Institut Paul Lambin

OSII

Synthèse du cours

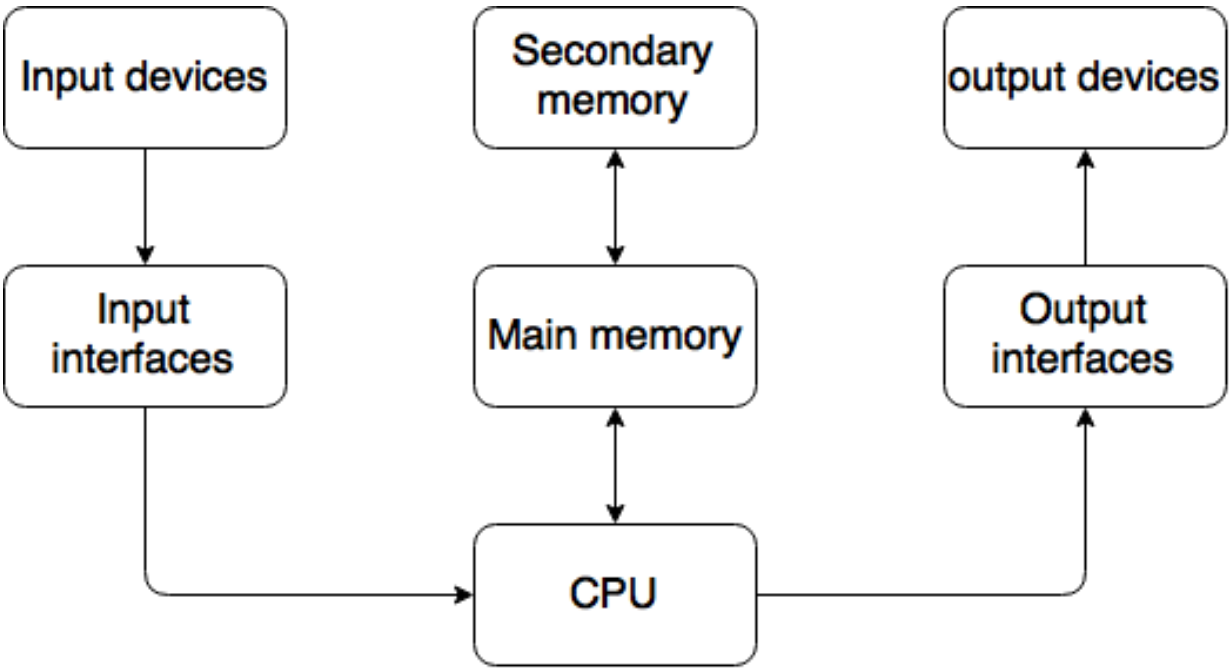
Nicolas van Gelder
21/12/2016

Table des matières

Introduction

dimanche 18 décembre 2016
12:41

Base component



La conception des ordinateurs actuels fût inventé au XIXe siècle (1834) par Charles Babbages. Sa machine exécutait étape par étape des instructions afin de fournir un résultat d'une opération arithmétique. Ses instructions étaient luent à partir de cartes perforées et le résultat obtenu était également obtenu par d'autres cartes perforées. Cette machine comportait tous les composants d'un ordinateur moderne :

- Une mémoire contenant les instructions à exécuter
- Une unité de contrôle assurant le transfert vers l'unité centrale
- Une unité centrale exécutant les instructions
- Des unités d'entrées et sorties permettant l'échange de résultats avec l'extérieur

L'architecture de cette machine ne fût améliorée que le siècle suivant grâce au développement de l'électronique.

Que nous réserve les ordinateurs quantiques?

Loi de Moore

"Le nombre de transistor dans un processeur double tous les 18 mois."

La limite a été atteinte en 2005-2006. La loi de Moore est modifiée et on recherche de nouveaux moyens pour améliorer la vitesse.

Solutions proposées :

- Multi-core
- Multi-threading

Challenge :

- Optimisation des ressources et non juste le CPU
- Déterminisme (Avoir toujours le même résultat)
- Bonne coordination

Architecture

- Mémoire
- instruction
- registre
- Interruption
- Gestion I/O
- Mode kernel

OS components

- Superviseur
- Dispatcher
- Gestion de la mémoire (virtuelle, réelle, auxiliaire)
- Gestion des ressources
- Etc…

--> Sera abordé pendant tout le cours

OS evolution

Avant	Après
1 OS	1 OS
1 programme	Plein de programme
	Gestion de la mémoire

- Chaque programme pense que toute la mémoire lui apartient
- C'est l'OS qui gère la mémoire

Quelques chiffres …

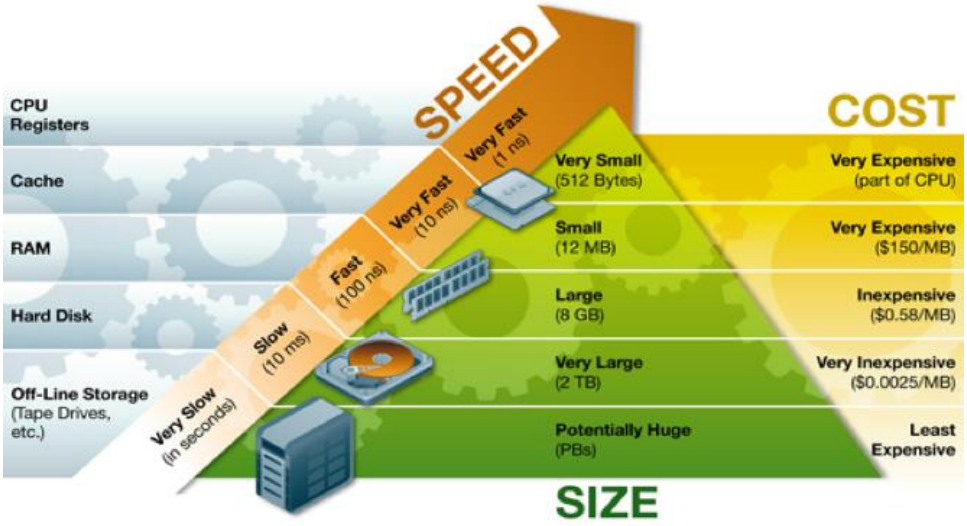
$$2^{10} = 1024$$
$$2^{6*10} = 1000^6$$
$$2^{64} = 16 * 10^9 * 10^9$$

Conversion

$$10^3 = KB\ (kilo)$$
$$10^6 = MB\ (méga)$$
$$10^9 = GB\ (giga)$$
$$10^{12} = TB\ (téra)$$
$$10^{15} = PB\ (péta)$$
$$10^{18} = EB\ (éxa)$$

Memory hierarchy

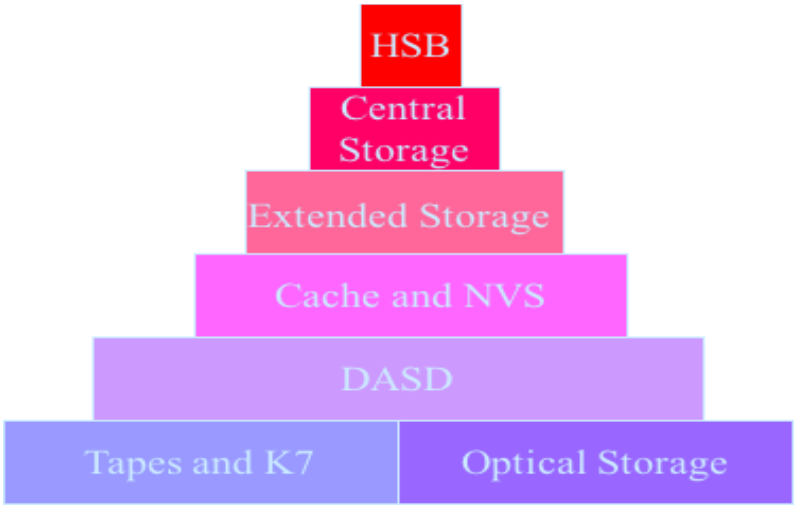
La configuration idéale serait d'avoir une seul grande mémoire sauf que cette mémoire est la mémoire cache qui coûte très cher même si elle est très rapide. Il faut alors hiérarchiser cette mémoire en ayant toujours la plus grande performance possible mais avec des coûts plus faible.



Memory characteristics

- temps d'accès
- Bande passante
- Cappacité d'extension
- Le prix
- Sa fiabilité
- Son coût environnemental

Hardware components



Data flow

But : Amener les informations le plus près possible du CPU avant qu'il n'en ait besoin. On appelle ça le "hit ratio".

Le contraire d'un hit ration est le miss ratio

Hit ratio	Miss ratio
96%	4%
98%	2%

Coût moyen de la mémoire

Memory	Size	Speed	Cost $/GB	Transfer
Registers	kB	<1ns	(incl.CPU)	Tbps
Cache L1	100 kB	<1 ns	(150,000)	Tbps
Cache L2	512 kB	~ns		100 Gbps
Cache L3	3-6 MB	10 ns	(1000)	
Main (RAM)	~10 GB	100 ns	3 – 4	200 Gbps
Online SSD	~500 GB	100 µs	0,25	1 Gbps
Onl. Hard disk	x TB	10 ms	0,06	1 - 10 Gbps
(cloud)		Sec.		
Nearline (robot CD-ROM, tapes)	PB	10sec-Min		
Offline		Min-hours		

Exemple :

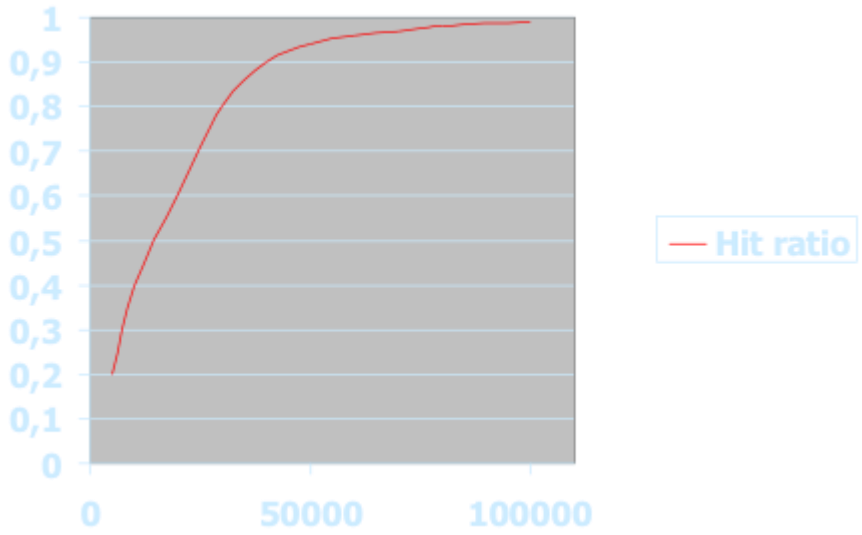
/	Memory size (GB	Coût ($)	M*C
1MB de cache	0,001	150.000	150
10 GB de RAM	10	4	40
1 TB SSD	1000	0,25	250
10 TB HDD	10.000	6	600
Total :	1.1010,001	/	1040$

Les diverses technologies à notre disposition nous permettent de mettre au point une structure hiérarchisée de différents niveaux de stockage. Ces différentes technologies nous permette d'obtenir un débit du système dans des conditions économiquement acceptables.

Hit ratio

L'éfficacité d'une cache est mesurée par le Hit Ratio qui est le pourcentage de référence satisfaites par un accès à la cache sans devoir accéder à la mémoire centrale. C'est donc la probabilité de trouver la donnée à un certain niveau de la mémoire hiérarchique.

A contrario, le Miss Ratio est le pourcentage de références insatisfaites à un certain niveau mémoire et qui ont dû être satisfaites par un accès à un des niveaux inférieurs de la mémoire.



Les différents types d'accès

Synchrone :	Le CPU attend la donnée
Asynchrone :	Le CPU continue de fonctionner mais sur une autre tâche. La tâche précédente est interrompue

Ce dernier est intéressant si l'interruption dure moins longtemps que le temps d'accès à la donnée

Exemple : Le temps d'une instruction pour un CPU de 2Ghz est de 0,5 ns

Access time

Le temps d'accès dépend de 2 facteurs :

- Le hit ratio
- L'accès ou pas à la mémoire principale

1 ns [L1]	• 96%	• 98%
5 ns [L2]	a. 4%	b. 2%

- 96/100 * 1 ns + 4/100 * 5 ns
  = 0,96 + 0,20
  = 1,16 ns

- 0,98 + 0,10
  = 1,08 ns

L'objectif est d'avoir un miss ratio de 1 ou 2% maximum

Memory usage rebilling

C'est ce que coûte de faire un virement sur PC banking

- Se fait via l'usage du CPU en fonction du temps

Locality of reference

L'objectif poursuivi est que les données doivent se trouver le moins possibles aux dernières mémoires

Exemple :

HIT = 95% **MISS = 5%**

- 95% dans L1 et 5% dans L2
- Dans ces 5% de L2 il y a 4,9%(95% hit) dans L2 et 0,1 dans L3(5% miss)
- Etc…

Exercice

Faire exercice du cours n°2-2 slide n°9 (commentaire)

HSB : Registre
DASD : Disque dur

La différence de miss ratio est divisée par 2 ce qui fait une grande différence sur la rapidité d'accès

Memory hierarchy

dimanche 18 décembre 2016
15:00

HSB / cache management

Lorsqu'on met des données dans la mémoire cache, il faut sacrifier d'autres données lorsque celle-ci est pleine.

- Comment les sacrifier, remplacer ? (Replacement policy)
- Comment gérer cette gestion ? (fetch policy)
- Comment garder ma mémoire principale et ma cache cohérente? (Write policy)

Fetch policy

- **Demand fetch**

  On attend que le CPU demande quelque chose avant de le charger dans la cache.

  Fonctionne à la vitesse des mémoires plus lentes

- **Prefetch**

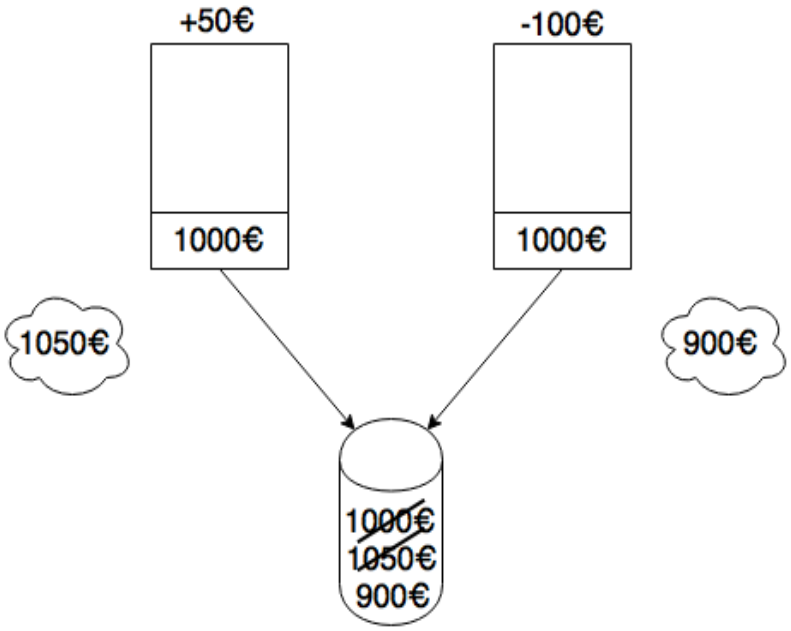
  On précharge les données dans la cache avant que celles-ci soient demandées avec "Promote" (i+1) et "demote" (i-1).

3. **Selective fetch**

   Ne choisi pas les données en fonction d'un critère défini (mémoire partagée sur un multi-processeur).

   Problème avec les multi-processeur

Exemple : compte en banque

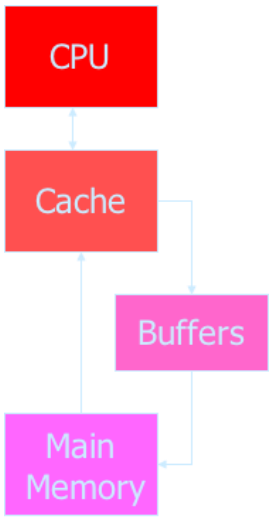


Write policy

1. Write-tru : Chaque opération sur la cache est répétée en même temps sur la mémoire principale.

Il existe des buffers pour pouvoir transférer des données en dehors de la cache. La mémoire est alors libérée. Il y a un pointeur pour l'écriture de la mémoire cache et un autre pointeur pour la lecture.

Exemple : Lors de la gravure d'un disque des buffers sont utilisés pour libérer la cache car c'est un périphérique lent.



2. Write-back : L'opération à la mémoire principale n'est effectuée qu'au moment où le bloc de données est remplacé.

Write-back

Chaque bloc de cache modifié est réécrit au moment de la perte

Ces deux modifications améliorent les performances de l'accès mémoire

Write policy with multi-processor

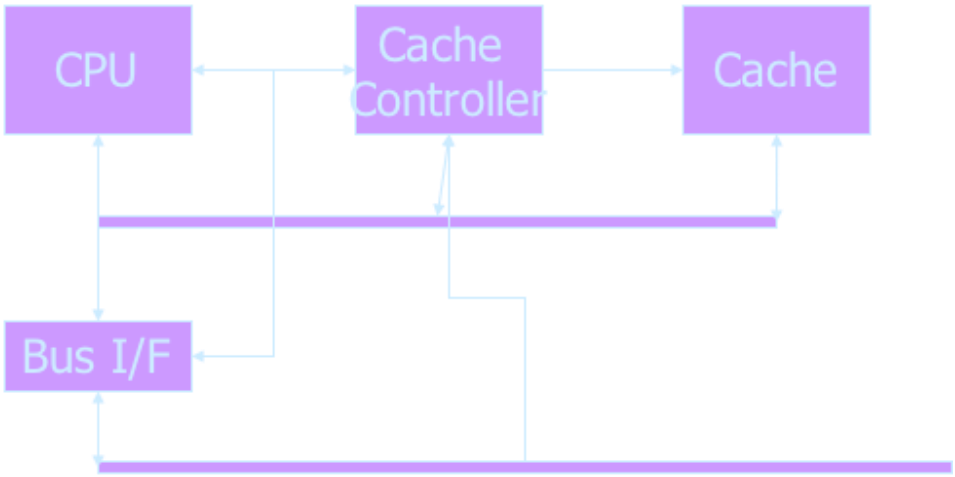
Lorsque deux processeurs lisent la même donnée et que le 1er la modifie et la met dans sa cache, il ne faut pas que le 2e processeur lise l'ancienne donnée restée dans la mémoire principale. Il faut alors une technique qui assure une certaine cohérence.

Solutions :

- Shared cache (cache partagée)
- Non-cacheable items (Eléments impossible de mettre dans la cache)
- Snoop bus mechanism
- Broadcast write mechanism
- Directory methods

Snoop bus

Les caches sont reliées par des bus ce qu'ils leur permet de communiquer.



Ces caches sont également dotées de 3 bits d'états qui permette d'avoir des informations sur les données traitées :

1	2	3
Valid / Invalid	Clean / dirty	Share / exlusive

C'est le plus courremment utilisé

Broadcast write

Chaque écriture est envoyée à toutes les caches

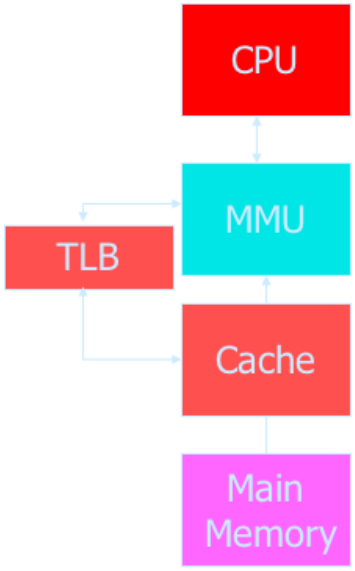
Processus assez lourd

Directory

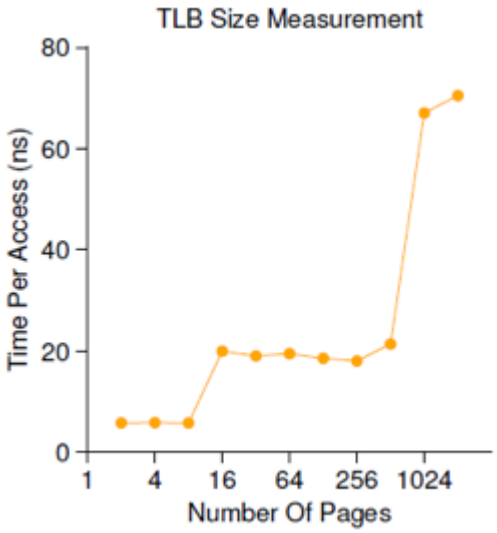
Mise en place d'un tableau de bits où chaque colonne correspond à un CPU plus une colonne de bit "modified". Ces bits d'état permettent de savoir si la donnée a été modifiée, si elle est présente dans une cache (dans ce cas il ne sera que dans cette cache durant le traitement) et si elle contient une copie valide.

CPU0	CPU1	CPU2	CPU3	Modified
1	0	0	0	1

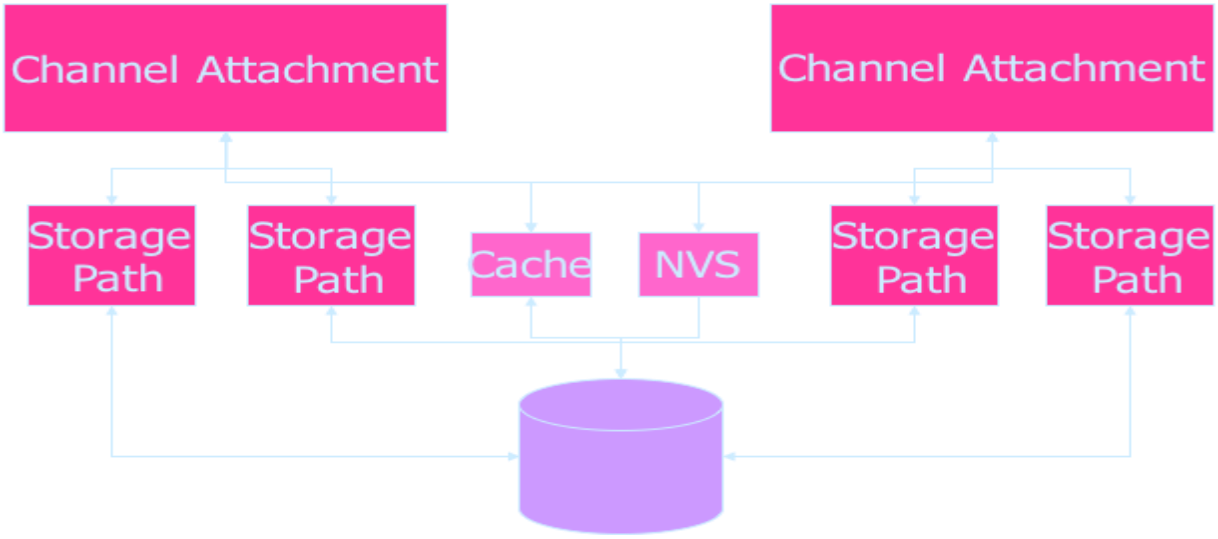
Translation look-aside buffer (TLB)



TLB sizes ans miss costs



Internal design RASD control



Cache residency

C'est le temps qu'une donnée reste dans la cache.

Pour connaitre la taille de la cache :

$$size = residency*(1-hit\ ratio)*I/O$$

Pour connaitre le temps qu'une donnée passe dans une cache :

$$residency = \frac{size}{1 - hit\ ratio} * I/O$$

R = 70 I/O par seconde
S = 336 emplacements
H = 85%

Temps de résidence = 336 / (1 - 0,85) * 70
= 32 secondes

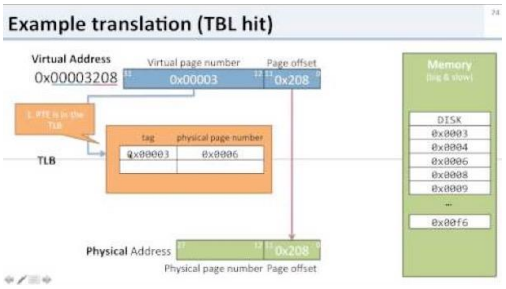
CPU 2
CPU 1

Solution : Mémoire cache commune ou fetch sélectif?

C'est la traduction d'adresse virtuelle en adresse physique

DAT :	Directly Adress Translation

Youtube : Virtual Memory: 11 TLB Example



Channel attachment : Bus d'accès rapide

S	Size
T	Cache residency time
H	Hit ratio
R	Number of I/O per unit

Les communications inter-processus

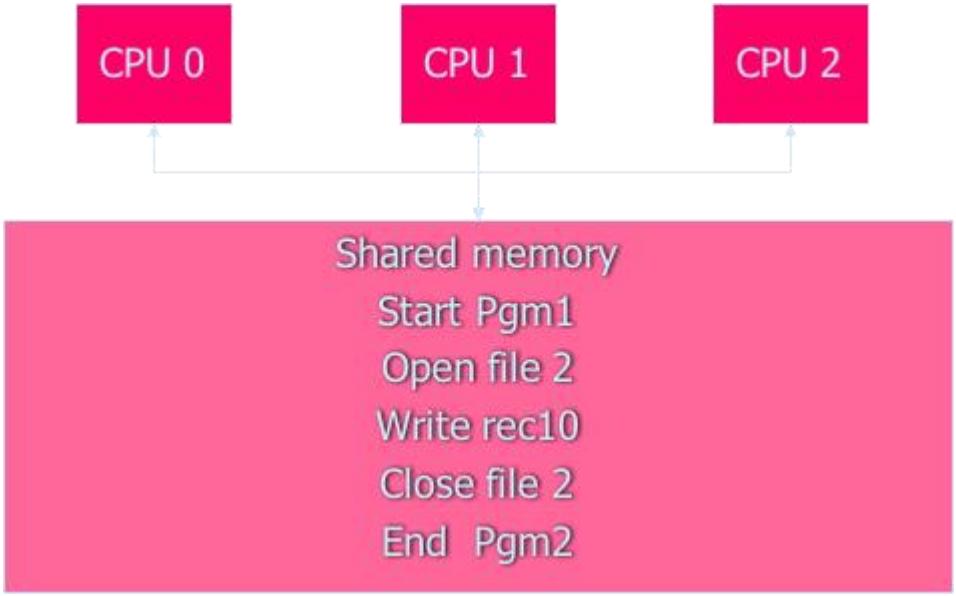
dimanche 18 décembre 2016
18:00

Problème : La loi de Moore est arrivé à ses limites. Comme le nombre de transistors ne sait plus augmenter on multiplie le nombre de processeurs.

Lorsqu'on a un processus qui tourne sur plusieurs processeurs cela peut provoquer des problèmes de données. Il faut absolument éviter toute erreur mais aussi tout blocage.

Parallelism

C'est le fait de distribuer ces tâches aux différents processeurs. Il faut alors gérer la cohérence et la gestion des interruptions.



Possibilité de les distribuer sur plusieurs processeurs?
Possibilité de lancer plusieurs fois le (même) programme en parallèle?

Flynn classification

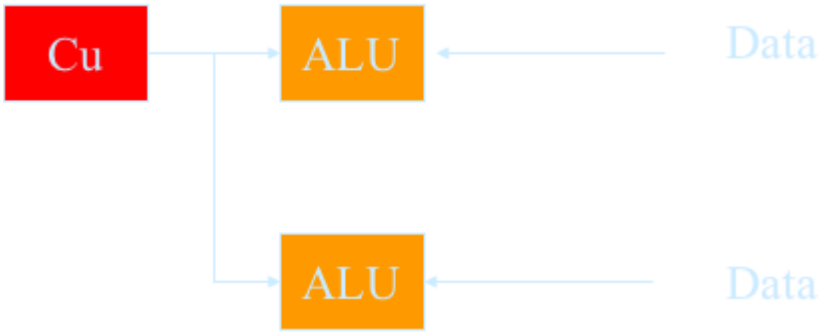
SISD

Une instruction traite une donnée (PC monoprocesseur). Il est composé d'une Unité de contrôle (CU) et d'une Unité Arithmétique et Logique (ALU).

Exemple : C'est ici l'architecture de Von Neumann donc aucun parralélisme possible.

SIMD

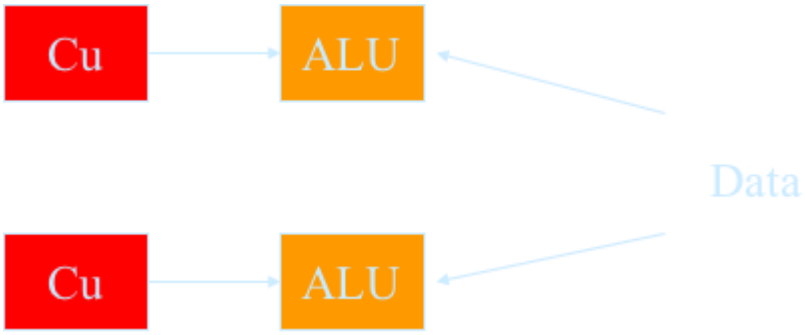
Une instruction traite plusieurs données (un tableau de données). Il est composé d'une Unité de contrôle (CU) et de plusieurs Unités Arithmétique et Logique (ALU). C'est souvent le cas des processeurs graphiques.



Exemple : Souvent utilisé pour les traitements qui ont des structures très régulière comme le calcul matriciel ou les calculs scientifiques.

MISD

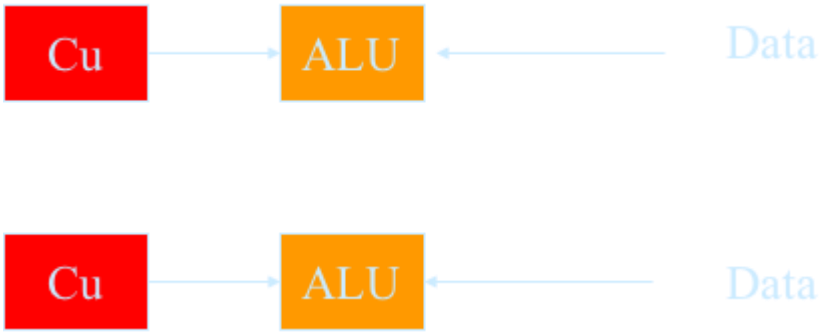
Plusieurs instructions traitent une donnée. Ces processeurs doivent ensuite contrôler le résultat et doivent également s'aligner. Cette architecture se fait principalement pour les calculs de précision.



Exemple : Ce système est utilisé pour le controle de vol des fusées, avions et prévisions météo. Les données sont envoyées par exemple dans 3 processeurs et la majorité (2/3 dans ce cas-ci) des réponses données gagnent. Cela permet de diminuer les erreurs de calcul.

MIMD

Plusieurs instructions traitent plusieurs données. Il s'agit ici de multi-processeurs indépendants. Cette architecture est la plus courante actuellement.



Exemple : Nos pc de tous les jours

Pour assurer la cohérence des données, il faut mettre en place des techniques de synchronisation qui dépendent de l'organisation de la mémoire.

On distingue 2 types d'architectures :

1. **MIMD à mémoire partagée**

   Les processeurs accèdent à une mémoire commune. Cette méthode est très largement utilisée. La synchronisation peut se faire au moyen de :

   - Sémaphore
   - Mutex
   - Barrière de synchronisation

   Expliqué plus tard dans le cours

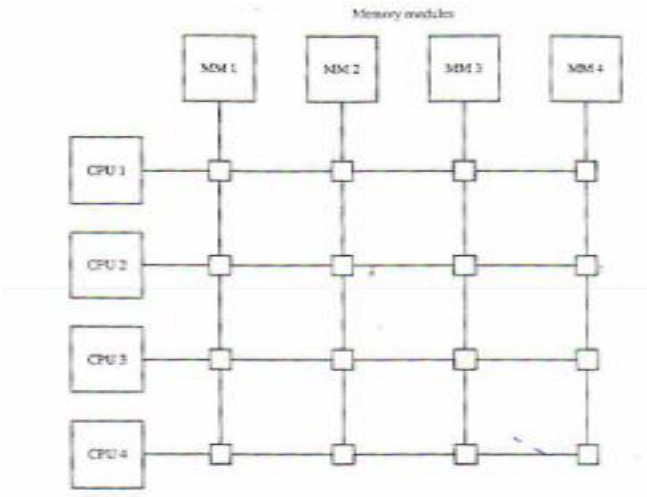
2. **MIMD à mémoire distribuée (message-passing)**

   Chaque processeur dispose de sa propre mémoire et n'a pas accès à celle d'autres processeurs. Les informations sont échangées entre les processeurs sous forme de message. Les processeurs sont donc reliés entre eux mais juste à leurs processeurs voisins en raison du coût de ces connexion. Les messages peuvent donc passer par plusieurs processeurs avant d'arriver à leur destination.

La mémoire partagée

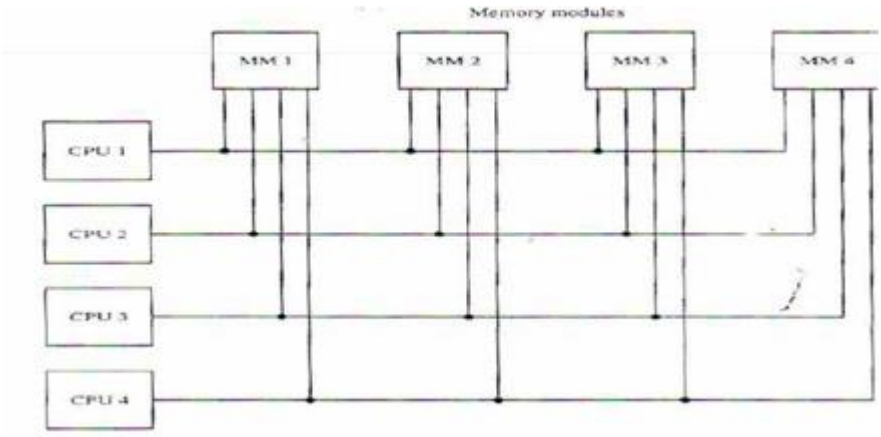
Crossbar switch

C'est un système matriciel qui a de multiple lignes d'entrées et de sorties. Une connexion peut être établie en fermant un commutateur situé à chaque intersection.
Cette configuration est utilisée lorsqu'il y a beaucoup de CPU avec beaucoup de mémoire. Elle supporte des transferts simultanés sur tout les modules mémoire.



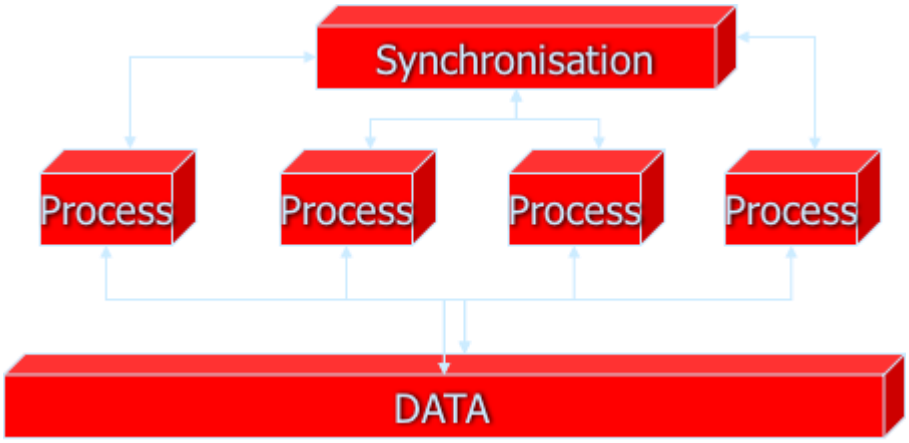
Multiport memory

Système constitué de plein de points de connexion. Il permet un taux de transfert très élevé mais génère des conflits en mémoire. Cette configuration est plus élaborée que la précédente.



Shared memory progamming model

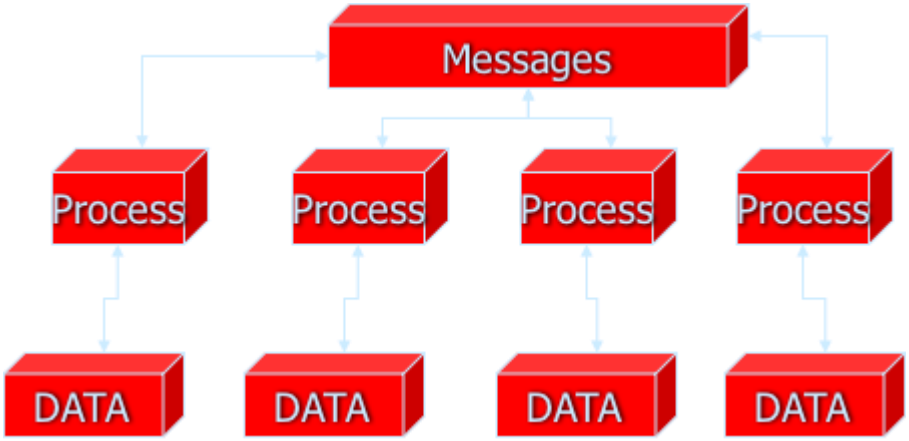
Ci-dessous, le modèle de base pour l'architecture MIMD à mémoire partagée.



Ce modèle reste fondamentalement le même. Différents processus accèdent aux données partagées en mémoire (à partir d'un certain niveau). Il faut une synchronisation pour régir cet accès. La très grande majorité des ordinateurs actuels utilisent ce modèle.

La mémoire distribuée (message-passing)

Cette configuration peut donner des schémas d'interconnexion compliqué. Voici le modèle de base de la mémoire distribuée :



Speedup factor

C'est la différence de temps d'une tâche entre un monoprocesseur et un multiprocesseur. Un programme n'est d'ailleurs pas forcément utilisé sur tout les CPU, cela dépend de l'implémentation de celui-ci. Sur Windows, on utilise le Ressource Monitor pour voir s'il utilise un ou plusieurs CPU.

Equation :

$$S(n) = \frac{Temps\ uniprocesseur}{Temps\ \ multiprocesseur}$$

Efficienty

A partir du speedup factor, on peut dériver un facteur d'éfficacité (en pourcentage).

$$E(n) = \frac{S(n)}{n} * 100\%$$

Exemple :

1. cas du flight simulator (S(4)=1) ➔ Efficiency = 25%
1. Video Converter (S(4) = ~4) ➔Efficiency ~100%

Parallel computation and serial section

C'est le fait de couper la tâche sur plusieurs processeurs.

/	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4
Open file	100 ko	100 ko	100 ko	100 ko	100 ko

T serial ➔ Data base : 500 ko

T parallel ➔ S(5) = 3 pour 5 CPU

$$= \frac{3}{5} * 100 = 60\% \text{ (speedup factor)}$$

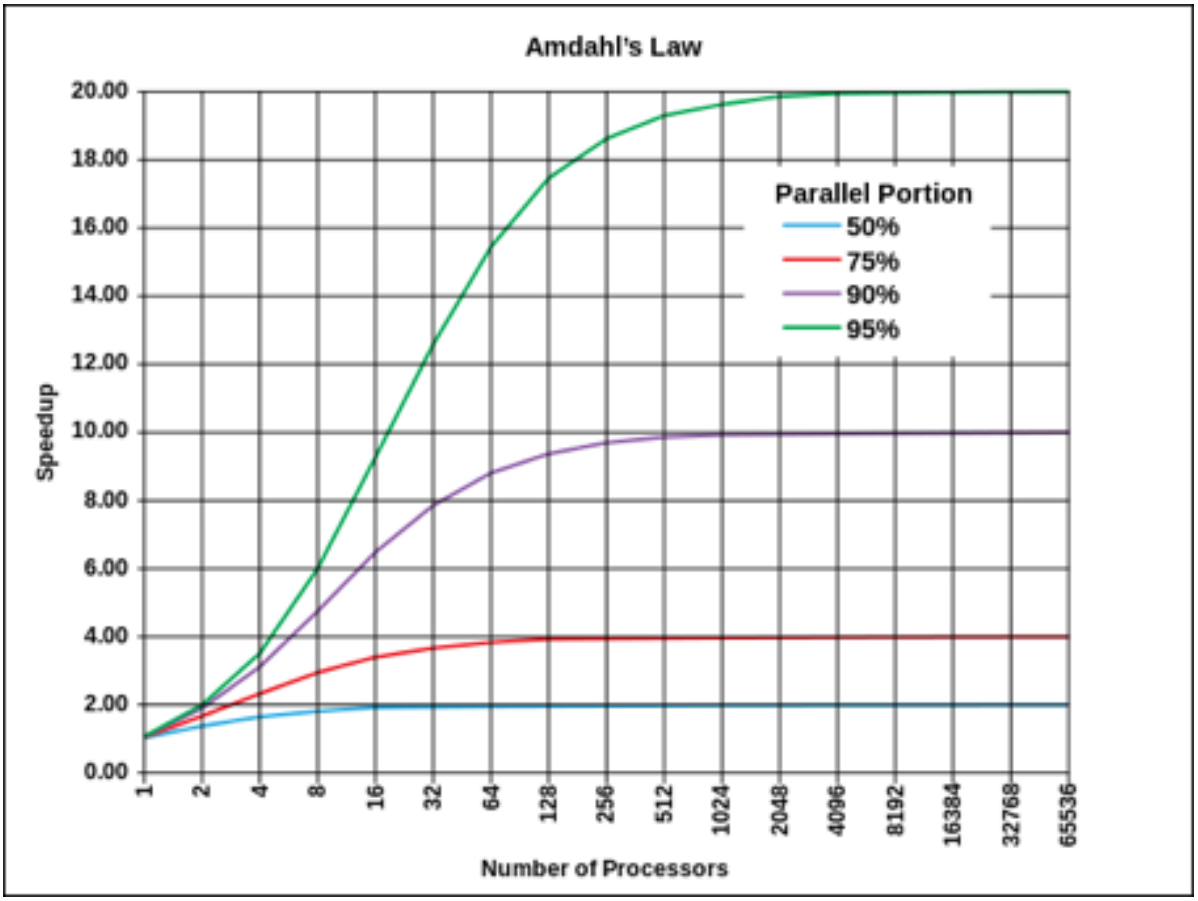
Amdhal's law

Certaines tâches peuvent être parallélisable (T parallel) tandis que d'autres non (T serial).

Equation :

$$Speedup = \frac{T\ serial + T\ parallel}{T\ serial + \left(T\frac{parallel}{N}\right)}$$

Schéma :



- Si 50% du travail est parallélisable, cela ne sert à rien d'avoir plus de 8 processeurs
- Si 90% du travail est parallélisable, cela ne sert a rien d'avoir plus de 512 processeurs
  Etc…

Qu'est ce qui peut empêcher le parallélisme entre processus?

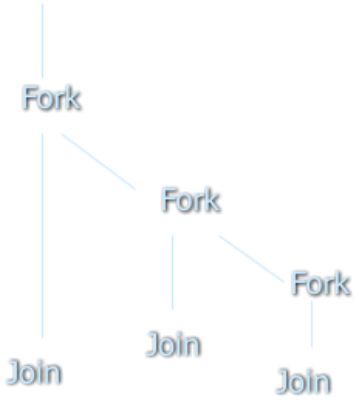
Exemple : un conflit d'accès au ressource

Parallelism

Un processus est réalisé par un processeur qui est défini par l'ensemble de ses entrées et sorties. Ce processus peut être parallélisé de 2 façons :

- **Parallélisme explicite**

  Ce parallélisme est fait directement par le programmeur et a la structure suivante :



- **Parallélisme implicite**

  Les compilateurs vérifient ici si les conditions sont remplies avant de générer du code parallèle(bernstein's condition). C'est en autre pour cela qu'il faut déclarer ses variables, fichiers, etc…

Bernstein's condition

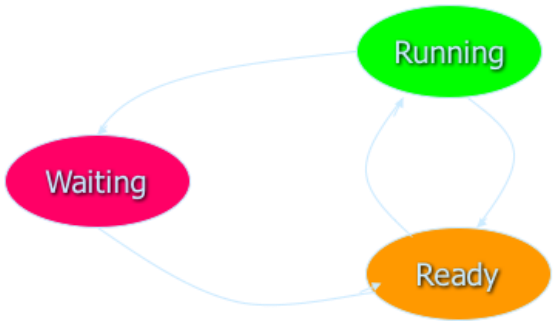
Il ne faut pas de dépendances entre les sorties sinon le système n'est plus déterministe. Un processus ne doit pas dépendre de la sortie du processus précédent pour continuer sa tâche (sinon pas de parallélisme).

Pseudo-parallelism

Cela se fait via les états des processus même sur monoprocesseurs (cf. OS 1).

Process status

Total Elapsed Time = T_{cpu} + T_{ready} + T_{wait}



Critical section

C'est un mécanisme d'utilisation unique d'une ressource. Il est donc fondamental dans un système multi-processeurs de :

- **Synchroniser**
  - Communication entre les processus
  - Gérer les changements d'états
- **Garantir le déterminisme, la cohérence**

  Il faut coordonner les changements de valeurs. Pour cela, il faut qu'un processus à la fois puisse modifier une ressource. Il ne doit pas être interrompu n'importe quand ni n'importe comment.

Non-déterministe : Le fait de ne pas avoir la garantie que plusieurs exécutions d'un même programme va produire le même résultat du à l'absence de section critique.

Race condition

Si deux processus parallèle utilisent la même ressource, c'est le dernier a avoir modifié la valeur de la ressource qui gagne.

Critical section rules

1.Mutual exclusion

Seulement un processus à la fois doit avoir accès à une section critique car si deux processus accèdent en même temps à une ressource, le résultat ne sera pas déterministe.

Exemple :	Si deux processus accèdent à l'imprimante, les résultats vont se superposer.

2.Progress

Supposons qu'une section critique soit libre. Si un processus indique qu'il a besoin d'entrer dans une section critique, il ne doit pas attendre.

3.Bounded wait

Il faut faire en sorte qu'aucun processus n'attende éternellement pour entrer dans sa section critique.

4.Number independent (optimisation)

La mise en oeuvre des sections critiques ne doit pas interférer sur les performance du processeur.

Implémentation des sections critiques

6 solutions ont été implémentée au fil des années :

Interrupt disabling

Une ressource ne peut être utilisée par interruption qu'une fois en même temps

Pertubation I/O

Exemple :

```
dcl shared Ice_cream;                /* Shared variables*/

Program for process Pa               Program for process Pb

disable Interrupts ();               disable Interrupts ();
    eat(Ice_cream);                      eat(Ice_cream);
enable Interrupts;                   enable Interrupts;
```

Shared variable testing

Une variable (boolean) permet de savoir si on peut accéder à la ressource

Race condition

Si le processus se fait interrompre par une quelconque raison avant qu'il ne change la variable partagée, un 2e processus peut y accéder. Les deux processus travaillent à un moment donné sur la même ressource et le système devient non déterministe.

Exemple :

```
dcl shared lock=FALSE;          /*Shared variables*/
dcl shared Ice_cream;

while (lock) {NULL};            while (lock) {NULL};
    lock = TRUE;                    lock = TRUE;
        eat(Ice_cream);                 eat(Ice_cream);
    lock=FALSE;                     lock=FALSE;
end while;                      end while;
```

3. Strict alternation

Implémentation d'une 2e variable "Turn" (boolean). Le 1er processus a la main lorsque cette variable est à 1 et le 2e processus est en attente car sa variable est à 0.

1. Si la vitesse des processus sont différentes, les performances sont fortement amoindrie.
2. Un processus peut garder la ressource indéfiniment

Exemple :

```
dcl shared turn;                /* Shared variables */
dcl shared Ice_cream;

while (turn != 0) {NULL};       while (turn != 1) {NULL};
    eat(Ice_cream);                 eat(Ice_cream);
    turn=1;                         turn=0;
    wash_up;                        wash_up;
```

4. Peterson's solution

Mise en place d'un flag ("Je suis interessé"). Tant qu'un autre processus n'est pas interessé, on ne lui donne pas la main. La variable turn est changée pour que le même processus garde la main.

Exemple :

```
flag[0]   = 0;
flag[1]   = 0;
turn;

P0: flag[0] = 1;
    turn = 1;
    while (flag[1] == 1 && turn == 1)
    {
        // busy wait
    }
    // critical section
      ...
    // end of critical section
    flag[0] = 0;
```

```
P1: flag[1] = 1;
    turn = 0;
    while (flag[0] == 1 && turn == 0)
    {
        // busy wait
    }
    // critical section
      ...
    // end of critical section
    flag[1] = 0;
```

Les turn sont inversé ! (voir synthèse prof)

TS/CS (amélioration)

Test and set : Faire en sorte qu'une opération soit indivisible tel que même si le processus est interrompu, le système ne se bloque pas

Il peut toujours y avoir le problème qu'un processus attend indéfiniment

Exemple :

```
dcl shared lock=0;

P0:
    while (test_and_set(lock) == 1)
    {
        // busy wait
    }
    // critical section
      ...
    // end of critical section
    lock = 0;
```

```
P1:
while (test_and_set(lock) == 1)
{
    // busy wait
}
// critical section
  ...
// end of critical section
lock = 0;
```

5. Sleep and wake-up

Mise en place de deux primitives : sleep & wake_up. Elles permettent d'éviter les attentes infinies.

Producer and consumer

Collaboration de deux processus un producteur et un consommateur : l'un rempli un buffer tandis que l'autre le vide (via le mécanisme de sleep & wake up).

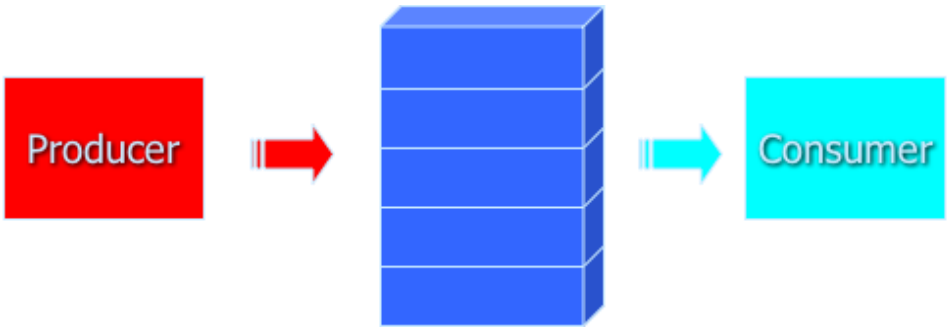
Producteur : Rempli un buffer d'informations

Consumer : Copie les informations en dehors d'un buffer

Il y a un nombre fini de place dans un buffer.

1. Un processus actif doit réveiller le processus dormant.
2. L'instruction pour endormir un processus doit se faire en une seule fois

Schéma :



Exemple :

```
Dcl N=100; dcl count = 0;          /* nbr of slots in buffer pool;nbr items*/

void producer; dcl ice_cream;
    while(TRUE)
    {       produce(&ice_cream);
            if (count == N) sleep();
               /* if buffer pool full, wait  */
            store(&ice_cream);
               /*put item in one buffer*/
            count = count + 1;
            if (count == 1) wakeup(consumer);
    }
```

```
Void consumer;
    while (TRUE)
    {       if (count == 0 ) sleep();
               /* buffer pool is empty*/
            remove(&ice_cream);
            count= count -1;
            if (count == N-1)
                wakeup(producer);
                      /* buffer was full */
            consume(ice_cream);
    }
     /* miam-miam */
```

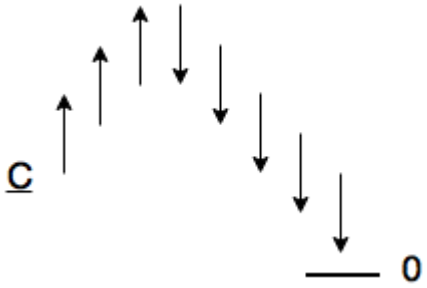
6. semaphores

Les sémaphores permettent de tester et modifier une variable par 3 opérations indivisibles. Elles peuvent être comparées à des feux rouges. Les 3 opérations opérations prises en charge sont : **Init**, **P** et **V** et doivent être implémentées au niveau matériel (microprocesseur) afin d'exister au niveau logiciel. P et V viennent du néerlandais Proberen et Verhogen qui signifie "tester" et "incrémenter". Ils sont souvent aussi appelé Up et Down en Anglais.

P :	Met en attente le processus courant jusqu'à ce qu'une ressource soit disponible.
V:	Elle rend une ressource disponible après qu'un processus ait terminé de l'utiliser.

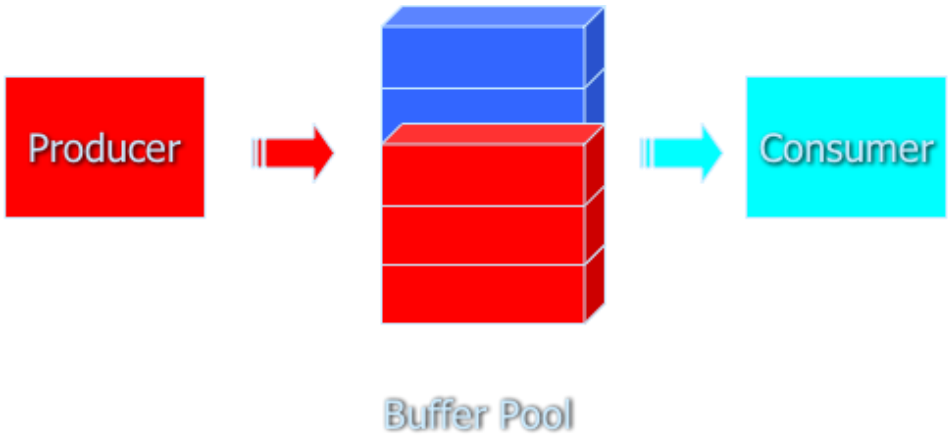
Init :	Elle permet de réinitialiser la sémaphore et ne peut être utilisée qu'une seule fois.

Ces 3 opérations sont indivisible ce qui signifie qu'un processus qui désire exécuter une opération qui est déjà en cours d'exécution par un autre processus doit attendre que le premier termine.



Ces sémaphores se font via des valeurs Integer. Elles servent à améliorer les primitives Wake_up et Sleep. Elles sont implémentées via une file d'attente. Wake_up enlève de la file d'attente et sleep la rajoute dans la file d'attente.

Producer and consumer



Mise en pratique dans l'exemple de consommateur et du producteur

1. Exclusion mutuelle pour l'accès au buffer

   Besoin d'un Mutex pour limité l'accès. (valeur initiale = 1)

2. Le producteur arrête de "produire" lorsque le buffer est plein

   Mise en place d'une sémaphore #free quand le producteur est dans l'état sleep

3. Le consommateur arrête de "consommer" lorsque le buffer est vide

   Mise en place d'une sémaphore #full quand le consommateur dors

Ces deux dernières valeurs doivent être rajouté car les deux processus sont asynchrone. Ils remplissent et vident le buffer à leur vitesse respectives.

Exemple (aller slide 23) :

<<S.E.2-5.pptx>>
On a un comptoir on doit produire et vendre (manger) de la glace

Producteur

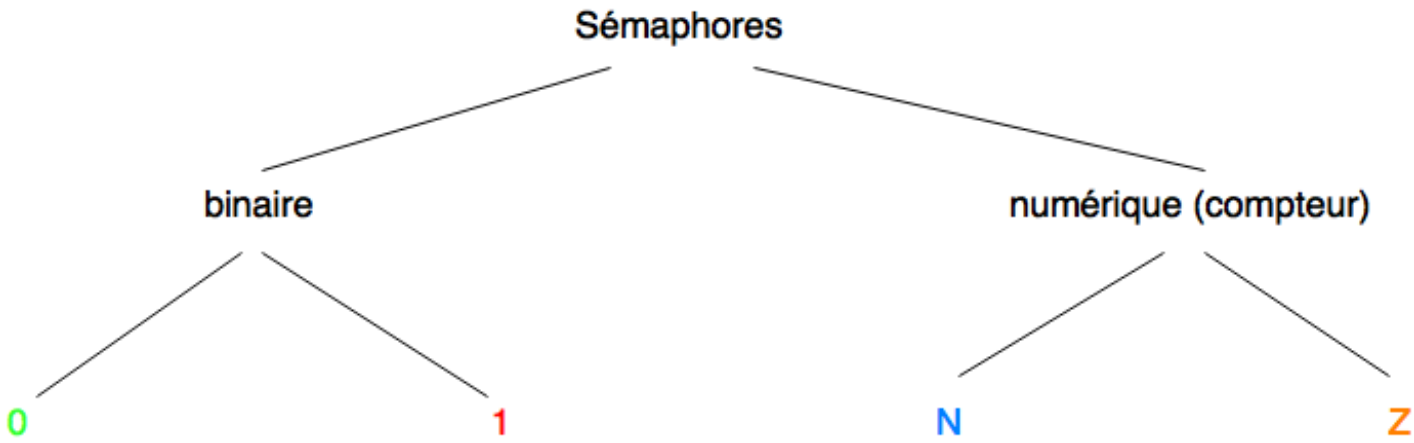
down (#free)	1. Si #free == 0 donc comptoir plein ===> sleep 2. Si #free >=1 je peux rajouter une glace (incrémenter)	
down (mutex)	Réserve l'accès au comptoir (ressource)	Section critique
produce(&ice_cream)		Section critique
up (mutex)	Libérer le comptoir (ressource)	Section critique
up(#full)	On indique qu'on a rajouté une glace au comptoir (buffer) = incrémentation	

Consommateur

down(#full)	1. Si #full == 0 donc rien sur le comptoir (buffer) ===> sleep 2. Si #full == 1 donc comptoir rempli (buffer), je peux enlever une glace (décrémenter)	
down(mutex)	Réserve l'accès au comptoir (ressource)	Section critique
eat(&ice_cream)		Section critique
up(mutex)	Libérer le comptoir (ressource)	Section critique
up(#free)	On indique qu'on a retiré un glace au comptoir (buffer) = décrémentation	

Ordre des opérations importants sinon on a des blocages

Les différents types de sémaphores



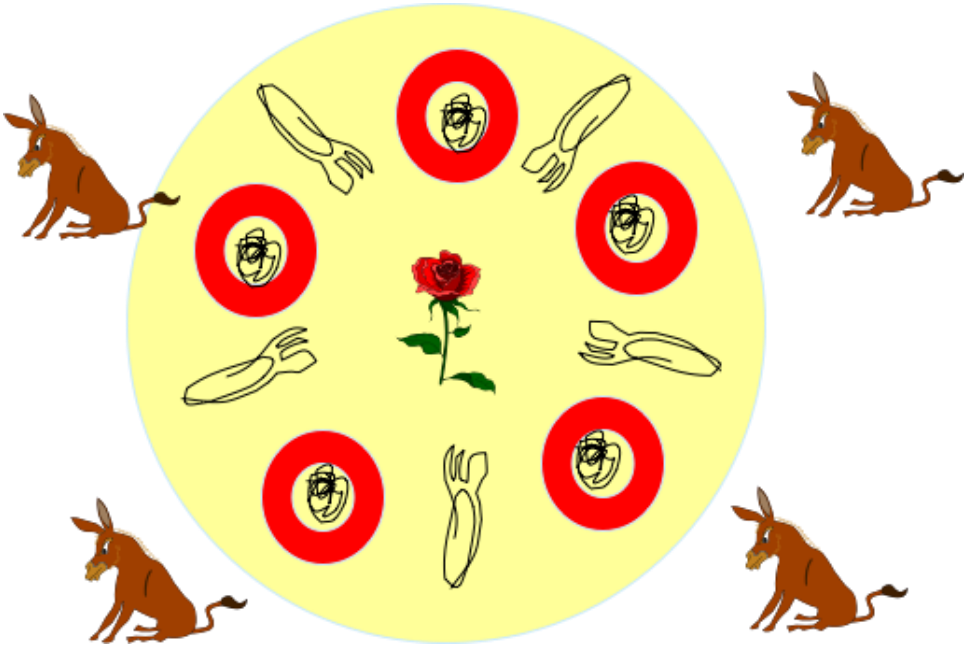
Synchronisation : Un processus A a besoin d'un processus B avant de commencer

Mutex : Utilisé quand on veut protéger une ressource (section critique)

Buffer : Ajoute et supprime dans le buffer mais ne retiens pas le nombre de processus en attente

Buffer : Ajoute et supprime dans le buffer et retiens le nombre de processus en attente

Philosophers



La situation est la suivante :

- cinq philosophes se trouvent autour d'une table
- chacun des philosophes a devant lui un plat
- à gauche de chaque plat se trouve une fourchette

Un philosophe n'a que trois états possibles :

- penser (pendant un temps indéterminé) ;
- être affamé (pendant un temps déterminé)
- manger pendant un temps déterminé et fini.

Problème :

- Un philosophe a besoin de 2 fourchettes pour manger (a gauche et à droite de son assiette)

Si chaque philosophe prend la fourchette de gauche il seront bloqué car aucune fourchette de droite ne sera disponible. Il faut donc trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

Illustration :

```
Base algorithm:

void philo(int i){
  while(TRUE){
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}
```

```
Void take_forks(int i) {
    ask_take_fork(i);
    ask_take_fork(i+1);
)

void put_forks (int i) {
    put_fork(i);
    put_fork(i+1);
}
```

Solution :

Le 5e philosophe prend la fourchette de droite et peut donc manger. Quand il a fini, il libère ses fourchettes ce qui permet au 4e philosophe de manger. Il libère ses fourchettes à son tour qquand il a fini et le 3e peut manger, …

Cet exemple illustre le fonctionnement des sémaphores

Illustration :

```
#define N                    5
#define LEFT       (i-1)%N
#define RIGHT     (i+1)%N
#define THINKING           0
#define HUNGRY 1
#define EATING   2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philo(int I){while(TRUE){
    think();
    take_forks(I);
    eat();
    put_forks(I); } }
```

```
Void take_forks(int I) {
    down(&mutex);
       state[I]= HUNGRY; test(I);
    up(&mutex);
    down(&s[I]); }

void put_forks (int I) {
    down(&mutex);
          state[I]= THINKING ;
          test(LEFT);test(RIGHT);
    up(&mutex);}

void(test int I) {
if (state[I] == HUNGRY &&
    state[LEFT] != EATING &&
    state[RIGHT] != EATING) {
        state[I] =EATING; up(&s[I]);
          } }
```

Sleeping barber

Un coiffeur a une salle d'attente pour ses clients mais en l'absence de ceux-ci il s'est endormi (sleep). Il arrive un moment où un client arrive car on ne peut pas l'empêcher de venir. Le but de cet exemple est qu'il faut réveiller le coiffeur pour que le client puisse rentrer.

Solution :

- Ajout d'une variable pour voir si il y a un client
- Il y a donc maintenant 2 sections critiques

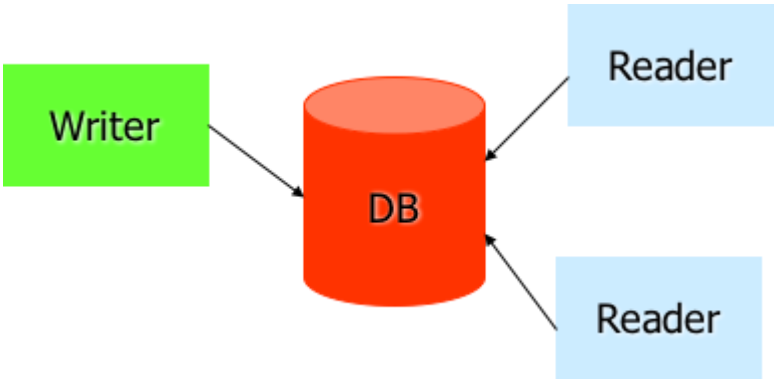
Illustration :

```
#define CHAIRS 5
typedef int semaphore;
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1 ;
int waiting = 0;

void Barber(void){
  while(TRUE){
     down(&customers);
     down(&mutex);
     waiting = waiting-1
     up(&barbers);
     up(&mutex);
     cut_hair();}}
```

```
Void Customer(void) {
    down(&mutex);
    if (waiting < CHAIRS) {
          waiting = waiting +1;
          up(&customers);
          up(&mutex);
          down(&barbers);
          get_haircut();
    } else {
          up(&mutex); } }
```

Readers and writers



Lorsqu'on lit (read) une ressource, on ne peut pas écrire (write) sur cette même ressource au même moment (et inversément). Dans cet exemple, on peut avoir plusieurs accès en lecture mais qu'un seul accès en écriture. On utilise alors le mécanisme de sleep and wake_up.

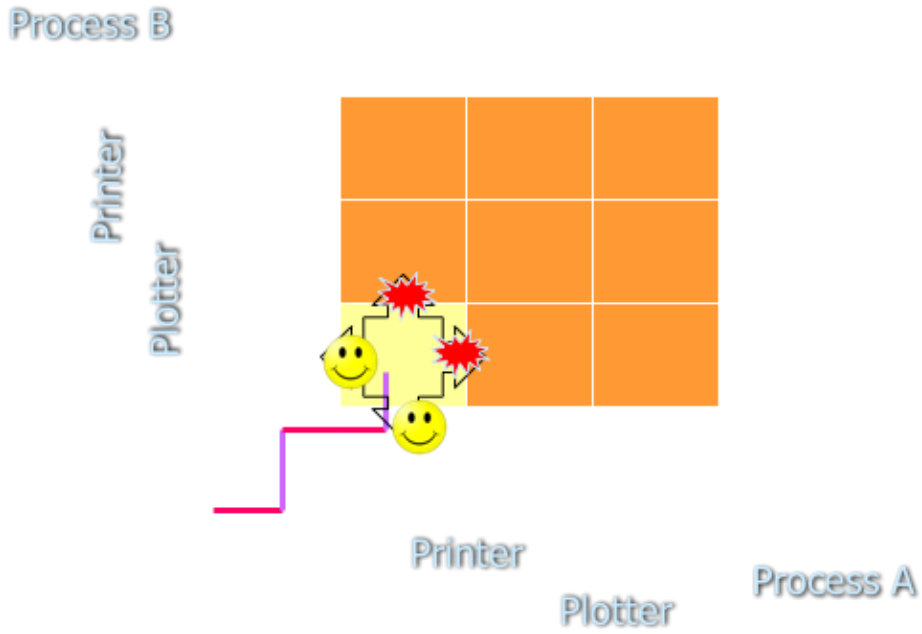
Si il y a plusieurs ressource, il est possible d'avoir des deadlock

Deadlock

C'est quand un ensemble de processus sont bloqués et attendent. Il faut alors un évènement extérieur pour pouvoir les débloquer :

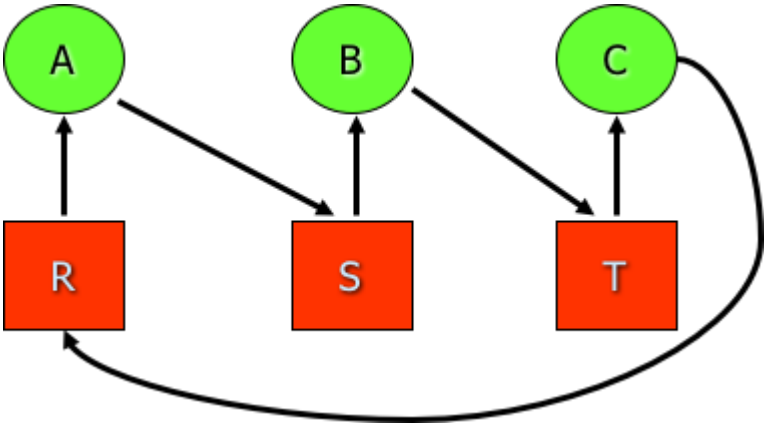
- **Le ressource manager :** Gère la gestion des graphes et des états
- **Un opérateur humain :** Avec le célèbre CTRL + ALT + DEL qui permet de trouver LE processus à tuer
- **L'OS :** Va gérer lui-même le deadlock

Safe and unsafe state



C'est lorsqu'un processus se trouve dans un état qui peut entraîner un deadlock (ici représenté par les carrés oranges).

Deadlock representation



Pour éviter qu'un deadlock n'arrive, il faut détecter s'il y a une boucle quelque part (voir schéma ci-dessus).

Les 4 conditions d'un deadlock

1. Exclusion mutuelle : Un processus utilise exclusivement une ressource
2. Hold & wait : Un processus garde une ressource et demande une nouvelle
3. Attente circulaire : Voir schéma ci-dessus

4. Pas de préemption : La ressource est libérée par une action explicite d'un processus

Les 4 conditions doivent être remplies pour qu'un deadlock se forme

Les 4 stratégies pour sortir d'un deadlock

1. **Ignorance** : Il faut alors une intervention manuelle
2. **Prevention** : En évitant une des 4 conditions :

   Exclusion mutuelle :

   - *Implémenter des fichier read-only qui n'entraîne plus de deadlock*
   - *Certaines ressources requiert un accès par un seul processus comme les imprimantes*

   Hold & wait :

   - *Allouer toutes les ressources au début*
   - *Libérer toutes les ressources si une nouvelle demande l'accès*

   Circulair wait

   - *L'allocateur de ressources empêche la création de boucle d'attente*
   - *Un ordre de priorité pour les allocations est la meilleure solution*

   Préemption

   - *Si la ressource n'est pas disponible pour qu'un processus se poursuive, l'allocation des ressources est supprimée*

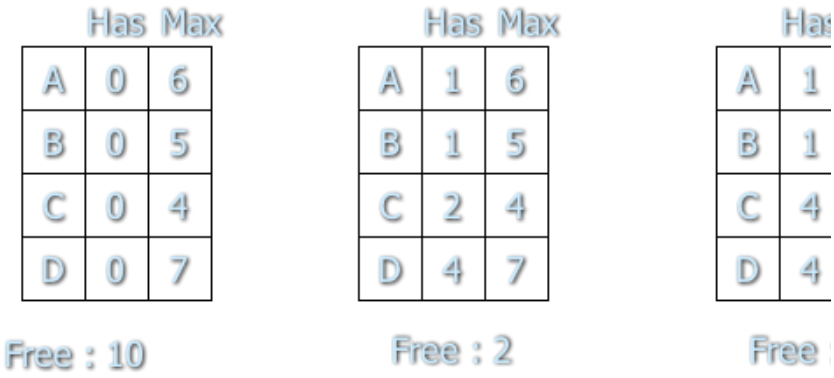
   - *Un processus peut ne pas progresser (starvation)*
   - *Le gestionnaire de ressource est constamment en attente*

   Avoidance : Avec une stratégie d'allocation des ressources traditionnel

   Banker's algorithm :

   - *Le but est de donner les ressources au processus à qui il reste le moins de ressources à obtenir afin qu'il puisse se terminer.*

     Has : nombre de ressources allouées
     Max : nombre de ressources max susceptible



- On donne les ressources à C pour qu'il puisse terminer

Detection & recovery :

Détection des conditions de deadlock et récupération

- Le ressource manager scanne le graphe par intervalle régulier pour détecter des conditions circulaires. Il commence d'un point quelconque et suit tous les points. Si il y a une entrée en double il y a un deadlock.
  Il va alors choisir une "victime" et tuer le processus ou enlever tous les changements.

Transaction atomicity

Chaque changements faites par une opérations est confirmé par des check-point. Si il n'y a pas de check-point, cela signifie qu'il n'y a pas de changement.
Dans le cas d'un deadlock, Le ressource manager tue l'opération est revient sur un check-point précédent.

Back-out / rollback

Chaque opération dans une base de données est enregistrée dans un "logging". De ce logging il y a moyen de :

- Faire un REDO c'est-à-dire une image disque "après", faire un replay
- Faire un UNDO c'est-à-dire une image disque avant le changement

Problems whith semaphores

Cette implémentation n'est pas facile à utiliser. En effet il est facile de mélanger le "Up" et le "Down", d'inverser les appels ni les manipulations des erreurs (cf. Barbier).
Hoare & Hansen inventent alors les "Moniteurs".

Monitors

C'est une classe thread-safe qui permet de faire de l'exclusion mutuelle pour plus d'un thread. Elle permet également d'aider le programmeur à résoudre certains problèmes comme les lock (mutex).
Chaque processus appele une interface qui manipule des données abstraites. Les moniteurs intègre des sections critiques à l'intérieur des données abstraites.

La donnée ici est la même pour les 2 ALU

Compliqué à implémenter

↳

Propiété : On ne peut pas faire plus de Down (V) qu'il n'y a de Up (P)

==> Ici on commence à un nombre C et on fait 5 down avant d'arriver à 0

Up = Libérer
Down = vérifier le compteur

Sens des fléches important

Memory management

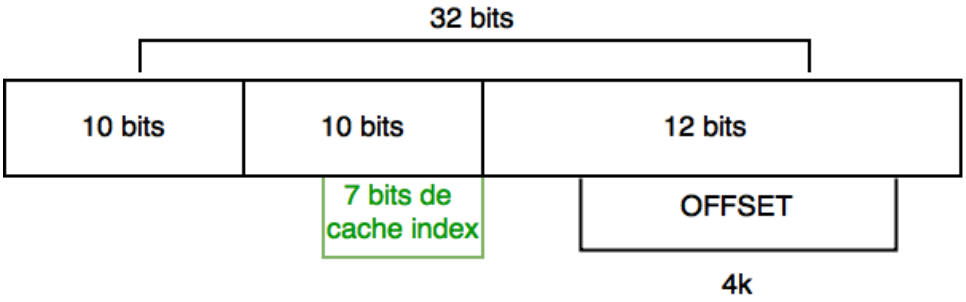
mercredi 21 décembre 2016
14:00

L'avènement des systèmes d'exploitation 64 bits, les processus savent adresser jusqu'à 2^{64} adresses mémoire (ce qui fait des exabytes). La plupart des ordinateurs actuels ne dépassent pas les 4 GB (2^{32}) de mémoire RAM.

Comment gérer cet excédent de mémoire ?

- On va utiliser de la mémoire virtuelle et physique

Virtual memory



Pour savoir le nombre de page : $\frac{512}{4} = 128\ pages$

Page table and DAT

La plupart des pages sont gardées dans la cache et dans la RAM qui sont plus rapide mais certaines sont tout de même sur des périphériques plus lent comme les disques durs.
Pour gérer cette adresse virtuelle, il nous faut une Page Table et un mécanisme de traduction de page virtuelle pour savoir où elles sont physiquement stockées sur le disque. Cette traduction est faite par le "Memory Management Unit" (MMU) et s'apelle la "Dynamic Address Translation (DAT).

La page table doit impérativement rester en cache et en RAM

Page table location

Mémoire physique :

Facile à adresser, pas de traduction d'adresse requise mais consomme de la mémoire.

Mémoire virtuelle (espace d'adressage virtuel de l'OS) :

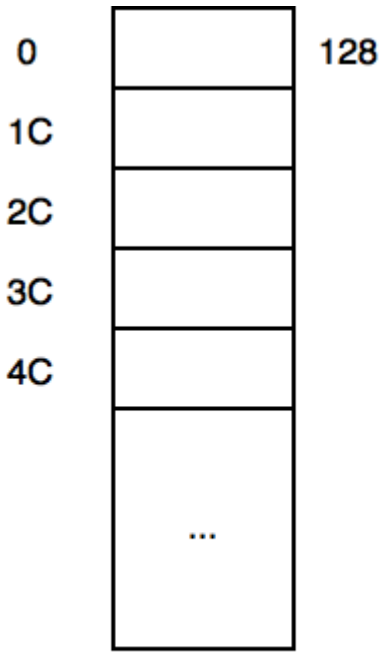
Page table non utilisé peut être référencé en dehors du disque mais mais les adresses requiert une traduction.

Basic hardware techniques for DAT

Direct mapping

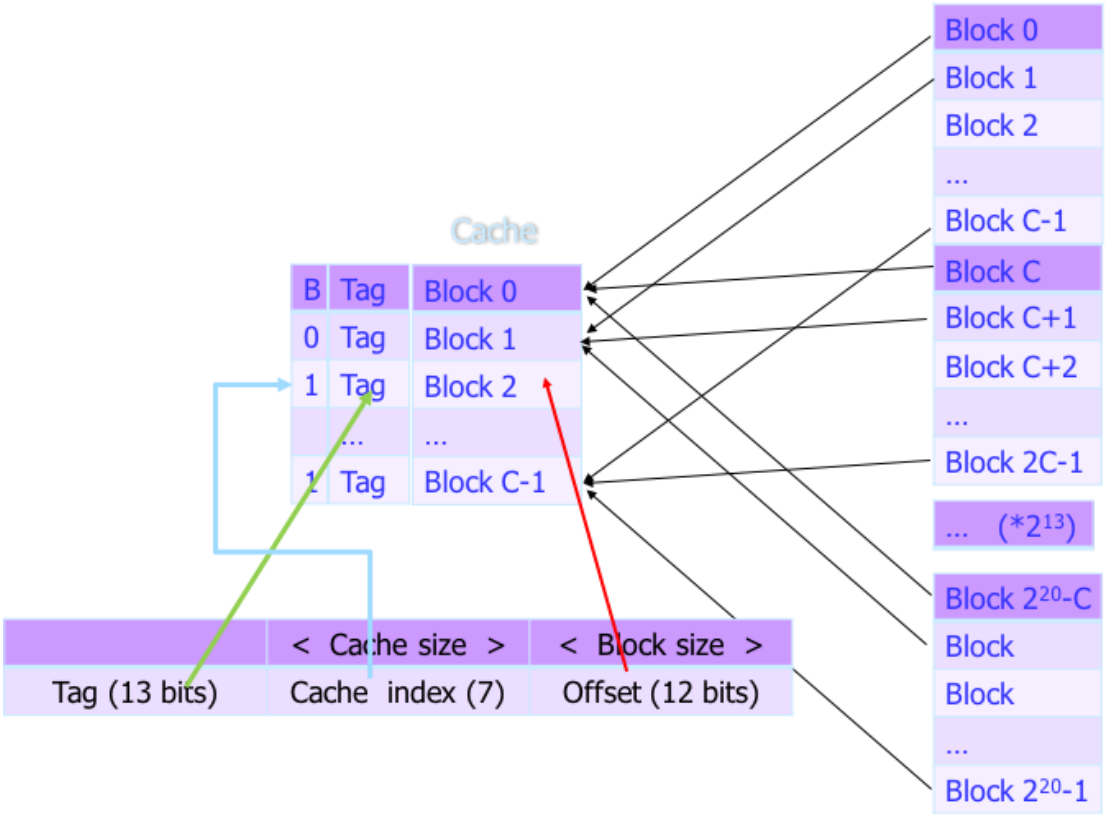
Les pages d'adressages directes sont stockées dans la RAM et la cache.

Exemple : 1. On considère que la RAM est divisée virtuellement par bloc de 128.



$10^6 = 2^{20}$

2)



Associative mapping

Même chose que pour le direct mapping mis à part que les pages ne se trouvent plus dans le même bloc dans la cache.

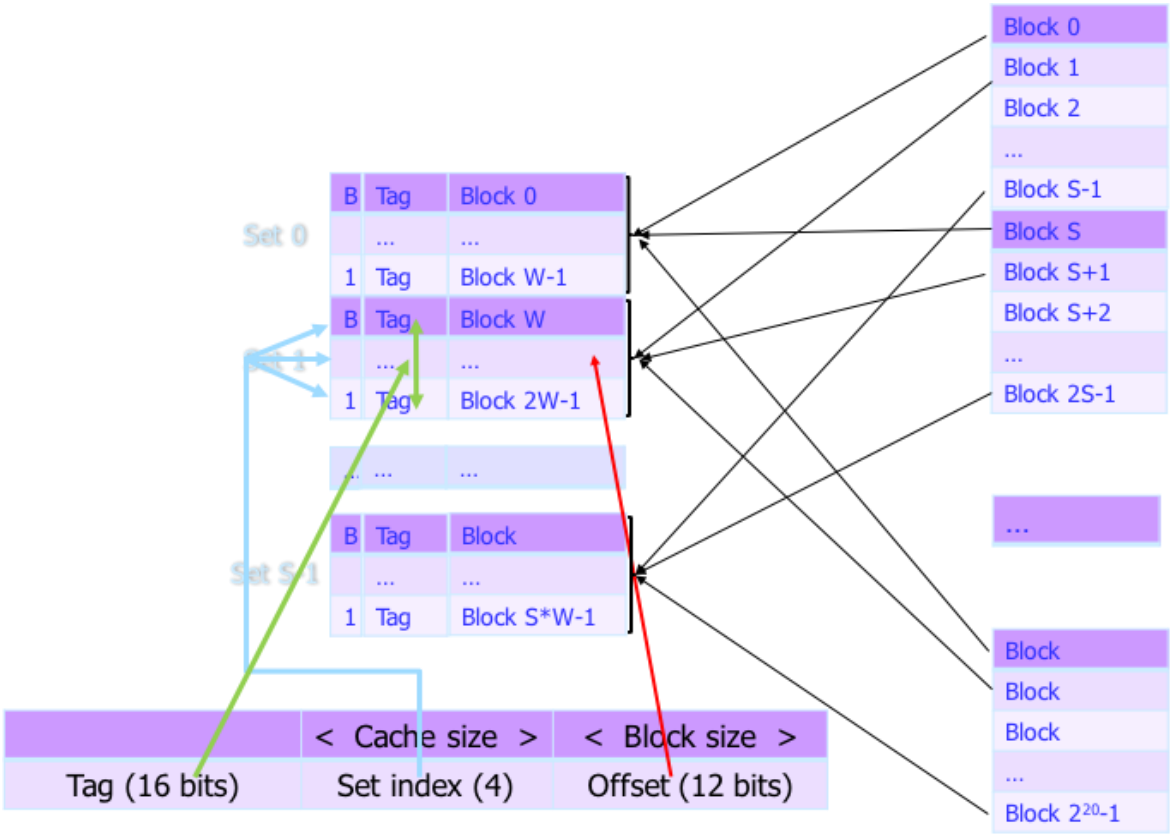
Les recherches sont donc moins rapide car il y a 128 entrées à vérifier à chaque fois

- Plus lente pour les recherches mais a un hit ratio plutôt élevé

Set-associative mapping

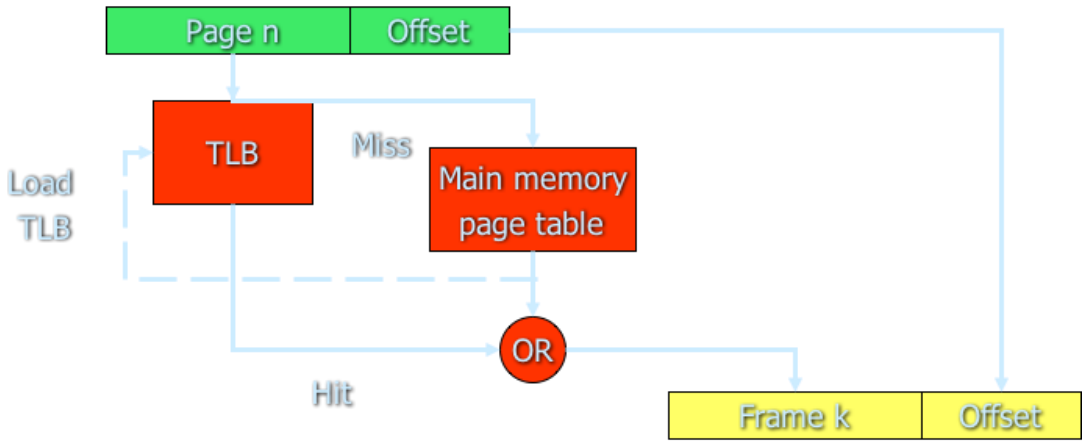
C'est la combinaison des deux premiers. Par exemple, la cache est divisé en 16 ensemble de 8 bits. La cache est donc coupé en plus petites parties.

Exemple :



Translation lookaside buffer (TLB)

C'est une cache hardware ultra rapide qui se trouve dans le MMU (et dans la page table). Elle contient les entrées de la page table les plus frequemment utilisées. Chaque mémoire virtuelle peut potentiellement référencer 2 ou plusieurs accès à la mémoire physique .Un (ou plus) pour chercher la page table de deuxième niveau et 1 pour faire sortir la donnée. La TLB est basée sur la technique Set-associative mapping et est relativement petite. Une TLB de niveau 1 contient de 32 à 64 entrées et une TLB de niveau 2 contient jusqu'à 512 entrées.

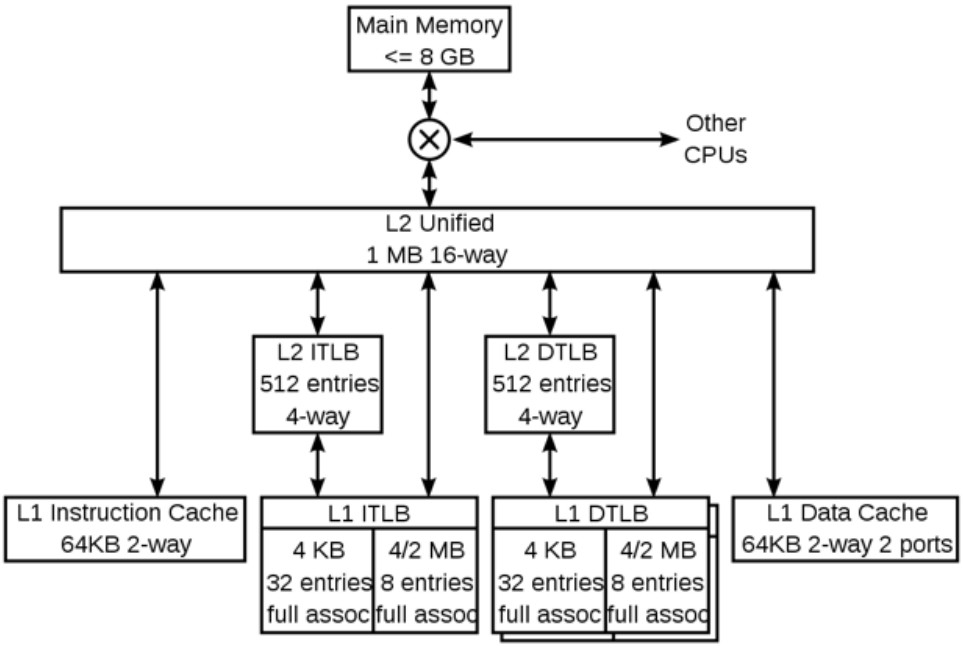


Deux cas sont possible quand le processeur cherche dans la TLB pour une adresse virtuelle donnée :

- Si la page est trouvée (TLB hit), le numéro de fenêtre est récupéré et l'adresse physique est formée.

- Si la page n'est pas trouvée (TLB miss), on la cherche tout d'abord dans la page table. Si elle n'est toujours pas là, on va la chercher dans la RAM. Dans les tous les cas, on remonte la donnée dans la TLB.

TLB and multi-level caching



Il existe plusieurs tailles de TLB (4k, 512 MB) car certaines situations particulières s'y porte mieux.

Paging policy

C'est la façon dont on enlève et met les données dans une mémoire virtuelle. Le but est d'avoir bien évidemment le meilleur hit ratio possible.

Fetch policy

Demande : Lorsqu'un processus à besoin d'une page, on va la chercher. Dans ce cas-ci on ne connait rien à l'avance, tout ce fait sur demande.

Anticipation : Il faut aller lire les pages suivantes pour ne pas avoir de page fault

Il faut avoir un buffer assez grand

Cela peut également être dangereux car on n'en a pas forcément besoin

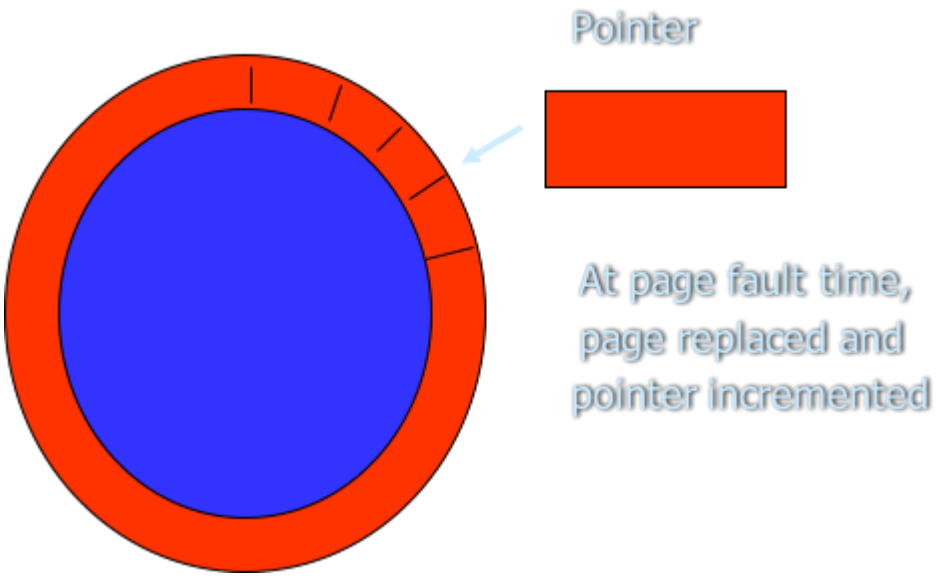
Replacement policies

Random : Une page est sacrifiée aléatoirement. On ne regarde pas l'historique d'utilisation de la donnée. Cet algorithme cause beaucoup de miss et a été rapidement abandonné au début des années 60.

FIFO Premier arrivé, premier dehors. En pratique, un pointeur tourne au fur et à mesure et sacrifie la page qu'il pointe.

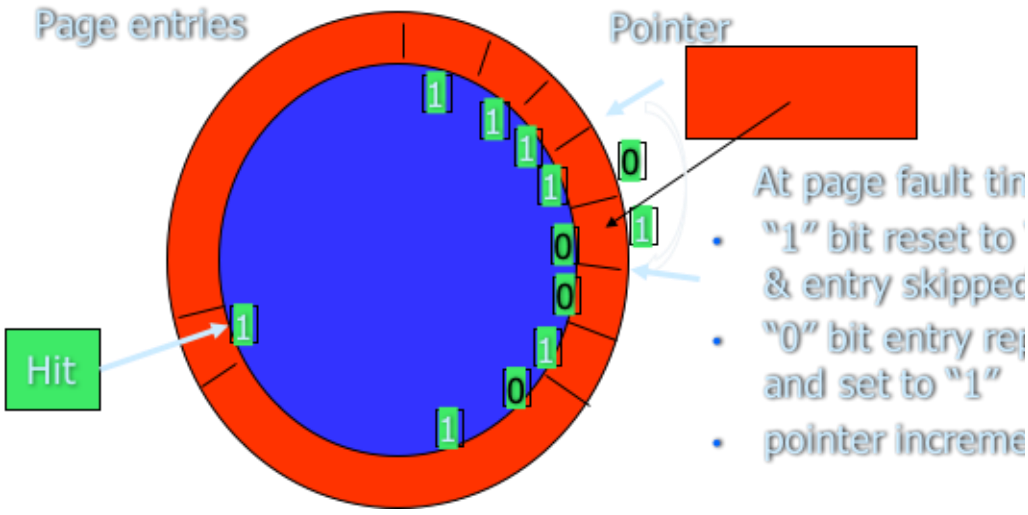
Il peut encore y avoir des doublons dans le buffer

Exemple :



Not recently used (NRU) Celui qui n'a plus été utilisé depuis longtemps est enlevé. En pratique, le pointeur navigue jusqu'au 1er bit à 0 et l'enlève.

Exemple :



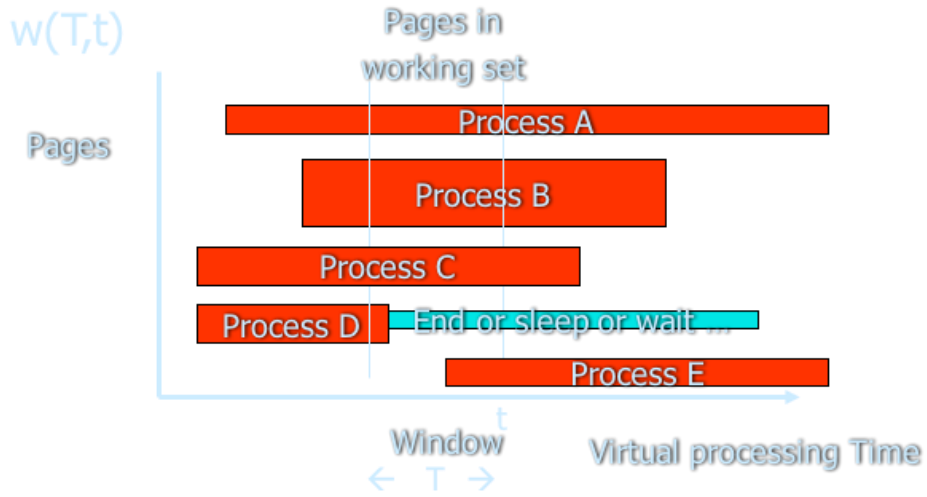
Least recently used (LRU) :	Celui qui est le plus vieux de tous est enlevé. On est alors obligé d'utiliser une référence de temps (timestamp) ou un tableau pour voir lequel est le plus vieux.

	Timestamp très lourd à implémenter

Least (or not) frequently used (LFU / NFU) :	Emploi de la fréquence d'utilisation de la page grâce à un compteur (beaucoup mins lourd qu'un timestamp). Ce mécanisme contient toujours un système de aging mais est beaucoup plus facile à implémenter.
	Si le processus a utilisé plein de fois au début une page tandis que les autres moins mais que maintenant il utilise celle qu'il n'a pas encore utilisé, il y aura un problème.

Working set algorithm

Le système alloue un espace mémoire variable (T interval) en fonction du processus. Lorsqu'il y a un page fault, le système augmente la mémoire allouée à ce processus. Une page est déchargée lorsqqu'elle n'est plus référencée dans l'interval T.

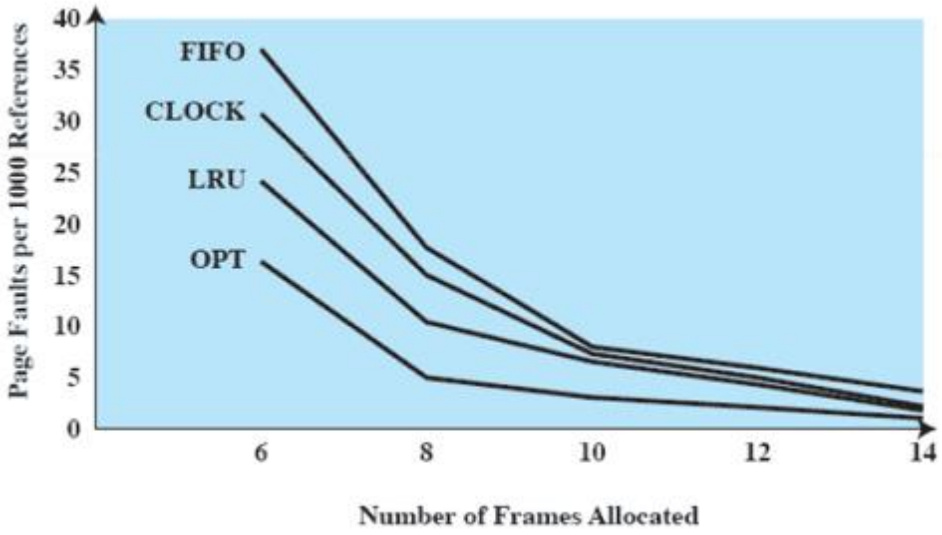


Belady's optimal algorithm (optimisation)

C'est un concept théorique. Pour avoir la meilleur efficacité il faudrait :

- Anticiper le futur
- Toujours diminuer le plus possible les pages fault
- Remplacer les pages les plus vieilles et éloignées
- Enlever les pages dont on a plus besoin

Algorithm efficiency



Stack algorithm

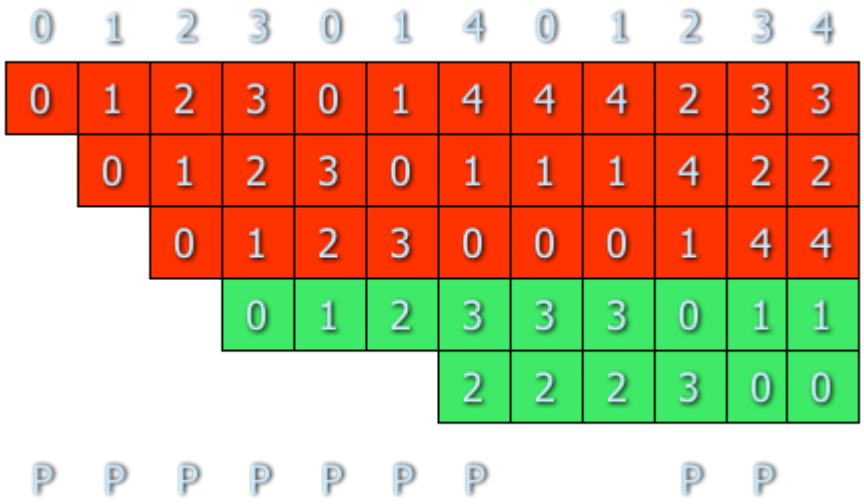
Intuitivement, une cache de taille X sera moins efficace qu'une cache de taille X+1

Belady's anomaly

: En cache

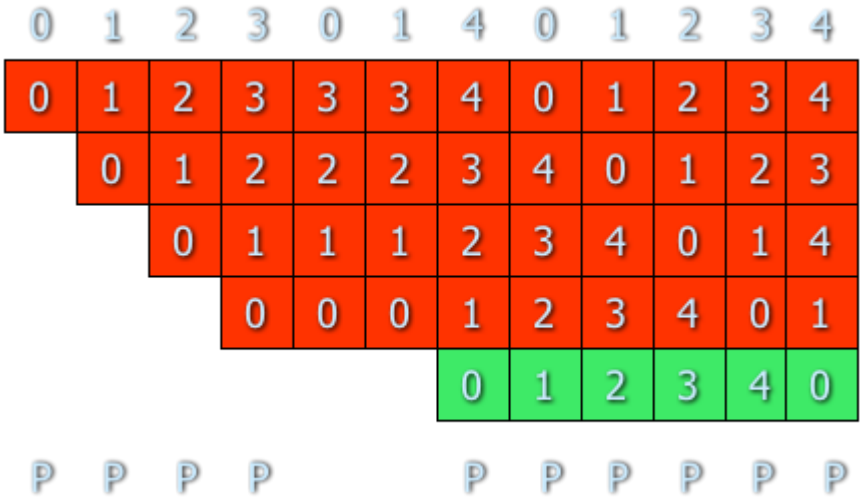
: Hors cache

Taille de cache 3



Nombre de page fault : 9

Taille de cache 4



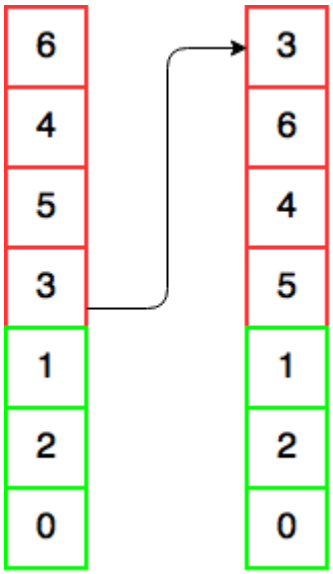
Nombre de page fault : 10

- L'exemple ci-dessus montre qu'une cache de taille plus grande ne résoud pas forcément le problème de page fault

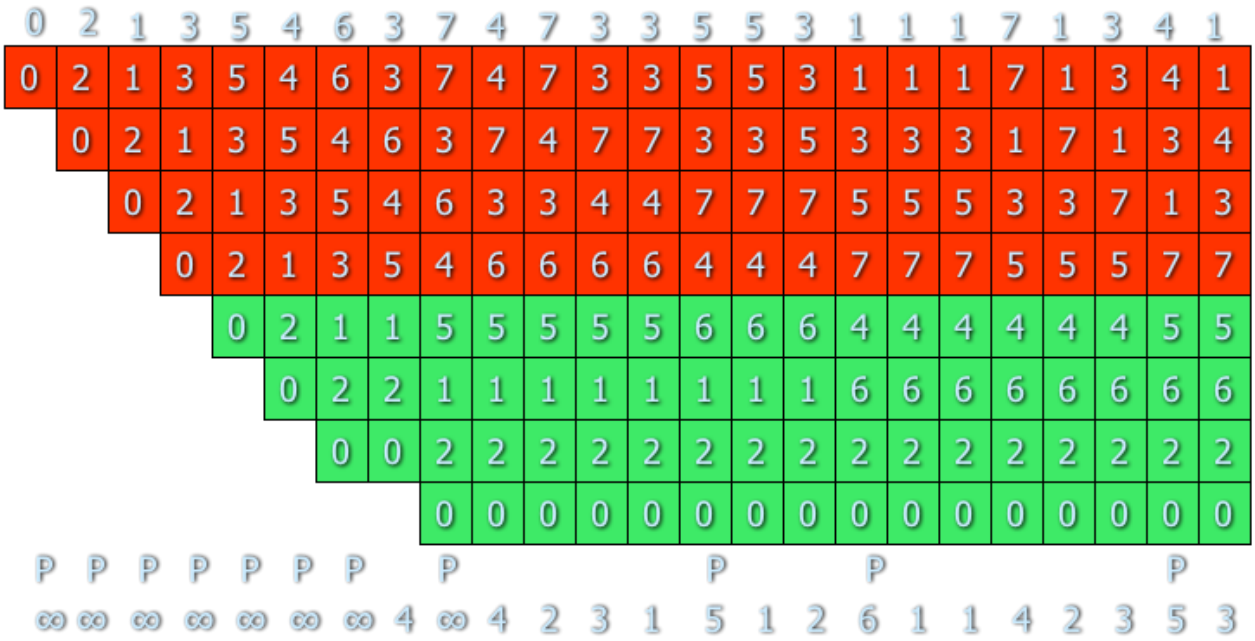
Distance string

A quelle profondeur dans la pile (stack) la page se trouve?

- Lorsqu'une page qu'on cherche se trouve quand même dans la pile, on la met en tête de la pile.



Exemple :



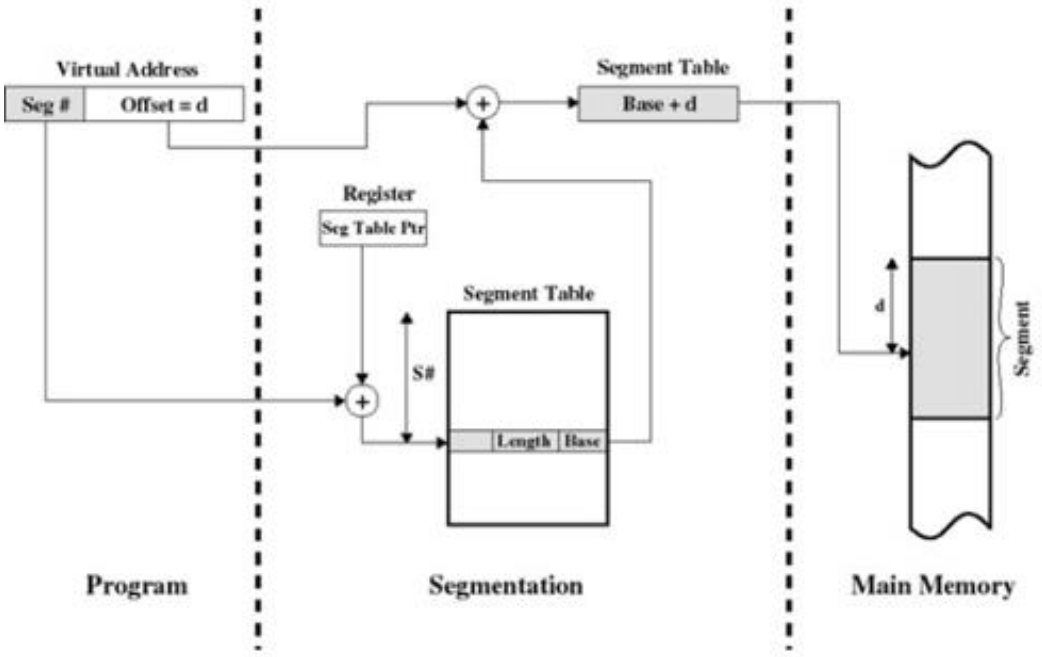
Segmentation

C'est une technique de découpage de la mémoire. La segmentation divise la mémoire physique en mémoire virtuelle en segments caractérisés par leur adresse de début et leur taille. Elle permet également la séparation des données et du programme facilitant alors la programmation et le partage interprocessus. Chaque segment offre une grande protection grâce au niveau de privilège de chaque segment.

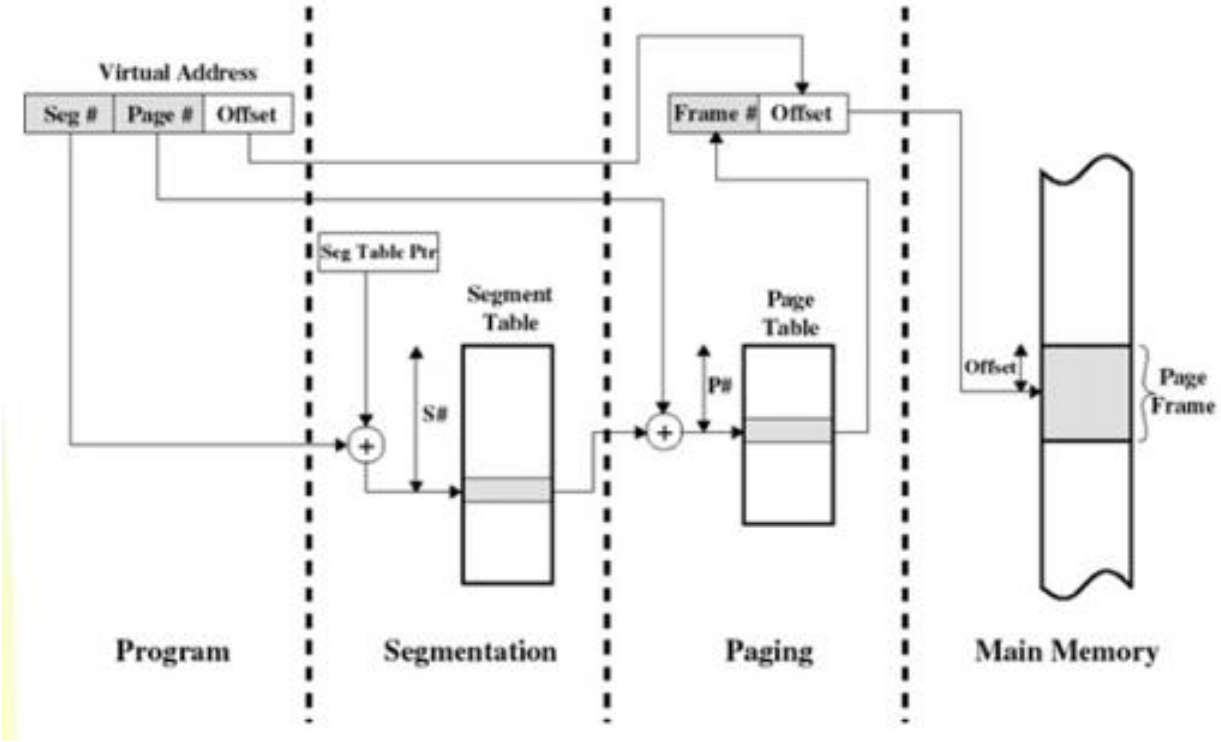
Les adresses générées sont alors composées de 2 éléments :

- Le numéro de segment (qui est l'emplacement de base)
- L'offset (qui est l'offset de la cellule cible dans le segment)

Exemple :



La segmentation et le paging peuve être combiné.



- Une toute petite partie va se trouver dans la cache (2^7)

Nombre de conflits possible :	$\frac{2^{20}}{2^7} = 2^{13}$

- On doit également savoir quel bloc de page est dans la cache. Le tag permet savoir si il y a un cache hit ou miss.

- Très rapide pour des recherches mais à un hit ratio très limité

L1 ITLB :	Cache de niveau 1 pour les instructions
L1 DTLB :	Cache de niveau 1 pour les données

On regarde également combien de fois on a dû chercher une page en page fault. Le but étant de diminuer le nombre de page fault en fonction de la taille du stack (trouver la taille optimale de la cache).

Ressource management

mercredi 21 décembre 2016
18:27

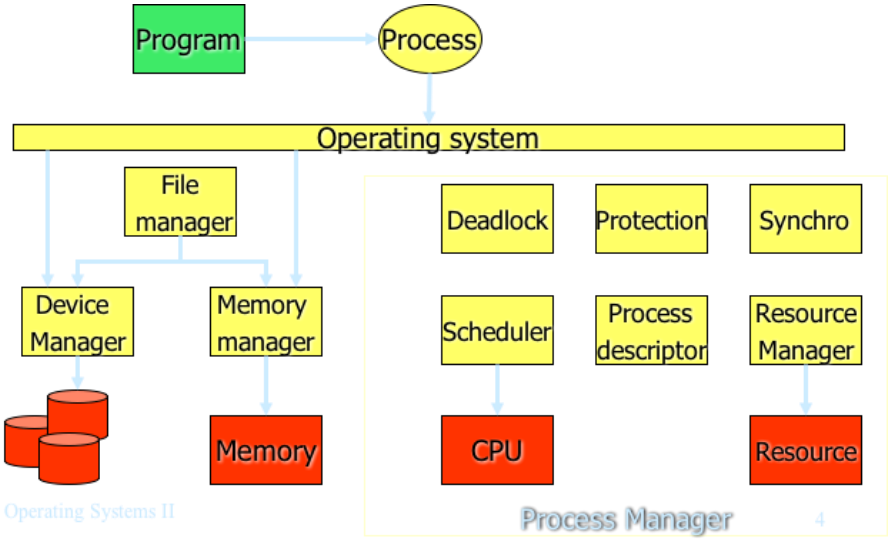
Process management

Le système d'exploitation gère les processus de telle sorte qu'ils pensent avoir accès à toutes les ressources. En réalité, il n'a pas accès à toutes ces ressources, l'OS décide qui prend qquoi à tel moment. Cette gestion permet d'avoir plusieurs processus en même temps.

OS view of a process

Un processus est composé de :

- Un programme qui défini son comportement
- Données traitées par le processus et ses résultats
- Un enregistreur de statut qui permet de surveiller la progression de celui-ci
- Un ensemble de ressource pour son exécution

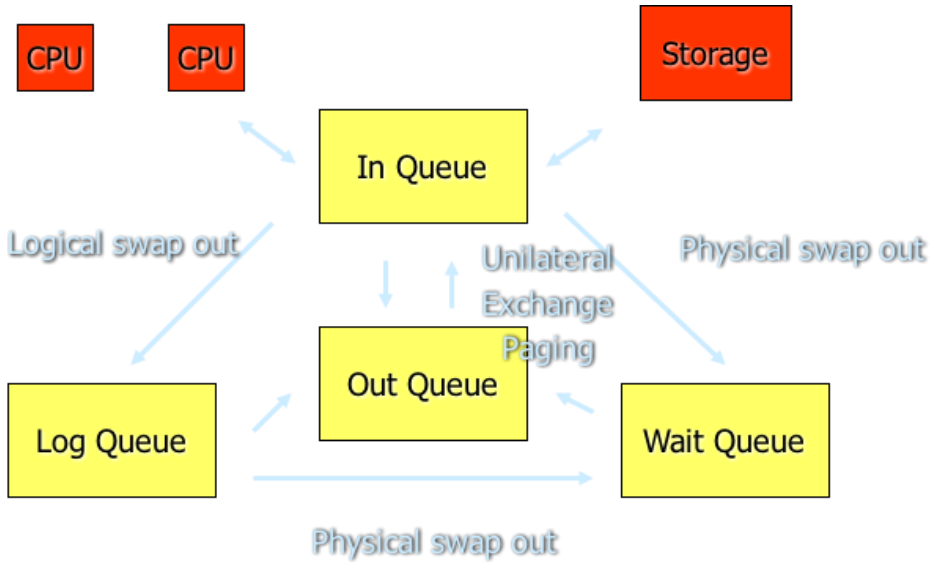


Process descriptor

Chaque processus dans le système possède une description. Cette description est composée de :

Process state diagram

Lorsqu'un processus change d'état, le système d'exploitation retient la cause de ce changement d'état.



In queue : Dans l'état RUNNING

Out queue : Dans l'état READING

Log queue : Dans l'état WAITING

- Ne relâche pas les ressources si aucune ne les demande
- Rapide

Wait queue Dans l'état WAITING

- Relâche les ressources
- Plus lent

- Les deux derniers états sont une optimisation pour létat waiting

CPU ressource management

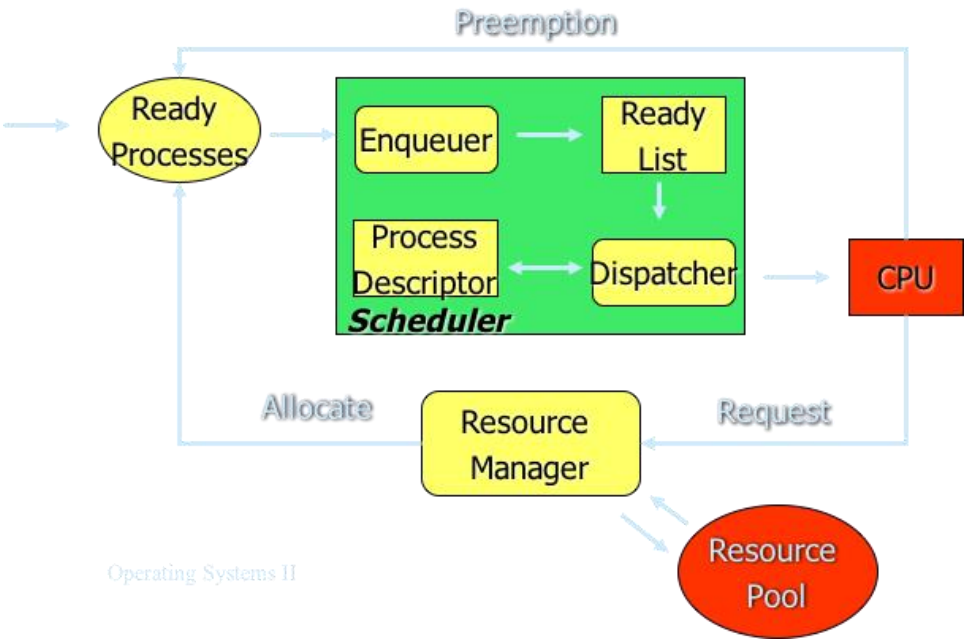
Le gestionnaire du precesseur est divisé en 3 niveaux :

Le dispatcher : Il envoie les processus sur un CPU disponible

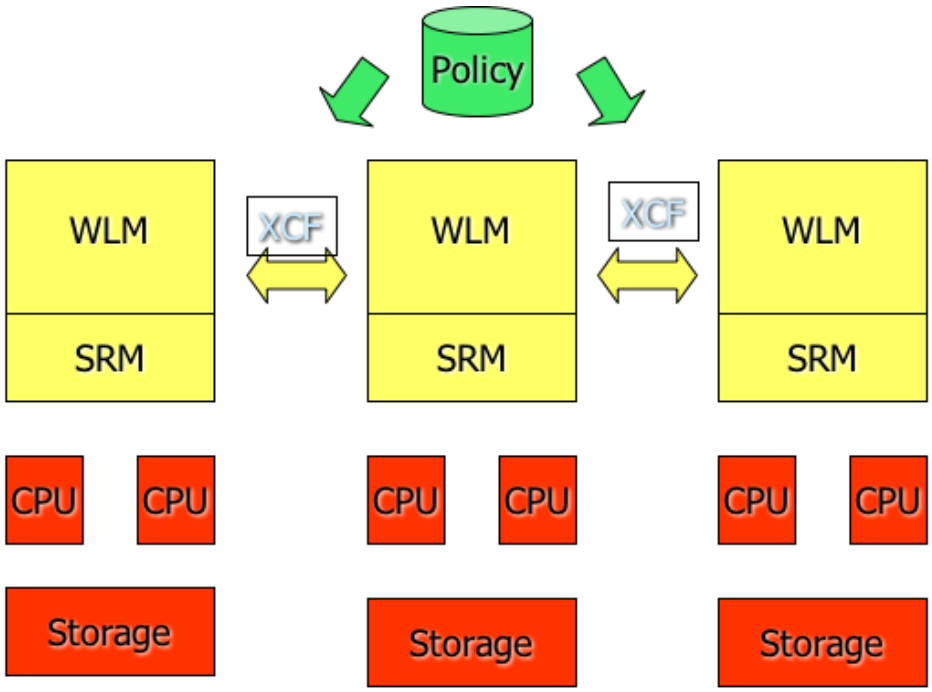
Le scheduler : Il utilise des règles de priorité

Le workload : Il distribue une tâche spécifique sur plusieurs CPU afin de diminuer le temps d'exécution (se fait beaucoup sur le réseau)

Le scheduler



Le workload manager



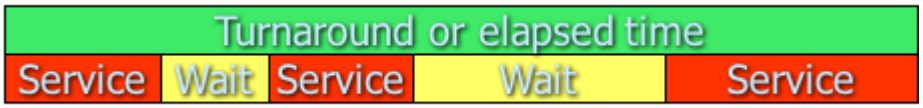
Scheduler strategy

Chaque processeur doit être gérer avec équité :

- Ils reçoivent un temps égal
- Ils doivent respecter les temps de réponse
- Ils reçoivent la même charge de travail (ou le plus proche possible)
- Un travail doit également ne pas rester éternellement dans le système
- Il faut également essayer de garder les CPU occupé au maximum

Mais ces différentes stratégies dépendent essentiellement du but de l'OS et de son utilisation. En effet, un ordinateur de bureau de doit pas satisfaire les mêmes exigences qu'un ordinateur de vol pour un avion.

Scheduling algorithms



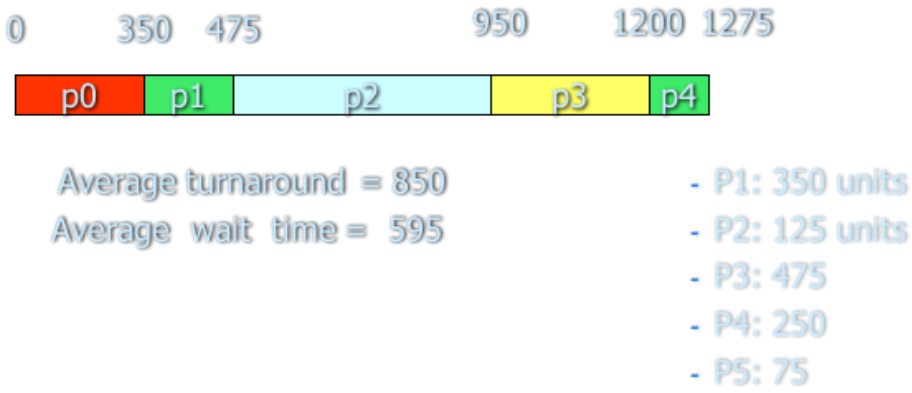
Il y a deux grands types de scheduler :

• **Non préventif :**	Lorsqu'un processus est dans l'état RUN, il n'est pas interrompu jusqu'à ce qu'il ait terminé.
• **Préventif :**	il y a une notion de priorité et les processus change en fonction de cette priorité.

- Voici quelques algorithme de scheduling **non préventif** (une fois lancé, ils ne sont jamais interrompu):

- First-come-first-served

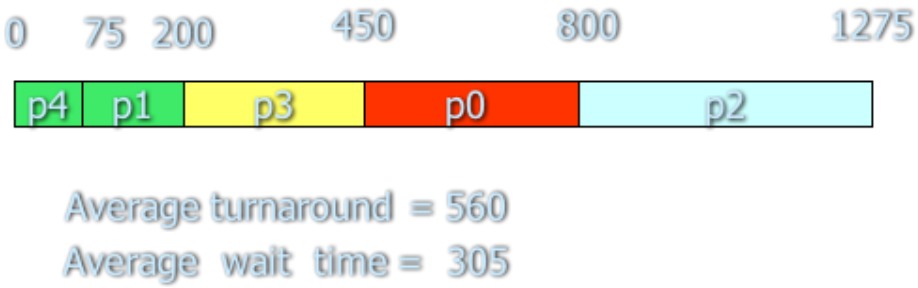
  Littérallement : Premier arrivé, premier servi.



  Des processus durent plus longtemps que d'autres

- Shorted job next

  On fait d'abord passer les plus petits processus. On constate alors que les moyennes d'attente et d'exécution sont diminués (voir schéma ci-dessous).



Les gros processus ne peuvent jamais passer s'il n'y a que des petits

3. Priority scheduling

Des priorités sont déterminée pour chaque processus

4. Deadline scheduling

Les processus doivent avoir fini avant un certain temps. Ce système est utilisé dans les OS à temps réel.

Les avantages du scheduling préventif

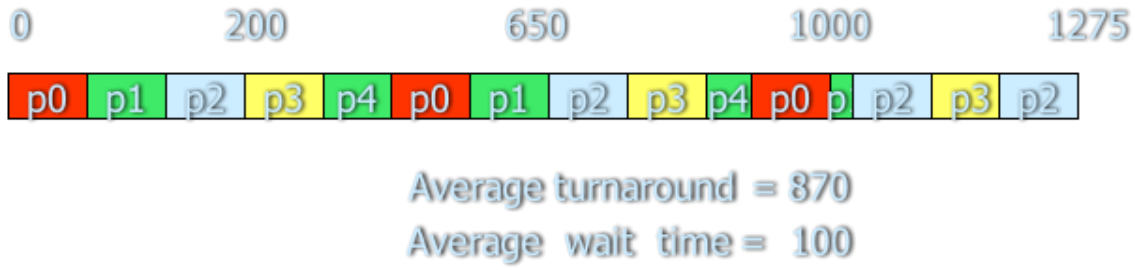
- Il ne requiert pas de connaissances préliminaires
- La distribution du temps CPU est équitable

- Voici maintenant 2 algorithmes **préventif** (avec notion de priorité) :

- Round-Robin

Chaque processus reçoit un temps CPU et lorque ce temps est écoulé on passe au suivant. Avec cet algorithme, on constate une amélioration du temps d'attente que se soit pour les gros ou les petits processus.

Malheureusemebt le temps d'exécution est plus grand (voir schéma ci-dessous)



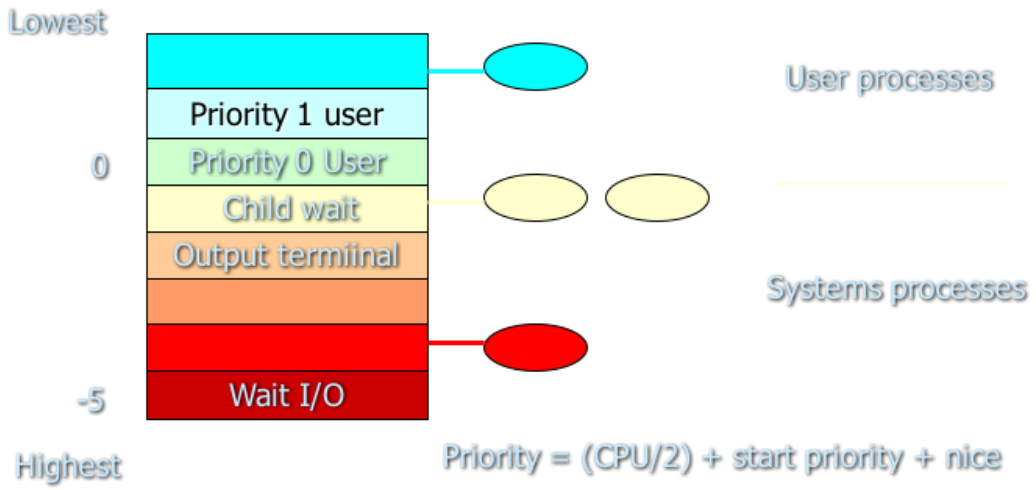
- Multiple-level queues

Chaque processus est mis dans sa propre file qui correspond à son temps d'exécution. Chaque file possède alors une priorité. Le scheduler peut alors changer la priorité de certain processus si ils attendent depuis trop longtemps ou même tuer des processus qui dépassent le temps imparti.

Mise en place d'un système de "aging" :

1. Si le processus consomme trop de ressources, on diminue sa priorité
2. Si le processus consomme pas assez, on augmente sa priorité

Exemple sur Unix :

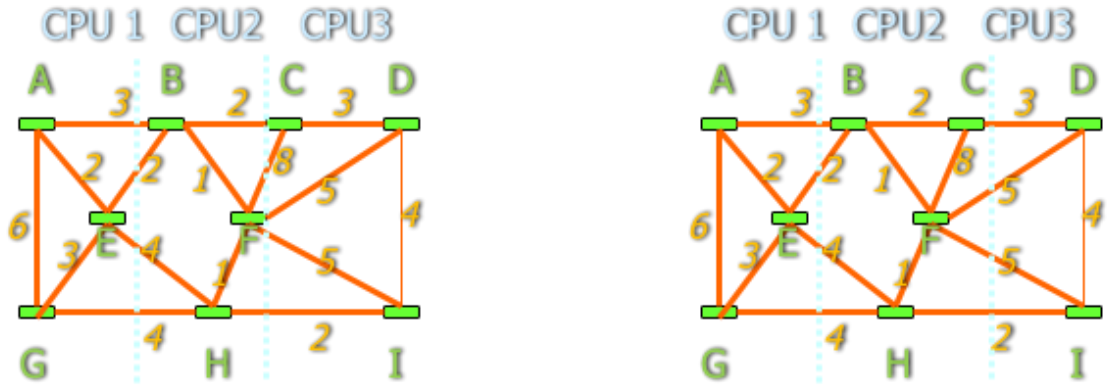


Processor allocation algorithm

Le but de cet algorithme est de chercher l'optimisation maximum sans que cela devienne une charge trop importante. Le minimum à avoir pour qu'un système réponde dans un temps convenable est :

80% IN et 20% de page fault (strict minimum)

Graph theoretic - deterministique



Le graphes contiennent des processus (en vert), le traffic de données (barre orange+ chiffre) et son répartit sur 3 processeurs. La différence entre le graphe de gauche et de droite se trouve au niveau du 3e processus (vert) en partant du coin supérieur gauche. L'un se trouve dans le CPU 3 et l'autre dans le CPU 2. Le graphe de droite

sera plus rapide car le coùut pour passer de F à C sera sur le même processus tandis que celui de gauche ne se trouve pas sur le même processus.

Co-scheduling

Le principe est de faire travailler deux processus qui communiquent entre eux à la chaîne. De ce fait, le résultat final du 1er processus peut directement être délivré au 2e processus qui en a besoin.
Les processus sont alors regroupés en time slice.

Ces time slice doivent avoir une taille équivalente pour être performant

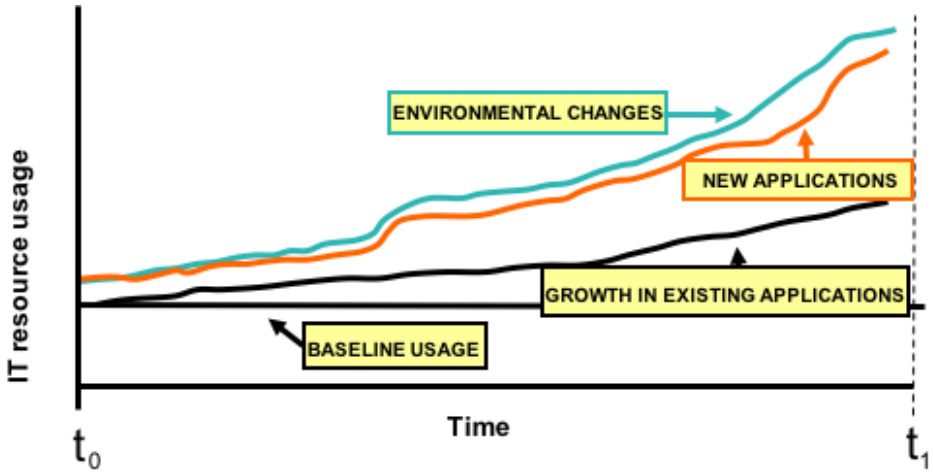
Performance management

SLA :	C'est un objectif à atteindre. Exemple : Une transaction bancaire doit se faire en moins de 1/2 seconde
SLO :	Ce sont des sous-objectifs. Exemple : Il faut que 95% des réponses soient faites en moins de 10 ms

- Toutes ces mesures doivent être calculées. Le système qui s'occupe de cela est le monitoring.

Capacity management

C'est le fait d'analyser une tendance générale. En informatique par exemple, on a besoin de plus en plus de puissance dû aux applications, nombre de clients croissant, …
Ils faut pouvoir faire des prévisions de tous ces changements pour l'avenir. Bon nombre d'entreprises doivent utiliser ce système. Elles peuvent avoir besoin, par exemple, de nouveaux disques durs, de nouveaux mécanisme de sécurité.



Le plan de capacité : Exemple :

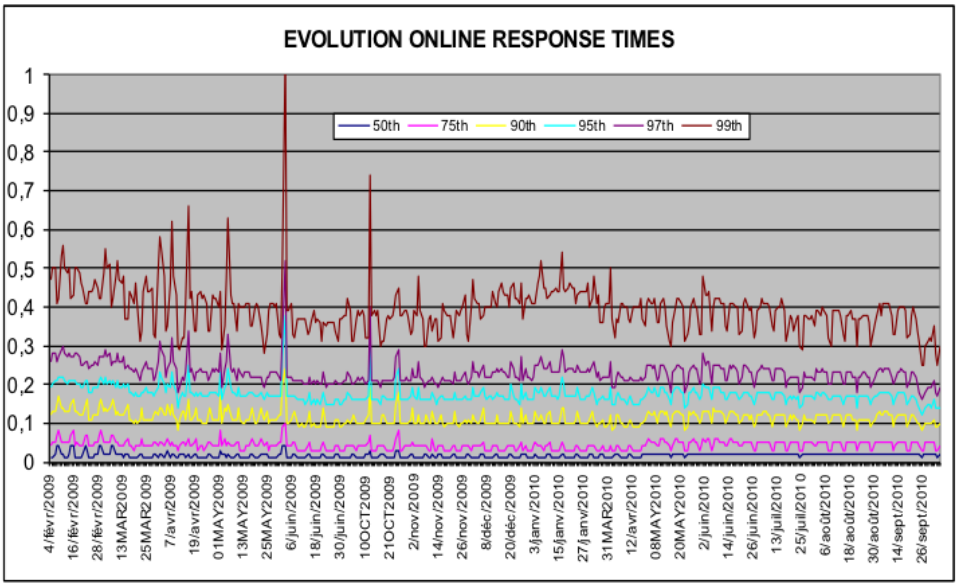
Les serveurs Windows et Linux sont peu utilisés (environ 15%) mais le coût d'entretien de ces serveurs sont peu cher donc ce n'est pas très grave.
Par contre, les mainframes sont beaucoup plus cher que les serveurs. Pour pouvoir les rentabiliser, il faut que le taux d'utilisation sur une journée soit beaucoup plus élevé. C'est le rôle de scheduler d'augmenter ce taux d'utilisation.

Systems monitoring

Le monitoring peut également servir pour voir si un processus particulier dure plus longtemps qu'avant. Si un processus durait 5 secondes et maintenant pour effectuer le même travail il dure 6 secondes cela est mauvais.

Il faut tout de même faire attention aux moyennes car elles peuvent être tompeuses
Le système de mesure (monitoring) ne doit pas ralentir le système

Exemple :



- Le processeur utilisé avec ses registres
- Le statut du processus
- Le statut de la mémoire
- Le pointeur de pile
- Les ressources attribuée
- Les ressources nécessaire
- …

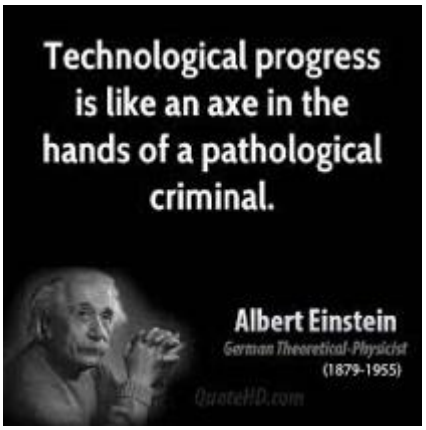
Security

jeudi 22 décembre 2016
16:01

Pourquoi faire de la sécurité ?

- On peut avoir des informations précieuses et confidentielles

- Si il y a un problème, on doit pouvoir reprendre rapidement ses activités
- Pour la valeur de la marque et sa réputation
- S'il n'y a pas de sécurité, la société est en danger



Motivations

1. CONFIDENTIALITE	2. INTEGRITE	3. DISPONIBILITE

- Il faut être protégé par les attaques éventuelles venant de l'extérieur. Les données d'une entreprise sont l'unes des plus précieuses ressources d'une entreprise. Il faut alors utiliser des privilèges d'accès pour chaque personne.
- Il faut pouvoir se protéger de toutes erreurs qui peuvent survenir. Il ne faut pas authoriser une tentative de connexion échouée à plusieurs reprise d'avoir accès au données(exemple).

3. Il faut prévoir des sytème de récupération en cas de perte de données. Cela peut venir d'une attaque de type DDOS, du code caché mal attentionné (cheval de troie, …), défaillance hardware et/ou software, Une défaillance d'un partenaire/ fournisseur, d'une panne d'électricité, d'une catastrophe naturelle, …

Méthodes pour prévenir d'un problème de sécurité

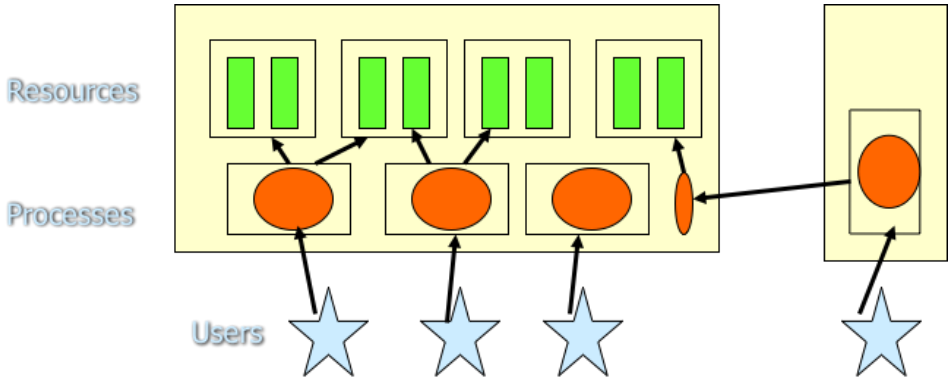
- **Protection**
- **Détection**
- **Eradication**
- **Récupération**
    - Mise en place de firewall, de Web Application Firewall (WAF), de détection d'intrusion, de scanners de vulnérabilité, …

Basic protection at facility

- Plusieurs site de stockage des données en cas de problème sur un
- Politique de bacup régulier
- Equipement tolérant les panne (RAID,…)
- Mise en place de droits d'accès et donc d'authentification
- Anti-virus, firewall,…
- Application de protection (cryptage,…)

Ressource protection model

Un utilisateur accède aux ressources via des processus



Authentication

Plusieurs possiblités sont possibles :

- Le modèle classique du login + mot de passe

Ici quelque alternative qui peuvent être complémentaire :

- Empreinte rétinienne
- Empreinte digitale
- Calculette synchronisée (banque)
- SMS

Il existe des centaines de façon pour pénétrer un sytème mais les trois plus communes sont :

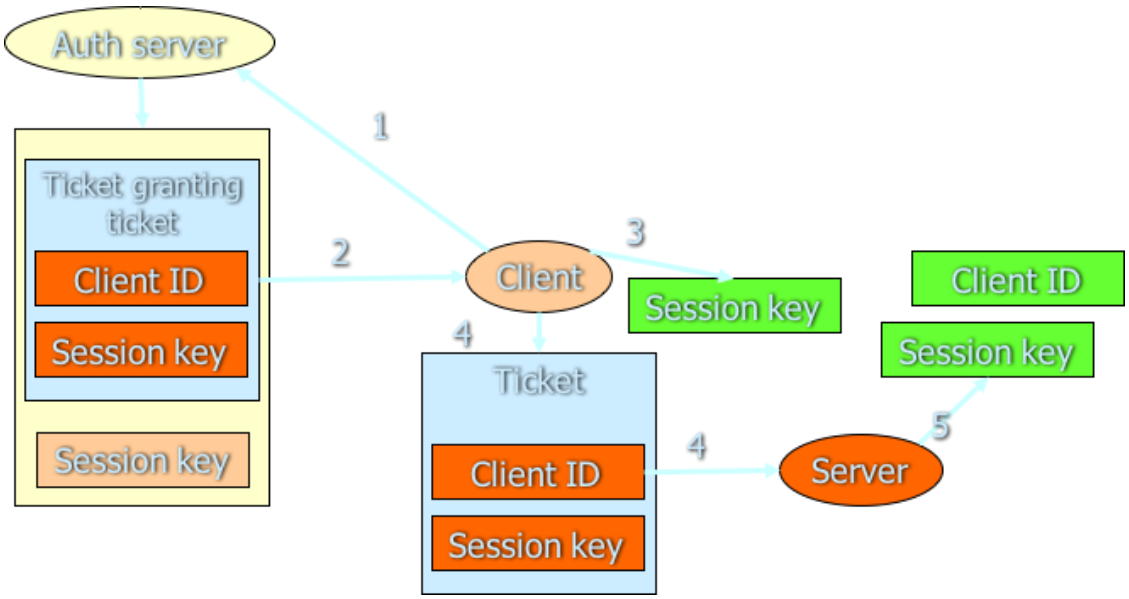
- Le transfert de fichiers
- Du code caché sur les pages web
- Les emails
- …

Kerberos

C'est un protocole d'authentification réseau créé par le MIT qui repose sur un mécanisme de clés secrètes et l'utilisation de ticket (pas de mot de passe).

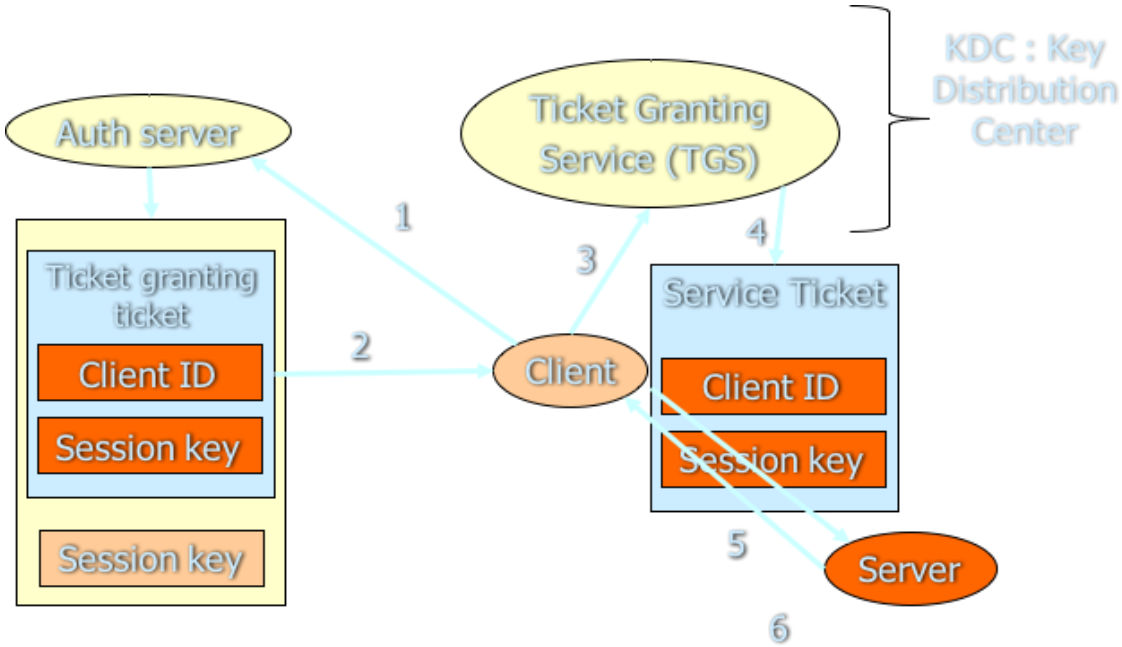
Phase 1 :
- Le client demande un ticket

- Le serveur lui donne un ticket et une clé de session
- Ce ticket est utilisé pour demander d'autre ticket pour des services divers
- Ce ticket véhicule l'identité du client au serveur
- La clé de session est utilisée alors pour la communication entre le client et le serveur



Phase 2 : • Le client utilise le ticket de la phase 1 pour avoir l'autorisation du serveur

Phase 3 : • Le client présente la clé au serveur



Chaque utilisateur partage une clé secrète avec le KDC
L'utilisateur entre le mot de passe sur sa machine locale uniquement
Le ticket généré la la KDC à une durée de vie limitée

Role based access protocol (RBAC)

Ce système est utilisé dans les grandes entreprises pour diminuer la complexité de la gestion des droits pour les utilisateurs. Chaque utilisateur appartient à un groupe dans le système qui ont des droits prédéfinis. Cela permet de changer les droits pour tout un groupe d'utilisateur à la fois plutôt que de les changer pour chaque utilisateur à la fois.

- Gestion facile

Cryptography

C'est iun sytème de cryptographie composé de clés publiques et privées. La clé publique est révélée à tout le monde sur le réseau tandis que la clé privée reste sur la machine de départ. Grâce à un algorithme bien précis, il est pratiquement impossible de recomposer les messages envoyé sans la clé privée. Seul l'envoyeur et le reçeveur savent les décrypter.
Le protocole HTTPS utilise ce système de clé privée et publique.

THE END