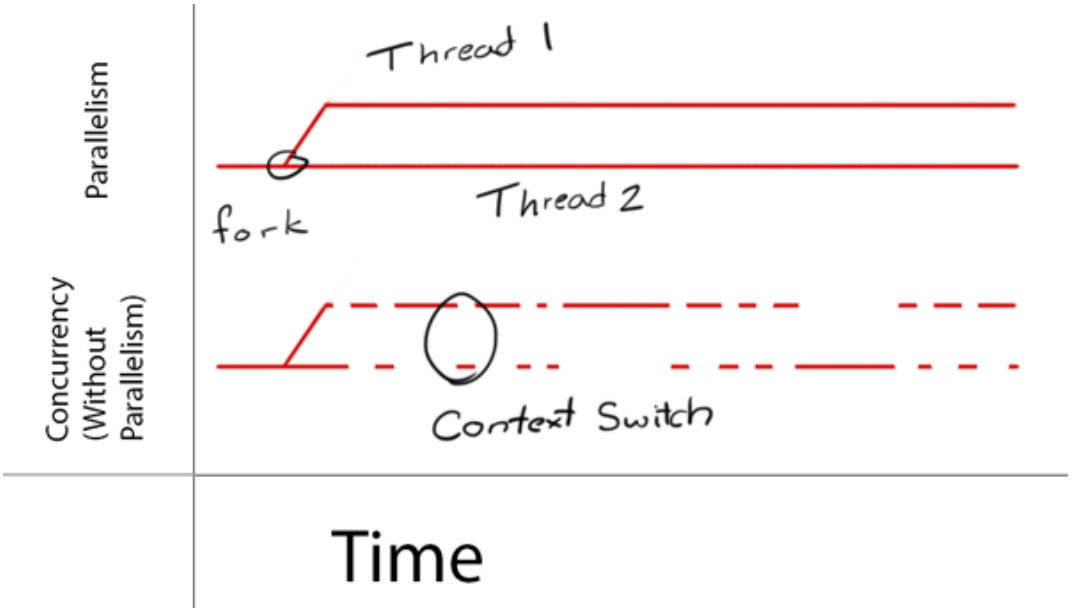
Faculté des Sciences & Techniques

Université de Limoges

Licence 3ème année



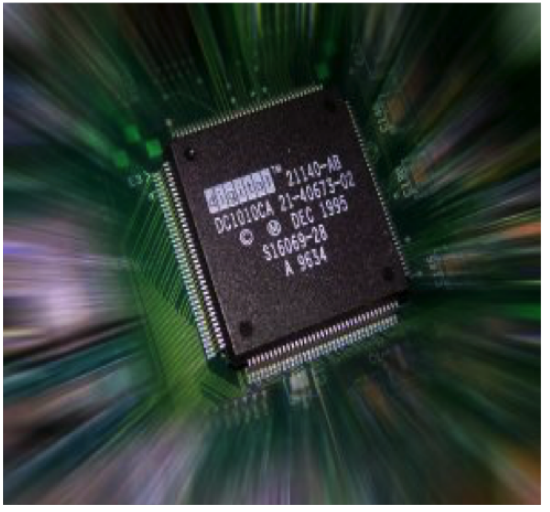
Programmation Concurrente — P-F. Bonnefoi

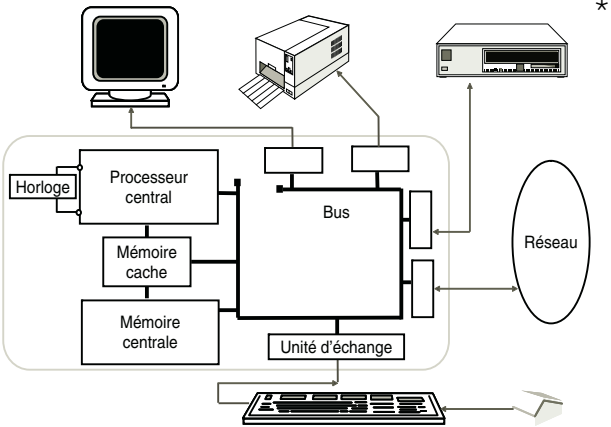
Version du 8 janvier 2024

Table des matières

Plan

- Architecture d'un processeur :
  - les différents composants ;
  - les multi-cores ;
  - l'hyperthreading ;
  - l'exécution d'un programme ;
- Le processus
  - segmentation/exécution et pile ;
  - contexte ;
- Le parallélisme
  - interruptions/horloge/timer ;
  - changement de contexte ;



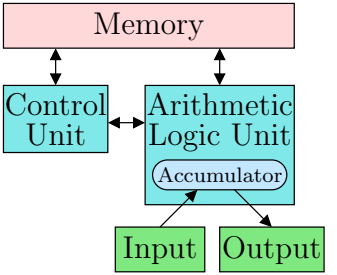


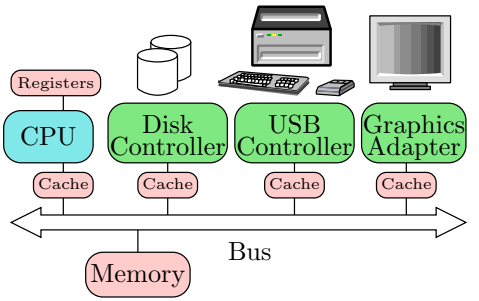
- une *mémoire centrale* :
  - architecture «Von Neuman» : programme et données résident dans la même mémoire.
    La lecture ou l'écriture dans la mémoire concerne une donnée ou une instruction.
  - architecture «Harvard» : programme et données sont dans des mémoires différentes ;
    Il est possible de charger simultanément une instruction et une donnée.

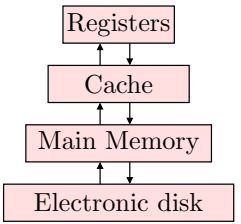
  L'architecture d'un ordinateur est de type «von Neumann».
  Des matériels embarqués, comme des modules Arduino ou des DSP, «*Digital Signal Processor*», utilisent une architecture de type «Harvard».

- un *processeur central* ou CPU, «*Central Processing Unit*» qui réalise le traitement des informations logées en mémoire centrale :
  - le processeur permet l'exécution d'un programme ;
  - chaque processeur dispose d'un langage de programmation composé d'**instructions machine** spécifiques ;
  - **résoudre un problème** : exprimer ce problème en une suite d'instructions machines ;
  - la solution à ce problème est **spécifique à chaque processeur** ;
  - le programme machine et les données manipulées par ces instructions machine sont placés dans la mémoire centrale.
- des *unités de contrôle* de périphériques et des périphériques :
  - périphériques d'entrée : clavier, souris, *etc* ;
  - périphériques de sortie : écran, imprimante, *etc* ;
  - périphérique d'entrée/sortie : disques durs, carte réseau, *etc*.
- un *bus de communication* entre ces différents composants.

- le processeur est la combinaison :
  - d'une « unité arithmétique et logique », «ALU», «*Arithmetic Logic Unit*» ;
  - d'une « unité de contrôle » : contrôle du bus servant à échanger données et instructions ;
- la mémoire n'est pas d'**accès uniforme**, c-à-d que la vitesse d'accès n'est pas toujours la même :
  - les registres : cases mémoires intégrées au processeur ;
  - le cache : zone mémoire **tampon** entre la mémoire centrale et le processeur :
    - ⋆ permet d'éviter l'accès à la mémoire centrale pour des données déjà accédées et mémorisées dans le cache ;
    - ⋆ évite d'utiliser le bus de données ;
  - le disque : permet de « *simuler* » de la mémoire.

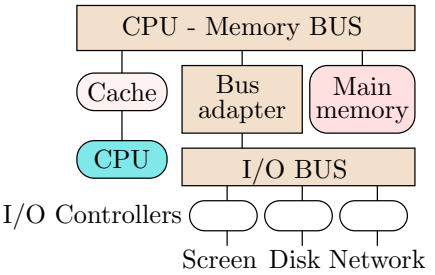




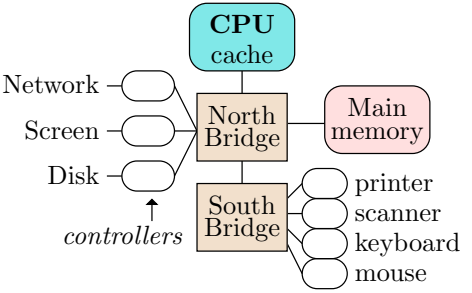


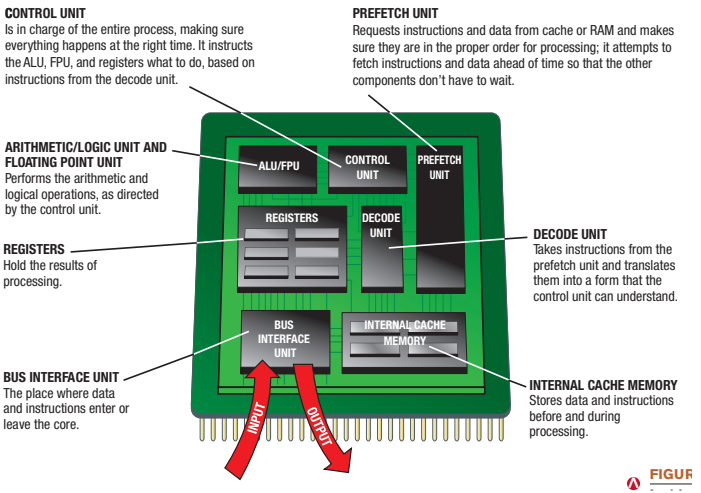
- la vitesse entre ces différentes mémoires est **très différentes** :
- la taille de ces memoires est très différente :
  - ⋆ registres $< 1ko$
  - ⋆ cache : qqs Mo ;
  - ⋆ mémoire centrale : qqs Go ;
  - ⋆ le stockage sur disque : qqs To.

"Archaic" design



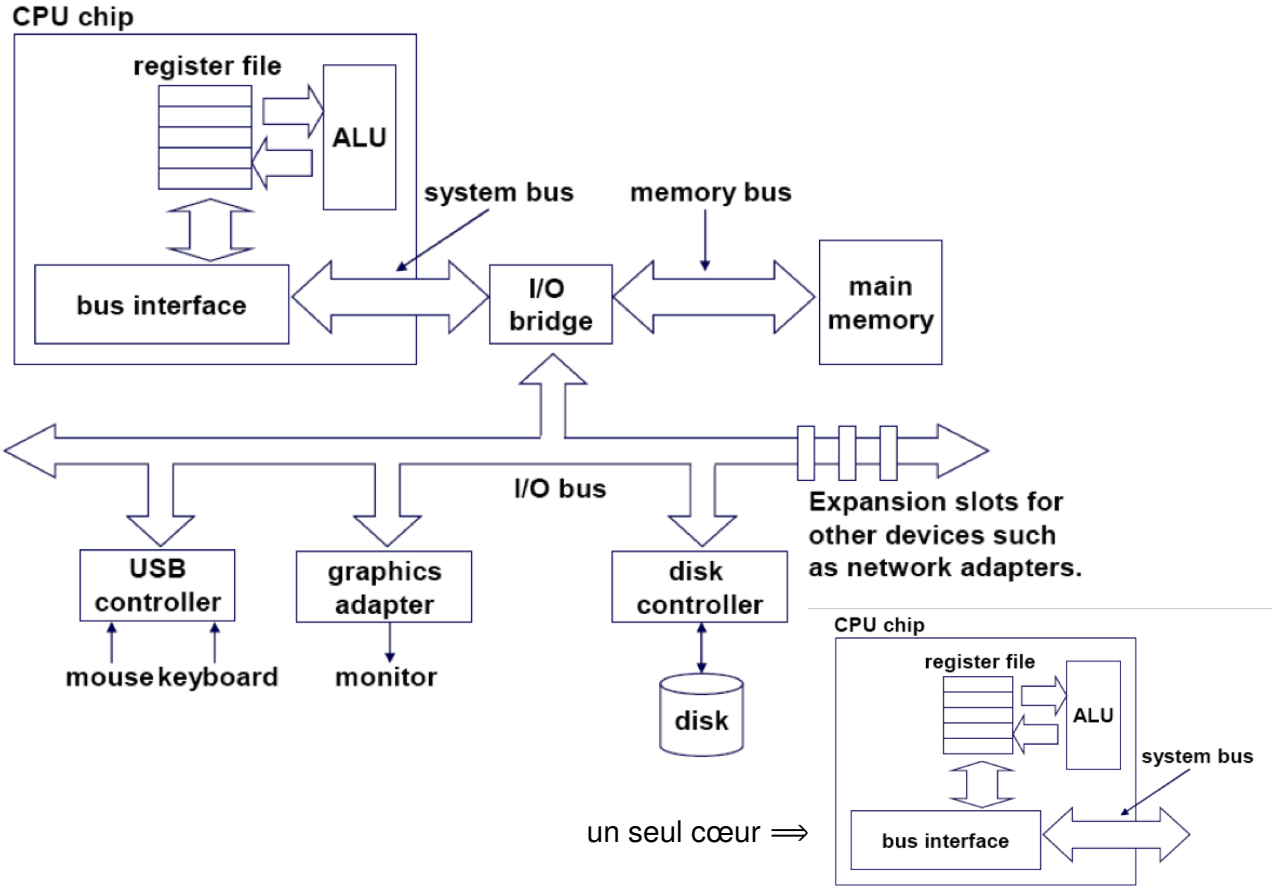
Current design



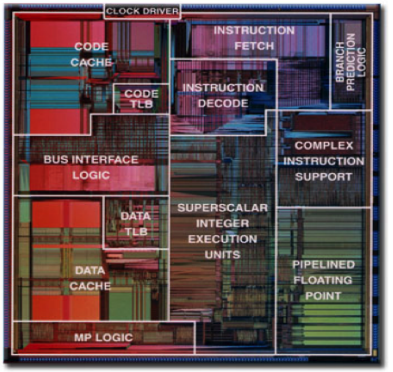


- «FPU», «*Floating Point Unit*» : gère les calculs sur nombre à virgule flottante ;
- «Prefetch Unit» : charger les données et instructions en amont de leur traitement pour accélerer le travail du processeur ;
- «Decode Unit» : analyse les instructions pour le processeur.

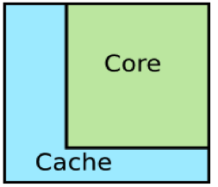
Et les multi-cores ? Qu'est-ce qu'un «core» ?



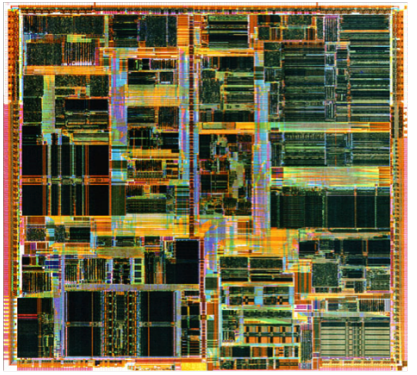
Pentium I

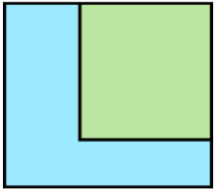


Chip area breakdown

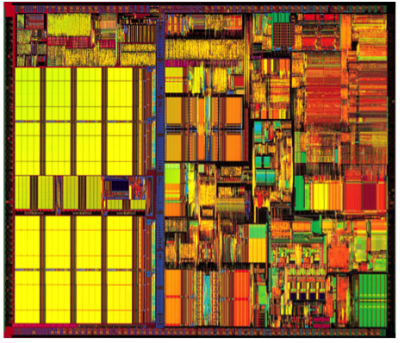


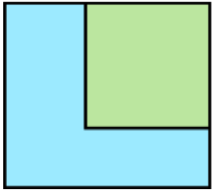
Pentium II



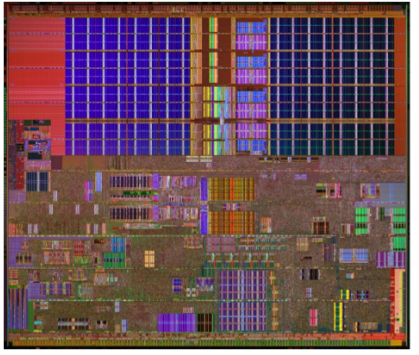


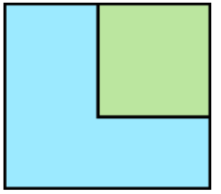
Pentium III





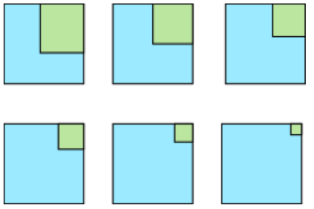
Pentium IV





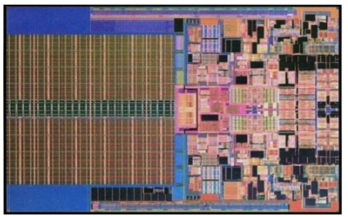
Et les multi-cores ?

L'avenir ?

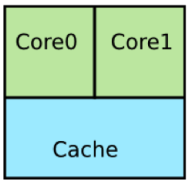


Non ! le multi-cores :

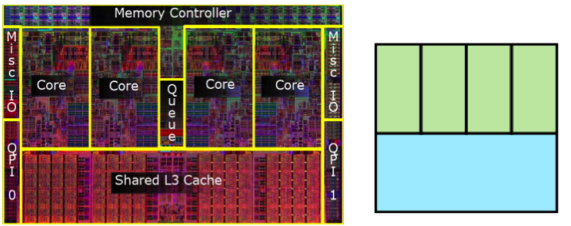
Penryn



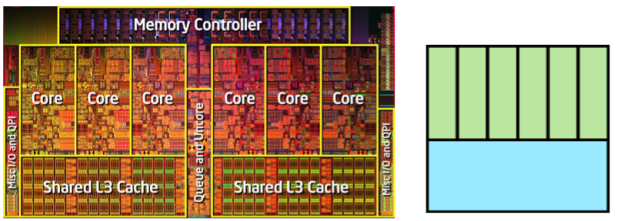
Chip area breakdown



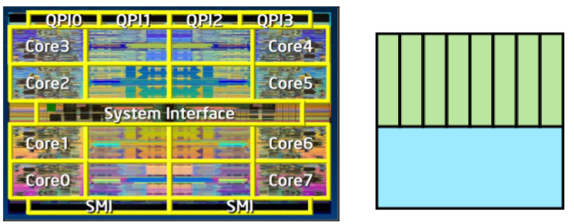
Bloomfield



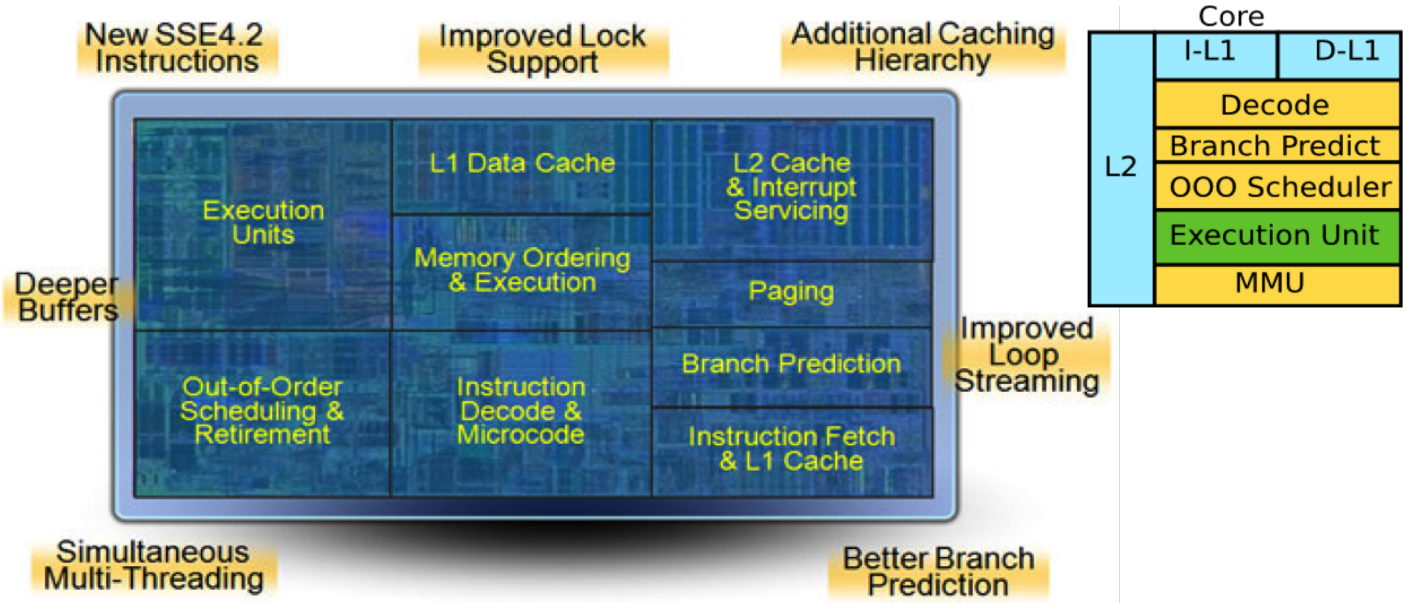
Gulftown



Beckton



De plus près

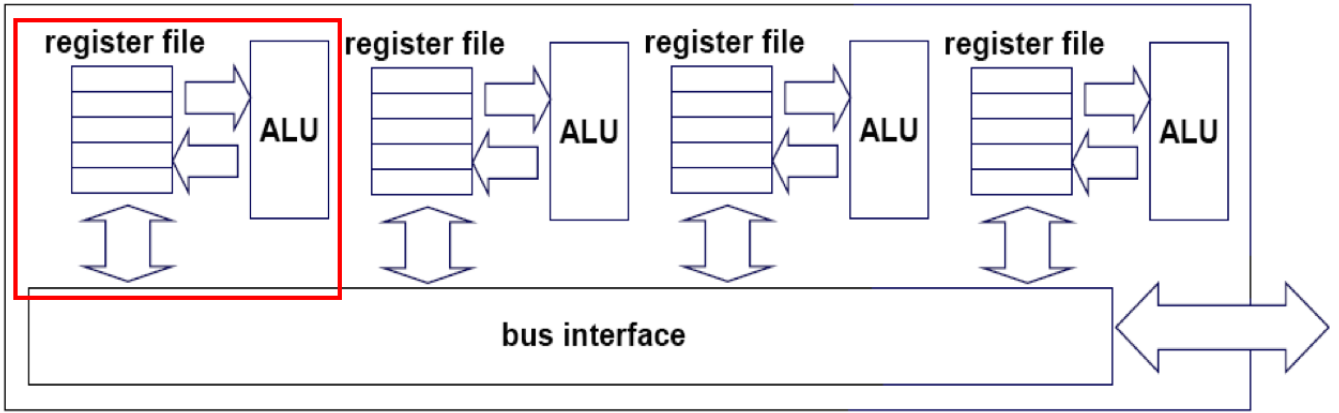


Moins de 10%de la surface sert à l'exécution réelle

«OOO» : Out of Order

Et les multi-cores ?

Processeur «multi-cores»

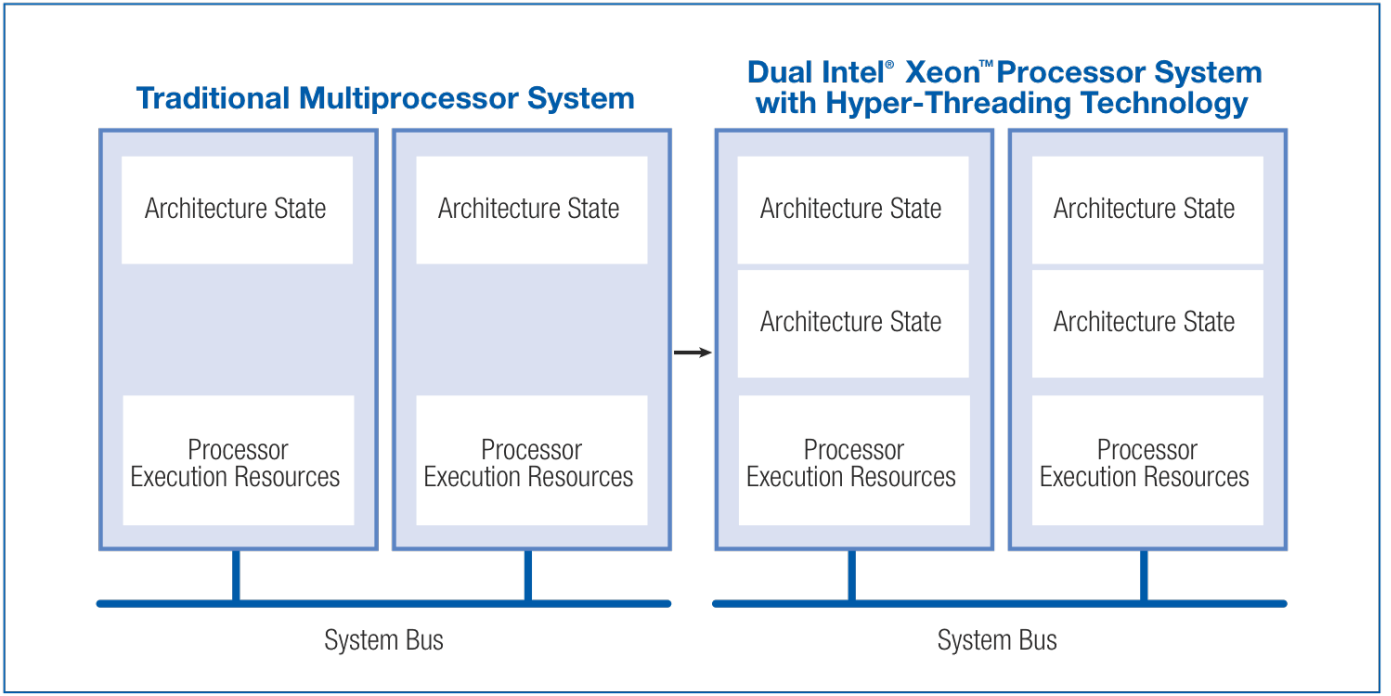


- ▷ On «grave» plusieurs processeurs sur le même support.
- ▷ Chaque cœur est vu par le système d'exploitation comme un processeur séparé.

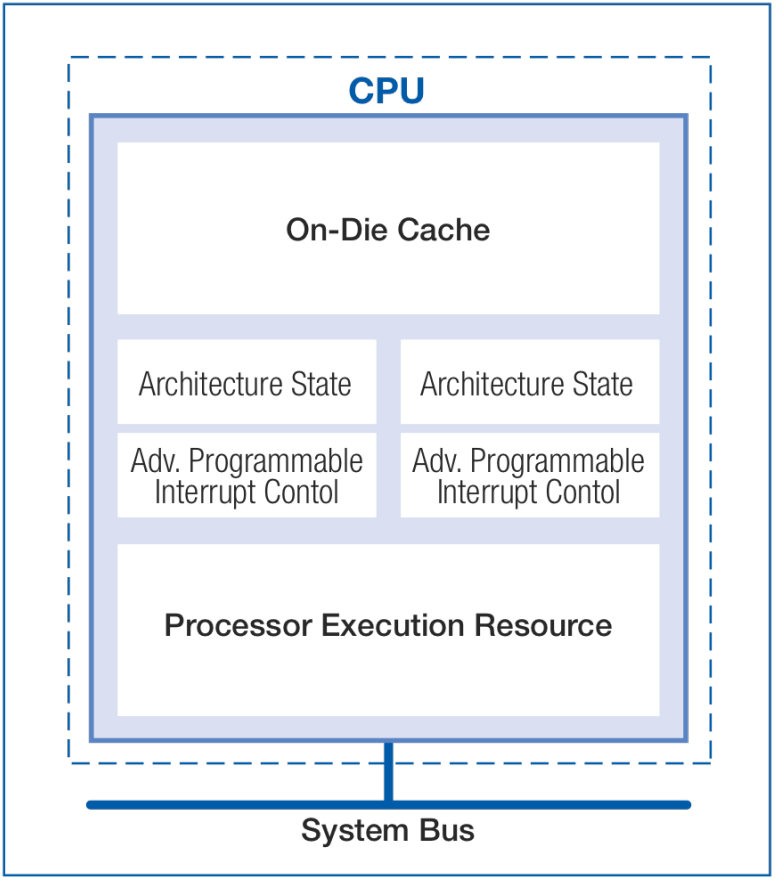
Avantages

- ☐ On **augmente moins la cadence** du processeur (échauffement, consommation, difficultés de conception)
- ☐ On va vers **plus de parallélisme** (bien !)

Et l'Hyperthreading ? Qu'est-ce que c'est ?



Une «Hyperthread»



Un processeur logique

- ▷ un «*architecture state*», c-à-d un état matériel :
  - ◇ registres, RI, CO, PSW, Interruptions ;
  - ◇ son propre flot d'instruction ;
- ▷ peut être interrompu et stoppé indépendamment.

Un processeur composé de «processeurs logiques»

Tous les processeurs logiques partagent la partie «exécution» :

- ☐ les caches mémoires ;
- ☐ les bus mémoires ;
- ☐ le CPU, ALU, FPU, *etc.*

D'après Intel

Each logical processor maintains a complete set of the architecture state. The architecture state consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers and some machine-state registers. From a software perspective, once the architecture state is duplicated, the processor appears to be two processors. The number of transistors to store the architecture state is an extremely small fraction of the total. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic and buses.

Programme *vs* Processus

Programme entité purement statique associée à la suite des instructions qui la composent.

Processus entité purement dynamique associée à la suite des actions réalisées par un programme.

- ◇ La notion de processus introduit implicitement le concept et le fonctionnement des *systèmes multiprogrammé*.
- ◇ Un processus est une abstraction de données définies par 2 parties : un **état** et un **comportement**.

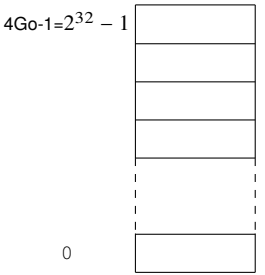
À chaque étape d'exécution du programme :

- ▷ le contenu des registres du compteur ordinal évolue ;
- ▷ le contenu de la mémoire centrale peut être modifié : écriture ou lecture.

*On appelle **processus** l'image de l'état du processeur et de la mémoire au cours de l'exécution d'un programme.*

Le programme est statique et le processus représente la dynamique de son exécution.

Un programme sur un adressage sur 32 bits voit la mémoire de l'adresse 0 à $2^{32} - 1$, c-à-d $4Go$:



Les segments :

- chaque section possède des droits d'accès différents : `read/write/execute`
- **Code** : les instructions du programme (appelé parfois «*Text*») ;
- **Data** : les variables globales initialisées ;
- **Bss** : les variables globales non initialisées (elles seront initialisées à 0) ;
- **Heap** : mémoire retournée lors d'allocation dynamique avec les fonctions «`malloc/calloc/new`» : *elle croît vers le haut* ;
- **Stack** : stocke les variables locales et les adresses de retour : *elle croît vers le bas* ;
- **Shared libraries** : les bibliothèques partagées, c-à-d le code des fonctions partagées par plusieurs processus (fonctions d'entrée/sortie, mathématiques, *etc.*).

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int global_init_data = 30;
int global_noinit_data;

void function_prints(void)
{ char c;
  int local_data = 1;
  int *dynamic_data = (int *)calloc(1,sizeof(int));

  printf("Pid of the process is = %d\n", getpid());
  printf("Addresses which fall into:\n");
  printf("1) Data  segment = %p\n", &global_init_data);
  printf("2) BSS   segment = %p\n", &global_noinit_data);
  printf("3) Code  segment = %p\n", &function_prints);
  printf("4) Stack segment = %p\n", &local_data);
  printf("5) Heap  segment = %p\n", dynamic_data);
  scanf("%c", &c);
}
```

```c
int main()
{
  function_prints();
  return 0;
}
```

xterm

```
$ ./segments
Pid of the process is = 4697
Addresses which fall into:
 1) Data  segment = 0x804a020
 2) BSS   segment = 0x804a02c
 3) Code  segment = 0x8048494
 4) Stack segment = 0xbfe47454
 5) Heap segment = 0x8d76008
```

xterm

```
$ more /proc/4697/maps
08048000-08049000 r-xp 00000000 08:01 424814   ./segments
08049000-0804a000 r--p 00000000 08:01 424814   ./segments
0804a000-0804b000 rw-p 00001000 08:01 424814   ./segments
08d76000-08d97000 rw-p 00000000 00:00 0        [heap]
bfe28000-bfe49000 rw-p 00000000 00:00 0        [stack]
```

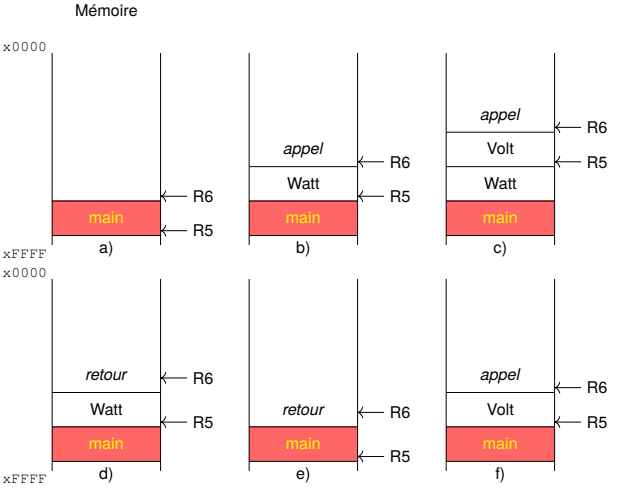
xterm

```
$ nm -f sysv segments
Symbols from segments:
Name                  Value   Class   Type     Size      Section
main                |08048558|   T  |     FUNC|00000012||.text
global_init_data    |0804a020|   D  |   OBJECT|00000004||.data
global_noinit_data  |0804a02c|   B  |   OBJECT|00000004||.bss
function_prints     |08048494|   T  |     FUNC|000000c4||.text
```

On remarque que sur la sortie du fichier «`maps`» :

- *le segment `code` possède les **droits d'exécution** ;*
- *les segments `bss`, `data` possèdent les droits de lecture/écriture mais **pas d'exécution**.*

```
 1 int main ()
 2 {
 3    int a = 23;
 4    int b = 14;
 5    ...
 6
 7    b = Watt(a); /* appel de Watt  */
 8    b = Volt(a,b); /* puis de Volt */
 9    ...
10 }
11
12 int Watt(int c)
13 {
14    int w = 5;
15    ...
16    w = Volt(w,10); /* appel Volt  */
17    ...
18    return w;
19 }
20
21 int Volt(int q, int r)
22 {
23    int k = 3;
24    int m = 6;
25    ...
26    return k+m;
27 }
```



a. l'état de la pile au démarrage du programme :
   - ◇ registre «R5», «*frame pointer*» : contient l'adresse du début du contexte courant (appelé «*frame*») ;
   - ◇ registre «R6», «*Top of Stack (TOS) pointer*» : contient l'adresse du sommet de la pile qui varie par empilement/dépilement ;

b. appel de la fonction «Watt» ;

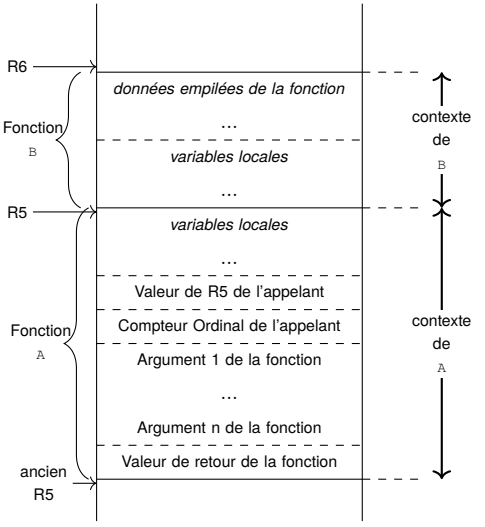
c. le registre R5 prend la valeur de R6 et pointe sur le «contexte» associé à l'exécution de la fonction, R6 sur le nouveau sommet qui va être décalé pour faire de la place aux arguments, valeur de retour et adresse de retour (empilement) ;

d. le processeur saute à l'adresse du code de la fonction appelée ;

e. lors du retour de la fonction, le processeur dépile l'adresse de retour et le registre R6 pointe sur la valeur de retour de la fonction ;

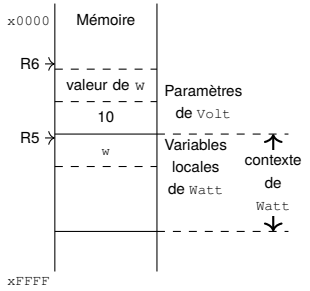
f. nettoyage de la pile pour l'exécution d'une autre fonction (dépilement valeur retour et arguments précédents) ;

g. ⟹**écrasement** de l'utilisation de la pile de l'ancien appel «Watt», par le nouvel de appel «Volt».



- soit l'état de la pile d'exécution après l'appel de la fonction A :
  - l'exécution de la fonction A crée un «contexte A» dans la pile :
    - la valeur de retour de la fonction A ;
    - les arguments : «*Argument 1 à n de la fonction*» ;
    - la valeur du «*PC*» ou «CO» de l'appelant de la fonction A ;
      Il est nécessaire de mémoriser l'ancienne valeur du CO, puisqu'il sert maintenant à exécuter le code de la fonction A.
    - la valeur du registre de R5 lors de l'exécution de l'appelant : obligatoire pour pouvoir remettre l'appelant dans un état correct lors du retour de la fonction ;
    - les variables locales à la fonction ;
    - les registres `R5` et `R6` contiennent maintenant des valeurs relatives au contexte de A ;

- ensuite, la fonction A appelle la fonction B :
  - l'appel de la fonction B crée le contexte de B dans la pile :
    - on empile les données de la fonction en rapport avec A (valeur de retour et arguments) ;
    - on empile le CO de l'appelant, c-à-d l'adresse de l'instruction de la fonction A à laquelle il faudra retourner (A ayant appelé B). On mets l'adresse du code de la fonction B dans le CO, ce qui force l'exécution de la fonction B.
- lors de l'exécution de B, on empile la valeur de R5 pour pouvoir la restaurer et on empile les variables locales de B ;
- cela crée une «liste chaînée» des différents **appels imbriqués** : il est possible de remonter dans les contextes d'appels en examinant la pile. *Les outils de «déboguage» permettent de visualiser et analyser le contenu de la pile d'appel*.
  Le mécanisme «d'exception» présent dans le langage C++, Java, Python etc. permet en cas d'erreur de remonter de contexte de fonction en contexte de fonction appelante à la recherche d'une fonction capable de traiter cette erreur.

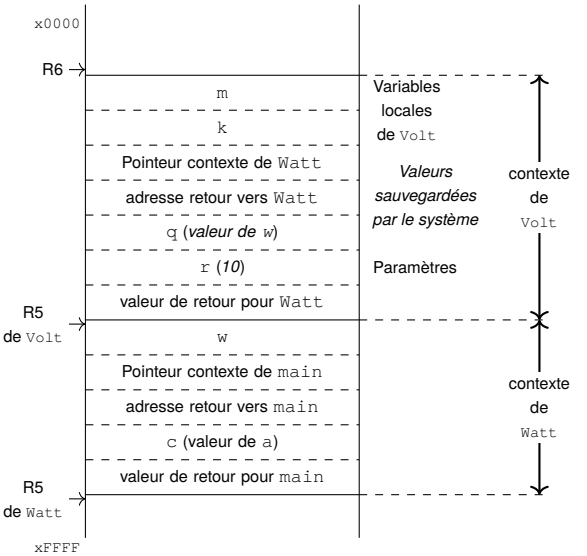


Lors de l'appel de la fonction «Volt», par la fonction «Watt», la valeur contenue dans la variable locale «w» de «Watt» est transmise en tant que paramètre à la fonction «Volt».

Cet appel correspond à la ligne 16 du code source.

Dans le contexte de la fonction «Volt» :

- la variable locale «q» reçoit la valeur du paramètre transmis, c-à-d la valeur de «w» de la fonction «Watt» ;
- la variable locale «r» reçoit la valeur du paramètre transmis, c-à-d la valeur directe 10 ;
- pour son travail, elle utilise deux variables locales «k» et «m» ;
- des sauvegardes des adresses concernant «Watt» :
  - de retour d'exécution ;
  - de **restauration de contexte**.



L'image de la course de voiture

Plusieurs véhicules veulent aller d'un point A à un point B le plus vite possible, ils peuvent :

- faire la course sur la route et finir par :
  - soit se **suivre** les uns les autres ;
  - soit essayer de se **voler** mutuellement leurs positions respectives ;
  - soit avoir un **accident** !
- rouler sur différentes voies parallèles et arrivés ensemble s**ans entrer en collision** ;
- emprunter des **routes différentes** pour aller de A à B.

Et le parallélisme ?

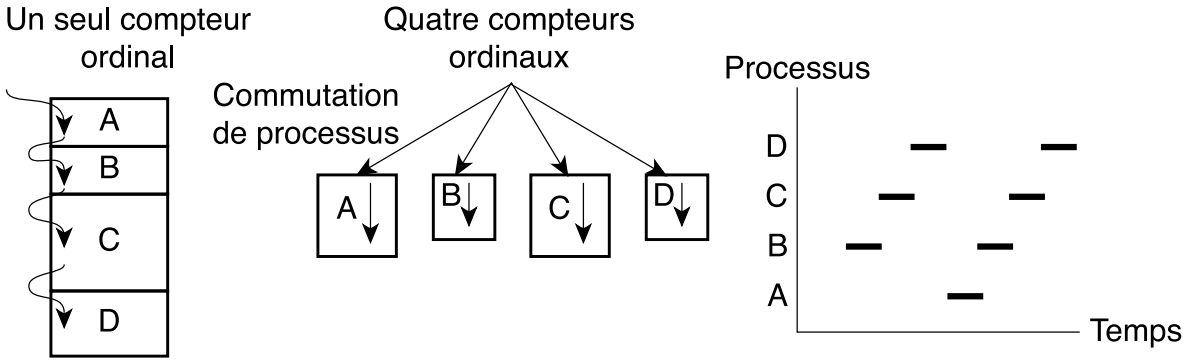
- Plusieurs tâches à réaliser : chaque voiture à acheminer ;
- Chacune de ses tâches peut s'exécuter :
  - une à la fois sur un **seul processeur** : une **seule route** ;
  - en paralléle sur **plusieurs processeurs** : **plusieurs voies** sur la même route ;
  - de manière **distribuées** sur plusieurs processeurs : des **routes séparées**.
- Ces tâches nécessitent souvent d'être **synchronisées** pour éviter les collisions ou de **s'arrêter** à des feux de trafic ou bien à des panneaux de signalisation (Stop).

On peut imaginer que :

- les voitures sont des **processus** ou **threads** ;
- les routes qu'elles veulent emprunter sont des **applications** ;
- la carte des routes correspond au **matériel** ;
- et le code de la route, aux **communications** et aux **synchronisations**.

Comment exécuter plusieurs programmes en «multi-tâche»

- charger différents programmes en mémoires :
  - répartir la mémoire entre les différents programmes ;
  - traduire les adresses *etc.*
- passer, «*switcher*», de l'exécution d'un programme à l'autre :
  - associer un contexte du processeur à chaque programme ⟶ notion de **processus** ;
  - passer d'un processus à un autre : sauvegarde un contexte et restaurer un autre contexte ;
  - passer d'un «CO» associé à un processus à un «CO» associé à un autre processus :



Comment organiser la mémoire entre les différents processus ?

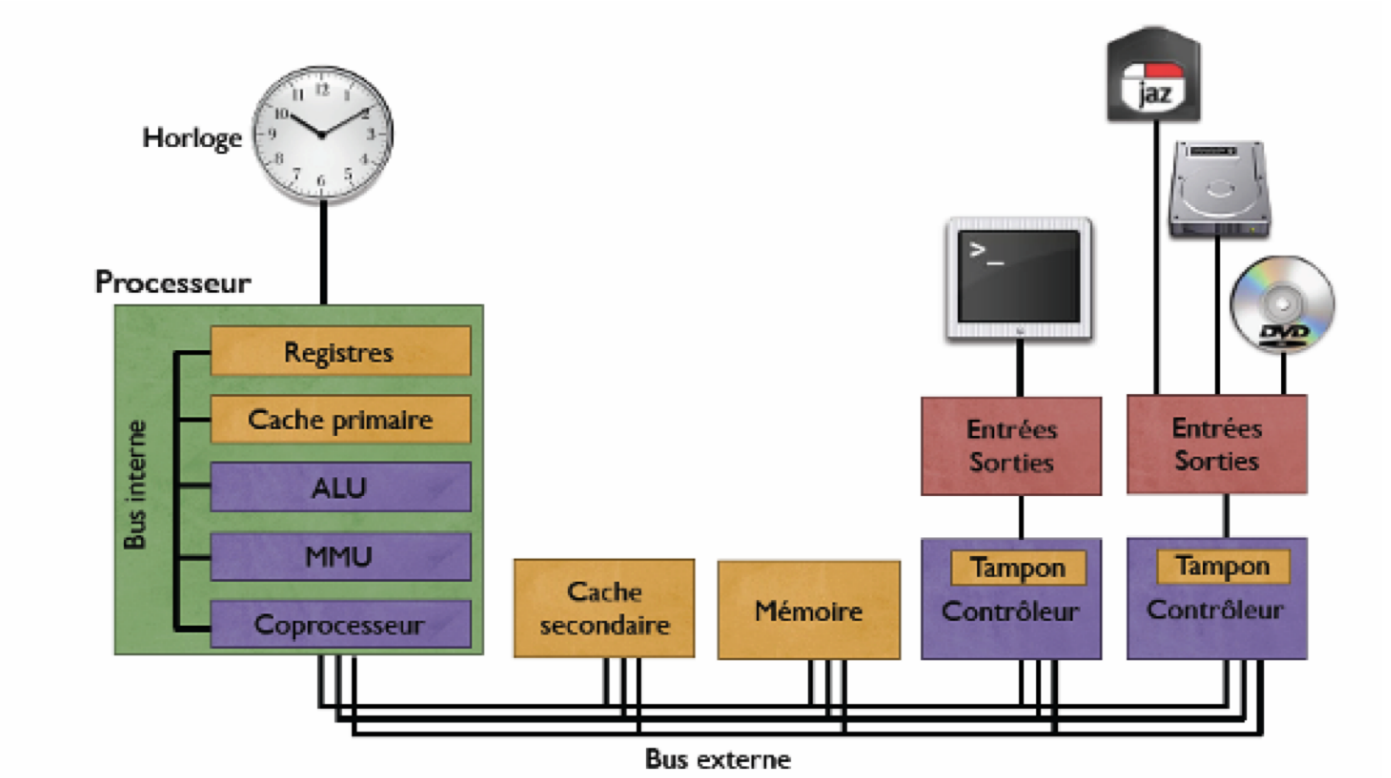
En utilisation le concept de «pagination».

Comment passer règulièrement et automatiquement d'un processus à l'autre ?

En utilisant le concept «d'horloge» et d'«interruption».

Représentation fonctionnelle d'un ordinateur

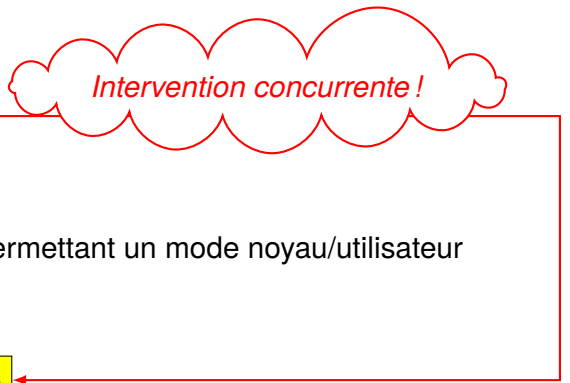
On ajoute dans cette représentation fonctionnelle, une horloge et des contrôleurs de périphériques



Représentation fonctionnelle d'un ordinateur

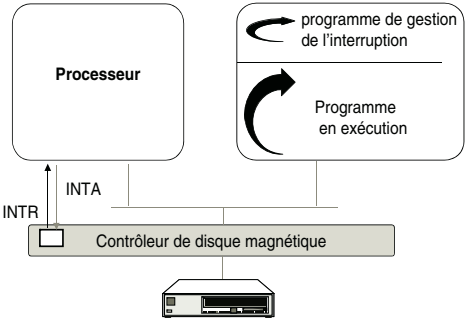
Les composants indispensables d'un système capable de supporter Unix :

- La mémoire principale
- La mémoire cache
- Les contrôleurs de périphériques
- Les bus de communication
- Une horloge
- Un processeur permettant un mode noyau/utilisateur
- Les interruptions



- permet d'arrêter l'exécution du programme en cours afin d'exécuter une tâche jugée plus urgente :
  «*Un périphérique peut signaler un événement important (fin de papier dans l'imprimante, dépassement de température signalée par une sonde placée dans un four...) au processeur qui, s'il accepte cette interruption, exécute un programme de service (dit programme d'interruption) traitant l'événement.*
  À la fin du programme d'interruption le programme interrompu reprend son exécution normale.»

- il en existe trois sortes provenant :
  - **d'un périphérique** : interruptions externes qui permettent à un périphérique d'avertir le processeur ;



  - **d'un programme en cours d'exécution** : interruptions logicielles internes nommées **appels systèmes**.
    Il s'agit de permettre à un programme en cours d'exécution de se dérouter vers un programme du système d'exploitation qui doit gérer une tâche particulière.
    Par exemple les instructions d'entrées/sorties, permettant les échanges d'informations entre le processeur et les périphériques, sont traitées de cette manière :
    un ordre d'entrées/sorties est un appel au système (une interruption logicielle) qui interrompt le programme en cours au profit du programme spécifique (driver ou pilote) de gestion d'un périphérique.

  - **du processeur** : traiter des événements exceptionnels de type division par zéro, dépassement de capacité

Principe

commutation de contexte provoquée par un **signal géré par le matériel**.

Ce **signal** est lui-même un événement qui peut être :

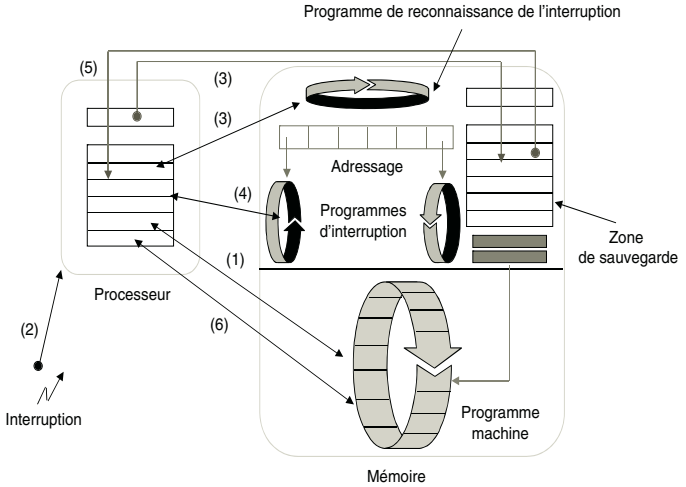
- ◇ interne au processus et donc résultant de son exécution ;
- ◇ extérieur indépendant de cette exécution.

Exemple

- ◇ L'événement concerne 1 périphérique d'entrée / sortie.
- ◇ L'événement concerne 1 utilisateur ou un opérateur.

Le signal :

- ▷ réalise un **changement de contexte processeur** lorsqu'il est matériel ;
- ▷ modifie un indicateur qui est **consulté régulièrement** par le système d'exploitation en vue de déterminer la cause de l'interruption lorsqu'il est logiciel.

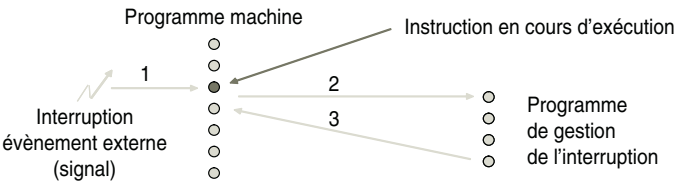


1. le programme utilisateur dispose du processeur. C'est lui qui est en cours d'exécution. Il dispose des registres, de l'unité arithmétique et logique, de l'unité de commande. Le compteur ordinal CO contient l'adresse de la prochaine instruction à exécuter ;
2. une interruption est déclenchée par un périphérique ;
3. sauvegarde du contexte matériel d'exécution du programme utilisateur et du compteur ordinal CO.
   Le programme de reconnaissance s'exécute et lit le numéro de l'interruption.
   Le numéro de l'interruption permet l'identification de l'adresse du programme de traitement ;
4. le compteur ordinal CO est chargé avec l'adresse du programme de traitement et celui-ci s'exécute ;
5. le contexte d'exécution du programme utilisateur est rechargé dans le processeur à la fin du programme d'interruption ;
6. le programme utilisateur reprend son exécution.

En version simplifiée :

Matériel — Prise en compte par le processeur d'événements externes (exemple : les périphériques positionnent un signal qui est reçu par le processeur).

Logiciel



Gestion logicielle de l'horloge

L'horloge est gérée par un «pilote d'horloge» pour fournir les services suivants :

- mettre à jour l'heure courante ;
- empêcher les processus de dépasser le temps qui leur est alloué ;
- comptabiliser l'allocation du processeur ;
- traiter l'appel système `alarm()` des processus utilisateur ;
- fournir des compteurs de garde au système lui-même ;
- fournir diverses informations au système (tracé d'exécution, statistiques).

Différentes horloges

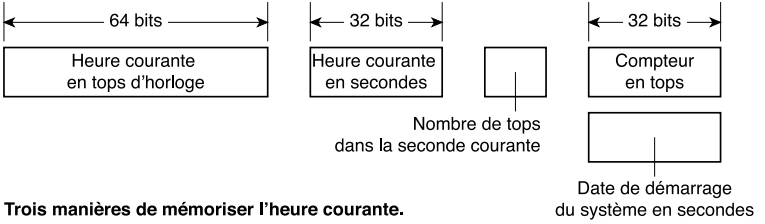
⋆ **horloge «temps réel»** : maintenir la date et l'heure «humaine».

▷ une date de référence : le 1er janvier 1970 ;

▷ un compteur exprimant le nombre de «tops» depuis la date de référence ;

▷ capacité du compteur : dépend du nombre de bits du compteur et de la fréquence de l'horloge ;
Exemple : un compteur sur 32bits, avec une horloge à $60Hz$ déborde en 2 ans…

Solutions :



Trois manières de mémoriser l'heure courante.

◇ compteur sur 64bits : augmente la complexité du calcul qui doit être réalisé de nombreuses fois par seconde ;

◇ compteur sur 32 bits par seconde + compteur supplémentaire «tops → seconde»
Solution retenue pour Unix : dépassement de capacité en 2038 !

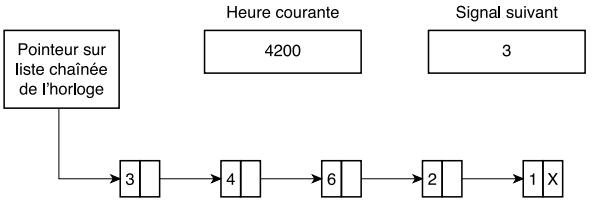
◇ compteur de tops à partir de l'heure du démarrage récupérée automatique (utilisateur ou réseau).

Autres services

- ⋆ **Ordonnanceur** : garantie qu'un processus ne s'exécute pas plus logntemps que son «quantum de temps» :
  - ◇ lorsqu'un processus démarre **un compteur lui est associé** avec un nombre de «tops» d'horloge ;
  - ◇ à chaque interruption de l'horloge **ce compteur est décrémenté** ;
  - ◇ lorsque le compteur est à zéro, **l'ordonnanceur est rappelé** et choisi **un nouveau processus**.
- ⋆ **Temps d'allocation d'un processus** : comptabiliser le temps alloué à chacun des processus :
  - ◇ on incrémente un compteur associé au processus à chaque top d'horloge ;
  - ◇ on distingue le temps passé en mode noyau et en mode utilisateur au service de ce processus : *on indique également le temps réel attendu par l'utilisateur devant son écran.*

```
pef@darkstar:~$ time ls
Segments  segments.c

real  0m0.010s
user  0m0.004s
sys 0m0.004s
```

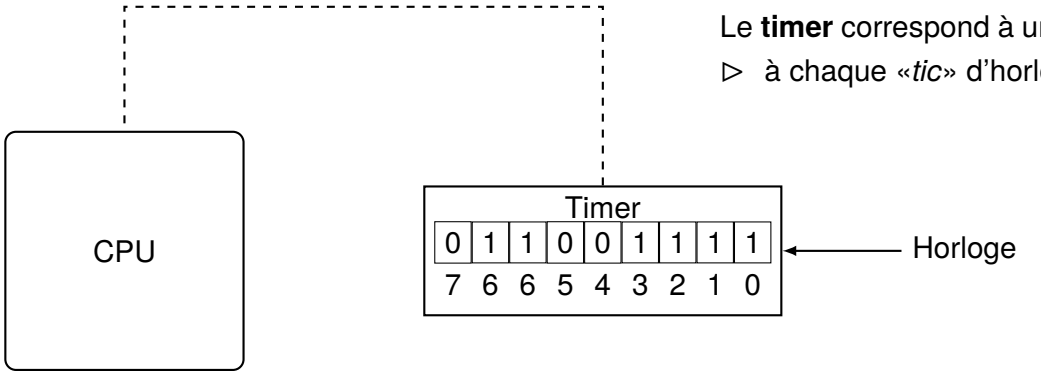
- ⋆ **Alarmes** : chaque processus peut demander d'être alerté après un certain temps ou à une certaine heure.



Simulation de plusieurs compteurs à partir d'une horloge unique

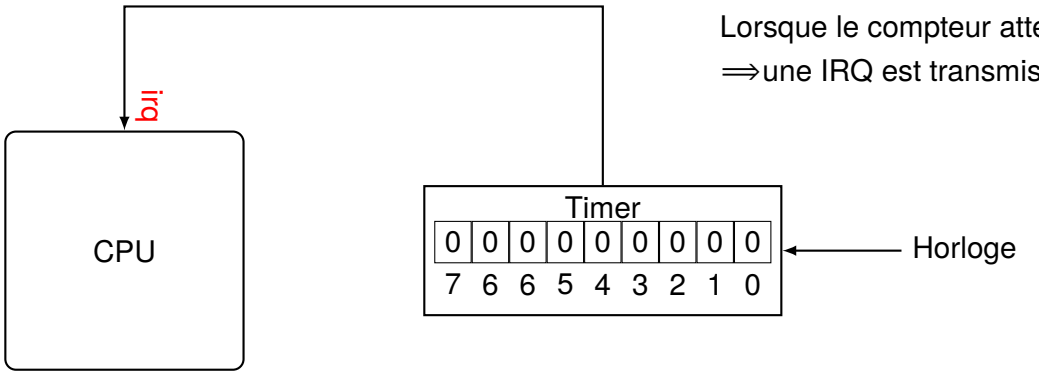
Optimisation : une seule horloge + une liste chaînée d'alarmes qui mémorise les délais entre chacune de ces alarmes : *sur la figure les signaux vont être envoyés pour* 4203, 4207, 4213, 4215 *et* 4216

Interruption + Timer $\Longrightarrow$ changement de contexte



Le **timer** correspond à un compteur **autonome** :

▷ à chaque «*tic*» d'horloge il est décrémenté.



Lorsque le compteur atteint la valeur zéro

$\Longrightarrow$une IRQ est transmise au CPU.

À chaque **interruption**, IRQ, déclenchée par le *timer*, le CPU «*saute*» vers le code de l'OS.

Le Système d'exploitation Unix

Unix est un système supportant :

- Plusieurs utilisateurs ;
- Plusieurs programmes :
  - Partage du temps d'utilisation ;
  - Protection de la mémoire ;
  - Partage des ressources ;
  - Accès à distance.
- Problèmes :
  - Équité ;
  - Gestion du matériel ;
  - Droits d'accès…

Unix est un système nécessitant un processeur récent disposant d'au moins **deux modes de fonctionnement** pour lui permettre de contrôler les accès aux ressources de la machine :

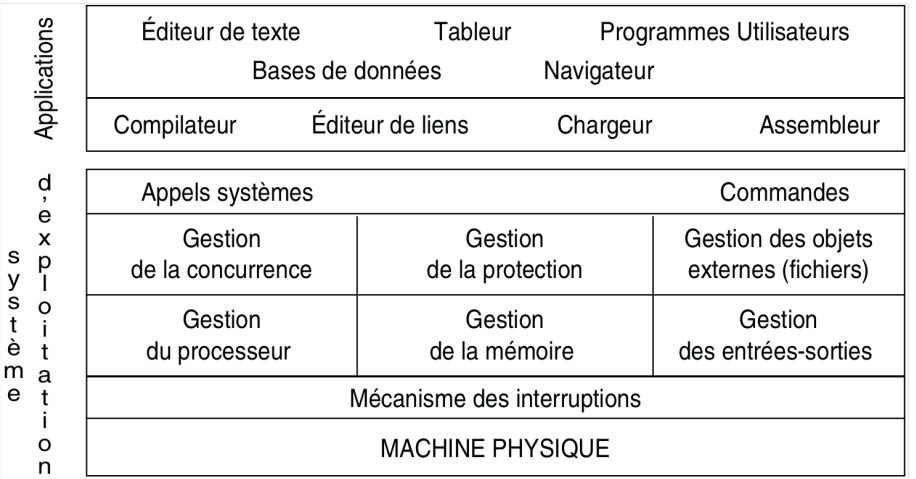
- un mode **noyau** (ou système ou superviseur)
  le processeur peut exécuter toutes les instructions disponibles du système.
- un mode **utilisateur**.
  certaines instructions lui sont interdites, pour des raisons de sécurité du système.

Protection fondamentale

Les programmes sont exécutés en mode utilisateur :

- obligation de faire appel au système d'exploitation pour les opérations à risque qui nécessite de passer en mode noyau.
  On appelle ces opérations des ***appels système*** *(exemple : opérations de gestion de fichier pour modification des données)*
- **Tous les accès aux ressources et aux données sont contrôlés par le système d'exploitation**.

Le système d'exploitation se présente comme une couche logicielle placée entre la machine matérielle et les applications.



Il s'interface avec :

- la couche matérielle, notamment par le biais du mécanisme des interruptions.
- les applications par le biais des primitives qu'il offre : appels système et commandes.

- **Gestion du processeur** : allouer le processeur aux différents programmes pouvant s'exécuter.
  Cette allocation se fait suivant un algorithme d'ordonnancement qui planifie de l'exécution des programmes.
- **Gestion de la mémoire** : allouer la mémoire centrale entre les différents programmes pouvant s'exécuter (pagination/segmentation).
  Mécanisme de **mémoire virtuelle** : *traduction entre adresse logique et physique, chargement uniquement des parties de code et de données utiles à l'exécution d'un programme.*
- **Gestion des E/S** :
  - accès aux périphériques,
  - faire la liaison entre appels de haut niveau des programmes (exemple : `getchar()`) et les opérations de bas niveau du système d'exploitation responsable du périphérique (exemple : le clavier).

  C'est le pilote d'E/S (driver) qui assure cette liaison.
- **Gestion de la concurrence** : plusieurs programmes coexistent en mémoire centrale :
  - assurer les besoins de communication pour échanger des données ;
  - synchroniser l'accès aux données partagées afin de maintenir la cohérence de ces données.

  Le système offre des outils de communication et de synchronisation entre les programmes.
- **Gestion du système de fichier** : stockage des données sur mémoire de masse (disque dur, mémoire flash, SSD, *etc.*)
  *Le système d'exploitation gère un **format d'organisation** (ext4 sous Linux), un **système de protection** (journalisation des modifications) et un **format de fichier** (inodes).*
- **Gestion de la protection** : mécanisme de garantie que les ressources (CPU, mémoire, fichiers) ne peuvent être utilisées que par les programmes disposant des droits nécessaires.
- Gestion des **comptes** et des **droits** ,
- **Protection du système d'exploitation** : séparation du système d'exploitation en mode noyau et des programmes en mode utilisateur.

Un **programme** est une entité purement statique associée à la suite des instructions qui la composent (le ou les fichiers stockés sur le disque dur).

Un **processus** est un programme en cours d'exécution auquel est associé :

- un environnement processeur (CO, PSW, RSP, registres généraux) ;
- un environnement mémoire appelés contexte du processus.

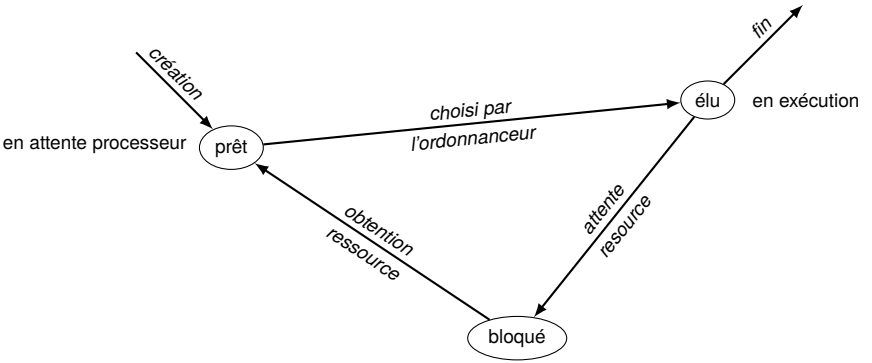
Un processus est :

- l'«instance dynamique» d'un programme ;
- le **fil d 'exécution**, thread, de celui-ci dans un espace d'adressage protégé (ensemble des instructions et des données accessibles en mémoire).

État d'un processus

C'est le système d'exploitation qui détermine et modifie l'état d'un processus sous l'effet des événements :

- le choix de l'**ordre d'exécution** des processus est appelé ordonnancement ;
- l'algorithme chargé de faire ce choix est appelé **ordonnanceur**.



État élu

Lors de son exécution, un processus est caractérisé par un état :

- lorsque le processus obtient le processeur et s'exécute, il est dans l'état **élu**.
- l'état **élu** est l'état d'exécution du processus.

État bloqué

Lors de cette exécution, le processus peut demander à **accéder à une ressource** (réalisation d'une entrée/sortie, accès à une variable protégée) qui n'est **pas immédiatement disponible** :

- le processus **ne peut pas poursuivre** son exécution tant qu'il n'a pas obtenu la ressource (par exemple, le processus doit attendre la fin de l'entrée-sortie qui lui délivre les données sur lesquelles il réalise les calculs suivants dans son code) ;
- le processus *quitte* alors le processeur et passe dans l'état **bloqué** : l'état bloqué est donc l'état d'attente d'une ressource autre que le processeur.

État prêt

Lorsque le processus a enfin obtenu la ressource qu'il attendait, celui-ci peut potentiellement reprendre son exécution. Cependant, dans le cadre de **systèmes multiprogrammés** , il y a plusieurs programmes en mémoire centrale, et donc plusieurs processus.

- lorsque le processus est passé dans l'état bloqué, le processeur a été **alloué** à un autre processus.
- le processeur n'est donc pas forcément libre : le processus passe alors dans l'état **prêt** .

L'état «Prêt» est l'état **d'attente** du processeur.

Changement d'état

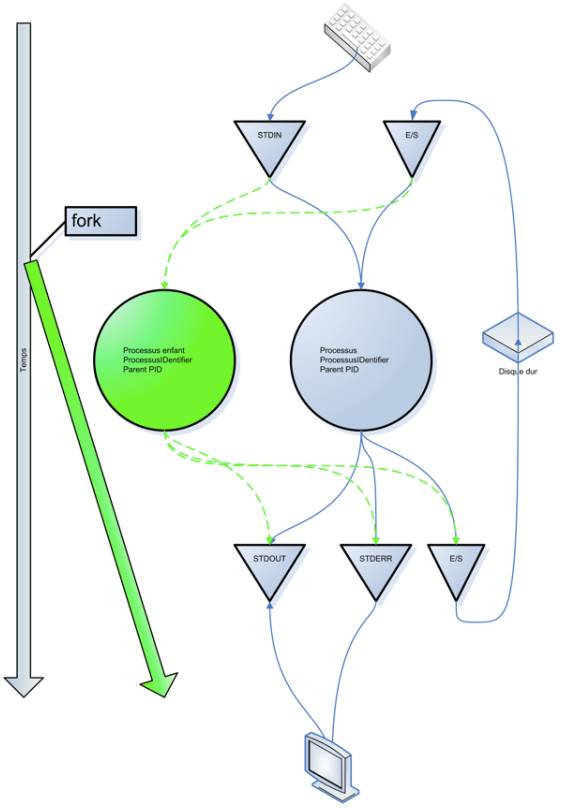
- ▷ Le passage de l'état prêt vers l'état élu constitue l'opération **d'élection** .
- ▷ Le passage de l'état élu vers l'état bloqué est l'opération de **blocage** .
- ▷ Le passage de l'état bloqué vers l'état prêt est l'opération de **déblocage** .

Remarques

- ☐ Un processus est toujours crée dans l'état **prêt** .
- ☐ Un processus se termine toujours à partir de **l'état élu** (sauf anomalie).

Création

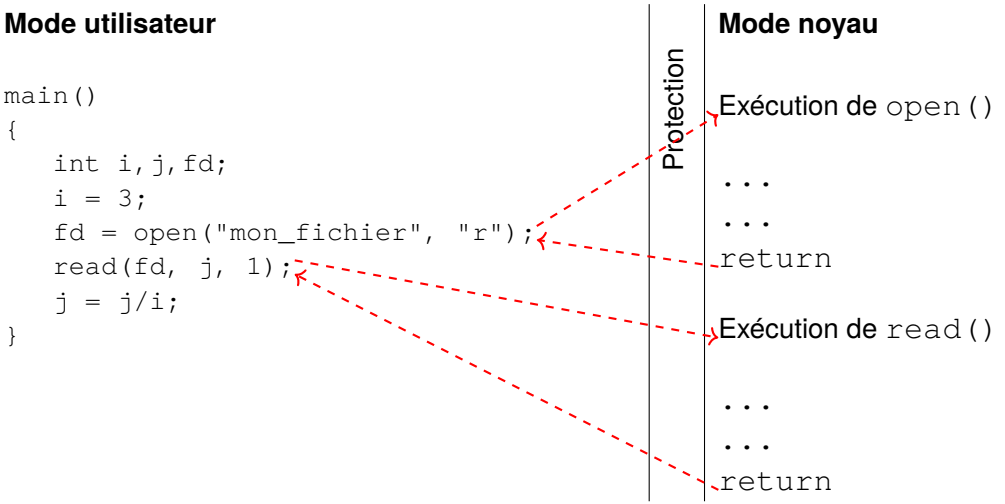
a. «clonage» du processus père.
   Le premier processus père : le Shell.
b. le nouveau processus partage ses E/S avec son père :
   - ◇ stdin, stdout, stderr ;
     la lecture de l'entrée se fait depuis le shell, les sorties se font dans le shell
   - ◇ tous les fichiers ouverts
     les fichiers disques, les tubes de communication
   - ◇
c. Possède le même code que son père :
   - ◇ différence de valeur de retour du fork :
     - ⋆ 0 pour le processus fils ;
     - ⋆ pid du fils pour le processus père ;
   - ◇ recouvrement du code par un autre avec «`exec()`



Les **appels systèmes** constituent l'interface du système d'exploitation et sont les **points d'entrées** permettant l'exécution d'une fonction du système :

- les **appels système** sont directement appelables depuis un programme.
- les **commandes** permettent d'appeler les fonctions du système depuis le prompt de l'interpréteur de commande (shell, «*Command Line Interface*»)

Déroulement d'un appel système



Lors de l'exécution d'un appel système, le programme utilisateur passe du mode d'exécution utilisateur au mode d'exécution **superviseur** ou **privilégié**, appelé «**mode noyau**».

Mode utilisateur

```
main()
{
    int i,j,fd;
    i = 3;
    fd = open("mon_fichier", "r");
```

Protection

Mode noyau

Exécution de `open()`

```
...
...

1 sauvegarde CO, PSW utilisateur
2 chargement CO ⟵ adresse de la fonction open
3 chargement PSW ⟵ mode superviseur
```

- **appel système** :demande d'exécution d'une fonction du système d'exploitation ;
- passage du **mode utilisateur** au **mode noyau** :
  - déclenchement d'une **interruption logicielle** visible dans le code assembleur du programme ;
  - passage dans un mode d'**exécution privilégié** qui est le mode d'exécution du système d'exploitation (mode noyau ou superviseur).
    Accès à un plus grand nombre d'instructions machine que le mode utilisateur (permet l'exécution des instructions de masquage et démasquage des interruptions interdites en mode utilisateur).
  - **sauvegarde du contexte** utilisateur ;

⟹**commutation de contexte**

Exemple Assembleur dans le système Linux

Assembleur x86, sous Linux, écrit pour le compilateur NASM

```
section .data
        helloMsg:       db 'Hello world!',10
        helloSize:      equ $-helloMsg
section .text
        global  _start
 start:
        mov eax,4               ; Appel système "write" (sys_write)
        mov ebx,1               ; File descriptor, 1 pour STDOUT (sortie standard)
        mov ecx,helloMsg        ; Adresse de la chaine a afficher
        mov edx,helloSize       ; Taille de la chaine
        int 80h                 ; Execution de l'appel système
        ; Sortie du programme
        mov eax,1               ; Appel système "exit"
        mov ebx,0               ; Code de retour
        int 80h
```

Appel au noyau de LINUX (int 80h)

Mode utilisateur

```
main()
{
   int i,j,fd;
   i = 3;
   fd = open("mon_fichier", "r");
```

Protection

Mode noyau

Exécution de `open()`

```
...
...
return
```

```
1 restauration du contexte utilisateur
2 chargement CO ⟵ CO sauvegardé
3 chargement PSW ⟵ PSW sauvegardé
```

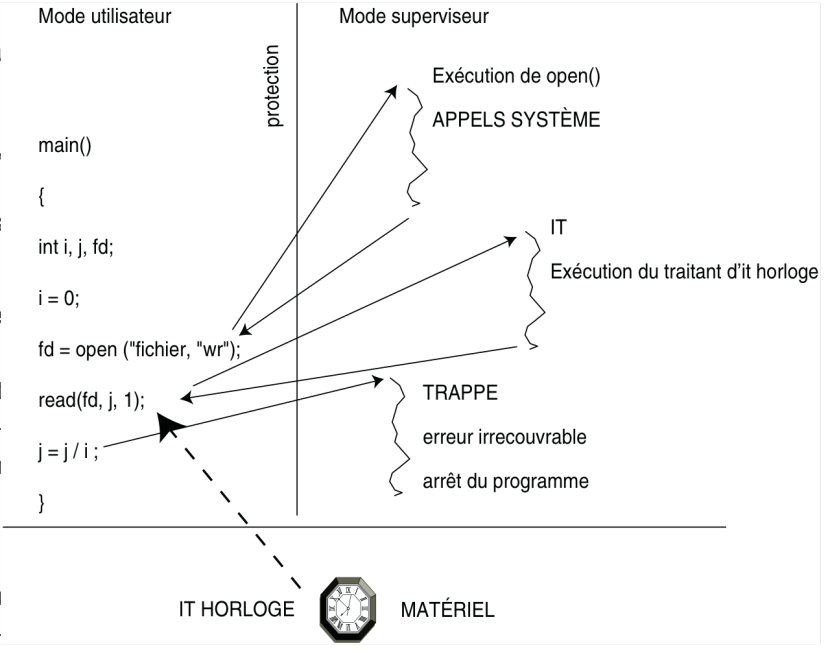
À la fin de l'**appel système** :

▷ passage du **mode noyau** au **mode utilisateur** ;

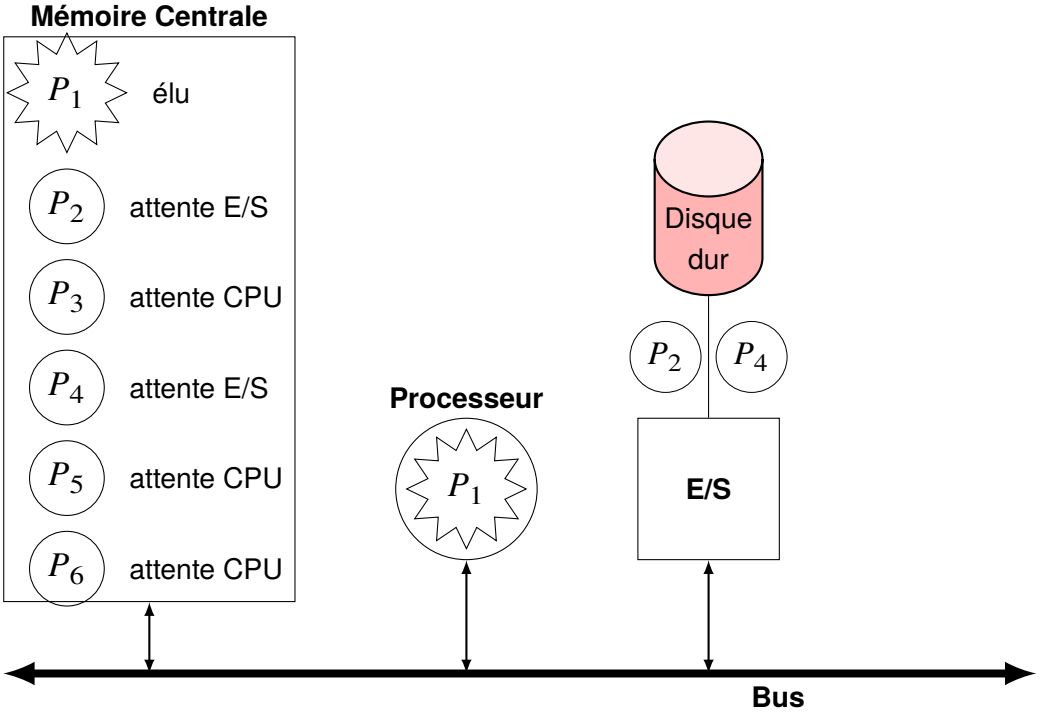
⟹ **commutation de contexte** : restauration du contexte utilisateur.

Trois causes de passage du mode utilisateur au mode noyau

- **appel d'une fonction du système** :
  demande explicite de passage en mode noyau
- **exécution d'une opération illicite**
  (division par 0, instruction machine interdite, violation mémoire…) : trappe.
  L'exécution du programme utilisateur est alors arrêtée.
- **interruption par le matériel et le système d'exploitation** :
  le programme utilisateur est alors stoppé et l'exécution de la routine d'interruption associée à l'interruption survenue est exécutée en mode superviseur.
- **interruption par l'horloge** :
  c'est une interruption matérielle qui permet au système d'exploitation de passer de l'exécution d'un processus à un autre dans le cadre de la «multiprogrammation».
  Cette interruption ne peut être ignorée par le processus.
  On appelle ce mécanisme : la **préemption**.



Préemption : commutation par l'horloge

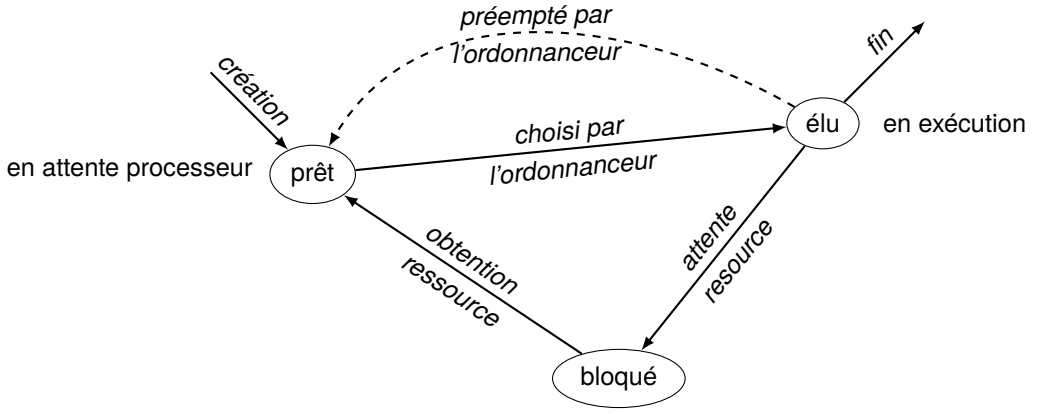


Dans ce schéma, plusieurs processus sont présents en mémoire centrale.

- P1 est élu et s'exécute sur le processeur.
- P2 et P4 sont dans l'état bloqué car ils attentent tous les deux une fin d'entrée- sortie avec le disque géréé par l'unité d'échange (UE).
- P3, P5 et P6 quant à eux sont dans l'état prêt : ils pourraient s'exécuter (ils ont à leur disposition toutes les ressources nécessaires) mais ils ne le peuvent pas car le processeur est occupé par P1.

Lorsque P1 quittera le processeur parce qu'il a terminé son exécution, les trois processus P3, P5 et P6 auront tous les trois le droit d'obtenir le processeur.
Mais le processeur ne peut être alloué qu'à un seul processus à la fois : il faudra donc choisir entre P3, P5 et P6.

C'est le rôle de **l'ordonnancement** qui **élira** un des trois processus.



L'automate montre les transitions pouvant exister entre l'**état prêt** (état d'attente du processeur) et l'**état élu** (état d'occupation du processeur).

Passage de :

- ▷ l'**état prêt** vers l'**état élu** constitue l'opération **d'élection** : allocation du processeur à un des processus prêts.
- ▷ l'**état élu** vers l'**état prêt** : **réquisition** du processeur, c-à-d que le processeur est retiré au processus élu alors que celui-ci dispose de toutes les ressources nécessaires à la poursuite de son exécution.

Cette réquisition porte le nom de **préemption**.

Ordonnancement préemptif ou non préemptif

- □ **non préemptif** ou **coopératif** : la transition de l'état élu vers l'état prêt est **interdite** : un processus quitte le processeur s'il a terminé son exécution ou s'il se bloque.
- □ **préemptif** : la transition de l'état élu vers l'état prêt est **autorisée** : un processus quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

Obtention du PID, «*Processus IDentifier*»

```
#include <unistd.h>

pid_t getpid(void) // retourne le pid du processus appelant
pid_t getppid(void) // retourne le pid du père du processus appelant
```

Obtenir des informations sur les processus

La commande `ps` : retourne la liste des processus avec leur caractéristiques (pid, ppid, état, terminal, durée d'exécution, commande associée...)

xterm

```
pef@cube:~$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1000  3889  3880  20   0  22840  4988 poll_s Ss+  pts/0      0:00 bash
0  1000 25566 25565  20   0  23120  5888 poll_s Ss+  pts/2      0:01 -bash
0  1000 29150 25566  20   0  54772  9300 signal T    pts/2      0:06 vi build_architecture
0  1000 30199 30198  20   0  22772  5252 wait   Ss   pts/1      0:00 -bash
0  1000 30772 30770  20   0  22900  5672 poll_s Ss+  pts/3      0:00 -bash
0  1000 32705 30199  20   0  31344  1580 -      R+   pts/1      0:00 ps l
```

Envoyer un signal à un processus

xterm

```
pef@cube:~$ kill -9 29150
```

Le signal nº9 correspond à `SIGKILL` : il termine de immédiatement le processus donné par son PID.

xterm

```
pef@cube:~$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1000  3889  3880  20   0  22840  4988 poll_s Ss+  pts/0      0:00 bash
0  1000 25566 25565  20   0  23120  5888 poll_s Ss+  pts/2      0:01 -bash
0  1000 30199 30198  20   0  22776  5256 wait   Ss   pts/1      0:00 -bash
0  1000 30772 30770  20   0  22900  5672 poll_s Ss+  pts/3      0:00 -bash
0  1000 32733 30199  20   0  31344  1604 -      R+   pts/1      0:00 ps l
```

Création de processus

```
#include <unistd.h>

pid_t fork(void)
```

La primitive `fork()` permet la **création dynamique** d'un nouveau processus qui s'exécute de manière concurrente avec le processus qui l'a créé :

- Tout processus Unix/Linux hormis le processus 0 est crée à l 'aide de cette primitive.
- Le processus créateur (le père) crée un processus fils par un appel à la primitive `fork()` qui est une **copie exacte** de lui-même (code et données)

Déroulement de l'appel système `fork`

MODE UTILISATEUR

PROCESSUS PID 10279

```
1 int main (void)
2 {
3   pid_t ret;
4   int i, j;

5   for(i=0; i<8; i++)
6     i = i + j;

7   ret = fork();
8 }
```

MODE NOYAU

Exécution de l'appel système fork :

Si les ressources noyau sont disponibles :

- allouer une entrée de la table des processus au nouveau processus
- allouer un **pid unique** au nouveau processus
- dupliquer le contexte du processus parent (code, données, pile)
- retourner :
  - ◇ 0 au processus fils
  - ◇ le pid du processus crée à son père

MODE UTILISATEUR

PROCESSUS PID 10279

```
1 int main (void)
2 {
3   pid_t ret;
4   int i, j;

5   for(i=0; i<8; i++)
6     i = i + j;

7   ret = fork();
8 }
```

PROCESSUS PID 10280

```
1 int main (void)
2 {
3   pid_t ret;
4   int i, j;

5   for(i=0; i<8; i++)
6     i = i + j;

7   ret = fork();
8 }
```

MODE NOYAU

Exécution de l'appel système fork :

Si les ressources noyau sont disponibles :

- allouer...
- allouer...
- dupliquer...
- retourner :
  - ◇ 0 au processus fils
  - ◇ le pid du processus crée à son père

10280

0

Chaque processus père et fils reprend son exécution après le `fork()` :

- le code et les données étant strictement identiques, il est nécessaire de disposer d 'un mécanisme pour différencier le comportement des deux processus après le `fork()` ;
- on utilise pour cela le code retour du `fork()` qui est différent chez le fils (toujours 0) et le père (pid du fils crée).

PROCESSUS PID 10279

```
1  int main (void)
2  {
3    pid_t ret;
4    int i, j;

5    for(i=0; i<8; i++)
6      i = i + j;

7    ret = fork();
8    if (ret == 0)
9      printf("Je suis le fils");
10   else
11   {
12     printf("Je suis le pere");
13     printf("pid de mon fils %d",ret);
14   }
15 }
```

Pid du fils	10280
getpid	10279
getppid	pid du shell

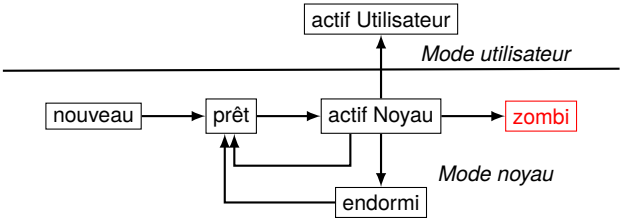
PROCESSUS PID 10280

```
1  int main (void)
2  {
3    pid_t ret;
4    int i, j;

5    for(i=0; i<8; i++)
6      i = i + j;

7    ret = fork();
8    if (ret == 0)
9      printf("Je suis le fils");
10   else
11   {
12     printf("Je suis le pere");
13     printf("pid de mon fils %d",ret);
14   }
15 }
```

getpid	10280
getppid	10279

```
1 #include <stdlib.h>
2
3 void exit (int valeur);
4 pid_t wait (int *status);
```

un appel à la primitive `exit()` provoque la terminaison du processus effectuant l'appel avec un code retour valeur. *Par défaut, en C, la sortie du dernier bloc d'instructions tient lieu d'exit.*



▷ un processus qui se termine passe dans l'état **Zombi** tant que son père n'a pas pris en compte sa terminaison ;

▷ le processus père «récupère» la terminaison de ses fils par un appel à la primitive `wait()`

PROCESSUS PID 10279

```
int main (void)
{
   pid_t ret;
   int i, j;

   for(i=0; i<8; i++)
      i = i + j;

   ret = fork();
   if (ret == 0)
   {
      printf("Je suis le fils");
      exit();
   }
   else
   {
      printf("Je suis le pere");
      printf("pid de mon fils %d",ret);
      wait();
   }
}
```

PROCESSUS PID 10280

```
int main (void)
{
   pid_t ret;
   int i, j;

   for(i=0; i<8; i++)
      i = i + j;

   ret = fork();
   if (ret == 0)
   {
      printf("Je suis le fils");
      exit();
   }
   else
   {
      printf("Je suis le pere");
      printf("pid de mon fils %d",ret);
      wait();
   }
}
```

synchronisation

5 Communications Inter-Processus

- Signaux ;
- Tubes ;
- Mémoire partagée.

Définition

Un signal est une **information atomique** envoyée à un processus ou à un groupe de processus par le système d'exploitation ou par un autre processus.

Le signal constitue un système de **communication entre processus** qui peut les amener à réagir en conséquence :

Un signal peut provoquer 3 types de réaction :

- l'exécution du processus est **immédiatement déroutée** vers une fonction spécifique, qui réagit au signal ;
- le signal est ignoré ;
- le signal provoque l'arrêt du processus (avec ou sans génération d'un fichier *core*).

Par défaut, la réception d'un signal termine le processus.

Exemple de signaux envoyés depuis le shell

- «`Ctrl-c`» : terminaison du processus en cours d'exécution depuis le shell ;
- «`Ctrl-z`» : suspension du processus ;
- à l'aide de la commande «kill» :

```
xterm
$ kill -9 34567
```

envoi le signal de terminaison `SIGKILL` au processus de PID 34567.

Signaux : les différents types

Deux catégories d'événement déclenchant un signal :

- **extérieur au processus** : frappe caractère, terminaison d'un autre processus, *etc.*
- **intérieur au processus**, c-à-d relatif à l'exécution du processus par le CPU :
  - ◇ une erreur arithmétique (division par zéro) ;
  - ◇ violation mémoire, «*segmentation fault*» : levée d'une trappe.

signaux relatifs à la fin de processus	
SIGCHLD (17)	mort du fils
SIGKILL (9)	signal de terminaison
signaux relatifs à des erreurs	
SIGILL (4)	instruction illégale
SIGFPE (8)	erreur arithmétique
SIGSEGV (11)	violation mémoire
SIGPIPE (13)	écriture dans un tube sans lecteur
signaux relatifs aux temporisations	
SIGALRM (14)	fin de temporisation (fonction alarm)
signaux relatifs aux interactions avec le terminal (extinction, frappe touche DELETE et BREAK)	
SIGHUP, SIGINT (crtl C), SIGQUIT (crtl \)	
signaux relatifs à la mise au point de programmes	
Deux signaux disponibles pour les utilisateurs	
SIGUSR1, SIGUSR2	

```
#include <signal.h>
void message1(int n) {
   if (n == SIGQUIT) {
      printf("SIGQUIT recu\n");
      exit(1) ;
     }
   else {
      printf("SIGINT recu\n");
      exit(2) ;
     }
}
void message2(int n) {
  printf("Signal %d : recu\n", n);
}

int main(int argc, char **argv) {
   int n;
   signal(SIGINT, message1);
   signal(SIGQUIT, message1);
   signal(SIGUSR1, message2);
   signal(SIGUSR2, message2);
   for( ; ; );
}
```

- Envoyer un signal à un processus :
  `int kill (pid_t pid, int sig)`
- Associer un «*handler*» à un signal :
  `signal(int sig, fonction)`
  Modifier le «*handler*» :
  `sigaction(int sig, struct sigaction action, NULL)`
- Armer une temporisation
  `int alarm (int secondes)`
  au bout de `secondes` *unités de temps, le signal SIGALRM est envoyé au processus*
- Attendre un signal
  `int pause();`

Signaux : l'exécution

Processus 1

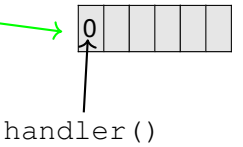
```
1  int main()
2  {
3    signal(sig1,handler);
4    _______
5    ____________
6    _________
7    _______
8  }
9
10 int handler()
11 {
12   _______
13   ____________
14 }
```

Processus 2

```
1  {
2    _______
3    ____________
4    _________
5    _______
6    kill(proc1,sig1);
7    _______
8    ____________
9    _________
10   _______
```

installation

Processus 1



Mode Utilisateur

Mode Noyau

Processus 1

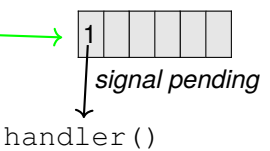
```
1  int main()
2  {
3    signal(sig1,handler);
4    _______
5    ____________
6    _________
7    _______
8  }
9
10 int handler()
11 {
12   _______
13   ____________
14 }
```

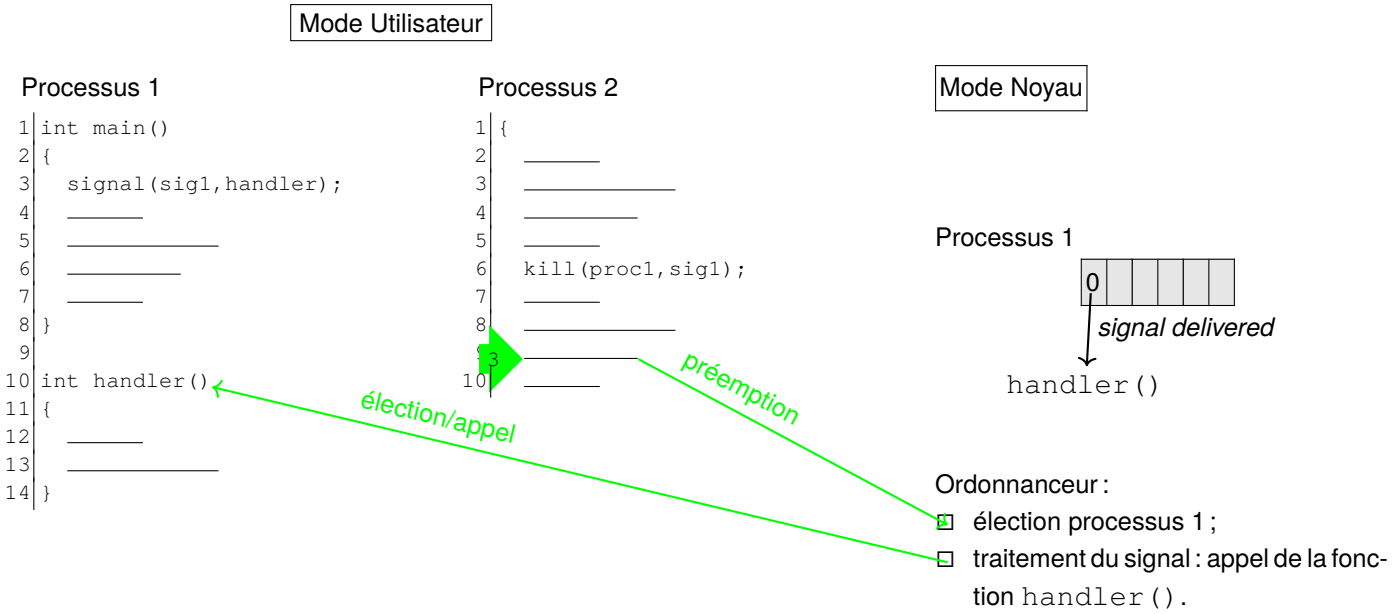
Processus 2

```
1  {
2    _______
3    ____________
4    _________
5    _______
6    kill(proc1,sig1);
7    _______
8    ____________
9    _________
10   _______
```

déclenchement

Processus 1





Signaux :

a. Le processus P_2 envoi un signal au processus P_1 (signal pendant chez P_1).
b. **Plus tard**, le processus P_1 est élu. Il quitte le **mode noyau**. Il exécute le handler du signal en **mode utilisateur** (signal délivré à P_1).

Interruptions :

a. Le dispositif matériel X envoi une interruption lors de l'exécution du processus P_1.
b. **Immédiatement**, le processus P_1 est dérouté en **mode noyau** pour exécuter le handler «*routine*» de l'interruption.

Définition et contexte d'utilisation

Les tuyaux permettent à un processus d'**envoyer des données** à un autre processus.
Ces données sont envoyées directement en mémoire sans être stockées temporairement sur disque, ce qui est donc très rapide.

Tout comme un tuyau de plomberie, un tuyau de données a deux côtés :

- un côté permettant d'écrire des données
- un côté permettant de les lire.

Chaque côté du tuyau est un descripteur de fichier ouvert soit en lecture soit en écriture, ce qui permet de s'en servir très facilement, au moyen des fonctions d'entrée / sortie classiques.

Caractéristiques

La lecture d'un tuyau est **bloquante**, c-à-d que : si aucune donnée n'est disponible en lecture, le processus essayant de lire le tuyau sera suspendu (il ne sera pas pris en compte par l'Ordonnanceur et n'occupera donc pas inutilement le processeur) jusqu'à ce que des données soient disponibles.

L'utilisation de cette caractéristique comme effet de bord peut servir à **synchroniser des processus entre eux** (les processus lecteurs étant synchronisés sur les processeur écrivains).

Caractèristiques

La lecture d'un tuyau est **destructrice**, c-à-d que si plusieurs processus lisent le même tuyau, **toute donnée lue par l'un disparaît pour les autres.**

Par exemple :

- si un processus écrit les deux caractères «ab» dans un tuyau lu par les processus A et B et que A lit un caractère dans le tuyau, il lira le caractère «a» qui disparaîtra immédiatement du tuyau sans que B puisse le lire.
- Si B lit alors un caractère dans le tuyau, il lira donc le caractère «b» que A, à son tour, ne pourra plus y lire.

Si l'on veut envoyer des informations identiques à plusieurs processus, il est nécessaire de **créer un tuyau vers chacun d'eux**.

Synthèse

Les tuyaux sont très utilisés sous UNIX pour faire communiquer des processus entre eux.
Deux contraintes :

- Les tuyaux ne permettent qu'une communication **unidirectionnelle** ;
- Les processus pouvant communiquer au moyen d'un tuyau doivent être issus d'un **ancêtre commun**.

L'appel système `pipe()`

Un tuyau se crée très simplement au moyen de l'appel système `pipe()` :

- L'argument de `pipe()` est un tableau de deux descripteurs de fichier (un descripteur de fichier est du type int en C) similaires à ceux renvoyés par l'appel système `open()` et qui s'utilisent de la même manière.
- Lorsque le tuyau a été créé, le premier descripteur, `tuyau[0]`, représente le côté lecture du tuyau et le second, `tuyau[1]`, représente le côté écriture.

Un moyen *mnémotechnique* pour se rappeler quelle valeur représente quel côté, est de rapprocher ceci de l'entrée et de la sortie standard :

- L'entrée standard, dont le numéro du descripteur de fichier est toujours 0, est utilisée pour lire au clavier : 0 lecture.
- La sortie standard, dont le numéro du descripteur de fichier est toujours 1, est utilisée pour écrire à l'écran : 1 écriture.

Pour faciliter la lecture des programmes et éviter des erreurs, il est préférable de définir deux constantes dans les programmes qui utilisent les tuyaux :

```
#define LIRE 0
#define ECRIRE 1
```


Méthodologie

La mise en place d'un tuyau permettant à deux processus de communiquer est relativement simple.

Prenons l'exemple d'un processus qui crée un fils auquel il va envoyer des données :

- ☐ Le processus père crée le tuyau au moyen de `pipe()`.
- ☐ Puis il crée un processus fils grâce à `fork()`.

Les deux processus partagent donc le tuyau.

- ▷ Le père va écrire dans le tuyau, il n'a pas besoin du côté lecture, donc il le ferme.
- ▷ Le fils ferme le côté écriture.
- ▷ Le processus père peut dès lors envoyer des données au fils.

Attention

Le tuyau doit être créé avant l'appel à la fonction `fork()` pour qu'il puisse être partagé entre le processus père et le processus fils (les descripteurs de fichiers ouverts dans le père sont hérités par le fils après l'appel à `fork()`).

Recommandations pour la fermeture d'un des bouts du tuyau

Un tuyau ayant plusieurs lecteurs peut poser des problèmes, c'est pourquoi le processus père doit fermer le côté lecture après l'appel à `fork()`.

Il en va de même pour un tuyau ayant plusieurs écrivains donc le processus fils doit aussi fermer le côté écriture.

Omettre de fermer le côté inutile peut entraîner l'**attente infinie** d'un des processus si l'autre se termine.

Exemple de scénario bloquant

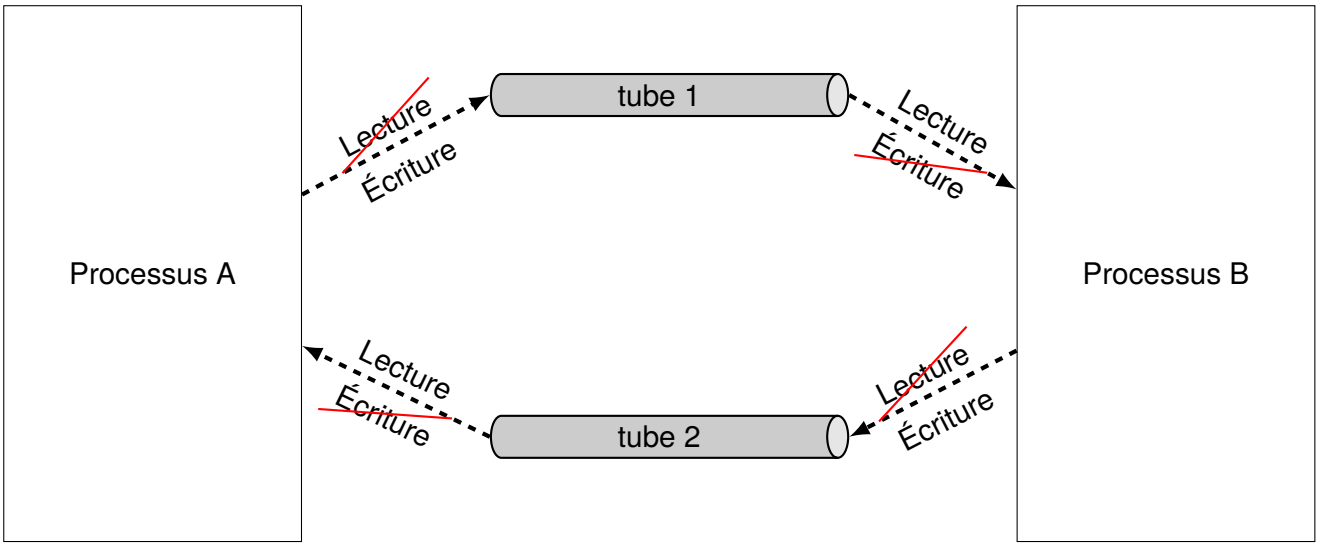
Imaginons que le processus fils n'ait pas fermé le côté écriture du tuyau.

– Si le processus père se termine, le fils va rester bloqué en lecture du tuyau sans recevoir d'erreur puisque son descripteur en écriture est toujours valide.

– En revanche, s'il avait fermé le côté écriture, il aurait recu un code d'erreur en essayant de lire le tuyau, ce qui l'aurait informé de la fin du processus père.

Tuyaux et signal

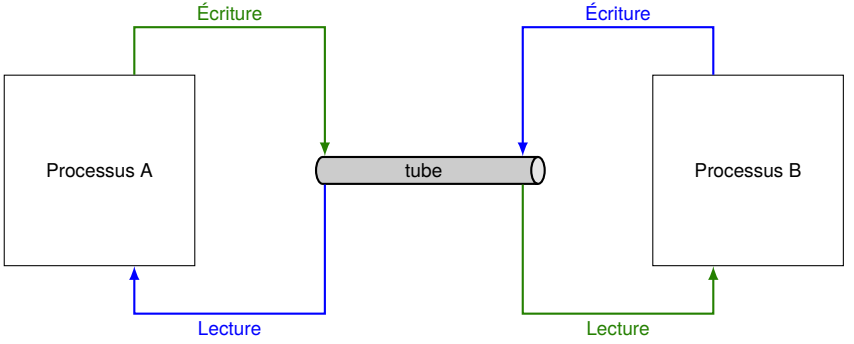
Lorsqu'un processus écrit dans un tuyau qui n'a plus de lecteurs (parce que les processus qui lisaient le tuyau sont terminés ou ont fermé le côté lecture du tuyau), ce processus recoit le signal `SIGPIPE`. *Comme le comportement par défaut de ce signal est de terminer le processus, il peut être intéressant de le gérer afin, par exemple, d'avertir l'utilisateur puis de quitter proprement le programme.*



Fermeture des descripteurs inutiles

Un effet de bord intéressant des tuyaux est la possibilité de **synchroniser** deux processus :

- un processus tentant de lire un tuyau dans lequel il n'y a rien est **suspendu** jusqu'à ce que des données soient disponibles.
- il est possible de **synchroniser** le processus **lecteur** sur le processus **écrivain** :



Limitation des tuyaux

- ils ne permettent qu'une communication unidirectionnelle ;
- les processus pouvant communiquer au moyen d'un tuyau doivent être issus d'un ancêtre commun.

D'autres moyens de communication entre processus sont possibles pour pallier ces inconvénients :

- les **tuyaux nommés** ;
- les **sockets réseaux** qui permettent une communication bidirectionnelle entre divers processus, fonctionnant sur la même machine ou sur des machines reliées par un réseau.

Les processus désirant communiquer **désignent le tuyau** qu'ils souhaitent utiliser.

Ceci se fait grâce au **système de fichiers**.

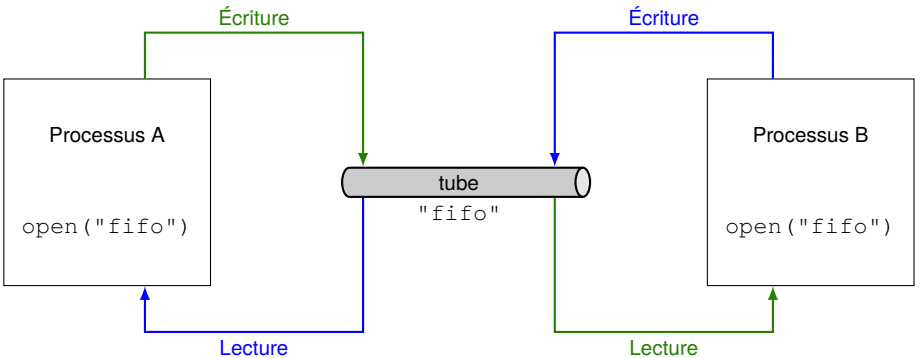
- Un tuyau nommé est un fichier :

```
xterm
linux$ mkfifo fifo
linux$ ls -l fifo
prw-r--r-- 1 pef pef 0 Jan 10 17:22 fifo
```

il s'agit cependant d'un fichier d'un type particulier, comme le montre le `p` *dans l'affichage de* `ls`*.*

- une fois créé, un tuyau nommé s'utilise très facilement :

```
xterm
linux$ echo coucou > fifo &
[1] 25312
linux$ cat fifo
[1]  + done       echo coucou > fifo
Coucou
```



Un processus tentant d'écrire dans un tuyau nommé ne possédant pas de lecteurs sera **suspendu** jusqu'à ce qu'un processus ouvre le tuyau nommé en lecture.
C'est pourquoi, dans l'exemple, `echo` *a été lancé en tâche de fond, afin de pouvoir récupérer la main dans le shell et d'utiliser* `cat`*.*
On peut étudier ce comportement en travaillant dans deux fenêtres différentes.
De même, un processus tentant de lire dans un tuyau nommé ne possédant pas d'écrivains se verra **suspendu** jusqu'à ce qu'un processus ouvre le tuyau nommé en écriture.
On peut étudier ce comportement en reprenant l'exemple mais en lancant d'abord `cat` *puis* `echo`*.*

Remarque

En particulier, ceci signifie qu'un processus ne peut pas utiliser un tuyau nommé pour **stocker des données** afin de les mettre à la disposition d'un autre processus une fois le premier processus terminé.

Synchronisation

On retrouve le même phénomène de synchronisation qu'avec les tuyaux classiques.

En C, un tuyau nommé se crée au moyen de la fonction `mkfifo()`.
Il doit ensuite être ouvert (grâce à `open()` ou à `fopen()`) et s'utilise au moyen des fonctions d'entrées / sorties classiques.

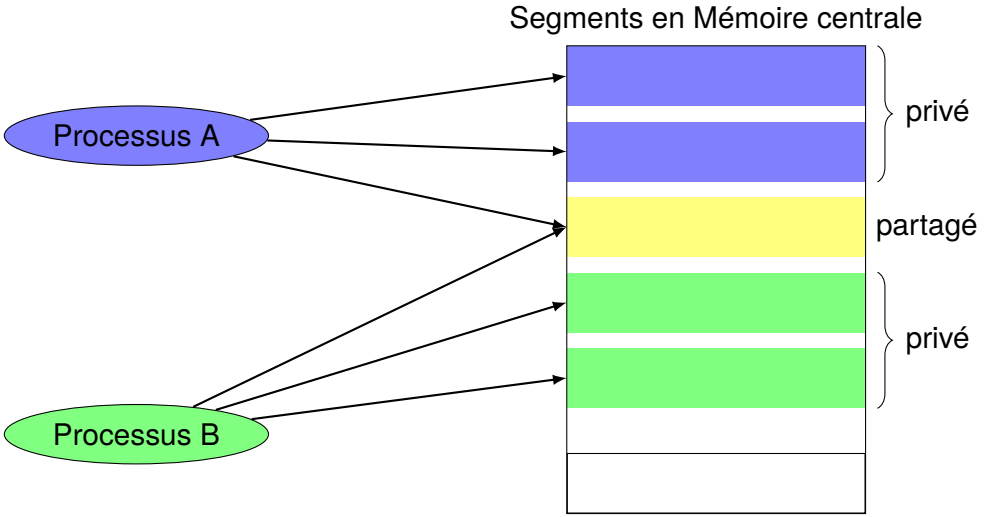
Processus écrivain

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #include <sys/stat.h>
 4 main()
 5 {
 6     mode_t mode;
 7     int tube;
 8     mode = S_IRUST | S_IWUSR;
 9     mkfifo ("fictube",mode); /* création fichier FIFO */
10     tube = open("fictube",O_WRONLY); /* ouverture fichier */
11     write (tube,"0123456789",10); /* écriture dans fichier */
12     close (tube);
13     exit(0);
14 }
```

Processus lecteur

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #include <sys/stat.h>
 4
 5 main()
 6 {
 7    int tube;
 8    char buf[11];
 9    tube = open("fictube",O_RDONLY); /* ouverture fichier */
10    read (tube,buf,10);  /* lecture du fichier */
11    buf[11]=0;
12    printf("J'ai lu %s\n, buf);
13    close (tube);
14    exit(0);
15 }
```


Mémoire partagée ou «*Shared Memory*»



La **mémoire partagée** est une **région de mémoire ou segment** dont l'utilisation est partagée entre plusieurs processus :

- un processus doit **attacher la région** à son espace d'adressage avant de pouvoir l'utiliser ;
- cette région de mémoire partagée est **identifiée par une clé unique** ;
- l'accès aux données présentes dans la région peut requérir une **synchronisation** ;
- la région ou segment partagé correspond à des pages de mémoire.

Mémoire partagée : les instructions

- Création ou accès à la mémoire partagée :

```
int shmget(key_t cle, int taille, int option);
```

  - ▷ `shmget` : retourne l'identifiant interne propre au processus de la région partagée ;
  - ▷ `cle` : identifiant externe de la région ;
  - ▷ `taille` : taille de la région ;
  - ▷ `option` : bits de configuration : IPC_CREAT, IPC_EXCL et droits d'accès.

- Attachement de la mémoire partagée :

```
void *shmat(int shmid,void *shmadd,int op
tion);
```

  - ▷ `shmat` : retourne l'adresse de la région partagée dans l'espace d'adressage du processus ;
  - ▷ `shmid` : identification interne de la région ;
  - ▷ `shmadd` : adresse dans la région partagée, en général 0 ;
  - ▷ `option` : similaire à l'instruction précédente.

- Détachement de la région partagée :

```
void *shmdt(void *adresse)
```

  - ▷ `adresse` : l'adresse interne au processus de la mémoire partagée.

- Destruction de la région partagée :

```
int shmctl(int shmid,int cmd,struct shmid_ds
*buf);
```

  - ▷ `shmid` : l'identifiant de la région ;
  - ▷ `cmd` : la commande, ici `IPC_RMID` ;
  - ▷ `buf` : une structure permettant la récupération des statistiques d'accès sur la région ;

Mémoire partagée : création et écriture

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE 256

int main(void)
{
  int shmid;
  char *mem;

  printf("Mon pid est %d\n",getpid());

  /* création du segment de mémoire partagée avec la clé CLE */
  shmid = shmget((key_t)CLE, 1000, 0750 |IPC_CREAT | IPC_EXCL);

  /* attachement */
  if((mem=shmat(shmid,NULL,0))==(char *)-1) {
    perror("shmat");
    exit(2);
  }

  /* écriture dans le segment */
  strcpy(mem, "voici une écriture dans le segment");
  exit(0);
}
```

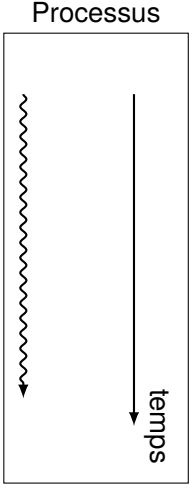

Mémoire partagée : attachement et lecture

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE 256

int main() {
  char *mem;
  int shmid;

  /* récupération du segment de mémoire */
  shmid = shmget((key_t)CLE, 0, 0);

  /* attachement */
  mem = shmat(shmid, NULL, 0);

  /* lecture dans le segment */
  printf("lu: %s\n",mem);

  /* détachement du processus */
  shmdt(mem);

  /* destruction du segment */
  shmctl(shmid, IPC_RMID, NULL);
  exit(0);
}
```

xterm

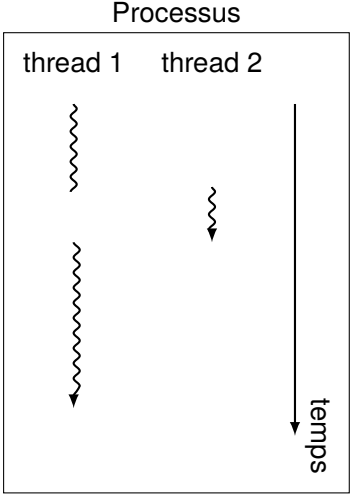
```
pef@beelink:~/ParallelismeI$ ./shmem_write
Mon pid est 17396
pef@beelink:~/ParallelismeI$ ipcs -mp
------ Shared Memory Creator/Last-op PIDs --------
shmid      owner      cpid       lpid
196608     pef        2430       2246
327681     pef        17396      17396
pef@beelink:~/ParallelismeI$ ./shmem_read
lu: voici une écriture dans le segment
pef@beelink:~/ParallelismeI$ ipcs -mp
------ Shared Memory Creator/Last-op PIDs --------
shmid      owner      cpid       lpid
196608     pef        2430       2246
```


Un **processus** peut être vu comme des instructions qui s'exécutent les unes après les autres : cela forme un «*fil d'exécution*», ou «*thread*» :



On parle aussi de «*processus de poids lourd*».

On peut créer plus d'une «*thread*» et **partager le temps** alloué au Processus, entre l'exécution de ces différentes threads :

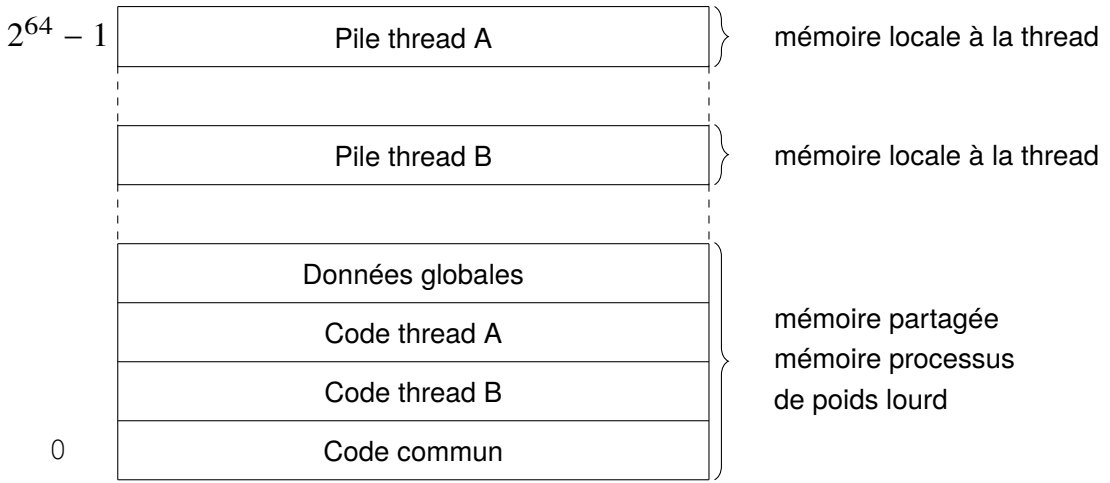


On parle de processus «*de poids léger*» pour chaque thread existant dans le processus.

Du point de vue de l'utilisateur, les différentes threads semblent se dérouler en **parallèle**.

Chaque thread :
- ▷ **partage** l'espace mémoire du processus ;
- ▷ possède sa propre **pile**.

Threads et espace d'adressage mémoire



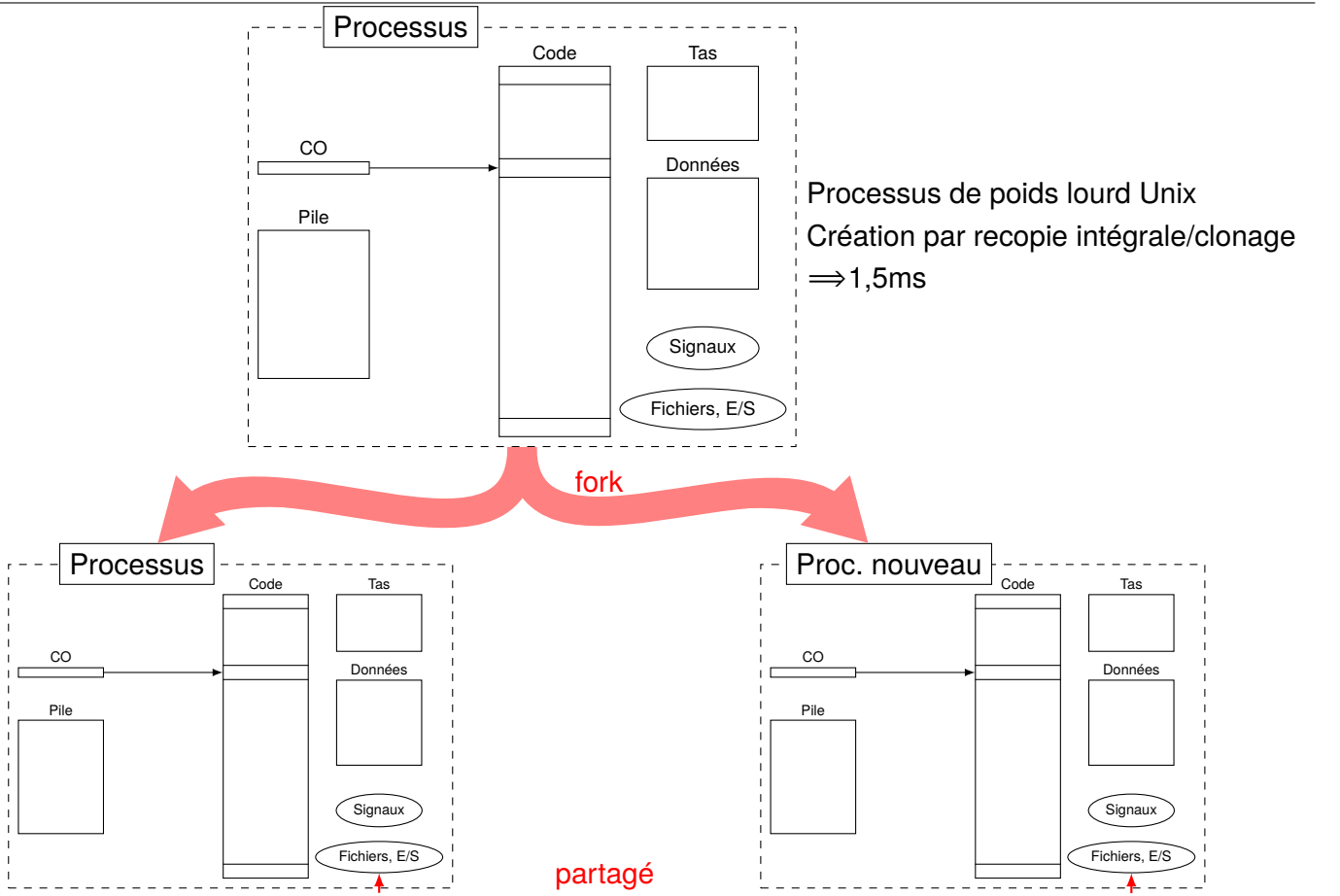
Une **thread** est plus légère à gérer et sa gestion peut être personnalisée.

La **commutation de contexte** est plus simple et plus entre threads qu'entre processus.

États uniques pour chaque thread :

- identificateur de thread ;
- état des registres : CO, «*compteur ordinal*», registre d'état, registre de pile ;
- pile ;
- masques de signaux (décrivent à quels signaux la thread répond) ;
- priorité ;
- données privées de la thread.

Processus de poids lourd

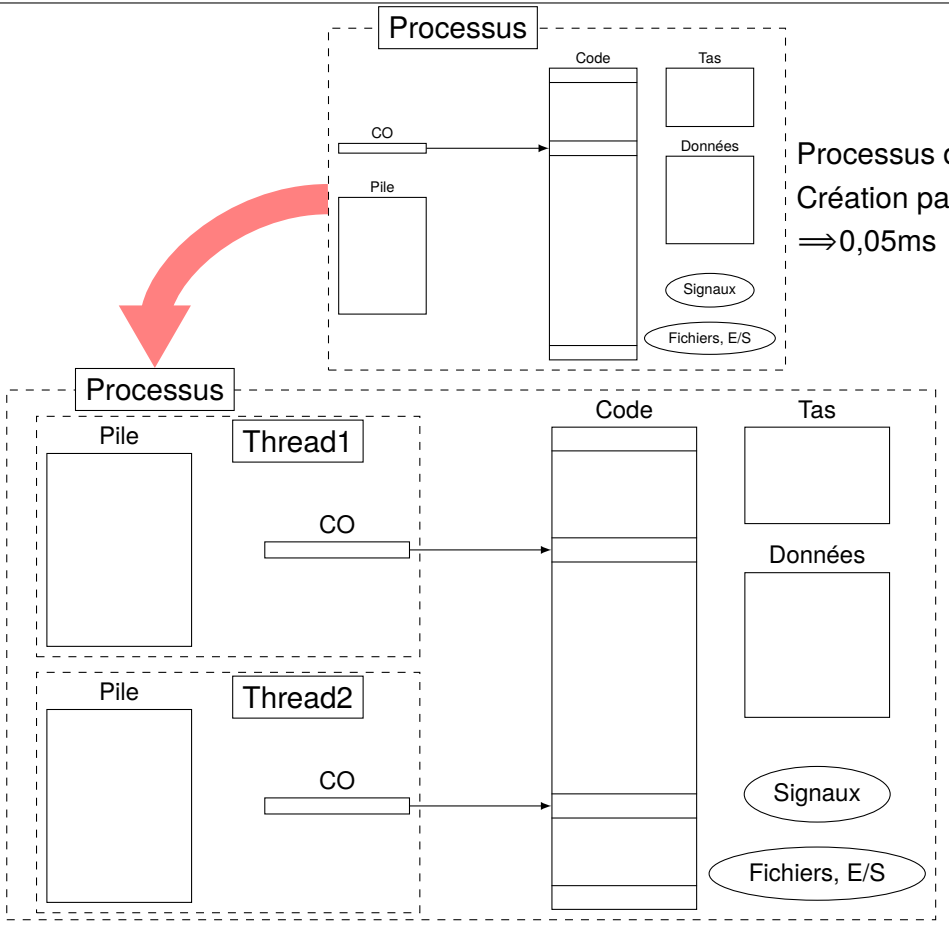


Processus de poids lourd Unix

Création par recopie intégrale/clonage

$\Longrightarrow$1,5ms

Processus de poids léger



Processus de poids léger/thread

Création par recopie de certains éléments

$\Longrightarrow$0,05ms

Critique des processus

Un processus ne convient pas toujours :

- ▷ **contexte lourd**, coûteux à créer et à détruire ;
- ▷ **pas efficace** : changement de contexte lent entre les processus ;
- ▷ **partage de données** «pas naturel» : il faut utiliser des IPC, «*Inter Process Communication*», pour passer par l'OS ;
- ▷ **pas intégré** dans les langages de programmation (à part le fork) ;
- ▷ fournis par des **opérations OS complexes**, pas souples du point de vue de l'utilisateur ;
- ▷ **programmation difficile**, non contrôlable par l'utilisateur.

Rappel sur l'exécution des processus

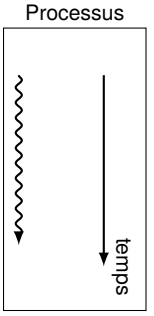
- ▷ le temps processeur est **partagé** entre les différents processus ;
- ▷ le processeur **passe d'un processus à l'autre** suivant un intervalle régulier ou lorsque un processus actif est retardé ;
- ▷ permet de dé-programmer (de-schedule) les processus bloqués en attente de fin d'E/S par exemple.

Thread : dérivée de la notion de processus

- □ **unité d'exécution concurrente** dans un même programme applicatif («*Light-weight process*») ;
- □ introduit la notion de programmation «*multi-threads*»
  - ◇ les threads partagent le même espace d'adressage, la mémoire du processus : code, données globales, tas
  - ◇ elles exécutent **même code** ou des **fonctions dédiées** ;
  - ◇ elles **partagent** tout ou partie des **ressources OS** (E/S par exemple) ;
  - ◇ elles peuvent disposer d'un **contexte de travail privé** alloué par l'application ;
- □ bibliothèque standardisée Posix : `pthreads`.

Processus classique dit «*lourd*»

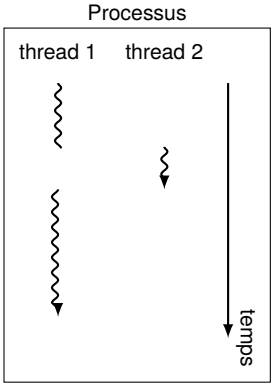
- espace d'adressage protégé à un fil d'exécution :



- **Commutation de contexte** :
  Toujours un changement d'espace d'adressage
- **Communication** :
  Outils noyau entre espace d'adressage : tubes, «*messages queues*»
- **pas de parallélisme dans le même espace d'adressage**.

Processus avec threads appelées «*léger*»

- espace d'adressage protégé à n fils d'exécution :



- **Commutation de contexte** :
  allégée, pas de changement d'espace d'adressage d'un fil à l'autre
- **Communication** :
  allégée :
  - les fils partagent les données ;

  $\Longrightarrow$ **attention à la synchronisation** !
- **parallélisme dans le même espace d'adressage**.

IEEE Portable Operating System Interface, POSIX, 1003.1 standard

- Création :

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
                    void *(*start_routine)(void *), void *arg);
```

- synchronisation entre threads :

```
int pthread_join ( pthread_t thread, void **value_ptr);
```

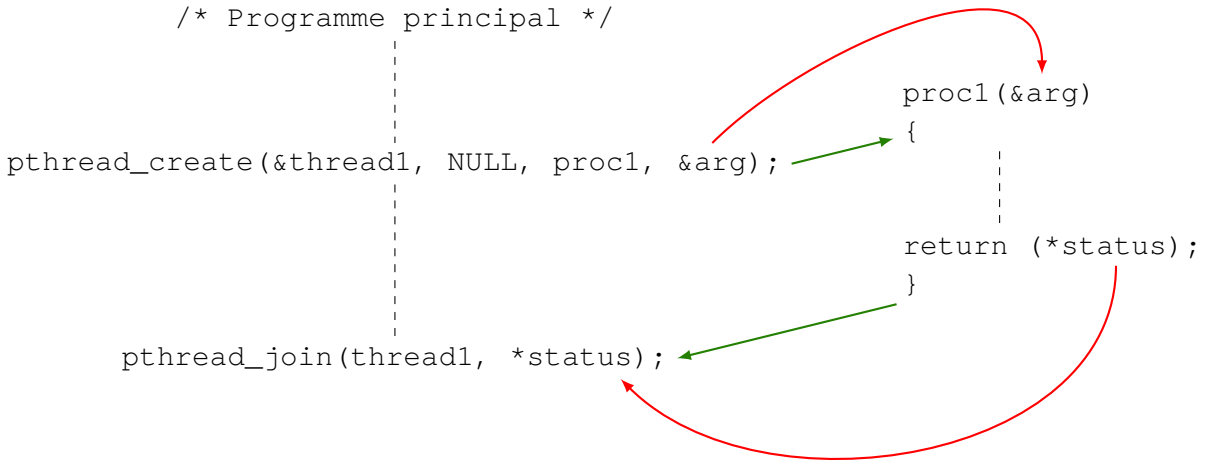
- terminaison de threads :

```
int pthread_exit (void **value_ptr);
```

La liste complète des instructions

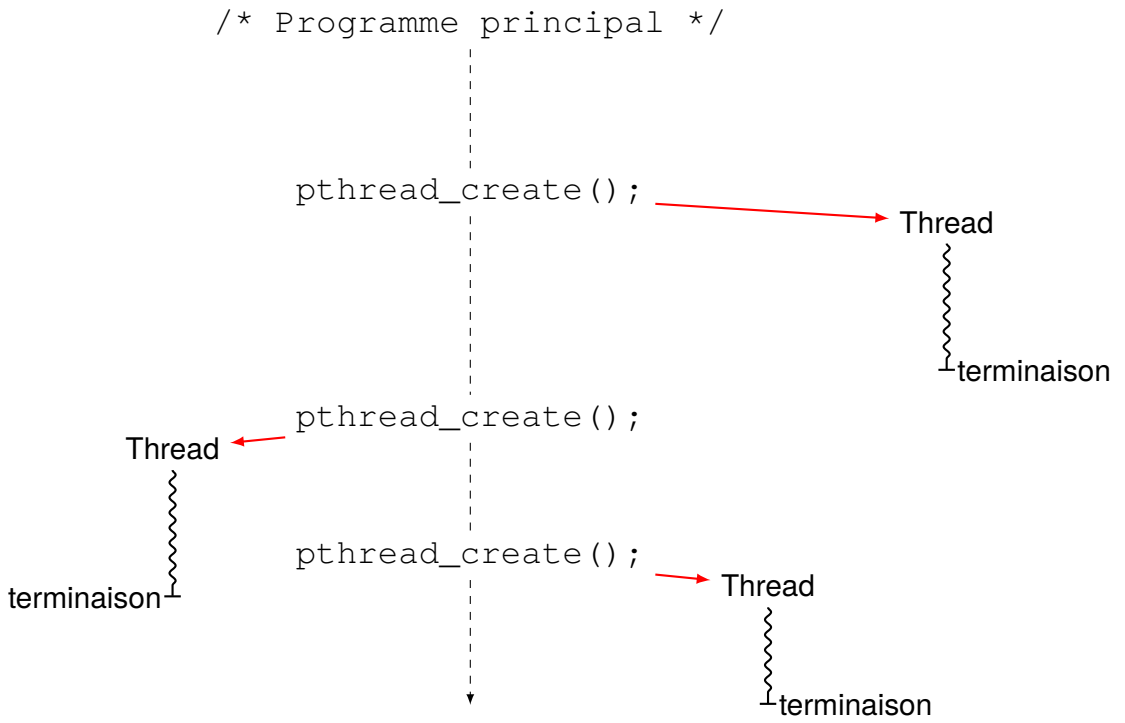
Thread call	Description
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run
pthread_attr_init	Create and initialize a thread's attribute structure
pthread_attr_destroy	Remove a thread's attribute structure

Exécution d'une thread POSIX avec attente de terminaison

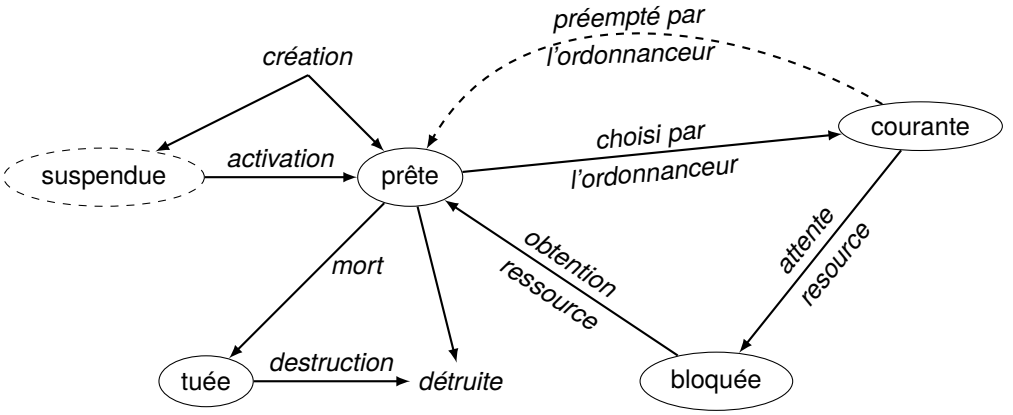


- depuis le programme principal, le `pthread_create` déclenche la **thread 1** et lui transmets `&arg` ;
- le programme principal devient lui-même une **thread** ;
- le programme principal attend la terminaison de la **thread 1** avec le `pthread_join` et reçoit `status`.

Exécution d'une thread POSIX avec attente de terminaison



- en les lançant en mode «*detached thread*» ;
- lorsque ces threads terminent, elles sont détruites et leur espace mémoire est récupéré.



La thread évolue suivant les différents états :

- ▷ la notion de «*courante*» correspond à la thread qui s'exécute.
- ▷ Courante ⟹ prête (préemption) :
  - ▷ le CPU est retiré à la thread courante pour être attribué à la thread la plus prioritaire ;
- ▷ Courante ⟹ bloquée (mise en attente)

  La thread doit arrêter son exécution pour attendre l'arrivée d'un événement (fin d'E/S)
  - ◇ elle libère volontairement le CPU ;
  - ◇ l'OS alloue le CPU à la thread prête la plus prioritaire ;
- ▷ Bloquée ⟹ prête (réveil)

  La thread est réveillée par :
  - ◇ un événement ou par une fonction d'interruption (fin d'E/S)
  - ◇ la thread courante (le scheduler ?)

  Elle rejoint le groupe des demandeurs de ressource CPU.
- ▷ Prête ⟹ courante (élection), le CPU est attribué à la thread prête la plus prioritaire ⟹ elle devient la thread courante.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>

pthread_t pthread_id[3];

void *f_thread(void *d)
{
  int i = *((int *) d);
  printf ("je suis la %d-eme pthread du processus %d, mon id est %ld\n", i, getpid(),
pthread_self());
}

int main(void)
{
  int i;

  for (i=0; i<3; i++)
    if (pthread_create(&pthread_id[i], NULL, f_thread, (void *) &i) == -1)
      fprintf(stderr, "erreur de creation pthread numero %d\n", i);
  printf ("je suis la thread initiale du processus %d, mon id est %ld\n", getpid(),
pthread_self());
  for (i=0; i<3; i++)
    pthread_join(pthread_id[i],NULL);
}
```

Attention

Ce programme d'exemple ne protège pas l'accès à la variable `i` utilisée par chaque thread pour indiquer son numéro.
⟹la valeur de `i` affichée peut être la mauvaise...

Communication entre processus

Il existe deux mécanismes principaux pour faire communiquer deux processus :

- la communication via la **mémoire partagée** ;
- la communication par **échange de messages** au travers d'un tube ;

Au niveau d'une thread

Les threads qui ont besoin d'échanger des données le font via la **mémoire** en partageant une partie de la mémoire appartenant à leur **espace d'adressage commun**.

Problèmes

▷ Que se passe-t-il si deux threads **lisent en même temps** ? ...rien de particulier.

▷ Que se passe-t-il si une thread **lit pendant qu'une autre écrit** ? ...un gros problème, la donnée risque de ne pas être dans un état cohérent et peut ne correspondre ni à la la donnée avant l'écriture, ni à la donnée après l'écriture.

Exécution de l'exemple de code précédent :

```
xterm
pef@beelink:~/ParallelismeI$ ./pthread_example
je suis la 2-eme pthread du processus 15214, mon id est 139747692533504
je suis la 2-eme pthread du processus 15214, mon id est 139747684140800
je suis la thread initiale du processus 15214, mon id est 139747701020480
je suis la 3-eme pthread du processus 15214, mon id est 139747675748096
```

la variable `i` utilisé par la thread a été modifiée par une autre thread !

▷ Que se passe-t-il si deux threads **écrivent simultanément** ? ...de **gros problèmes** en perspective.

⟹**Problèmes de synchronisation** des processus et de **protection** des accès.

Guichet bancaire et accès simultanée en dépôt et en retrait

▷ depôt de s euros au guichet ;
▷ retrait par carte de s euros ;
} *simultanément*

Dépôt

❶ x := moi.solde
❷ x := x + s
❸ moi.solde := x

Retrait

❹ x' := moi.solde
❺ x' := x' - s
❻ moi.solde := x'

Entrelacement des instructions :

□ ❶❷❹❺❸❻
x := moi.solde; x := x + s; x' := moi.solde; x' := x' - s ; moi.solde := x; moi.solde := x'
⟹ **Une perte d'argent !**

□ ❶❷❹❺❻❸
x := moi.solde; x := x + s; x' := moi.solde; x' := x' - s ; moi.solde := x'; moi.solde := x
⟹ **Un gain d'argent !**

Illustration en assembleur Z80

```
int solde;

void main()
{
  int x;
  x = solde;
  x = x + 100;
  solde = x;
}
```

```
_main::
;compteur.c:6: x = solde;
   ld    bc, (_solde)
;compteur.c:7: x = x + 100;
   ld    hl, #0x0064
   add    hl, bc
   ld    (_solde), hl
;compteur.c:8: solde = x;
;compteur.c:9: }
```

La valeur 100 en hexa sur 16bits

Utilisation du compilateur C `sdcc` *vers processeur z80.*

Problème

Les opérations de lecture/écriture en mémoire ne sont pas **atomiques** :
$\Longrightarrow$ une lecture, comme une écriture, peut être **interrompue** au cours de son action.

La solution

Utiliser une **exclusion mutuelle** :

▷ elle rend l'opération de lecture/écriture **atomiques** en restreignant l'accès à **un segment de mémoire partagé** à **une seule thread à la fois**.

Risques ?

Imaginons qu'il existe :

□ **plusieurs segments de mémoire partagée** s_1, s_2 et s_3 ;

□ **plusieurs threads** t_1, t_2 et t_3 ;

▷ t_1 bloque l'accès à s_1 , t_2 bloque l'accès à s_2 et t_3 bloque l'accès à s_3 ;

▷ t_1 pour libérer s_1 a besoin d'accéder à s_2 ;

▷ Mais t_2 pour libérer s_2, a besoin de lire s_3.

▷ Si t_3 n'a pas besoin de quoi que ce soit d'autre, il fait ce qu'il a à faire, puis libère s_3 , ce qui libère s_2 et par la suite s_1 ... ouf ! !

▷ par contre, si t_3 a besoin de lire s_1 pour libérer s_3 ... $\Longrightarrow$ situation **d'interblocage**.

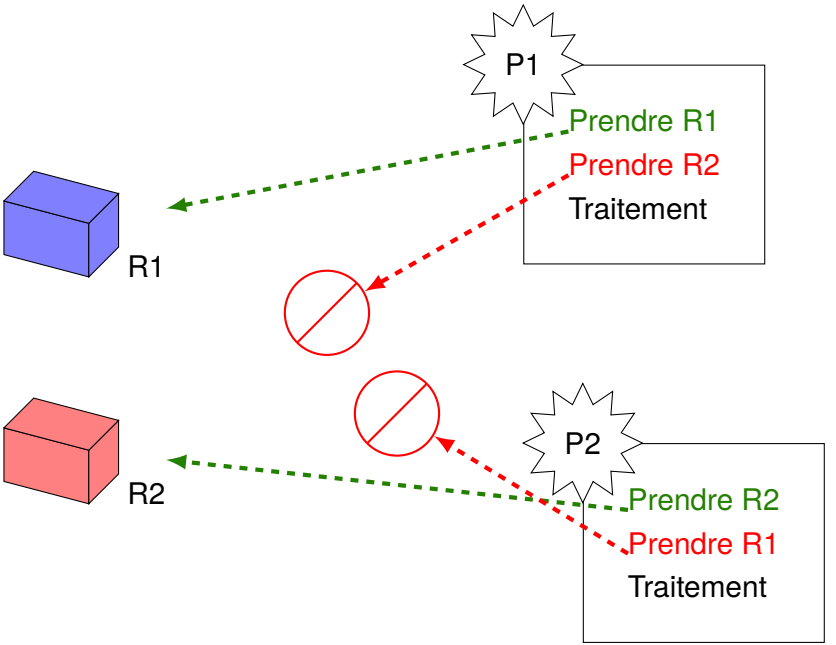
Définitions

- Une ressource désigne toute entité dont a besoin un processus pour s'exécuter.
  - Ressource matérielle (processeur, périphérique)
  - Ressource logicielle (variable)
- Une ressource est caractérisée
  - par un état : «*libre*» ou «*occupée*»
  - par son **nombre de points d'accès** : nombre de processus pouvant l'utiliser en même temps.
- Utilisation d'une ressource par une thread, «*processus de poids léger*» :
  - Trois étapes :
    - Allocation
    - Utilisation
    - Restitution
  - Les phases **d'allocation** et de **restitution** doivent assurer que le ressource est utilisée conformément à **son nombre de points d'accès**.
- Une **ressource critique à un seul point d'accès**.

Interblocage

Ensemble de n processus **attendant** chacun une ressource déjà **possédée** par un autre processus de l'ensemble.

R1 et R2 : un seul point d'accès



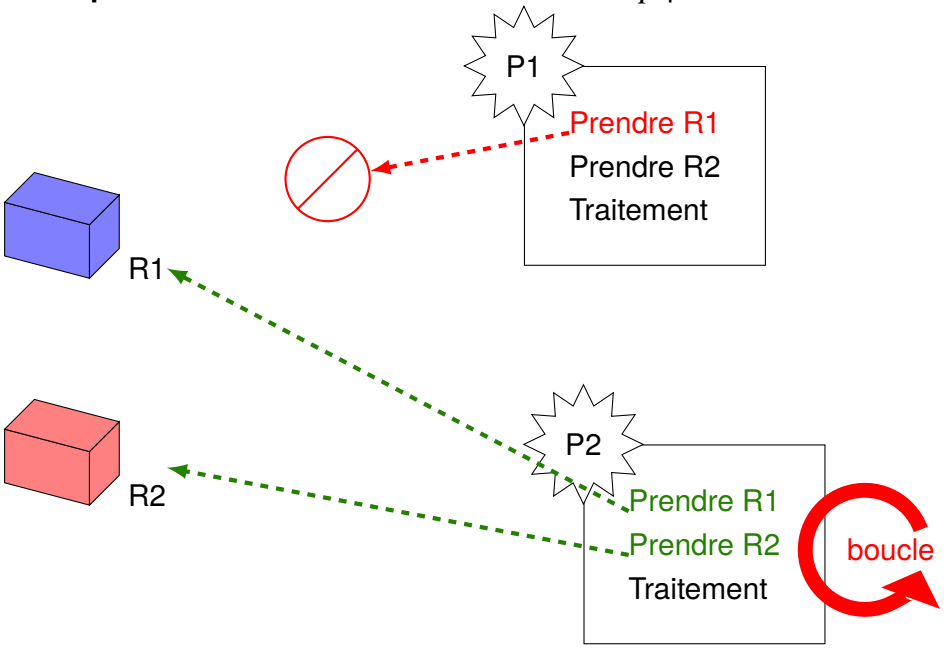
Aucun processus ne peut poursuivre son exécution :

$\Longrightarrow$ **Attente infinie !**

Coalition

Ensemble de n processus **monopolisant** les ressources **au détriment** de p processus.

R1 et R2 : un seul point d'accès



Le processus P1 n'obtiendra jamais R1, car P2 la monopolise :

$\Longrightarrow$ **Famine !**

$\Longrightarrow$ Attente finie, mais **indéfinie**.

P1 sera débloqué lorsque P2 terminera.

Conditions nécessaires à l'obtention d'un interblocage

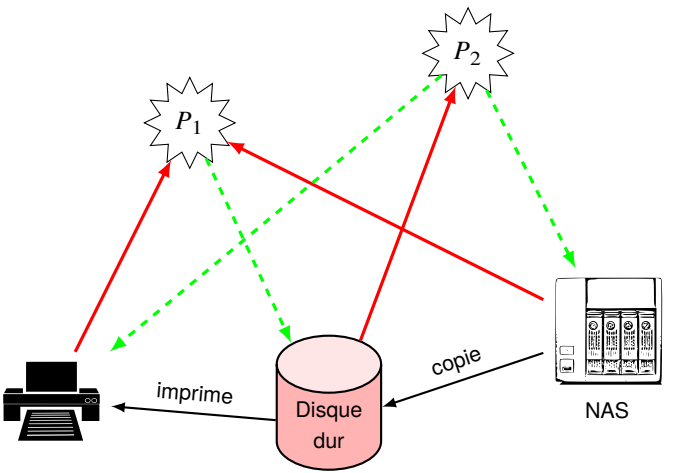
- **Ressource critique** : Une ressource au moins doit se trouver dans un mode non partageable
- **Occupation et attente** : Un processus au moins occupant une ressource attend d'acquérir des ressources supplémentaires détenues par d'autres processus
- **Pas de réquisition** : Les ressources sont libérées sur seule volonté des processus les détenant
- **Attente circulaire** : au moins un processus P1 attend une ressource détenue par un autre processus P2, P2 attend lui-même une ressource détenue par P1

Détection automatique ?

Attente Circulaire... *la détection est possible mais complexe à réaliser...et que faire ensuite ?*

- ▷ P_2 attend l'imprimante détenue par P_1 ;
- ▷ P_1 attend le disque dur détenue par P_2

P_1 possède le NAS
P_2 possède le disque dur
P_1 possède l'imprimante
P_2 attend l'imprimante
P_2 attend le NAS
P_1 attend le disque dur



Un premier problème de synchronisation est celui de l'accès par un ensemble de processus à une **ressource critique**, c'est-à-dire une ressource à un **seul point d'accès** donc utilisable par un seul processus à la fois.

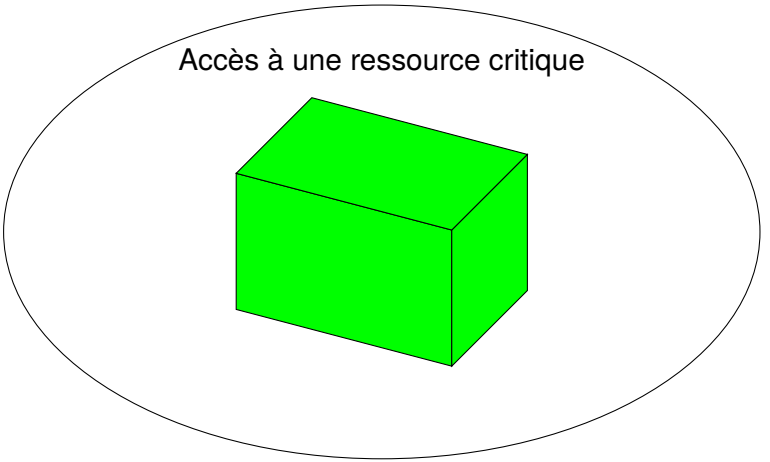
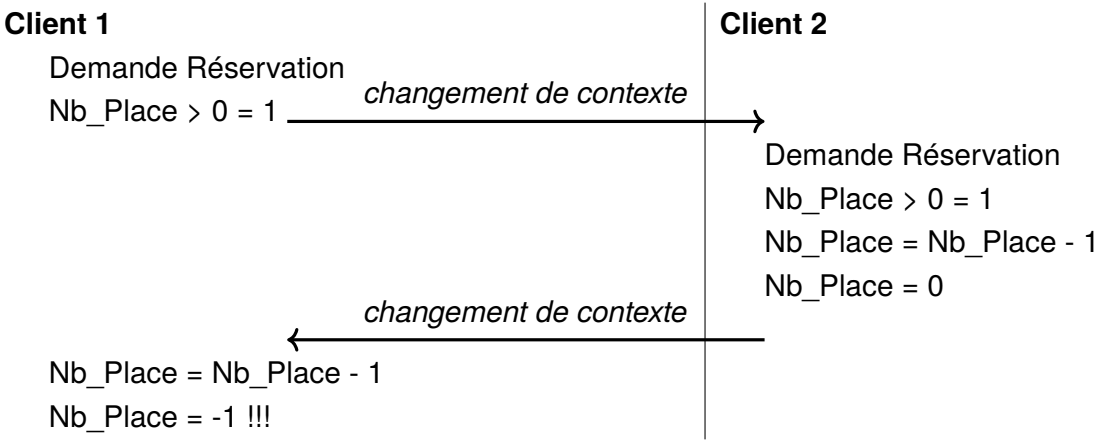


Illustration du problème : réservation d'une place

```
1 Réservation :
2 Si nb_place > 0
3 alors
4    Réserver une place
5    nb_place = nb_place - 1
6 fsi
```

Utilisation par deux clients s'exécutant en concurrence



Définitions de la section critique

Processus

Début

Entrée Section Critique

Ressource critique
Nb_Place

SECTION CRITIQUE
(code d'utilisation
de la ressource critique)

Sortie Section Critique

Fin

L'entrée et la sortie de la «*Section Critique*» doivent assurer qu'**à tout moment**, un **seul processus** s'exécute en SC $\Longrightarrow$ **exclusion mutuelle**.

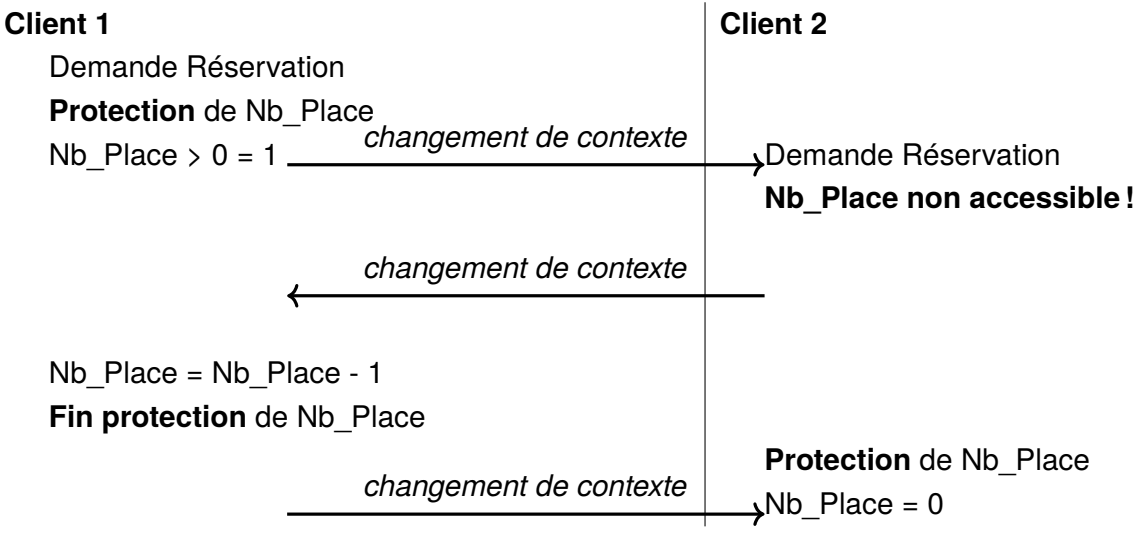
Section critique

Partie d'un programme dont l'exécution de doit pas **s'entrelacer** avec d'autres programmes.

*Une fois qu'une tâche y entre, il faut lui **permettre de terminer** cette section sans permettre à d'autres tâches de travailler sur les mêmes données.*

```
1 Réservation :
2 Protection de nb_place
3 Si nb_place > 0
4 alors
5    Réserver une place
6    nb_place = nb_place - 1
7 fsi
8 Fin protection de nb_place
```

Utilisation par deux clients s'exécutant en concurrence avec exclusion mutuelle



Problématique présente dans :

- les bases de données
- le partage des ressources (fichiers,connections,réseaux,...)
- les automates (exemple du DAB, «distributeur automatique de billets»)
- le matériel et périphérique (Disquedur,...)
- le développement (MMORPG, clients/serveur,...)
- *etc.*

Ecriture algorithmique

Repeat
  Section d'entrée
  Section critique, SC
  Section de sortie
  Section restante, SR
Forever

Plusieur types de solutions

- ☐ Solution matérielle ;
- ☐ Solutions algorithmiques ;
- ☐ Solutions fournies par le Système d'Exploitation.

Exclusion mutuelle : solution matérielle

Processus 1

```
Demande Réservation
Masquer IT
Si Nb_Place > 0
alors
    Réserver une place
    Nb_Place = Nb_Place - 1
fsi
Démasquer IT
```

préemption ordonnanceur

Processus 2

```
Demande Réservation
Masquer IT
Si Nb_Place > 0
alors
    Réserver une place
    Nb_Place = Nb_Place - 1
fsi
Démasquer IT
```

Masquage des interruptions

Section critique : interdire la prise en compte des interruptions durant l'utilisation de la ressource critique.

Inconvénients majeurs :

- ▷ attente de tous les processus car le **processeur est monopolisé** ;
- ▷ exécution uniquement en **mode noyau**.

Il faut trouver une solution **bloquant seulement les processus concernés** : *solution algorithmique* ou solution par le *système d'exploitation*.

Algorithme 1 : Threads se donnant mutuellement le tour

- ▷ la variable partagée «`turn`» est initialisée à 0 ou à 1 ;
- ▷ la *SC* de T_i est exécutée seulement si `turn = i`
- ▷ T_i est occupé à attendre si T_j est dans *SC*.

```
1 Thread Ti:
2 repeat
3    while (turn != i) {};
4       SC
5    turn = j;
6       SR
7 forever
```

- □ SC, «*Section critique*» : les instructions en exclusion mutuelle ;
- □ SR, «*Section Restante*» : des instructions sans accès conccurent.

Exécution de l'algorithme

```
1 Thread T0:
2 repeat
3    while(turn!=0){};
4    SC
5    turn = 1;
6    SR
7 forever
```

```
1 Thread T1:
2 repeat
3    while(turn!=1){};
4    SC
5    turn = 0;
6    SR
7 forever
```

On considère que :

- ▷ chaque tâche exécute le **même algorithme** de contrôle de concurrence : l'indice de la tâche permet de les différencier ;
- ▷ les parties SC et SR sont **spécifiques** à chaque tâche ;
- ▷ chaque tâche recommence **indéfiniment** son travail.

Modélisation de l'exécution : table des états d'exécution

L'état d'un processus est **entièrement déterminé** par la combinaison de son CO et de la valeur des variables.

Imaginons que l'ordonnanceur alterne entre T0 et T1 et que chaque ligne prend le même temps d'exécution :

CO		CPU		variable	explication
T0	T1	T0	T1	turn	
2	2			0	initialisation
3	2	✓		0	T0 est exécuté par le CPU et teste la condition ⟹ il peut sortir du `while`
3	3		✓	0	T1 est exécuté par le CPU et teste la condition ⟹ il reste dans le `while`
4	3	✓		0	T0 exécute la section critique
4	3		✓	0	T1 teste la condition ⟹ il reste dans le `while`
5	3	✓		1	T0 change la variable turn à 1
5	3		✓	1	T1 teste la condition ⟹ il peut sortir du `while`
6	3	✓		1	T0 exécute sa section restante
6	4		✓	1	T1 exécute la section critique
7	4	✓		1	T0 recommence la boucle `repeat` et va repartir à la ligne 3
7	5		✓	0	T1 change la variable turn à 0
3	6	✓		0	T0 teste la condition ⟹ il peut sortir du `while`
3	6		✓	0	T1 exécute sa section restante
4	6	✓		0	T0 exécute la section critique
4	7		✓	0	T1 recommence la boucle `repeat` et va repartir à la ligne 3
5	7	✓		1	T0 change la variable turn à 1
5	3		✓	1	T1 teste la condition ⟹ il peut sortir du `while`

temps

état déja connu

*Lorsque l'on rencontre un **état déjà connu**, c-à-d une combinaison de (CO_{T0},CO_{T0},variable), on sait que l'on va recommencer la même séquence dans le tableau ⟹ le comportement du programme est entièrement détaillé.*

Bilan de l'algorithme 1 : Généralisation à n threads : avant qu'une thread ne puisse rentrer dans sa section critique, il lui **faut attendre** que tous les autres aient eu cette chance !

Algorithme 2 : Threads exprimant leur souhait de rentrer en SC

```
1 Thread Ti:
2 repeat
3    flag[i] = vrai;
4    while (flag[j] == vrai) {};
5       SC
6       flag[i] = faux;
7       SR
8 forever
```

- ▷ une variable booléenne par thread : `flag[0]` et `flag[1]` ;
- ▷ T_i signale qu'elle désire exécuter sa SC par `flag[i]=vrai`

Exécution de l'algorithme

```
1 Thread T0
2 repeat
3    flag[0] = vrai;
4    while (flag[1] == vrai) {};
5    SC
6    flag[0] = faux;
7    SR
8 forever
```

Après vous !

```
1 Thread T1
2 repeat
3    flag[1] = vrai;
4    while (flag[0] == vrai) {};
5    SC
6    flag[1] = faux;
7    SR
8 forever
```

Après vous !

Analyse

- ▷ Que se passe-t-il si les deux threads exécutent l'une après l'autre ligne 3 ? $\Longrightarrow$ **blocage !**
  Exemple : T_0 exécute 3, puis on passe à l'exécution de T_1 qui exécute aussi la ligne 3.

Algorithme de Dekker

```
int c1, c2, Turn = 1;
```

Thread T1

```
 1 while(True)
 2 {   nc1: /* Section non critique*/
 3       c1 = 0;
 4       while (c2 == 0)
 5       {   c1 = 1;
 6           while (Turn == 2) {};
 7           c1 = 0;
 8       }
 9    criti1: /* Section critique */
10       Turn = 2;
11       c1 = 1;
12 }
```

Thread T2

```
 1 while(True)
 2 {   nc2: /* Section non critique*/
 3       c2 = 0;
 4       while (c1 == 0)
 5       {   c2 = 1;
 6           while (Turn == 1) {};
 7           c2 = 0;
 8       }
 9    criti2: /* Section critique */
10       Turn = 1;
11       c2 = 1;
12 }
```

La symétrie est rompue grâce à la variable `Turn`.

Analyse

Le fonctionnement correct de l'algorithme de Dekker dépend du fait que chaque processus **finit toujours par progresser**.

$\Longrightarrow$***propriété d'équité***

L'**hypothèse d'équité** simple, suffisante pour raisonner au sujet de l'algorithme de Dekker, est la suivante :

$\Longrightarrow$ **Tout processus qui a la possibilité d'exécuter une instruction finira toujours par le faire**.

Propriétés souhaitées pour un programme parallèle

Deux catégories :

- Propriétés de **sûreté** :
  - certains états d'exécution du système **indésirables** ne sont **jamais atteints**.
    Elles ne dépendent pas d'une hypothèse d'équité.

- Propriétés de **vivacité** :
  - les états d'exécution du système **désirables** seront forcément **atteints**.

  En général ces propriétés ne sont vraies qu'en prèsence d'une hypothèse d'équité.

L'état d'exécution d'un système est défini par l'ensemble des valeurs associées à chacune des threads du système :

- état de la pile et ensemble des valeurs des variables propre à chaque thread ;
- contenu du Compteur Ordinal exprimant l'endroit du code où chaque thread se trouve ;
- valeur des variables partagées.

Un **état désirable** d'exécution du système est un état où le système se comporte conformément aux attentes du programmeur.

Un **état indésirable** d'exécution du système est un état où :

- le système est **corrompu** : valeurs des variables perdues ou erronées ;
- système **bloqué** : le système ne plus atteindre d'état désirable.

L'algorithme de Peterson

Combine les deux idées : `flag[i]`=intention d'entrée et `turn`=à qui le tour

Initialisation

- flag[0] = flag[1] = faux
- turn = i ou j
- le désir d'exécuter SC est indiqué par `flag[i]=vrai`
- à la sortie de la SC : `flag[i] = faux`

```
1 repeat
2    flag[i] = vrai; /* Je veux entrer */
3    turn = j; /* je donne une chance à l'autre */
4    do while ((flag[j] == vrai) && (turn == j)) {};
5       SC
6    flag[i] = faux;
7       SR
8 forever
```

Thread T0

```
1 repeat
2    flag[0] = vrai;
3    turn = 1;
4    do while ((flag[1] == vrai)
5          && (turn == 1)) {};
6       SC
7    flag[0] = faux;
8       SR
9 forever
```

Thread T1

```
1 repeat
2    flag[1] = vrai;
3    turn = 0;
4    do while ((flag[0] == vrai)
5          && (turn == 0)) {};
6       SC
7    flag[1] = faux;
8       SR
9 forever
```


L'instruction Test and Set

```
bool testset(int *i)
{
    if (i == 0)
    {
        i = 1;
        return true;
    }
    else
        return false;
}
```

Instruction(s) atomique(s)

Un algorithme utilisant `testset` pour l'exclusion mutuelle :

- ▷ variable partagée b initialisée à zéro ;
- ▷ c'est le premier T_i qui met `b` à 1 qui entre en SC :

  Thread Ti :

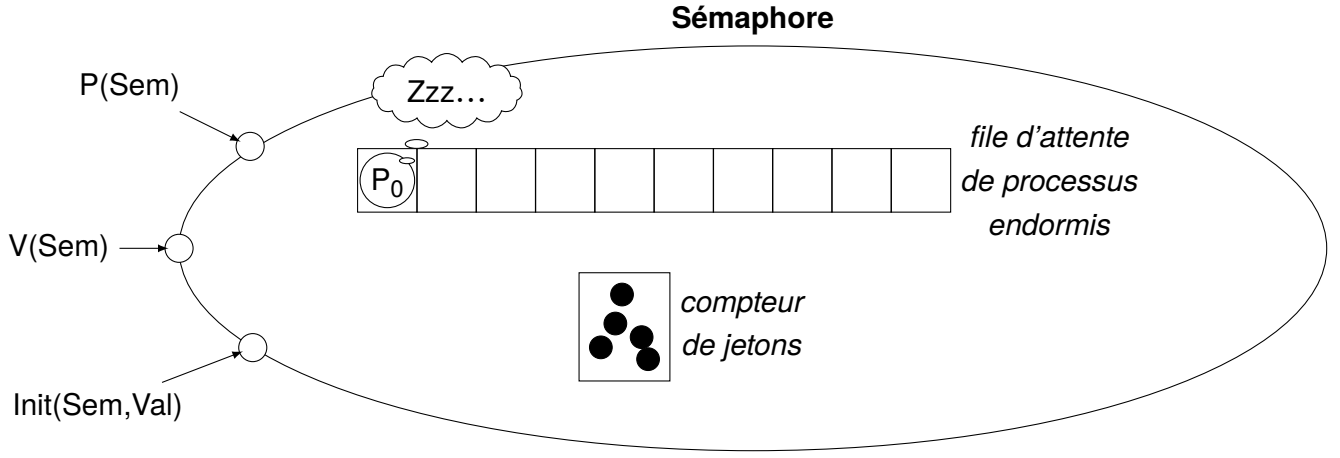
```
while testset(b) == false{};
SC
b = 0;
SR
```

Le testset peut être intégrer au processeur : une instruction du processeur réalise l'opération de test/modification ce qui en fait une opération réellement atomique.

Bilan des solutions algorithmiques

L'**inconvénient majeur** de ces solutions logicielles est l'**attente active**.

Une autre solution est d'utiliser un outil de synchronisation offert par le **système d'exploitation** :
⟹les **sémaphores**.

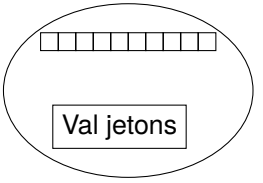


Une **sémaphore**, Sem, peut être vue comme un **distributeur de Jetons** manipulable au travers des opérations **atomiques** :

- **Init(Sem,Val)** : **fixe** le nombre de jetons initial ;
- **P(Sem)** : **attribue** un jeton au processus appelant s'il en reste, sinon **bloque** le processus et le mets dans la file d'attente des processus endormis ;
- **V(Sem)** : **restitue** un jeton et **débloque** un processus de la file d'attente s'il y en a un et lui **attribue** le jeton.

Opération Ini(Sem,Val)

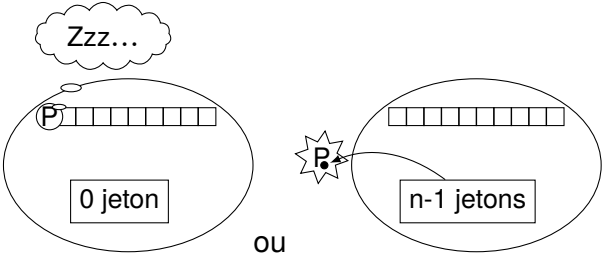
```
Init(Sem,Val)
début
   masquer_it
      Sem.compteur_jetons := Val jetons
      Sem.file_attente := vide
   démasquer_it
fin
```



«masquer_it» et «démasquer_it» servent à empêcher l'interruption de l'opération par une interruption : rendre atomique l'opération «Init» (les interruptions sont suspendues pendant un très court instant).

Opération P(Sem)

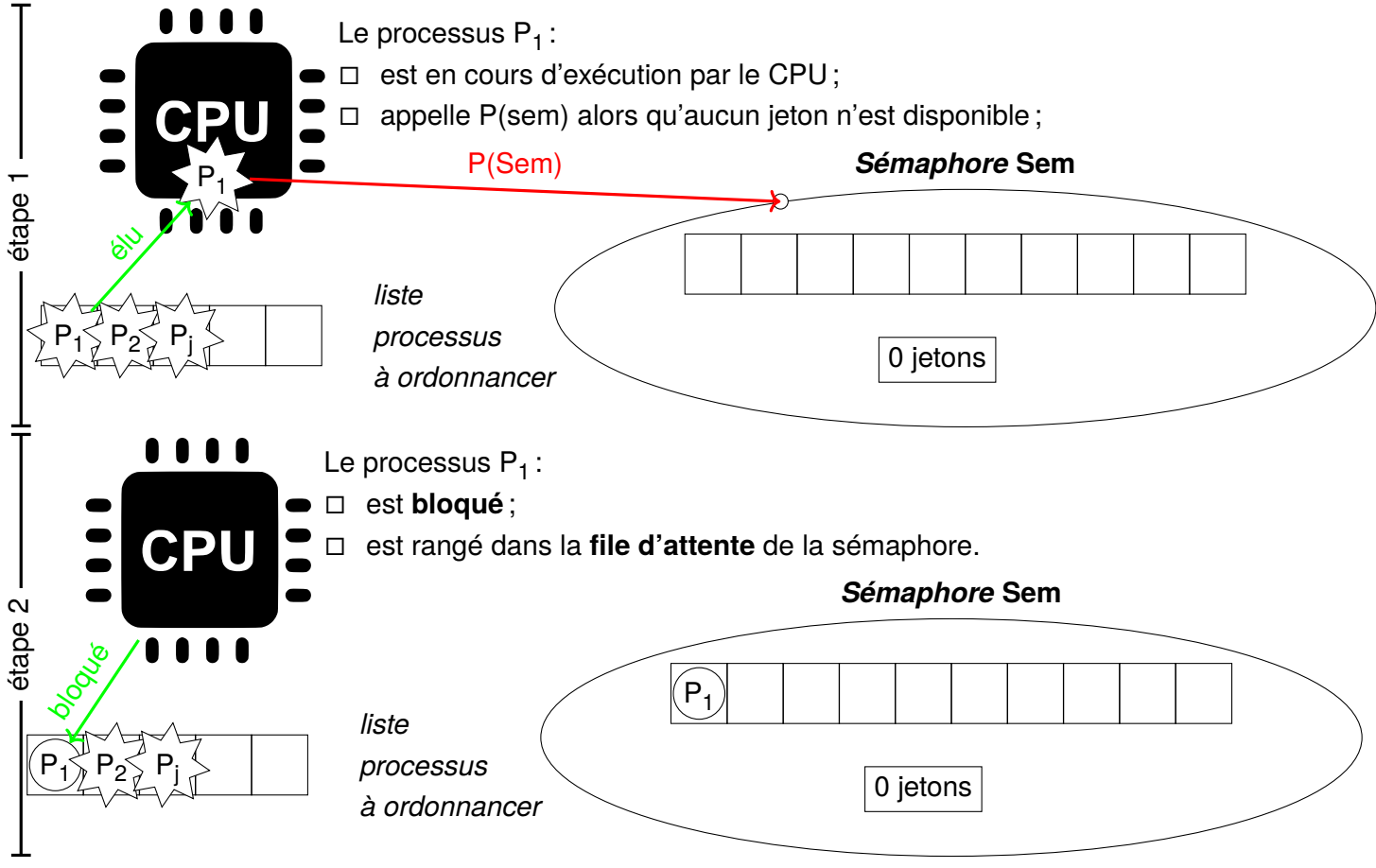
```
P(Sem)
début
   masquer_it
   Si il reste un jeton
   alors
      le donner à ce processus
   sinon
      ajouter ce processus à Sem.file_attente
      bloquer ce processus
   fsi
   démasquer_it
fin
```



Le processus qui ne peut obtenir de jeton est :

- ▷ ***endormi** : il ne sera plus ordonnancé tant qu'un jeton ne devient pas libre : pas d'attente active ;*
- ▷ *il est **mis dans la file d'attente** : il sera traité suivant la priorité de son entrée dans la file d'attente, ce qui garantit **l'équité**.*

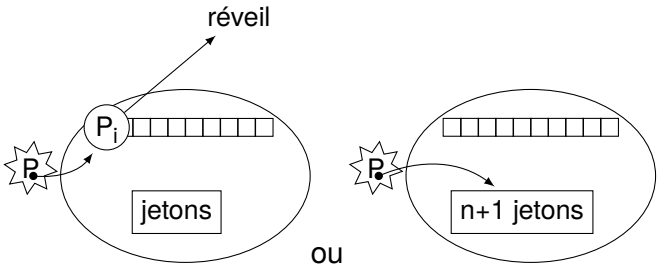
Définition d'une sémaphore : explications pour P(Sem) bloquant



Définition d'une sémaphore : les opérations

Opération V(Sem)

```
V(Sem)
début
   masquer_it
   Ajouter un jeton à S.K
   Si il y a un processus P en attente
   alors
      sortir P de Sem.file_attente
      donner un jeton à P
      réveiller P
   fsi
   démasquer_it
fin
```



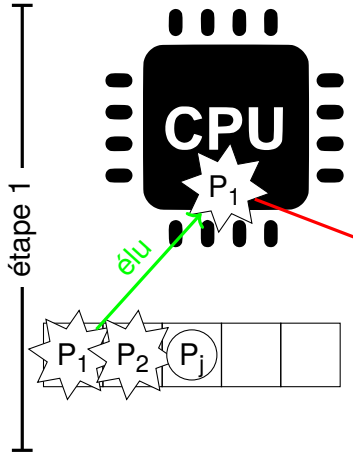
Lorsqu'un jeton est rendu alors qu'un processus était endormi en attente de celui-ci :

- ▷ *ce processus obtient immédiatement le jeton ;*
- ▷ *il redevient «ordonnançable» ;*

Passage de l'état «*Bloqué*»⟹«*Prêt*» par la sémaphore :

- ☐ le **plus efficace** :
  - ◇ le processus a été suspendu sans attente active⟹pas de «*temps CPU*» perdu ;
  - ◇ à son réveil le processus possède le jeton ;
- ☐ **équitable** : il utilise la file d'attente pour garantir cette équité : les processus sont sélectionnés dans leur ordre d'entrée dans la file d'attente.

Définition d'une sémaphore : explications pour V(Sem) débloquant



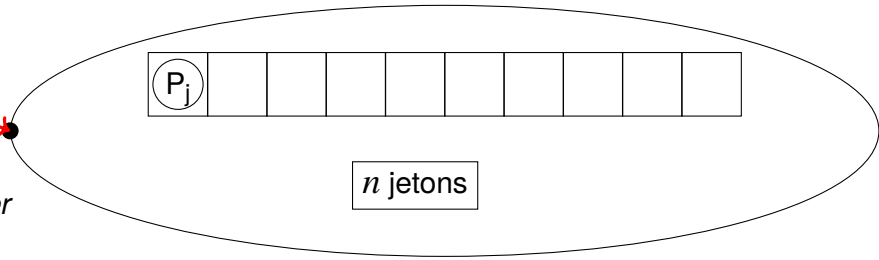
Le processus P_1 :

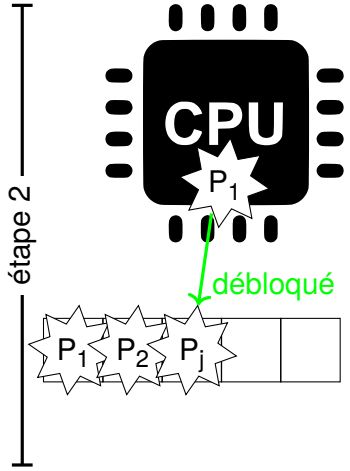
- ☐ est en cours d'exécution par le CPU ;
- ☐ appelle V(sem) ;

V(Sem)

liste processus à ordonnancer

***Sémaphore* Sem**



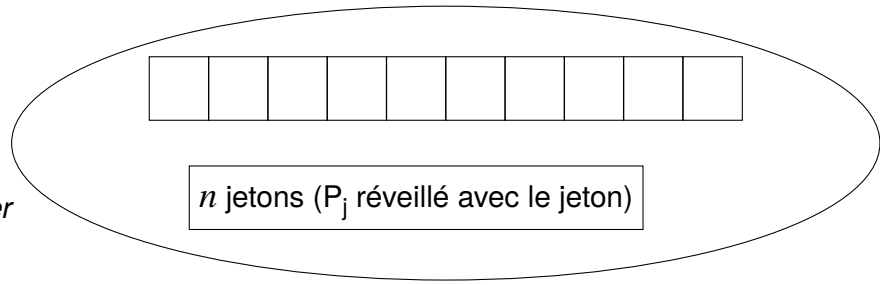


La sémaphore Sem vérifie la file d'attente :

- ☐ Le processus P_j est en **attente** $\Longrightarrow$
  - ◇ il devient **prêt** ;
  - ◇ P_j **reçoit le jeton** ;

liste processus à ordonnancer

***Sémaphore* Sem**

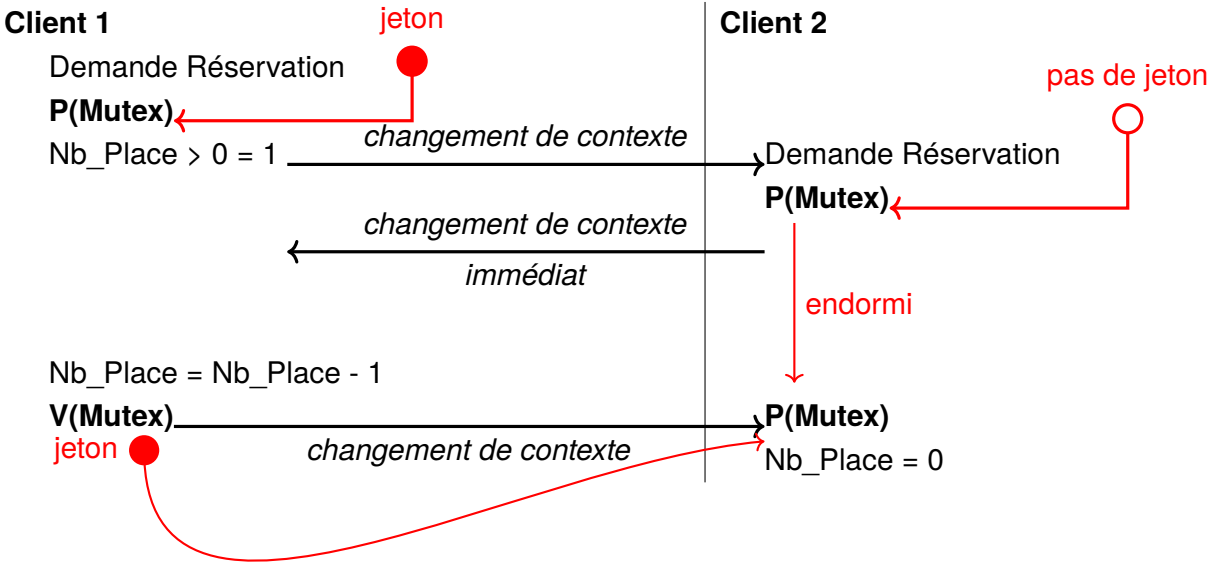


Un seul processus en section critique à la fois ⟹ un seul jeton

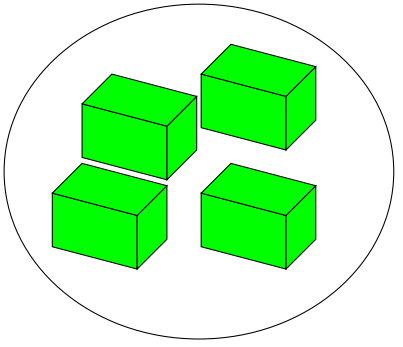
```
Sémaphore Mutex initialisée à 1
P(Mutex)          /* Entrée section_critique */
     |
     |            /* Section critique */
     |
V(Mutex)          /* Sortie section_critique */
```

```
1 Réservation :
2 P(Mutex)
3 Si nb_place > 0
4 alors
5    Réserver une place
6    nb_place = nb_place - 1
7 fsi
8 V(Mutex)
```

Utilisation par deux clients s'exécutant en concurrence avec exclusion mutuelle

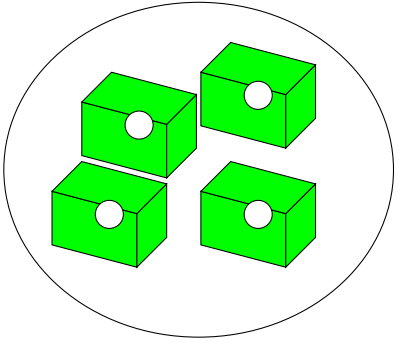


Accès à un ensemble de N ressources critiques



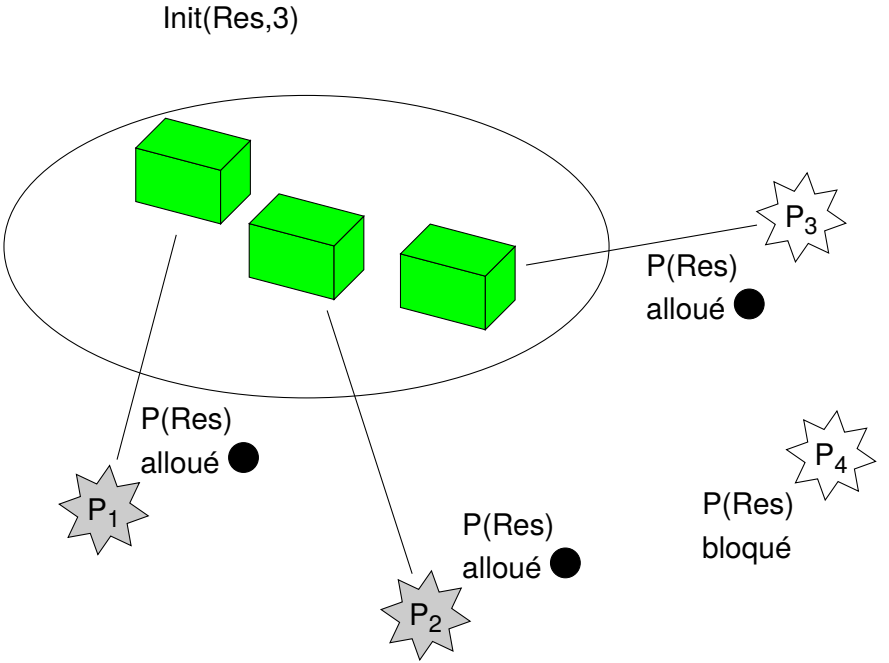
Un deuxième problème de **synchronisation** est celui de l'accès par un ensemble de processus à n ressources critiques :

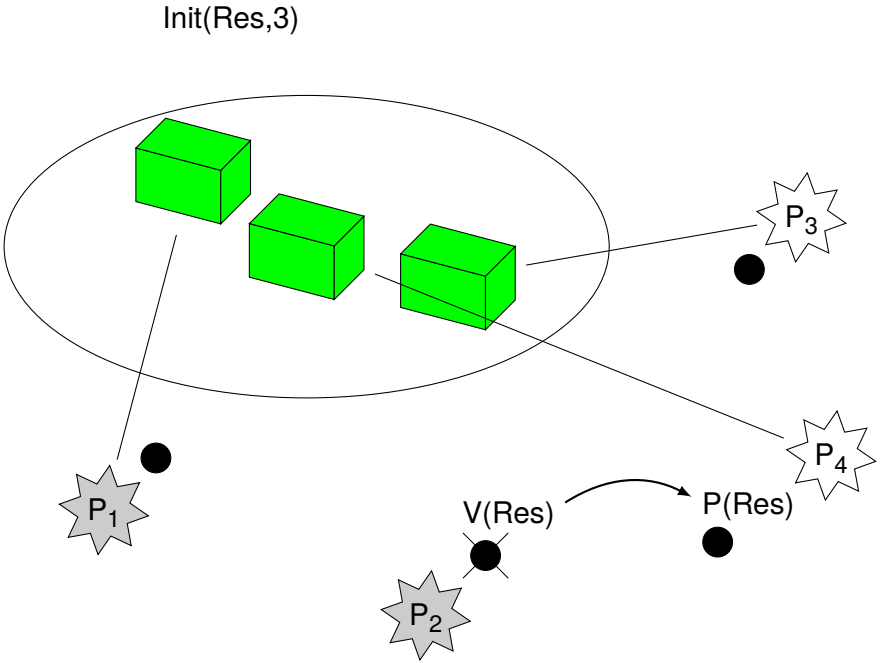
$\Longrightarrow$ c'est une **généralisation** du cas précédent :

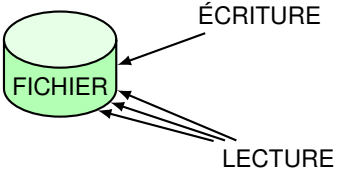


- N ressources exclusives de **même type**
- Allouer une ressource $\Longrightarrow$ prendre un jeton Res
- Libérer une ressource $\Longrightarrow$ rendre un jeton Res

```
Sémaphore Res initialisée à N /* 1 jeton par ressource */
P(Res) /* Allocation */

/* Utilisation Ressource */

V(Res) /* libération resource */
```


Init(Res,3)
P3
P(Res)
alloué
P(Res)
alloué
P1
P4
P(Res)
bloqué
P(Res)
alloué
P2


Init(Res,3)
P3
P4
P1
V(Res)
P(Res)
P2



- Le contenu du fichier doit rester **cohérent** : pas d'écritures simultanées ;
- Les lectures doivent être **cohérentes** : pas de lectures en même temps que les écritures.

Code du lecteur et rédacteur

- Lecteur :

```
Consulter_Fichier_annonce:
début
   ouvrir(fichier_annonce, lecture);
   pour chaque annonce du fichier faire
      lire(annonce) ;
   fin pour
   fermer(fichier_annonce);
Fin
```

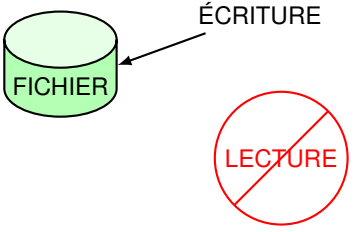
- Rédacteur :

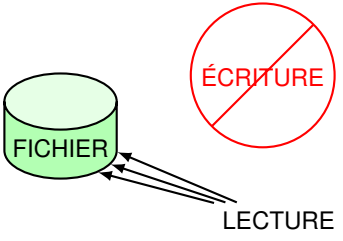
```
Modifier_Fichier_annonce (annonce, opération) avec opération=ajouter,modifier ou effacer
début
   ouvrir(fichier_annonce, ecriture);
   opération(annonce);
   fermer(fichier_annonce);
fin
```

Écriture seule – Lectures simultanées

Un **écrivain** exclut :

- ☐ les écrivains : un seul écrivain à la fois ;
- ☐ les lecteurs : pas d'écriture et de lecture simultanées.





Un **lecteur** exclut :

- ☐ les écrivains.

Solution ? Un écrivain exclut les écrivains et les lecteurs :

- ▷ Un écrivain accède toujours seul ;
- ▷ Un écrivain effectue des accès en **exclusion mutuelle** des autres écrivains et des lecteurs.

⟹ **Sémaphore d'exclusion mutuelle** «Accès» initialisée à 1.

Intégration de la sémaphore Accès

Travail de l'écrivain :

```
M'assurer que l'accès au fichier est libre
P(Accès)
   entrer en écriture
   Libérer l'accès au fichier
V(Accès)
```

d'où, le code de l'écrivain :

```
Modifier_Fichier_annonce (annonce, opération)
avec opération = ajouter, modifier ou effacer :
début
P(Accès)
   ouvrir(fichier_annonce, ecriture);
   opération(annonce);
   fermer(fichier_annonce);
V(Accès)
fin
```

Et pour les lecteurs ?

Un lecteur exclut les écrivains :

- NL, nombre de lecteurs courants, initialisé à 0 ;

$\Longrightarrow$ Un premier lecteur doit s'assurer qu'il n'y a **pas d'accès en écriture en cours** ;

$\Longrightarrow$ Le dernier lecteur doit réveiller un éventuel Écrivain

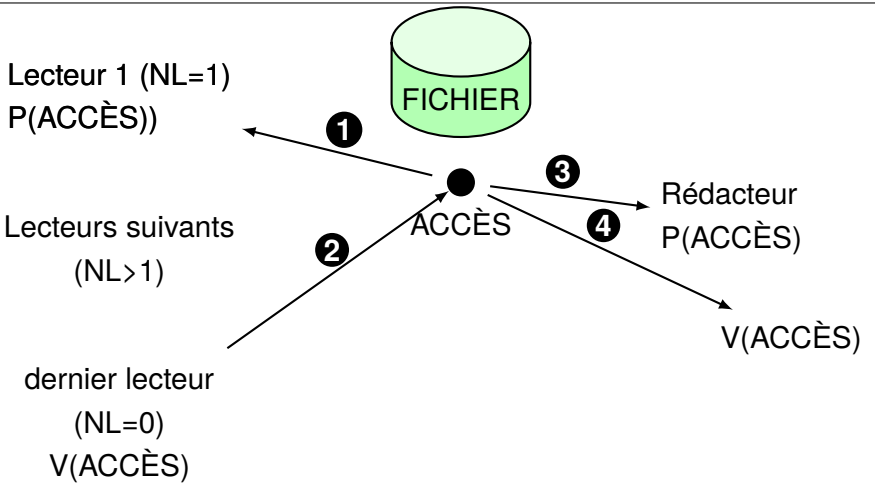
```
Compter un lecteur de plus
Si je suis le premier lecteur alors
   Y-a-t-il un écrivain ?
   si oui, attendre
Fsi
```

```
P(Mutex)
NL : = NL + 1
Si (NL = 1) alors
   P(Accès)
fsi
V(Mutex)
```

```
entrer en lecture
```

```
Compter un lecteur de moins
Si je suis le dernier, réveiller un écrivain
```

```
P(Mutex)
NL:= NL - 1;
Si (NL = 0) alors
   V(Accès)
fsi
V(Mutex)
```



❶ le premier lecteur prends la sémaphore «Accès», ce qui interdit à l'écrivain de pouvoir la prendre ;

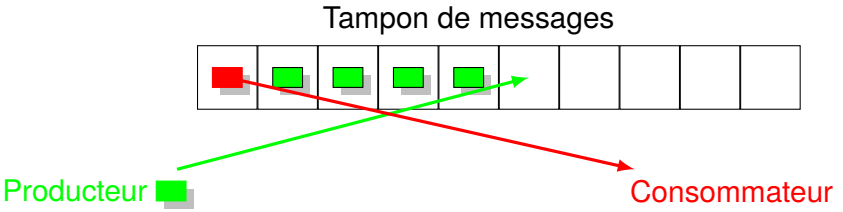
◯ de nouveaux lecteurs peuvent se présenter et consulter le contenu du fichier ;

❷ le dernier lecteur libère la sémaphore «Accès», ce qui permet au rédacteur de la prendre ;

❸ le rédacteur peut prendre la sémaphore ;

❹ le rédacteur peut ensuite libérer la sémaphore.

Gestion d'un tampon de messages en FIFO



- Le producteur ne doit **pas produire** si le tampon est **plein** ;
- Le consommateur de doit **pas faire de retrait** si le tampon est **vide** ;
- Producteur et Consommateur ne doivent **jamais travailler** dans une même case.

Producteur :

```
Si il n'y a pas de case libre alors
   attendre
sinon
   déposer le message
fsi
```

Consommateur :

```
Si il y a pas de case pleine alors
   attendre
sinon
   prendre le message
fsi
```

Solution ?

- ▷ Modéliser le «*tampon*» par une sémaphore de n jetons ;
- ▷ le «*jeton*» représente le «*produit*» ;

⟹ le consommateur est **bloqué** s'il n'y a pas de produit/jeton, ok...

⟹ le producteur n'est **jamais bloqué** !

⟹ On ne peut pas y arriver avec **une sémaphore** !

Solution

- On associe un ensemble de jetons aux **cases vides** :
  - N jetons VIDE à l'initialisation ;
  - deposer le message = prendre un jeton VIDE
- On associe un ensemble de jetons aux **cases pleines** :
  - 0 jetons PLEIN à l'initialisation ;
  - prendre le message = prendre un jeton PLEIN.

```
Producteur
Si il n'y a pas de case libre alors   } P(Sem Vide)
   attendre
sinon
   déposer le message
   nouvelle case pleine  <-------- V(Sem Plein)
fsi
```

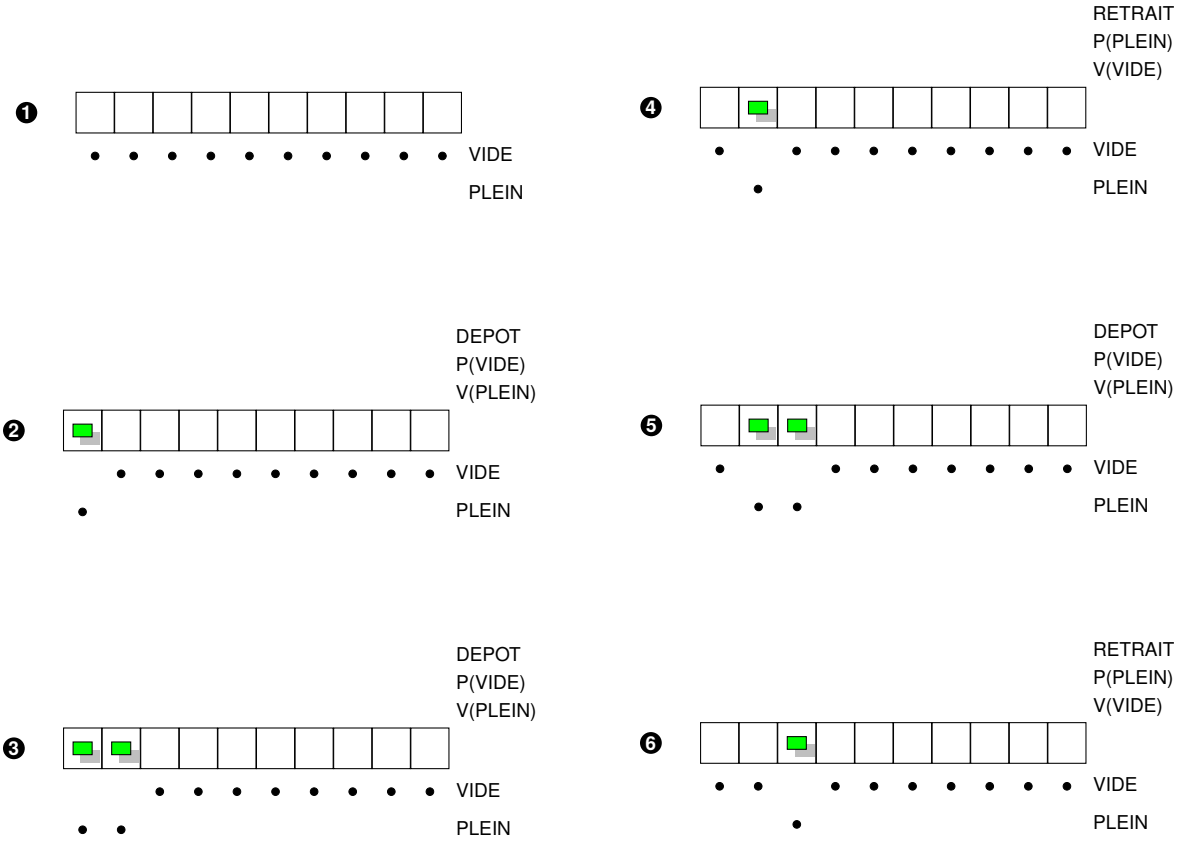
```
Consommateur
Si il y a pas de case pleine alors   } P(Sem Plein)
   attendre
sinon
   prendre le message
   nouvelle case vide  <-------- V(Sem Vide)
fsi
```

D'où le code final :

```
Sémaphore Vide initialisé à N : Init (Vide, N)
Sémaphore Plein initialisé à 0 : Init (Plein, 0)
```

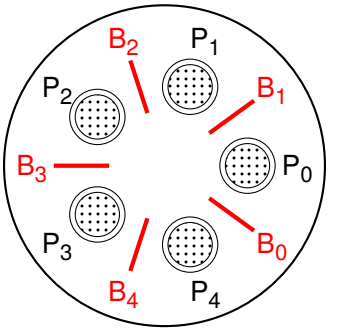
```
Producteur :
P(Vide)
   déposer le message
V(Plein)
```

```
Consommateur :
P(Plein)
   retirer le message
V(Vide)
```


❶
VIDE
PLEIN
❷
DEPOT
P(VIDE)
V(PLEIN)
VIDE
PLEIN
❸
DEPOT
P(VIDE)
V(PLEIN)
VIDE
PLEIN
❹
RETRAIT
P(PLEIN)
V(VIDE)
VIDE
PLEIN
❺
DEPOT
P(VIDE)
V(PLEIN)
VIDE
PLEIN
❻
RETRAIT
P(PLEIN)
V(VIDE)
VIDE
PLEIN

Hypothèses

- chaque assiette a une place fixe ;
- chaque baguette a une place fixe ;
- l'accès à chaque baguette est en **exclusion mutuelle** ;



Propriétés souhaitées

- pas d'interblocage ;
- pas de famine.

Première solution

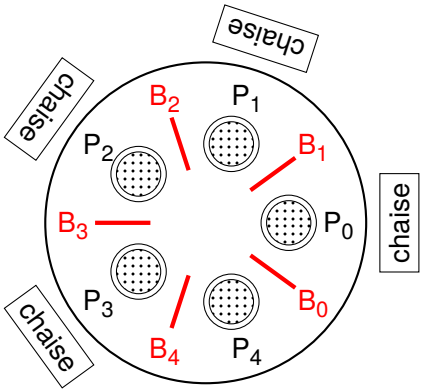
- on associe une sémaphore :
  - à chaque baguette B_i ;
  - à chaque assiette/philosophe P_i ;
- chaque philosophe pense, puis mange et recommence.

```
Sémaphore baguette[n] // chaque sémaphore doit être initialisée à 1
processus Philosophe(i) // i identifiant 0..n-1
tant que vrai faire
   penser()
   P(baguette[i])    // prendre la baguette i
   P(baguette[(i+1)%n]) // prendre la baguette de droite i+1
   manger()
   V(baguette[i])    // rendre la baguette i
   P(baguette[(i+1)%n]) // rendre la baguette de droite i + 1
```

$\Longrightarrow$ ***Interblocage possible !***

Seconde solution

```
Sémaphore baguette[n] // chaque sémaphore doit être initialisée à 1
Sémaphore chaise // initialisée à n-1

processus Philosophe(i) // i identifiant 0..n-1
tant que vrai faire
   penser()
   P(chaise)
   P(baguette[i])   // prendre la baguette i
   P(baguette[(i+1)%n]) // prendre la baguette de droite i+1
   manger()
   V(baguette[i])   // rendre la baguette i
   V(baguette[(i+1)%n]) // rendre la baguette de droite i + 1
   V(chaise)
```



La **solution** pour éviter l'interblocage :

▷ autoriser seulement $n-1$ philosophes à manger ;

▷ $\Longrightarrow n-1$ chaises !

$\Longrightarrow$ **pas de famine** si la file de chaise est gérée à l'ancienneté.

Conclusion sur les sémaphores

- Les exécutions de processus ne sont pas **indépendantes** : les processus peuvent vouloir
  - **communiquer** ;
  - accéder de manière **concurrente** à des ressources ;

- La **sémaphore S** est un **outil système** de synchronisation :
  - assimilable à un **distributeur de jeton bloquant** ;
  - manipulable par seulement **trois opérations atomiques** P(S), V(S) et Init(S) ;

- Il existe **plusieurs schémas typiques** de synchronisation à partir desquels sont élaborés des outils de communication entre processus :
  - **l'exclusion mutuelle** ;
  - **les lecteurs/rédacteurs** ;
  - **les producteur/consommateur** ;
  - **le repas des philosophes**.

Pour utiliser les sémaphores, il faut :

- inclure le fichier de définition :

```
#include <semaphore.h>
```

- relier avec la bibliothèque `-lpthread`

```
xterm
$ gcc -o mon_programme mon_source.c -lpthread
```

Pour créer la sémaphore

```
sem_t semaphore;

sem_init(&semaphore, p, n);
```

- *p* : indique si la sémaphore doit être partagée entre différents processus ;
  on mettra la valeur 0 pour indiquer qu'elle n'est pas partagée entre processus mais entre threads.
- *n* : indique la valeur initiale de la sémaphore.

Prendre et libérer la sémaphore

- prendre la sémaphore :

```
sem_wait(&semaphore);
```

- libérer la sémaphore :

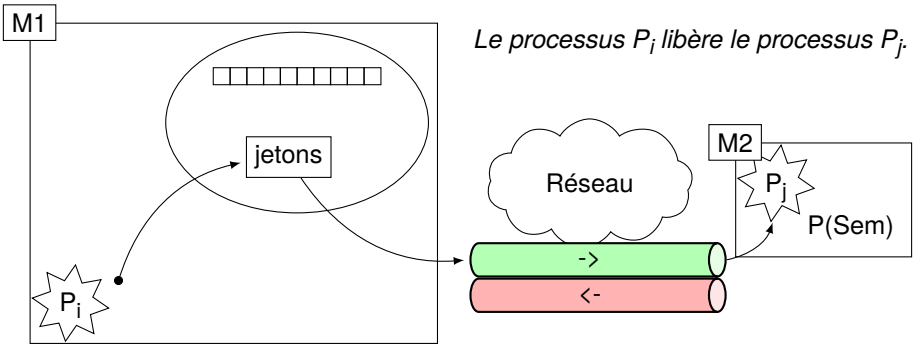
```
sem_post(&semaphore);
```

Pour détruire la sémaphore

```
sem_destroy(&semaphore)
```

Une sémaphore peut être utilisée pour **synchroniser** différents processus :

- dans **la même machine** : l'éxecution, le blocage et l'échange des jetons a lieu au sein du même système d'exploitation ;
- entre **différentes machines au travers d'un réseau de communication**.
  Exemple entre deux machines M_1 et M_2 :
  - un processus P_i s'exécute sur la machine M_1 : il gère la sémaphore (comportement serveur) ;
  - un processus P_j s'exécute sur la machine M_1 : il accède à la sémaphore de P_i au travers du réseau par deux tubes qui le synchronisent avec la sémaphore, chacun utilisé pour un sens différent d'échange (comportement client) :
    - il **lit** sur le tube :
      - **sans blocage** si la sémaphore contient un jeton, ce jeton est décrémenté du compteur : il obtient un «jeton» ;
      - **avec blocage** si la sémaphore ne contient plus de jeton : il sera débloqué lorsqu'un processus rendra un jeton qu'il pourra ensuite lire sur le tube ;
    - il **écrit** sur le tube un jeton qui s'ajoute au compteur, ce qui peut ou non débloquer un processus en attente de jeton.

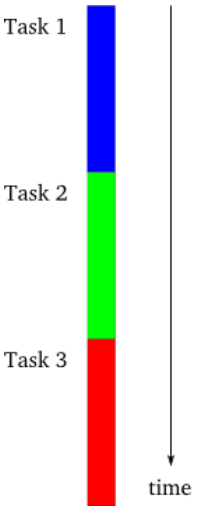


Le processus P_i libère le processus P_j.

On peut généraliser ce fonctionnement à plusieurs machines clientes utilisant la même sémaphore.

Qu'est-ce que la programmation asynchrone ?

Le contraire de la programmation synchrone !

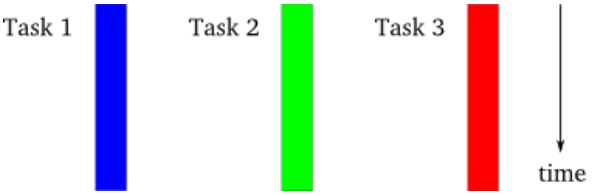


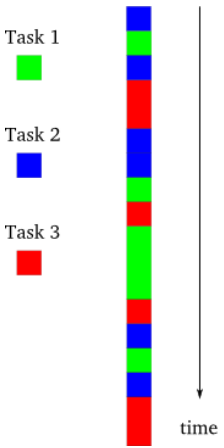
Imaginons que nous voulions exéctuer un programme décomposable en 3 tâches «*Task{1,2,3}*» :

- la version «*single threaded*» indiquée à gauche est la version la plus simple : chaque tâche est exécutée une à la fois, l'une s'achevant **complètement** avant que l'autre ne commence :
  - les tâches s'exécutent dans un **ordre prédéfini** ;
  - la dernière tâche peut ainsi être **sûre** que :
    - les tâches précédentes se sont exécutées **sans erreur** ;
    - que tous les **résultats** de ces tâches précédentes sont **disponibles**.

La programmation asynchrone *vs* programmation «multi-thread»

- la version «*multi-threaded*» permet à chaque tâche de s'exécuter dans une *thread* de contrôle différente :
  - les threads sont **ordonnancées** par le système d'exploitation ;
  - les threads peuvent s'exécuter de manière **concurrente** :
    - **réellement** si l'ordinateur dispose de différents processeurs ou cœurs ;
    - de manière **entrelacée** si l'ordinateur ne dispose que d'un processeur ;
  - le programmeur pense en terme de **flots d'instructions** s'exécutant simultanément ;
  - la coordination et la communication entre ces différentes threads est **complexe** et **difficile**.





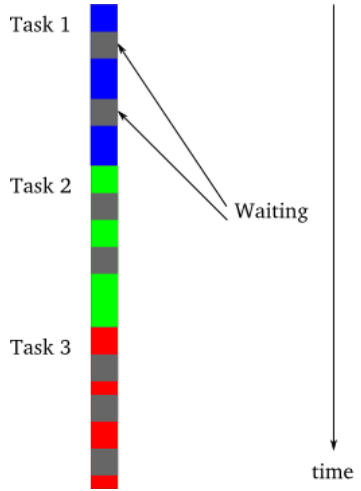
- la version «asynchrone» :
  - les tâches s'exécutent de manière **entrelacées** au sein de la même «*thread*» de contrôle ;
  - le modèle de programmation est **plus simple** que celui «*multi-threaded*» car le programmeur sait «qu'une seule tâche s'exécute à la fois» :

    lorsqu'une tâche est exécutée, aucune autre ne s'exécute
  - ***bien que*** *dans un environnement avec un seul processeur, la version «multi-thread» s'exécute aussi de manière entrelacée, le programmeur* ***doit toujours penser*** *en terme de threads pouvant s'exécuter simultanément* ;

- la version «asynchrone» s'exécute **toujours** de manière entrelacée même dans un environnement multi-processeurs ;
- dans la version «multi-thread», **c'est le système d'exploitation** qui décide de l'ordonnancement des différentes threads et le programmeur doit en tenir compte : chaque thread peut être suspendue à n'importe quel moment pour laisser la place à l'exécution d'une autre thread.
- dans le modèle «asynchrone», une tâche s'exécute jusqu'à ce qu'**elle décide de libèrer** le contrôle à une autre tâche.

La programmation asynchrone : mieux utiliser le processeur

Quel est l'intérêt de la programmation «asynchrone» ? *les performances !*



- le modèle «asynchrone» est **plus simple** que le modèle «multi-thread» : un seul fil de contrôle et le passage du contrôle d'une tâche à une autre est **explicite** ;
- le modèle «asynchrone» impose au programmeur de «penser» chaque tâche comme une succession d'étapes plus petites s'exécutant de manière **intermittente** :
  - si une tâche utilise la sortie d'une autre tâche, la tâche dépendant de cette sortie doit être écrite de manière à obtenir ses entrées par **morceaux** au lieu d'une réception en totalité ;
- le modèle «asynchrone» **n'exploite pas de parallélisme** : le temps d'exécution de l'application est le même que dans le modèle synchrone…
- **Mais** lorsqu'une tâche est forcée d'attendre ou est bloquée en attente d'une ressource externe (accès réseau, accès disque, *etc*), c'est l'**ensemble** de l'application qui est **retardé** : le temps d'exécution du modèle «asynchrone» peut être très **nettement inférieur** à celui synchrone (voir schéma du modèle asynchrone) !

- lorsqu'une tâche est **bloquée dans le modèle synchrone**, il est possible d'exécuter une tâche qui est **capable de progresser** :
  - le passage d'une tâche à une autre correspond :
    - soit à la terminaison de la tâche courante,
    - soit à l'arrivée à un point où la tâche va se bloquer ;
  - avec un nombre important de tâches pouvant potentiellement se bloquer, le modèle «asynchrone» peut **obtenir de bien meilleures performances** que le modèle «synchrone».

Le modèle «asynchrone» est meilleur dans le cas où :

- il y a de **nombreuses tâches** à réaliser, de telle manière à ce qu'il y est toujours une tâche capable de progresser dans son exécution ;
- les tâches réalisent **beaucoup d'entrées/sorties**, qui méneraient un programme synchrone à gaspiller beaucoup de temps bloqué alors qu'une tâche serait capable de s'exécuter ;
- les tâches sont **indépendantes** les unes des autres, ce qui fait qu'il y a peu de communication nécessaire entre elles.

Dans le cas de nombreuses entrées/sorties

Les performances sont très différentes en fonction des périphériques :

device	périphérique	cycles d'horloge
L1-cache	cache de niveau 1 sur le processeur	3 cycles
L2-cache	cache de niveau 2 sur le processeur	14 cycles
RAM	mémoire vive	250 cycles
Disk	disque dur	41 000 000 cycles
Network	réseau	240 000 000 cycles

L'utilisation d'une **seule thread** permet de travailler en gaspillant le moins possible de cycles d'horloge : moins de défaut de cache, pas de changement de contexte.

Les entrées/sorties mémoire, disque et réseau étant très lentes, on peut effectuer beaucoup de travail pendant qu'elles se déroulent, ce qui justifie l'utilisation d'E/S **asynchrones**.

La programmation asynchrone : intérêt ?

Dans quel domaine d'application le modèle asynchrone peut-il s'appliquer ? *le réseau !*

Une application serveur (comme un serveur Web) peut tirer parti de ce modèle :

- les bibliothèques «Twisted» et «Tornado» pour Python ;
- le serveur applicatif basé sur Javascript «Node.js».

Pourquoi le modèle «multi-threaded» n'est-il pas une bonne réponse ?

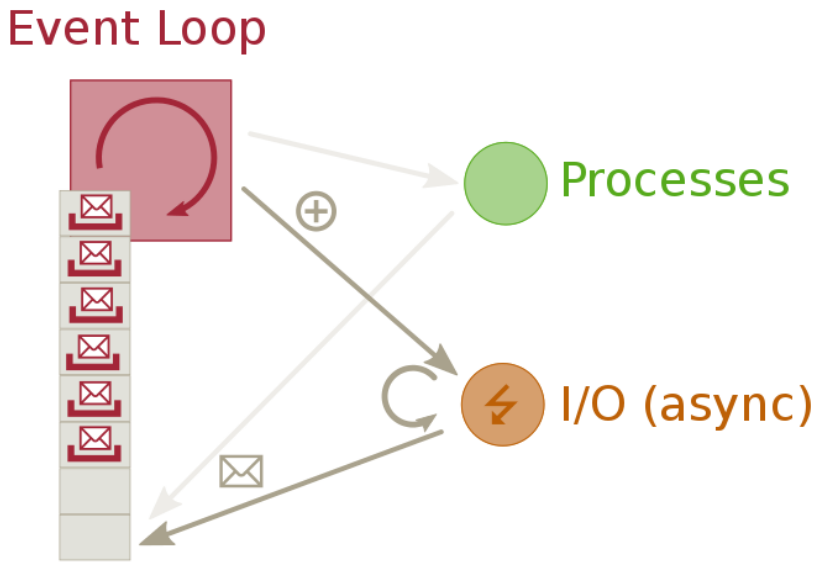
- lorsqu'une thread se bloque sur une entrée/sortie, une autre thread peut s'exécuter : pareil que dans le modèle asynchrone ;
- **mais** avec chaque thread, il y a un **surcoût** :
  - de **mémoire** : une **pile d'exécution** associée à chaque thread plus les structures diverses de gestion et de protection des threads ;
  - de **temps** : pour passer d'une thread à une autre : **changement de contexte** (sauvegarde de registres, manipulation mémoires *etc.*) et restauration du contexte de la nouvelle thread.

Ce surcoût peut **alourdir considérablement** le travail de gestion d'un serveur.

```
1  var produit = 0;
2
3  function appel_producteur() {
4      producteur.next();
5  }
6
7  function* p() {
8      var compteur = 0;
9      while(true)
10     {
11         compteur++;
12         produit = compteur;
13         console.log("Production " + produit);
14         setTimeout(appel_consommateur,0);
15         yield null;
16     }
17 }
```

```
1  function appel_consommateur() {
2      consommateur.next();
3  }
4
5  function* c() {
6      while(true)
7      {
8          console.log("Consommation "+produit);
9          setTimeout(appel_producteur, 0);
10         yield null;
11     }
12 }
13
14 var producteur = p();
15 var consommateur = c();
16
17 producteur.next();
```

- ▷ l'instruction «`yield`» permet de retourner une valeur et de conserver l'état de la fonction. L'appel à la méthode «`next()`» permet de retourner à l'instruction suivant le «`yield`».
- ▷ l'instruction «`setTimeout(function,0)`» permet d'ajouter une fonction à exécuter dans la boucle d'événement de NodeJs.
  Cet événement ne sera déclenché que lors de la terminaison de la fonction courante : *on prépare l'avenir de l'application*.



L'«event loop» correspond à une file d'événements qui peuvent déclencher l'exécution de «processes» qui sont suspendus en attente de leur arrivée.

Les E/S asynchrones :

- *se déroulent pendant qu'un «process» s'exécute ;*
- *produisent des événements lorsqu'elles se terminent qui pourront ensuite déclencher l'exécution de «process».*

*L'**Event Loop** représente le «**fil de contrôle**» de l'application asynchrone.*

- un «*process*», fonction ou méthode, ou une E/S asynchrone qui se termine ⟹ retour à l'**Event Loop** ;
- un événement ajouté lors la terminaison de l'E/S ou lors d'une soumission d'un «*process*» ne peut être pris en compte qu'à la fin du «*process*» courant (lors du retour à l'**Event Loop**) ;
- un «*process*», qui s'exécute peut **ajouter un événement**, lié à un délai ou à une E/S, en indiquant la fonction à exécuter lors de son déclenchement (*callback*).
  Le plus simple des événements correspond à celui d'un temps d'attente, qui s'apparente à de l'ordonnancement («setTimeout» avec la valeur zéro).

```
 1 var liste_philosophes = [];
 2 var liste_philosophes_a_traiter = null;
 3 var philosophe_a_reveiller = null;
 4
 5 var compteur_chaises = 4;
 6
 7 function reveil_philosophe()
 8 {
 9     philosophe_a_reveiller.next();
10 }
11
12 function* philosophe(n)
13 {
14     while(true)
15     {
16         if (compteur_chaises > 0)
17         {
18             compteur_chaises --;
19             console.log("Philosophe : "+n+" prend une chaise");
20             yield null;
21             console.log("Philosophe : " + n + " mange");
22             yield null;
23             console.log("Philosophe : " + n + " se leve");
24             compteur_chaises ++;
25         }
26         else
27         {
28             console.log("Philosophe : " + n + " reflechit");
29         }
30         yield null;
31     }
32 }
```

*C'est l'ordonnanceur qui garantit l'**équité** de traitement entre les philosophes.*
C'est le programmeur qui est responsable de l'entrelacement.

Il n'y a pas de problème d'accès concurrent aux variables partagées.

- lignes 7-10 : la fonction de réveil d'un philosophe qui sera attachée à un événement de l'«**Event Loop**» ;
- lignes 12-32 : le travail du philosophe décomposé en étapes :
  - ◇ chaque étape se termine sur un «`yield`» pour permettre de :
    - ⋆ retourner à l'«**Event Loop**» ;
    - ⋆ traiter des événements présents : activer l'ordonnanceur ;
  - ◇ cette décomposition permet d'**entrelacer** le travail des différents philosophes.

```
33 for(i=0; i<5; i++)
34         liste_philosophes.push(philosophe(i));
35
36 liste_philosophes_a_traiter = liste_philosophes.slice(0);
37
38 function scheduler()
39 {
40      var numero_philosophe = Math.floor(Math.random() * liste_philosophes_a_traiter.length);
41      philosophe_a_reveiller = liste_philosophes_a_traiter[numero_philosophe];
42      liste_philosophes_a_traiter.splice(numero_philosophe, 1);
43      if (liste_philosophes_a_traiter.length == 0)
44      {
45              liste_philosophes_a_traiter = liste_philosophes.slice(0);
46      }
47      setTimeout(reveil_philosophe,0);
48      setTimeout(scheduler,0);
49 }
50
51 scheduler();
```

- ligne 33 :
  - on lance 5 fonctions philosophes en état suspendu (utilisation de la déclaration «`function*`») ;
  - on ajoute chacune de ces fonctions suspendues dans une liste ;
- ligne 36 : on duplique la liste des philosophes ;
- ligne 38 : la fonction «`scheduler`» permet de :
  - ligne 40 : de choisir un philosphe de manière aléatoire dans la liste des philosophes à traiter ;
  - ligne 41 : on positionne le philosophe à réveiller sur celui que l'on vient de sélectionner $\Longrightarrow$ un événement le réveillera dans la fonction «`reveil_philosophe`» ;
  - ligne 42 : on enlève le philosophe choisi de la liste qui sera réinitialisée en ligne 43-46 quand elle est vide ;
- lignes 47-48 : l'ordonnancement : ré-activation du philosophe et rappel de l'ordonnanceur.