est pas que contenu, il y a des informations multiples tels que:

- Date de création;
- Créateur;
- Protection (qui peut accéder au fichier);
- Nombre de bytes (taille);
- Date/heure dernier accès;
- Type: ASCII, binaire, lien,...
- ...

C'est, concrètement, un ensemble de blocs.

FAT

L'ennui, dans un système de fichiers dynamique, il peut y avoir des problèmes lors de la suppression ou le déplacement de fichiers.



On a supprimé les fichiers D et F, on a donc 5 et 6 blocs libres.

Si on veut ajouter un fichier d'une taille de 8 blocs, on ne peut pas les placer dans ces espaces vides, qui font pourtant 11 blocs au total.

➔ **Fragmentation**: des espaces trop petits apparaissent, rendant une partie de l'espace libre inutilisable.

L'idée est donc de faire en sorte que chaque bloc puisse être lié à un bloc suivant grâce à une liste chaînée.

On a donc un pointeur et une quantité de données dépendant de l'espace du bloc. Un bloc pourra donc en appellera un autre, qui en appellera un autre, qui en appellera un autre, ... jusqu'à ce qu'on arrive au bout des données du fichier.

C'est très pratique pour éviter la fragmentation et en cas de lecture séquentielle mais, si on veut accéder au bloc 4, par exemple, on va devoir passer par le bloc 0, 1, 2 puis 3 pour savoir où se trouve le bloc 4. On a donc ajouté de la complexité, ce qui tend à augmenter le temps de lecture.

La solution est de placer cette liste chaînée en mémoire qui sera chargée au moment du lancement de l'OS
=> la **FAT (***File Access Table*)

Si on cherche à avoir accès au fichier A, on suit la table

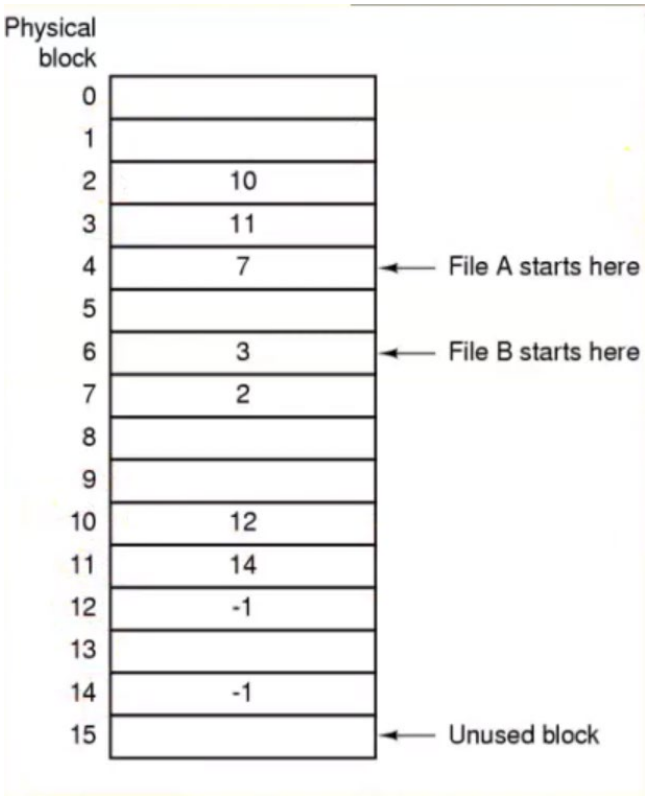
4 -> 7 -> 2 -> 10 -> 12 -> -1

Certes, on doit toujours parcourir le tout, mais en mémoire, ce qui sera beaucoup plus rapide que sur le disque.

GROS problème de la FAT: place en mémoire

En effet, pour un disque de 1TB avec allocation de bloc de 1Kb, on aura 10^9 entrées dans la FAT.

3bytes par entrée => taille de la table = 3Gb



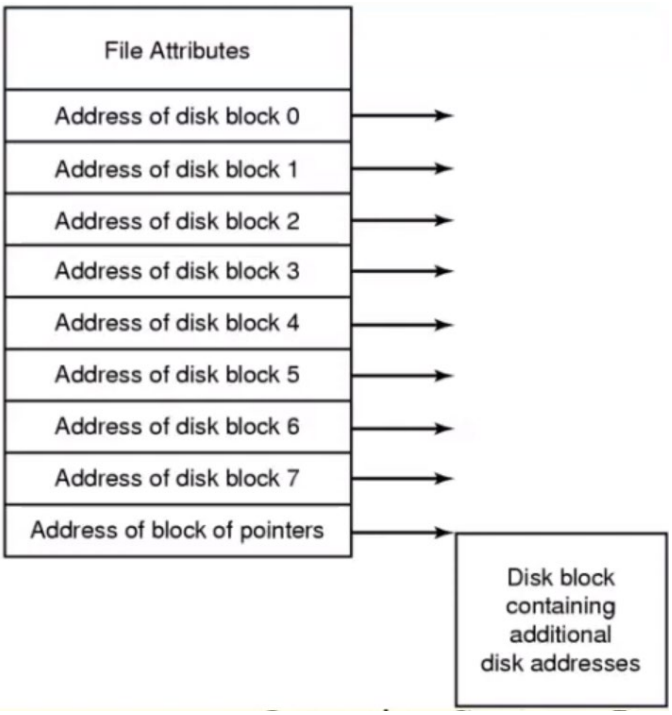
I-NODE

C'est le choix des systèmes UNIX.

Ici, il s'agit de ne charger en mémoire que les blocs des fichiers ouverts.

Un I-node est donc un bloc disque qui va contenir tous les attributs du fichier (auteur, taille, ...) suivi de la liste des blocs.

Si on n'a pas assez de place dans la table, le dernier pointeur pourra pointer vers une autre table.

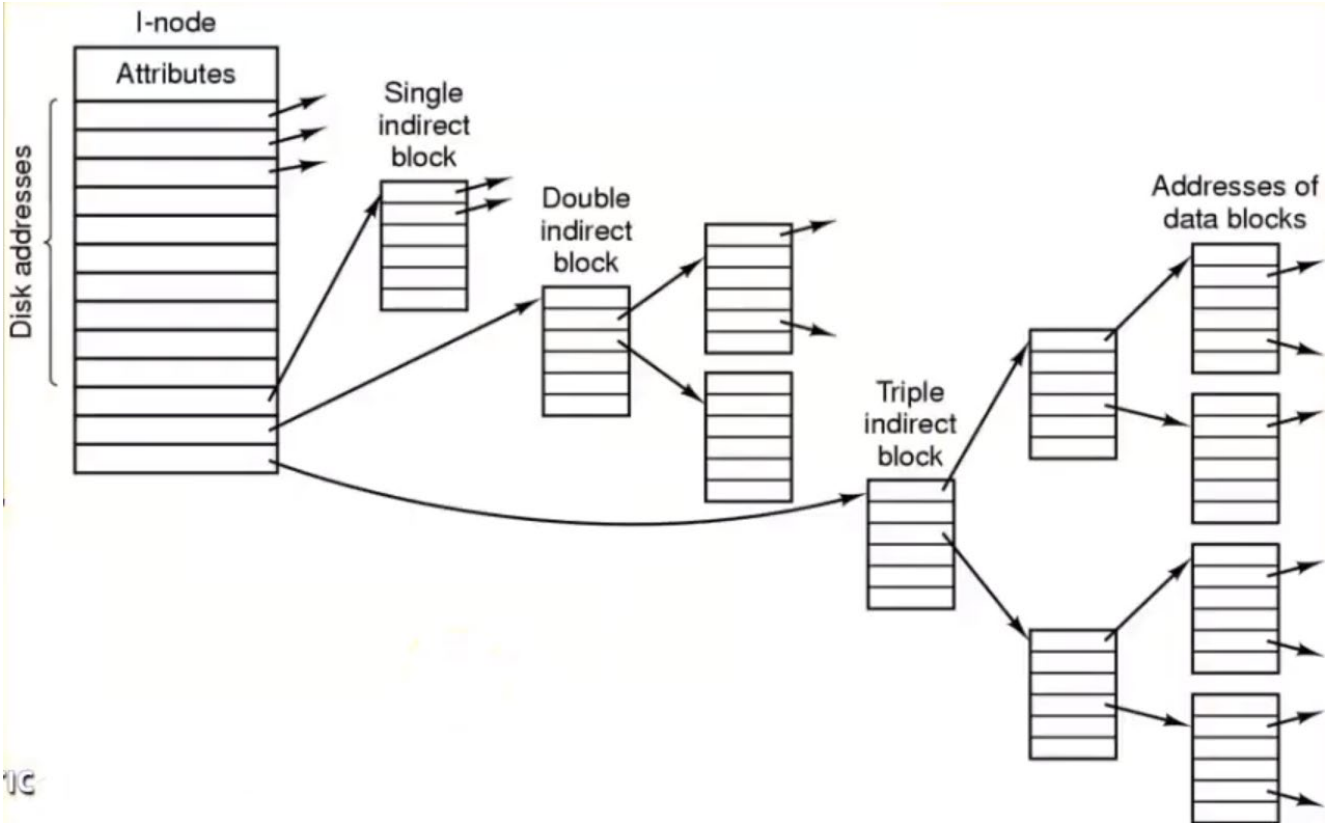


Ext 4

Sous Linux, le système de fichiers Ext 4 utilise l'i-node mais de manière un peu modifiée.

Il y a 3 niveaux de blocs supplémentaires en fin de table:

1. Une table
2. Une table de tables
3. Une table de tables de tables



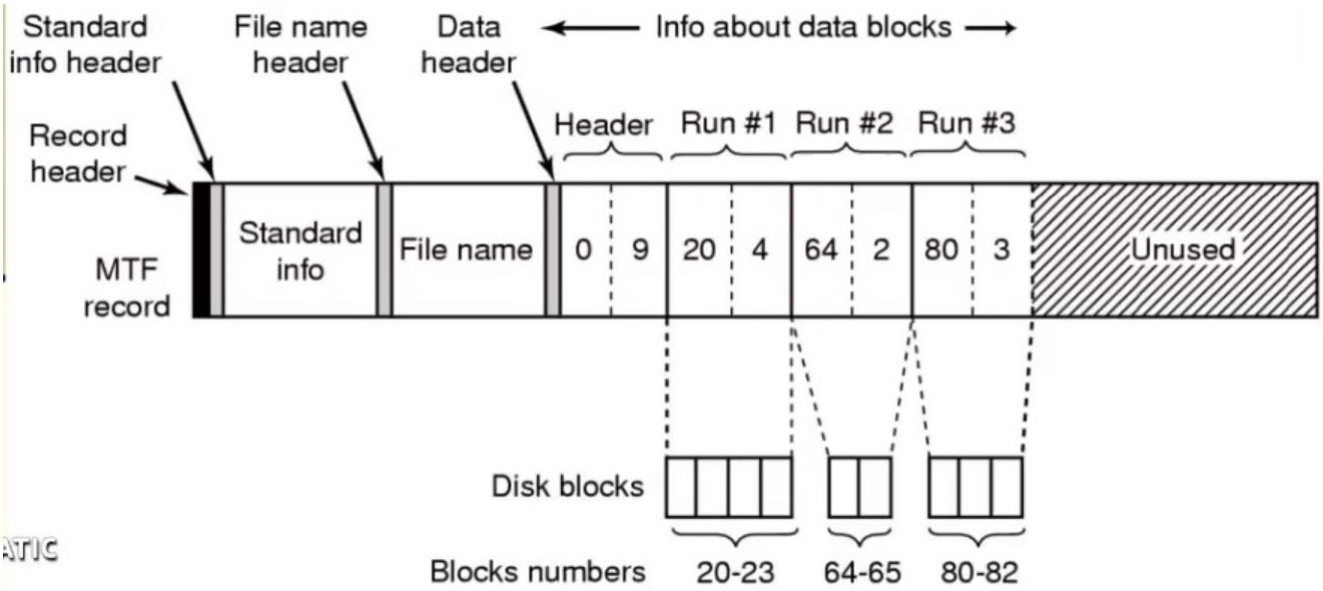
NTFS

New Technology File System (nouvelle technologie des années 90...).

Pas de FAT mais une **MFT** (Master File Table).
Ici, chaque entrée fait 1Ko et contient les informations standards à propos du fichier.

La différence se situe au niveau du renseignement des blocs.
Il est considéré que, la plupart du temps, des blocs de suivent (on dit qu'ils sont **contigus**), et donc qu'un range existe (appelé ***run*)**.
L'avantage d'un *run*, c'est qu'il y a 2 données: celle du début et, dans ce cas, le nombre de blocs qui se suivent.

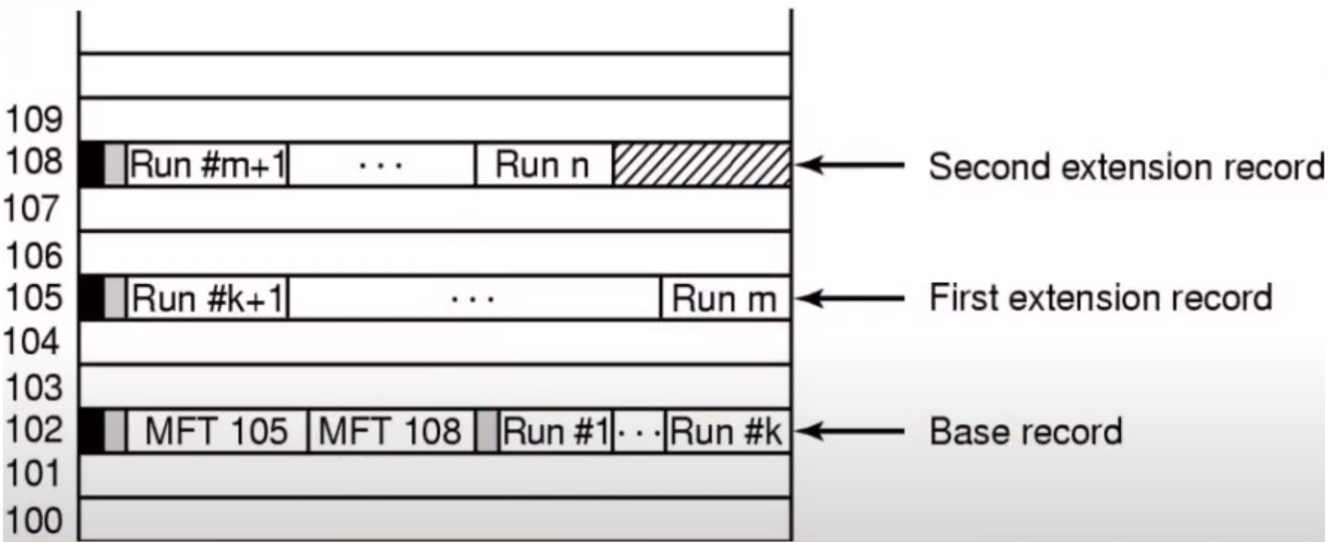
Il s'agit littéralement de compression de données.



Donc, si on a les blocs [20,21,22,23,64,65,80,81,82], on situe 3 runs:

1. 20-23
2. 64-65
3. 80-82

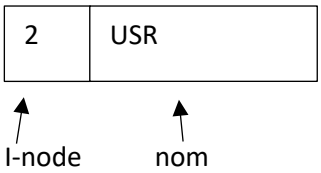
Bien entendu, puisqu'il n'est pas rare d'avoir des centaines de blocs utilisés, si l'entrée de 1Ko ne suffit pas, on peut l'étendre.



Répertoires

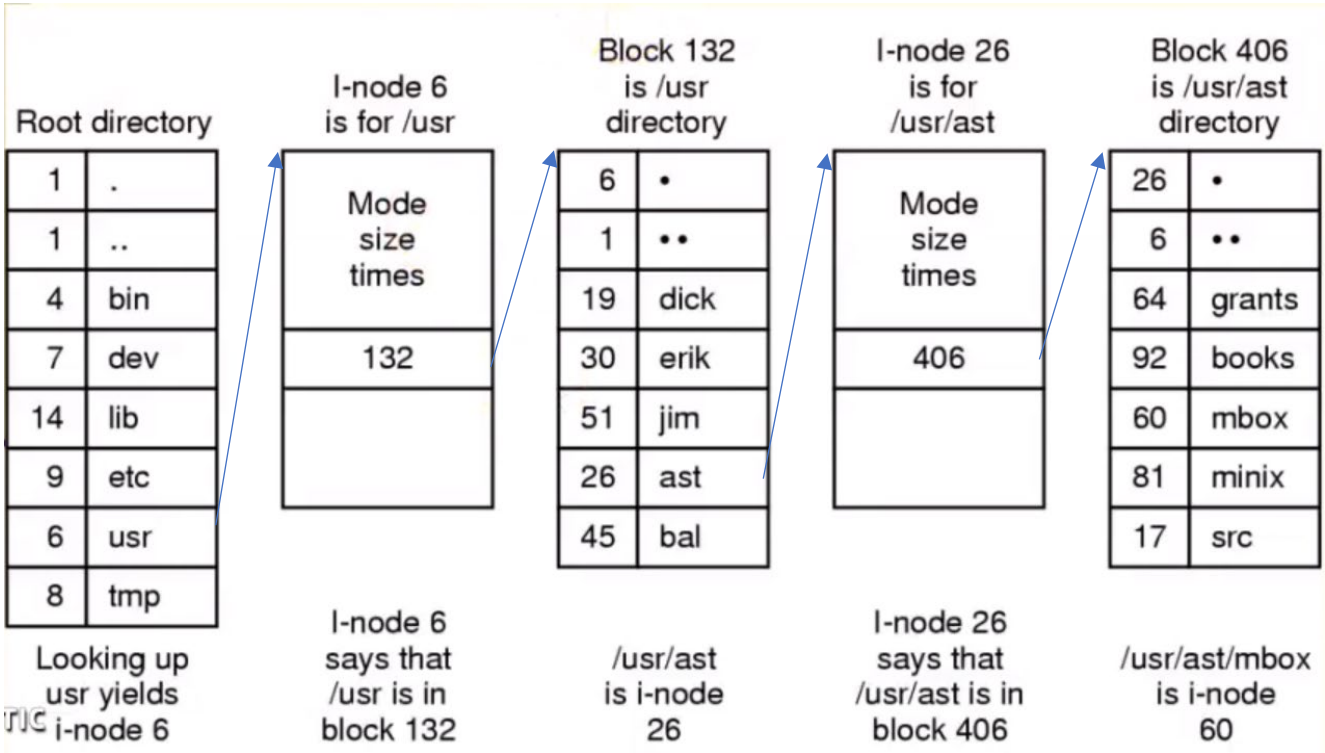
Leur hiérarchie est en arbre: une racine et des éléments dedans, qui sont racines d'autres sous-éléments.

Un dossier est un fichier, son contenu est d'autres fichiers et dossiers.



I-node: référencie les blocs appartenant à un fichier

Recherche de /usr/ast/mbox



Taille des blocs

La taille des blocs (*cluster*) peut être choisie lors du formatage.

Puisqu'on ne peut avoir que maximum 1 fichier par bloc, en choisissant des blocs de grande taille, on diminue donc la quantité de data que l'on peut placer sur le disque si on a beaucoup de petits fichiers (car ils prendront plus de place). La vitesse de lecture s'en verra en revanche augmentée car il faudra moins se déplacer de blocs en blocs, ce qui prend un temps considérable.

À l'opposé, en choisissant des blocs de petite taille, on augmentera la quantité de données que l'on peut placer sur le disque mais, pour tous les fichiers plus importants que la taille d'un bloc, il faudra beaucoup plus de déplacements de blocs en blocs, ce qui impactera la vitesse de lecture.

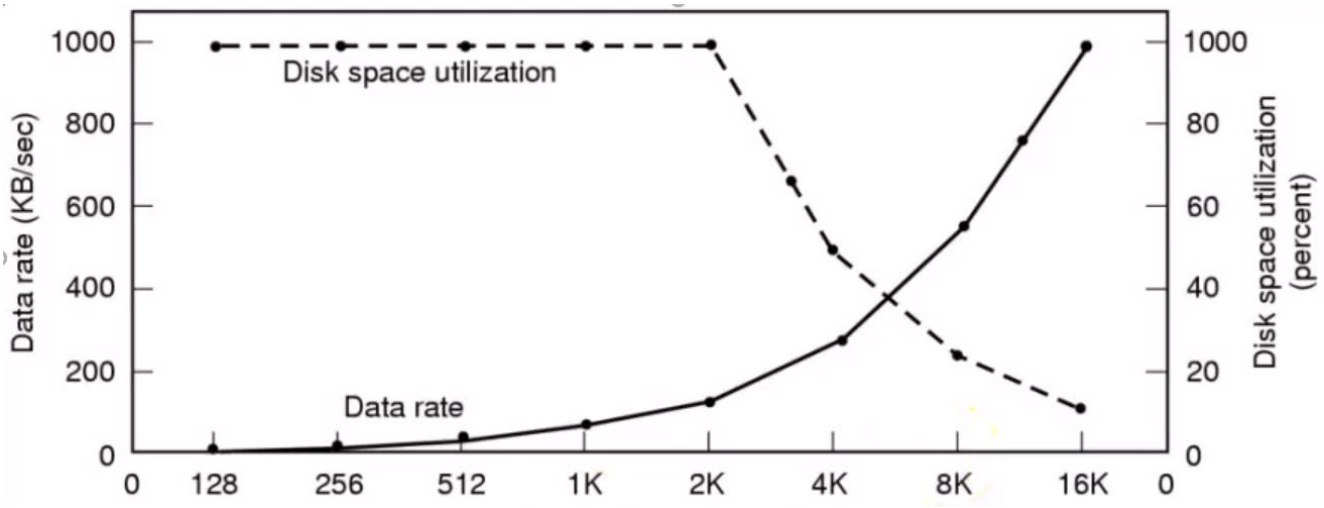
Pour résumer:

	Data	Vitesse
Grands blocs	-	+
Petits blocs	+	-

La taille d'allocation doit donc être choisie en fonction de l'utilisation.

Dans un système de fichier UNIX, on a beaucoup de petits fichiers (taille moyenne = 1Ko)

➔ Si on prend des blocs de 32Ko, on perd 97% d'espace disque!



Ce schéma explique bien le fonctionnement du disque selon l'allocation choisie.

Il a été déterminé que 4k était le choix le plus équilibré.
C'est d'ailleurs le choix proposé par défaut lors de formatage par Windows.

rmq: On pourrait donc envisager d'augmenter la taille des blocs sur un HDD stockant uniquement des fichiers de type films 4K. La vitesse de lecture serait optimalisée et, étant donné la taille des fichiers, il n'y aurait pas de perte significative d'espace.

Blocs cache

Le temps d'accès du disque est beaucoup plus grand que le temps d'accès à la RAM.

Certains blocs ont besoin d'être lus de manière répétée, le système de cache va donc permettre d'optimaliser la vitesse d'accès à ces blocs.

L'idée est de copier les derniers blocs lus dans la RAM. Lorsqu'on a besoin d'accéder à un bloc, on va d'abord vérifier s'il est dans la RAM, si c'est le cas, on l'y lit, sinon, on va le chercher sur le disque et on le copie dans la RAM.

Une liste des blocs est conservée pour rapidement vérifier la présence d'un bloc en mémoire.
Cette liste se débarrassera donc du dernier bloc (LRU: least recently used) de la liste auquel l'accès a été fait.
L'indice est un hash du bloc (HashSet donc).
Si on lit un bloc qui se trouve déjà dans la liste, il repassera en tête de liste (MRU: most recently used).

Problèmes potentiels:

Si des modifications sont opérées, le block dans la RAM peut être mis à jour mais pas le disque.
Lors d'un arrêt brutal de communication (panne de courant, retrait du support), il se peut que les dernières modifications n'aient pas été copiées sur le support, entraînant donc la perte des données.
(Problème connu lors du retrait d'une clef USB sans la déconnecter depuis l'OS)

Solutions:

1. Write-through caches: écriture du bloc sur le disque dès qu'il est modifié.
2. Écriture immédiate des blocs sensibles (i-nodes et autres)
3. Mix (Unix): écriture immédiate des blocs sensibles et sauvegarde toutes les x secondes des autres blocs (configurable).

Systèmes journalisés

La journalisation des actions ou comment ne pas corrompre tout son système.

Prenons l'exemple de la "simple" suppression d'un fichier.

1. Suppression de l'entrée du fichier dans le directory;
2. Libération de l'i-Node et placement de celui-ci dans les i-nodes libres;
3. Libération de tous les blocs et placement de ceux-ci dans les blocs libres.

S'il y a un plantage durant cette action?

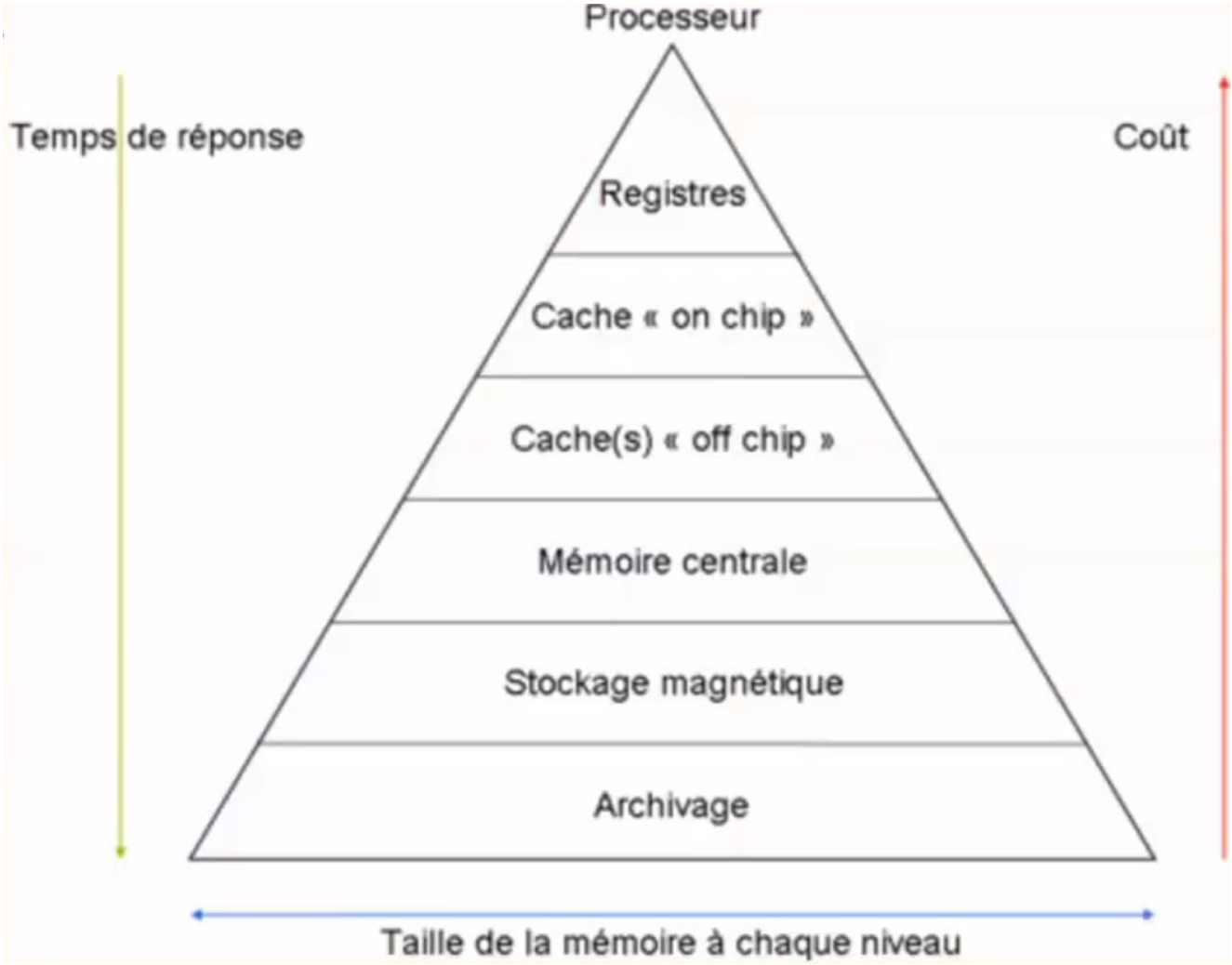
- Après 1: on perd l'i-node et les blocs;
- Après 2: on perd les blocs.

Les OS modernes vont donc, avant toute action potentiellement dangereuse, écrire dans un journal toutes les actions nécessaires à l'aboutissement de la demande initiale.

Les actions sont ensuite effectuées, et une fois la dernière terminée, le journal peut être effacé.

Si un plantage quelconque survient entre-temps, le journal sera relu lors du redémarrage du système et toutes les actions seront à nouveau effectuées.

La mémoire

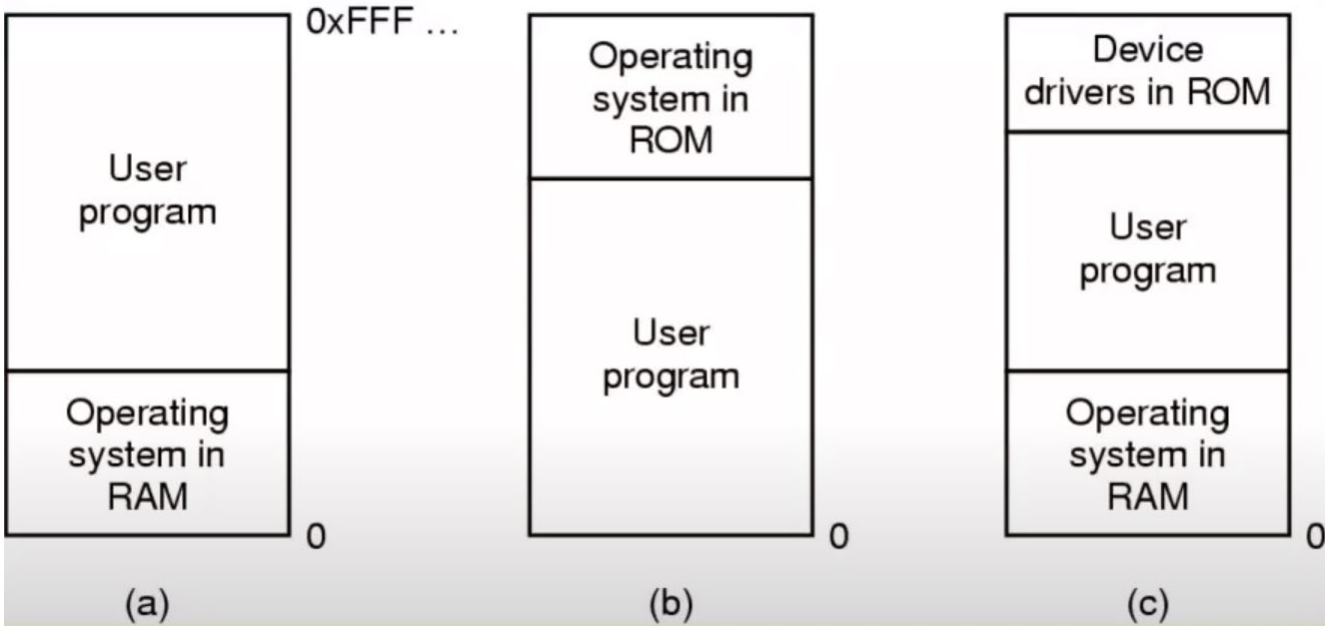


Stockage magnétique: disques

Archivage: Disques optiques, web, systèmes de stockage grande échelle

Relocation

Au début, les OS n'avaient pas de gestion de la mémoire: une adresse du programme = une adresse physique dans la mémoire.



L'OS se trouvant en mémoire, on ne sait pas forcément où, c'était un problème.

Il a donc fallu résoudre 2 problèmes

- Relocation: où se trouve la vraie adresse physique?
- Protection mémoire: ne pas permettre à un programme d'en écraser un autre

2 solutions:

1. Relocation software (calcul au moment du chargement)
2. Relocation hardware (calcul par le CPU) --> Choix dans les OS modernes

Relocation: correction de l'adresse d'un symbole pour avoir son adresse réelle.

Espace d'adressage d'un processus: l'ensemble des adresses entre les deux registres BASE + LIMITE
C'est donc la zone dans laquelle un processus peut travailler.

C'est le processeur qui ajoute les valeurs d'adresse à la base. Tout programme ne se préoccupe donc pas de son emplacement en mémoire car il fait comme s'il commençait à zéro.

Intel: Registres de segments --> le décalage de la relocation

- CS: segment de code
- DS: segment des données
- SS: segments de la pile

Protection de la mémoire:

Seul l'OS peut modifier les registres BASE et LIMITE. L'accès est limité aux adresses entre [BASE, BASE+LIMITE]
Si une tentative d'accès en dehors de ce range: Error Segmentation Fault-> interruption processus

Mémoire virtuelle

La taille du programme est souvent supérieure à la place allouée en mémoire.

Historiquement: on ne charge que ce dont on a besoin

Aujourd'hui: L'OS va découper les processus en petits morceaux et ne chargera dans la RAM qu'uniquement les parties les plus utilisées, les autres sont stockées sur le disque.
Au besoin, on sort les parties non utilisées pour placer, en copiant du disque, les parties dont on a besoin.
Lors de ce *swap* le programme est bloqué en I/O.

On aura donc un espace d'adressage virtuel. Le processus va donc croire être entièrement chargé en mémoire mais, lorsqu'on voudra accéder à une adresse virtuelle, cette adresse sera traduite en adresse physique, un *swap* chargera les données manquantes si nécessaire. Cela permettra donc d'avoir un espace d'adressage virtuellement plus grand que celui disponible en mémoire.

La gestion de la mémoire virtuelle se fait via un élément hardware: le **MMU** (memory management unit)

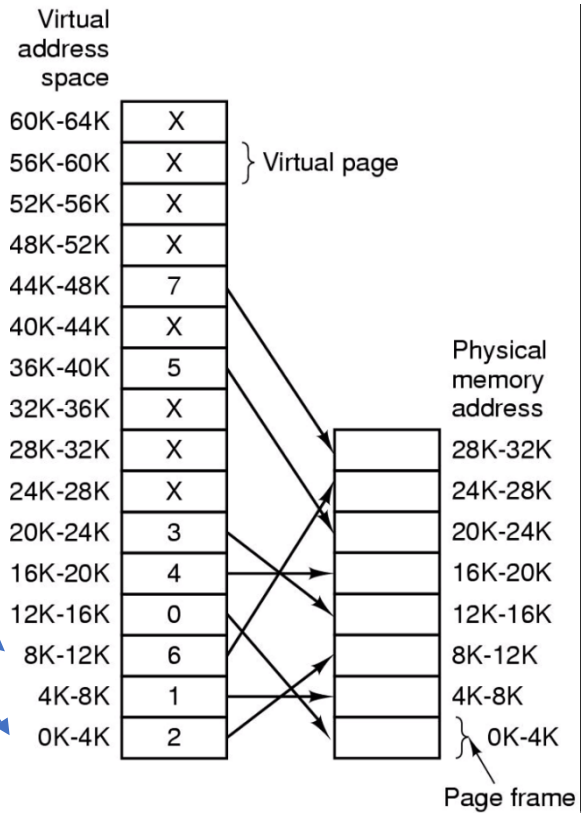
Le MMU va traduire les adresses virtuelles en adresses physiques pour le CPU (qui, lui, croira disposer de plus de mémoire qu'il n'en a) et inversement dans le sens opposé.

Aujourd'hui, CPU et MMU sont dans le même boitier mais restent bien deux éléments différents.

Pour la traduction de l'adresse virtuelle vers physique, on prendra la valeur de la taille d'une zone, ici 4096 (4k), et on le multipliera par l'indice de la zone (commence à 0).

ex:

- Adresse virtuelle 0 = 2*4096 = 8192
- " " 8192 = 6*4096 = 24576
- " " 36864 = 5*4096 = 20480



X --> page non chargée en mémoire

Si on veut accéder à cette zone, il y aura un **page fault**. ex: MOV [25631]
Le processus va donc être suspendu, une page virtuelle va être retirée de la mémoire et la nouvelle zone sera chargée et sa page virtuelle créée. L'action interrompue va ensuite être relancée.

Comment choisir quelle page sera sortie de la mémoire?

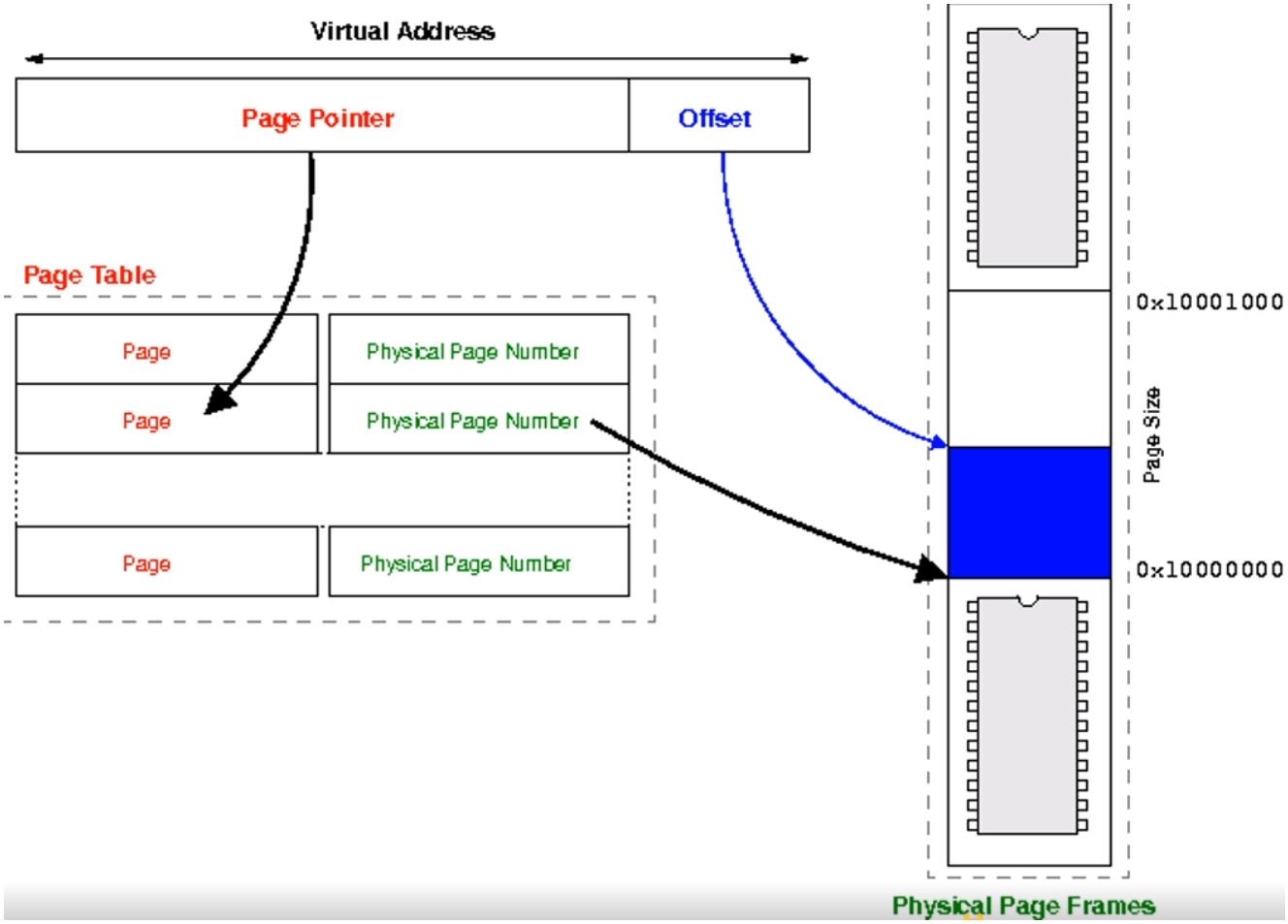
LRU est trop cher, FIFO n'est pas très efficace.
--> NRU (Not Recently Used): working set qui finira, après quelques swaps, par représenter un bon ensemble des pages les plus utilisées.

Working Set:

Dans la liste des informations à propos des pages, on va retenir l'heure du dernier accès à la page ainsi qu'un bit qui signale si la page a été accédée, ou pas, depuis le dernier *page fault*.

Lorsqu'un *swap* sera nécessaire, il y aura donc une vérification qui va voir si:

- Vérifier s'il existe des pages qui n'ont pas été accédée
- Si oui: On supprime celle dont le timestamp est le plus ancien de ces pages.
- Si non: On supprime celle dont le timestamp est le plus ancien.

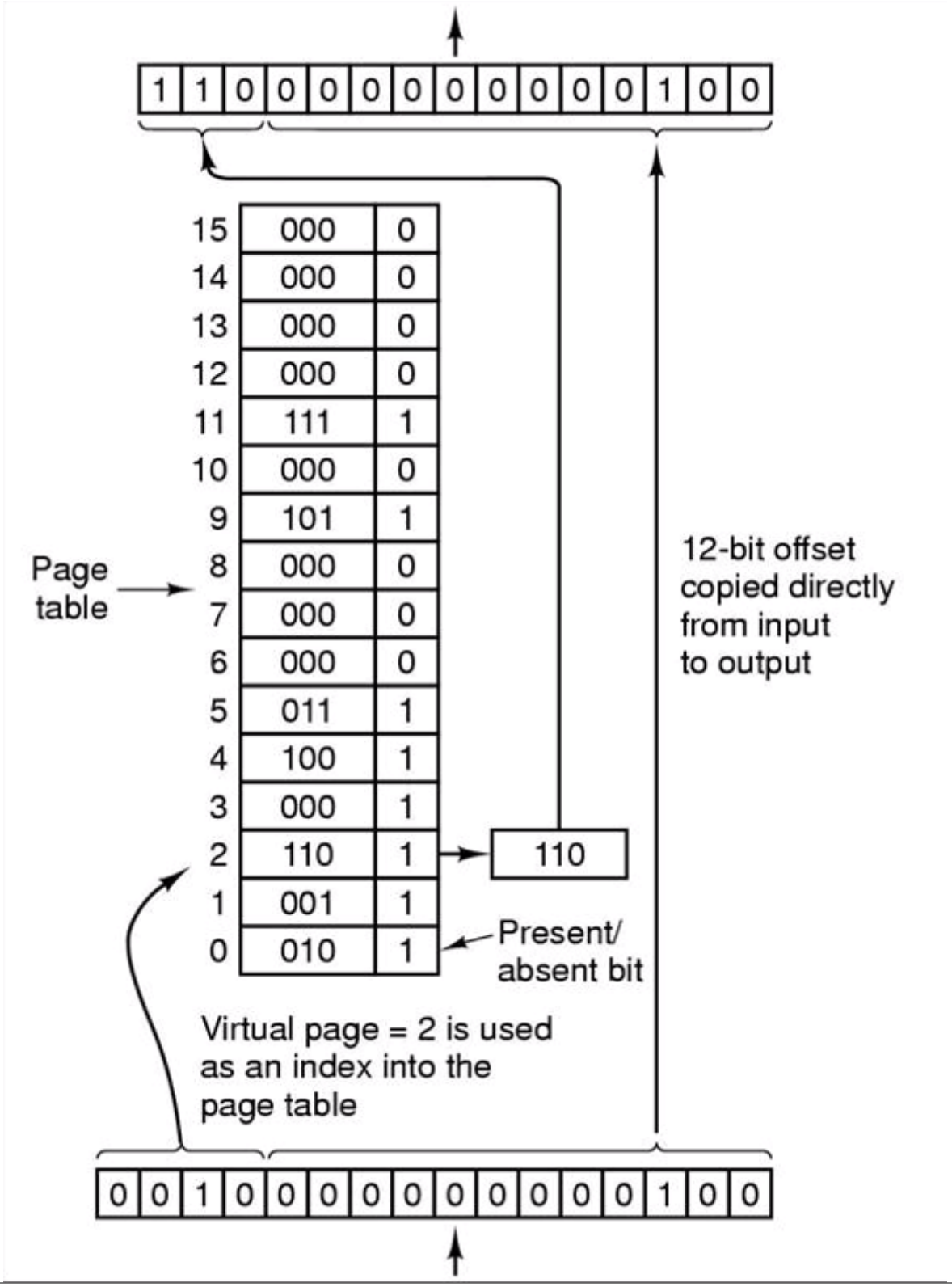


Une adresse virtuelle est divisée en 2 parties:

1. Bits de poids fort: n° de page virtuelle
2. Bits de poids faible: offset

Offset: déplacement, décalage dans la page

Ainsi, les bits de poids fort dirigent vers la page virtuelle, laquelle contiendra le coefficient de l'adresse physique (sans l'offset). Ces bits seront donc remplacés par la valeur du coefficient.



En hexadécimal:

0x2050	=> 0x2000 + 0x050
Page virtuelle	= 0x2 = 2_d
Page 2 chargée?	=> 1 => OK
Valeur de la page	= 110_b => 0x6
Déplacement	= 0x050
Adresse traduite	=> 0x6 050

Détail d'une entrée de table des pages:

- Généralement 32 bits;
- Numéro du cadre de page (plus important)
- Bit de présence/absence 1= présent 0=absent=>default page
- Bits de protection (1 ou 3)
  - Si 1: lecture seule ou lecture/écriture
  - Si 3: lecture, écriture, exécution
- Modifié: 1 si la page a été modifiée (dit ***dirty***), 0=clean
  Si elle a été modifiée, l'OS saura qu'il faut la copier sur le disque
- Référencé: 1 si a été consultée
- Cache inhibé: important pour les pages mises en correspondance avec des registres matériel plutôt que mémoire.

///////	Cache inhibé	Référencé	Modifié	Protection	Présence	Numéro du cadre de page

/! L'adresse disque qui sert à obtenir une page qui n'est pas en mémoire ne fait pas partie de la table des pages. Les informations nécessaires à traiter les default pages sont dans une table logicielle à l'intérieur de l'OS.