

# Les processus

**Thomas Vantroys**

[thomas.vantroys@polytech-lille.fr](mailto:thomas.vantroys@polytech-lille.fr)

Polytech'Lille  
Université de Lille

**IMA4**  
2020 - 2021



Université  
de Lille

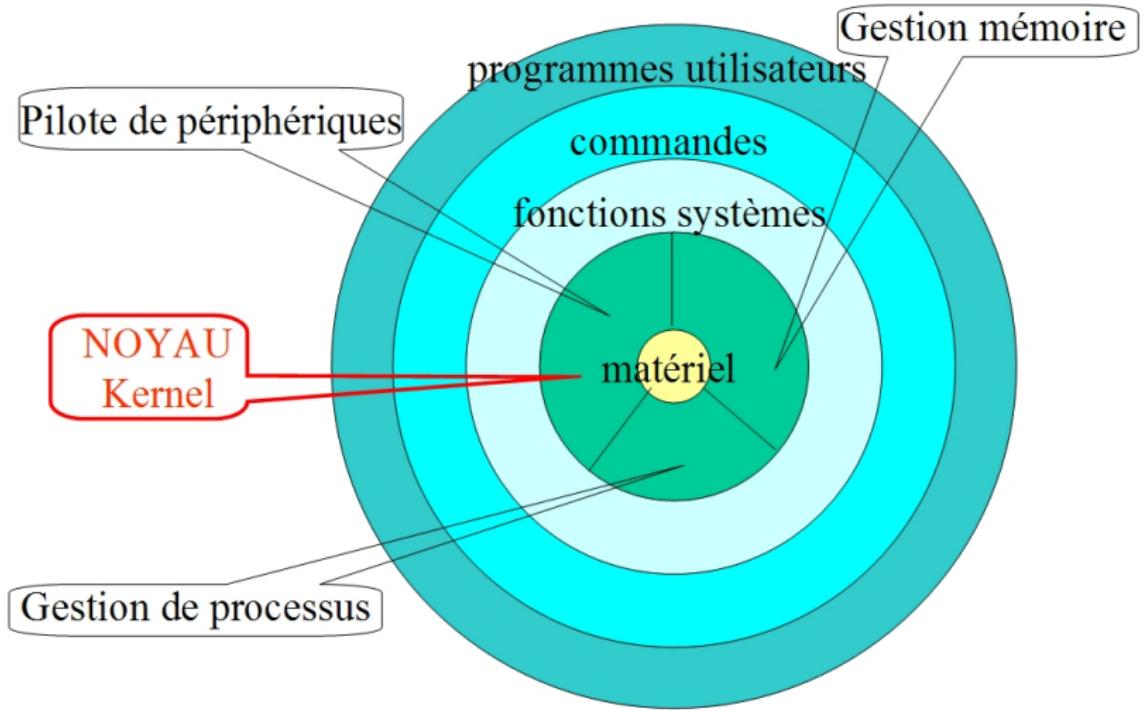


POLYTECH®  
LILLE

# Les processus

- état d'un processus
- interruption
- changement de contexte
- ordonnancement des processus
- synchronisation des processus
- communication entre processus

# Cœur du système d'exploitation



# Les processus

- concept clé de tous les systèmes d'exploitation
- un processus est le procédé temporel d'exécution d'un programme
- il est caractérisé par :
  - son état
  - son identificateur
  - son compteur ordinal
  - sa pile d'exécution
  - pointeur de pile et autre registres
  - ses données
  - toutes informations utiles à son exécution (E/S, fichiers ouverts, ...)
- l'état d'un processus évolue dans le temps

# Mode noyau et mode utilisateur

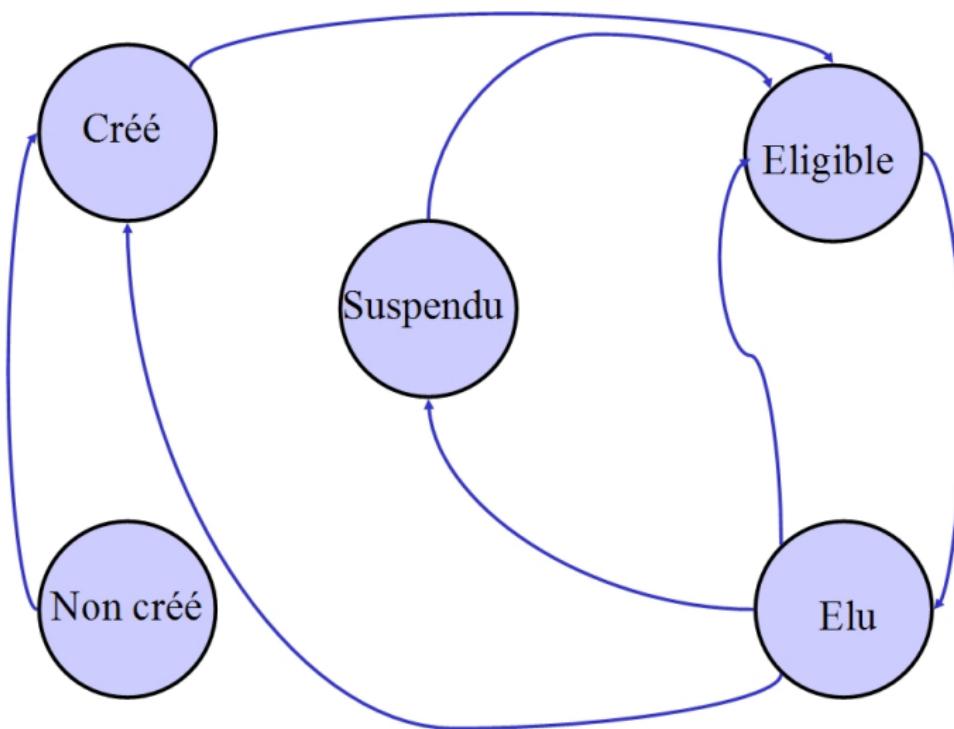
Un processus possède deux niveaux d'exécution :

- **mode noyau** : mode privilégié : accès sans restriction  
(manipulation de la mémoire, dialogue avec les contrôleurs de périphériques, ...)
- **mode utilisateur** : mode d'exécution normal d'un processus.  
Il ne possède aucun privilège et certaines instructions lui sont interdites. Il peut être interrompu à tout moment.

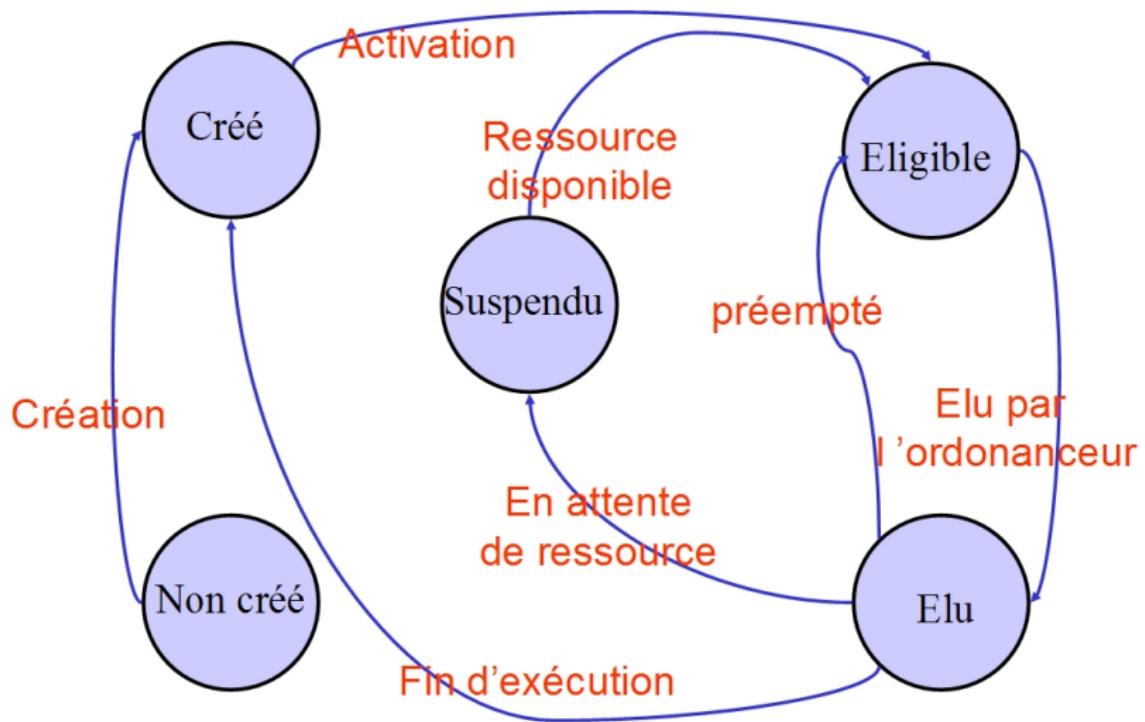
Appels systèmes :

- permet à un processus en mode utilisateur d'accéder à des fonctions nécessitant le mode noyau

État d'un processus



## État d'un processus



# État d'un processus

- **non créé**

- le code n'est pas écrit ou n'est pas en mémoire

- **créé**

- le code est en mémoire, en attente d'activation

- **éligible**

- l'activation a été demandée. il ne manque qu'une ressource, le PROCESEUR

- **élu**

- état où le processus s'exécute

- **suspendu**

- lors de l'exécution, une ressource est indisponible

# État d'un processus

- **non créé ➔ créé**
  - création du code et implantation mémoire
- **créé ➔ éligible**
  - une activation a été demandée
- **éligible ➔ élu**
  - c'est l'ordonnanceur qui choisit le processus
- **élu ➔ suspendu**
  - le processus a demandé une ressource indisponible
- **suspendu ➔ éligible**
  - la ressource est devenue disponible

# API Création d'un processus

Création d'un processus fils :

- `pid_t fork(void)`

Le fils hérite :

- du même code
- d'une copie de la zone de données
- d'une copie de la zone de pile
- de l'environnement
- du propriétaire
- des descripteurs de fichiers ouverts
- du traitement des signaux

La distinction père/fils grâce à la valeur de retour.

# API Synchronisation

Attendre la fin d'un fils :

- `pid_t wait(int *status)`
- `pid_t waitpid(pid_t pid, int *status, int options)`

La fonction `wait` est bloquante.

# Identificateurs d'un processus

Plusieurs identificateurs sont associés à un processus :

- numéro de processus (pid)
- identificateur d'utilisateur réel
- identificateur d'utilisateur effectif (bit setuid)
- identificateur de groupe réel
- identificateur de groupe effectif (bit setgid)
- liste d'identificateurs de groupes

Possibilité de connaître ces informations dynamiquement via des appels systèmes

# API Lecture des attributs

- `pid_t getpid(void)`
- `pid_t getppid(void)`
- `uid_t getuid(void)`
- `uid_t geteuid(void)`
- `gid_t getgid(void)`
- `gid_t getegid(void)`
- `int getgroups(int size, gid_t list[])`

# API Modification des attributs

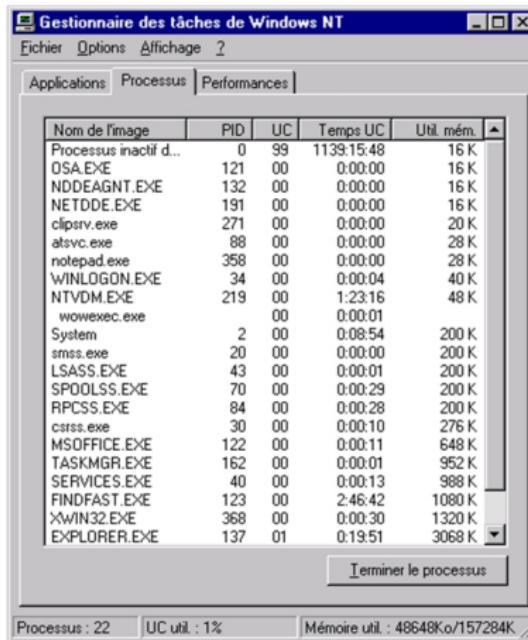
- `int setuid(uid_t uid)`
- `int setreuid(uid_t ruid, uid_t euid)`
- `int seteuid(uid_t euid)`
- `int setgid(gid_t gid)`
- `int setregid(gid_t rgid, gid_t egid)`
- `int setegid(gid_t egid)`

# API Exécution de programmes

- `int execl(const char *path, const char *arg0,  
..., const char *argn, NULL)`
- `int execlp(const char *path, const char *arg0,  
..., const char *argn, NULL)`
- `int execv(const char *path, char *argv[])`
- `int execvp(const char *path, char *argv[])`

# Visualisation des processus

Sous WinNT :



# Visualisation des processus

Sous UNIX :

- l'information sur les processus en cours peut être obtenue par :

**ps [options]**

- pour les options consultez le **man**

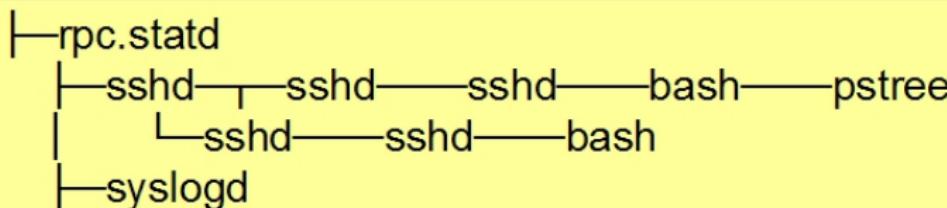
```
jduthill@pevele:~ >ps -eo pid,ppid,tt,user, fname|grep jduthill
25716 25715 tttyp3 jduthill bash
23226 23225 tttyp4 jduthill bash
23231 23226 tttyp4 jduthill top
23234 25716 tttyp3 jduthill ps
23235 25716 tttyp3 jduthill grep
```

jduthill@pevele:~ >ps	PID	TTY	TIME	CMD
	25716	tttyp3	00:00:00	bash
	23442	tttyp3	00:00:00	ps

# Visualisation des processus

Sous UNIX :

- l'information sur les processus en cours peut être obtenue par :  
**pstree [options]**



```
jduthill@pevele:~ >ps -eo pid,ppid,tt,user, fname|grep jduth
25716 25715 tttyp3 jduthill bash
23226 23225 tttyp4 jduthill bash
23231 23226 tttyp4 jduthill top
23234 25716 tttyp3 jduthill ps
23235 25716 tttyp3 jduthill grep
```

# Visualisation des processus

Sous UNIX :

- visualisation dynamique des processus :

**top [options]**

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	LIB	%CPU	%MEM	TIME	COMMAND
23231	jduthill	19	0	1308	1308	688	R	0	1.7	0.6	0:09	top
549	root	2	0	4312	4260	916	S	0	0.1	2.2	20:30	named
25818	root	1	0	1196	1172	1060	S	0	0.1	0.6	0:00	apache
1	root	0	0	452	452	396	S	0	0.0	0.2	0:08	init
2	root	0	0	0	0	0	SW	0	0.0	0.0	0:01	kflushd
3	root	0	0	0	0	0	SW	0	0.0	0.0	0:13	kupdate
4	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	kpiod
5	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	kswapd
6	root	0	0	0	0	0	SW	0	0.0	0.0	0:00	md_thread
504	root	1	0	388	372	272	S	0	0.0	0.1	34:24	syslogd

# Visualisation des processus

Sous UNIX :

- interruption d'un processus en cours :

**kill [numéro\_de\_signal] numéro\_de\_processus**

- **kill -l** donne l'ensemble des signaux disponibles

- exemple de signaux :

- SIGHUP (1) : émis à tous les processus associés à un terminal lorsque celui-ci se déconnecte
- SIGINT (2) : émis à tous les processus associés à un terminal lorsque <ctrl + C> est tapé
- SIGKILL (9) : tue un processus quel que soit son état. C'est l'arme absolue.
- SIGTERM (15) : signal de terminaison normale d'un processus

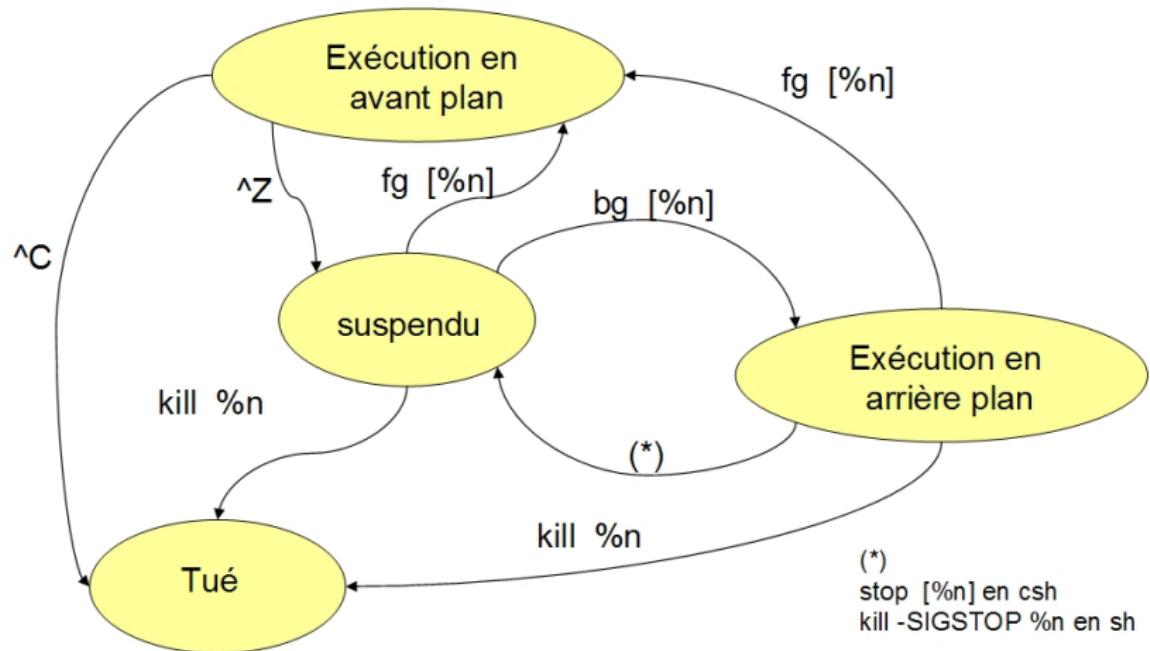
# Visualisation des processus

- lancement de commandes en arrière plan
  - il est possible de lancer une commande sans que le shell courant en attende la terminaison :  
**commande &**
  - elle ne peut plus lire au clavier
  - les sorties standards sont toujours associées, par défaut, au terminal
  - elles ne sont plus interruptibles à partir du clavier (`ctrl-c` par exemple)
- connaître les processus lancés en arrière plan
  - donne tous les processus lancés en arrière plan dans un shell donnée :

**jobs**

# Visualisation des processus

Sous UNIX :



# Classification des systèmes

- **gestion des processus**

- mono-tâche : CPU dédié à un processus
- multi-tâche : CPU partagé entre les processus
- multi-tâche préemptif : processus gérés par un ordonnanceur

- **gestion des utilisateurs**

- mono-utilisateur : pas de cohabitation
- multi-utilisateur : cohabitation possible

# Exemples de système d'exploitation

- MS-DOS : mono-utilisateur, mono-tâche
- Windows : mono-utilisateur, multi-tâche
- WinNT : mono-utilisateur, multi-tâche préemptif
- OS/2 : mono-utilisateur, multi-tâche préemptif
- UNIX : multi-utilisateur, multi-tâche préemptif

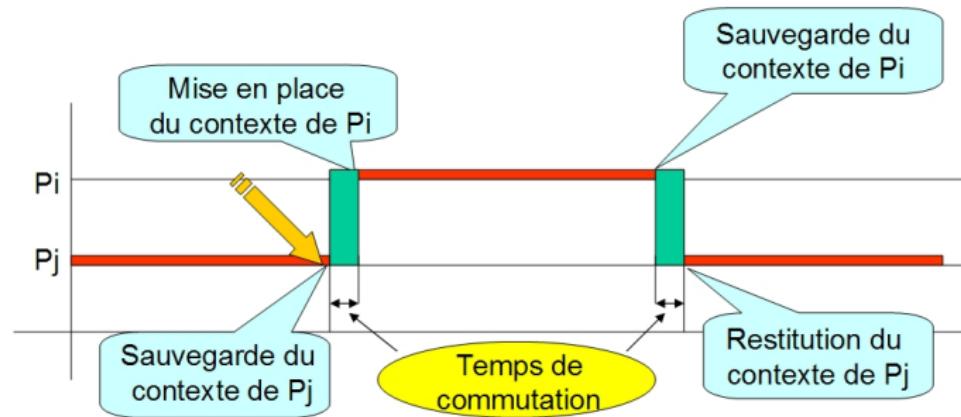
# Changement de contexte

- sans réquisition du processeur
  - le processus est exécuté jusqu'à ce qu'il fasse appel à un service du noyau
  - il n'est peut être pas nécessaire de sauvegarder le contexte en entier
- avec réquisition du processeur
  - le processus peut, à tout instant, perdre le processeur au bénéfice d'un processus de priorité supérieure

# Changement de contexte

Contexte d'un processus :

- valeur du compteur ordinal
- adresse de la pile
- état courant du processus
- valeur des registres virtuels



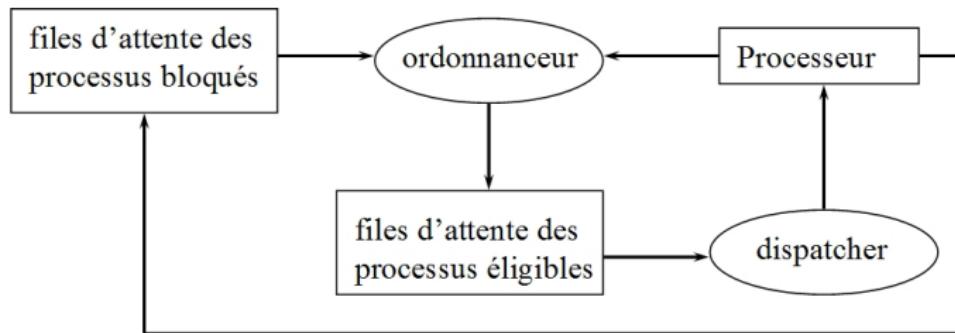
# Ordonnancement des processus

# Ordonnancement des processus

- le système d'exploitation rend l'ordinateur plus "productif"
- l'ordonnancement (*scheduling*) est une fonction fondamentale d'un système d'exploitation
- à la base de la multiprogrammation
  - plusieurs processus en mémoire en même temps
  - assignation du processeur à un processus en fonction de certains critères

# Ordonnancement des processus

- l'ordonnanceur est responsable de l'organisation des file d'attente des tâches éligibles
- le dispatcher réalise l'élection d'un processus et le changement de contexte associé



# Ordonnancement des processus

Les décisions de l'ordonnanceur peuvent avoir lieu dans les circonstances suivantes :

- ① quand un processus passe de l'état "élu" à "suspendu" (attente de disponibilité d'une ressource)
- ② quand un processus passe de l'état "élu" à l'état "éligible" (à cause d'une interruption)
- ③ quand un processus passe de l'état "suspendu" à l'état "éligible" (libération d'une ressource)
- ④ quand un processus se termine

# Ordonnancement des processus

- ordonnancement sans réquisition :
  - si le mécanisme d'ordonnancement n'a lieu que dans les cas 1 et 4
  - une fois le processeur alloué à un processus, celui-ci le garde jusqu'à ce qu'il ait terminé ou passe en attente
- ordonnancement avec réquisition :
  - si l'ordonnanceur s'occupe des cas 1 à 4
  - problèmes à résoudre :
    - le partage des données (mise en place de nouveaux mécanismes)
    - accès à des périphériques non partageables

# Ordonnancement des processus

Critères d'ordonnancement :

- être efficace : le processeur doit consacrer un maximum de temps à l'exécution des processus
- avoir un bon temps de réponse : pour réagir rapidement à un événement extérieur
- être impartial : doit assurer un partage équitable du processeur
- avoir un bon débit : minimiser le temps pendant lequel un processus est dans l'état éligible (en attente d'exécution)
- veiller à ce que le temps passé à l'ordonnancement ne devienne pas prépondérant par rapport au temps consacré aux processus

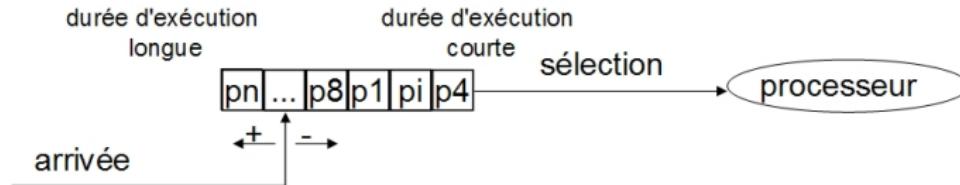
# Premier arrivé, premier servi (FCFS)

- First Come, First Serve
- c'est le plus simple
- gestion d'une file FIFO
- une fois le processeur alloué à un processus celui-ci le garde
  - ...
- peu utilisable pour le temps partagé
- désastreux si le processus en exécution ne se termine pas



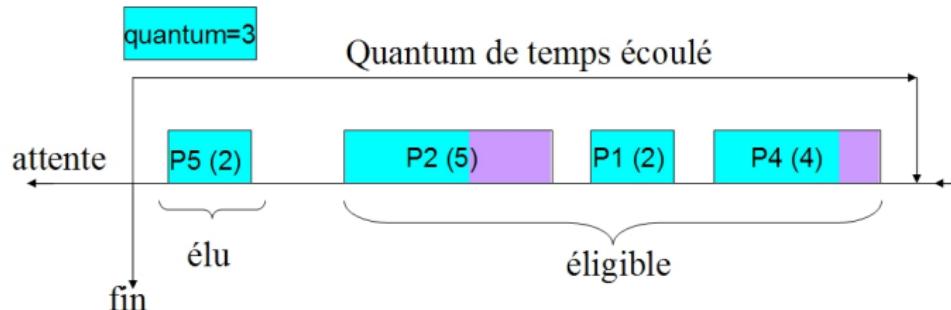
# Plus court temps de traitement

- les temps d'exécutions des tâches sont supposés connus
- technique favorisant les traitements courts
- en cas de charge, le temps de réponse des traitements longs sera catastrophique

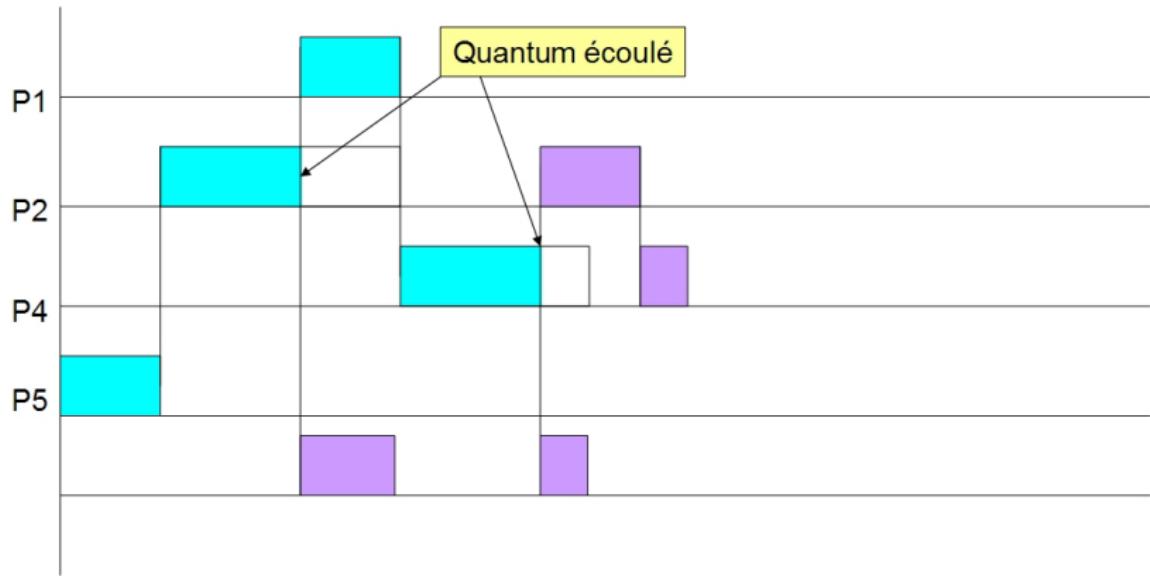


# Le tourniquet (round robin)

- spécialement conçu pour le temps partagé
- file d'attente circulaire
- un quantum de temps est alloué à chaque processus
- tout processus éligible est certain d'être élu une fois

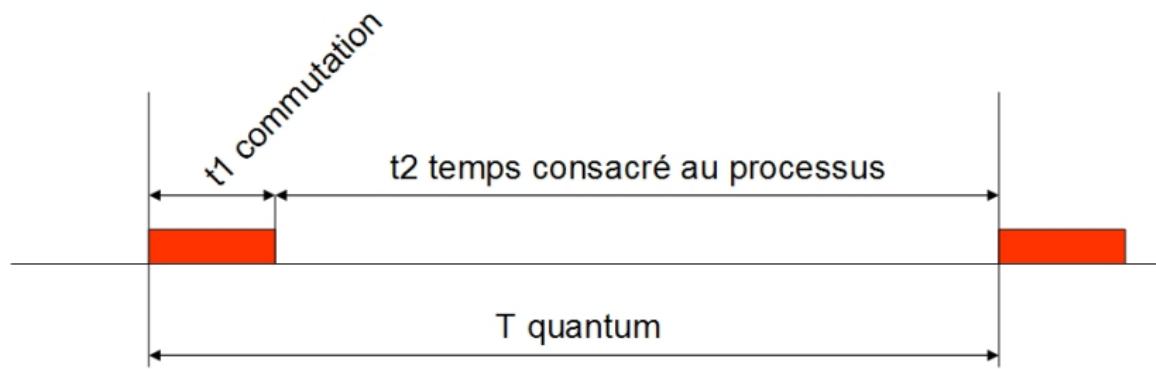


# Le tourniquet : exemple



# Le tourniquet : choix du quantum

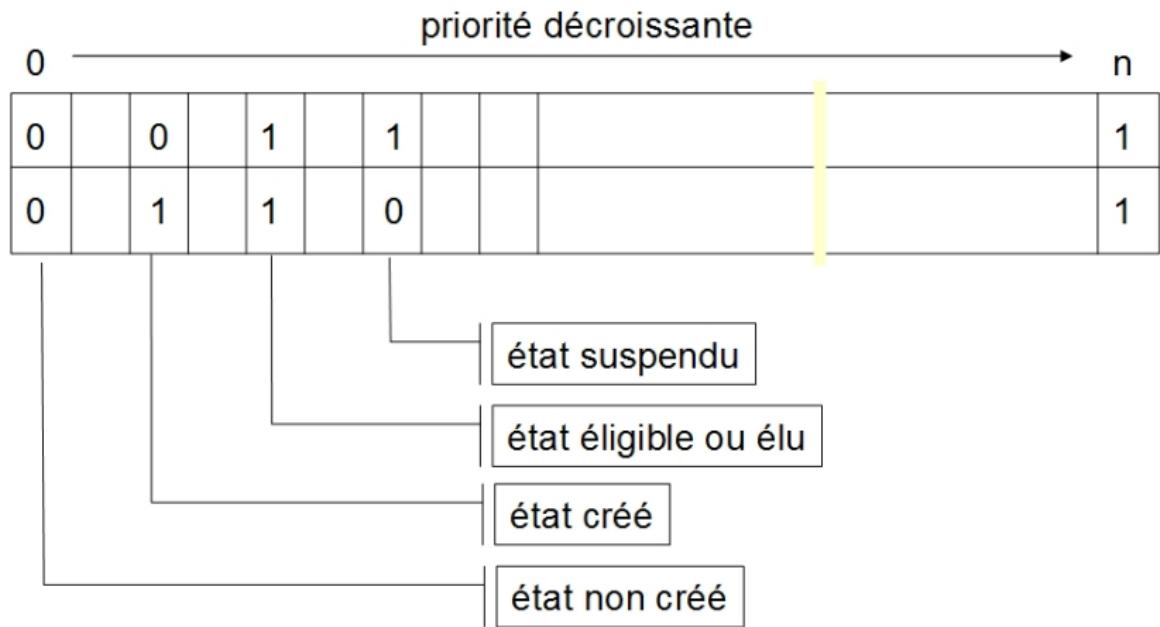
- c'est une affaire de compromis
  - s'il est trop court, le temps de commutation  $t_1$  devient prohibitif devant le temps consacré au processus  $t_2$
  - s'il est trop long, le temps d'attente des autres processus peut devenir gênant par rapport au temps de réponse à fournir



## Par priorité :

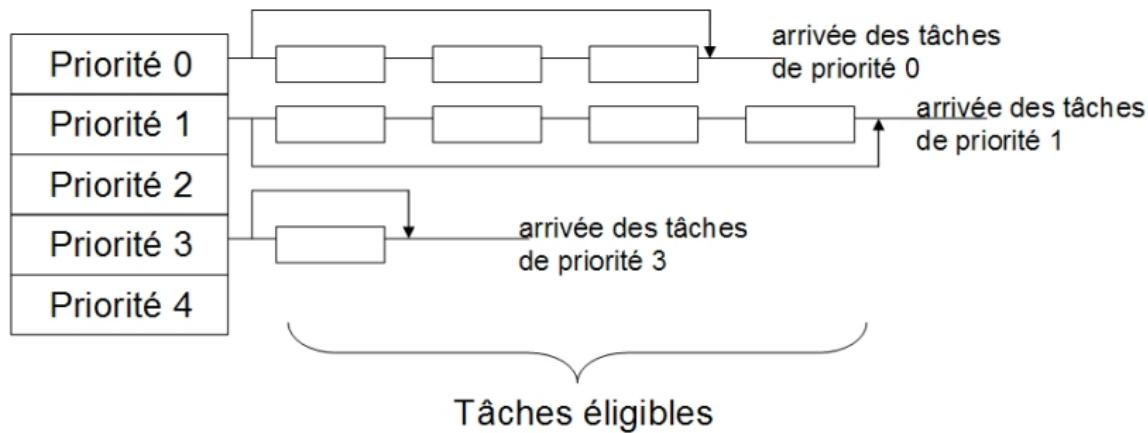
- attribution d'une priorité à chaque processus
- priorité allouée statiquement ou dynamiquement
- sans ou avec réquisition du processeur
  - sans : le processus est rangé en file d'attente en fonction de sa priorité par rapport aux processus en attente
  - avec : si le processus qui arrive dans la file d'attente est plus prioritaire que celui en cours d'exécution, il y aura réquisition du processeur en faveur du processus arrivant.

# Par priorité : exemple possible de caractérisation des états



# Par files d'attente multi-niveaux

- mixage des systèmes précédents
  - Gestion par priorités et si priorités égales, gestion par tourniquet ou FIFO



# Par priorité + temps partagé + âge

- les processus sont élus en fonction de leur priorité
- à chaque période d'horloge (*tick*), les processus en attente d'activation voient leur priorité augmenter.
- si un processus devient plus prioritaire que celui en cours d'exécution, alors il y a changement de contexte et le processus interrompu reprend sa place dans les processus en attente d'activation avec sa priorité initiale

# Par priorité + temps partagé + âge (exemple)

P1 (3)	P2(5)	P3(8) #
P1 (4)	P2(6) ←	P3(8) #
P1 (5)	P2(7)	P3(8) #
P1 (6)	P2(8) # ←	P3(8)
P1 (7)	P2(5) ←	P3(9) #
P1 (8) #	P2(6)	P3(8)
P1 (3)	P2(7)	P3(9) #
P1 (4)	P2(8) #	P3(8)
P1 (5)	P2(5)	P3(9) #
P1 (5) #	P2(5)	
P1 (3)	P2(6) #	
P1 (4)	P2(5) #	
P1 (5) #	P2(5)	

À chaque tranche de temps la priorité des tâches éligibles voient leur priorité augmenter.

A priorité égale c'est la la plus ancienne éligible qui est élue.

Une tâche suspendue retrouve sa priorité initiale

La tâche P3 s'arrête

# Communication entre processus

# Communication entre processus

- les processus concurrents d'un système d'exploitation peuvent être :
  - **indépendant** : le processus ne peut pas influer sur les autres processus (ou être influencé par les autres processus)
  - **coopérant** : le processus peut influer ou être influencé. Tout les processus qui partagent des données sont coopérants
- plusieurs raisons pour la coopération :
  - **le partage d'informations**
  - **accélération du calcul**
  - **la modularité**
  - **commodité**
- la communication peut se faire :
  - via une zone de données commune
  - par fichiers
  - par échange de messages

# Communication entre processus

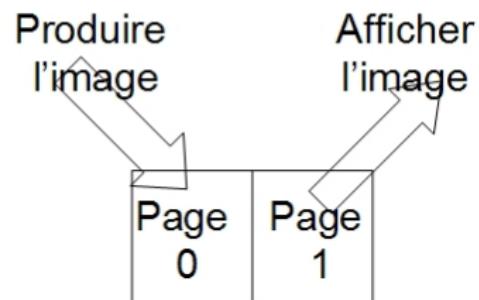
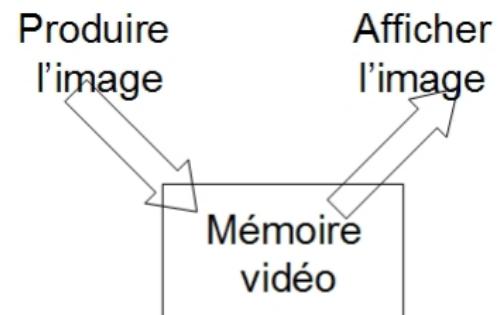
- problèmes d'accès concurrentiels
  - nécessité de fournir un environnement permettant l'accès concurrentiel : communication et synchronisation
- principe classique du **producteur** et du **consommateur**

# Zone de données commune

Processus 1 → produit l'image

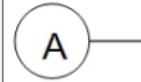
Processus 2 → affiche l'image

- problème : un seul processus peut accéder à la ressource "mémoire vidéo"
- solution : deux pages de mémoire. Pendant l'affichage de la page 1, la page 0 peut être écrite. Il faut un formalisme pour éviter l'écriture et la lecture simultanée.



# Verrou et attente active

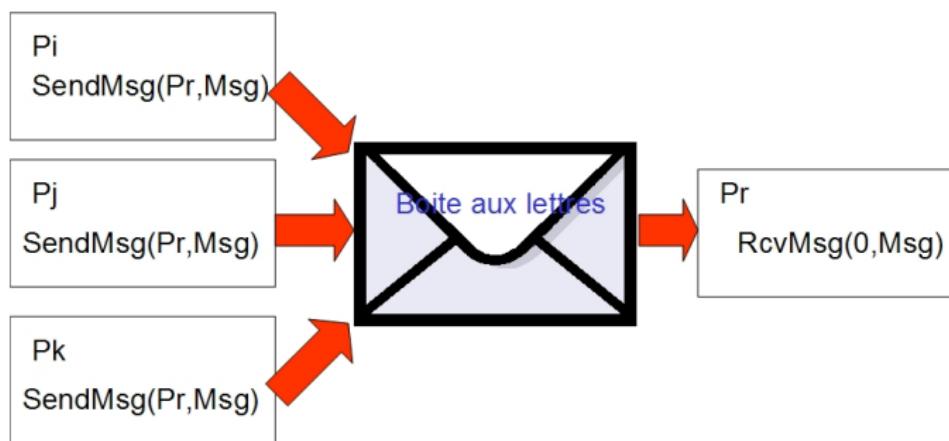
- la variable "verrou" indique (avec un 1) que la ressource est utilisée par un autre processus
- si "verrou" = 0 le processus met à 1 et utilise la ressource
- la variable "verrou" doit être commune aux processus coopérants
- problèmes : travail inutile (attente active) et si interruption au point "A"



```
verrou = 0;
while (verrou == 1); // attente
// active
verrou = 1;
// début de section
// critique
|
|
// fin de section
//critique
verrou = 0;
```

# Par messages : création de deux primitives

- `sendMsg(destinataire, message)`
- `recvMsg(expéditeur, message)`
- différentes modes de remise des messages
  - direct
  - indirect



# Synchronisation des processus

# Synchronisation des processus

## Mécanisme de synchronisation direct :

- ces mécanismes désignent la tâche concernée
- trois primitives sont souvent utilisées :
  - **bloque(tâche)** : permet de suspendre la tâche désignée
  - **dort()** : suspend la tâche y faisant appel
  - **réveille(tâche)** : réveille la tâche désignée (l'information peut être perdue si la tâche ne l'attend pas)

# Synchronisation des processus

## Mécanisme de synchronisation indirect :

- les tâches à synchroniser ne sont pas directement désignées dans l'opération
- la synchronisation s'effectue par l'intermédiaire d'objets communs aux tâches et des primitives assurant l'exclusion mutuelle de ces objets

# Synchronisation des processus

## Sémaphore d'exclusion mutuelle :

- proposé en 1965 par E.W. Dijkstra
- constitué d'une variable  $e(s)$  pouvant prendre des valeurs +, -, 0 et une file d'attente  $f(s)$ . La gestion de  $f(s)$  dépend de la politique d'ordonnancement choisie
- toute opération sur un sémaphore est une opération indivisible
- deux primitives :
  - $P(s)$  (néerlandais « proberen », prendre)
  - $V(s)$  (néerlandais « verhogen », rendre)
- doit être commun aux processus qui l'utilisent
- la valeur initiale de  $e(s)$  détermine le nombre de processus pouvant s'approprier la ressource

# Synchronisation des processus

## Sémaphore d'exclusion mutuelle :

### Wait ou P(s)

$$e(s) = e(s) - 1$$

si  $e(s) < 0$  alors

soit r le processus ayant effectué P(s)

état(r) = suspendu et le mettre dans f(s)

### Signal ou V(s)

$$e(s) = e(s) + 1$$

si  $e(s) \leq 0$  alors

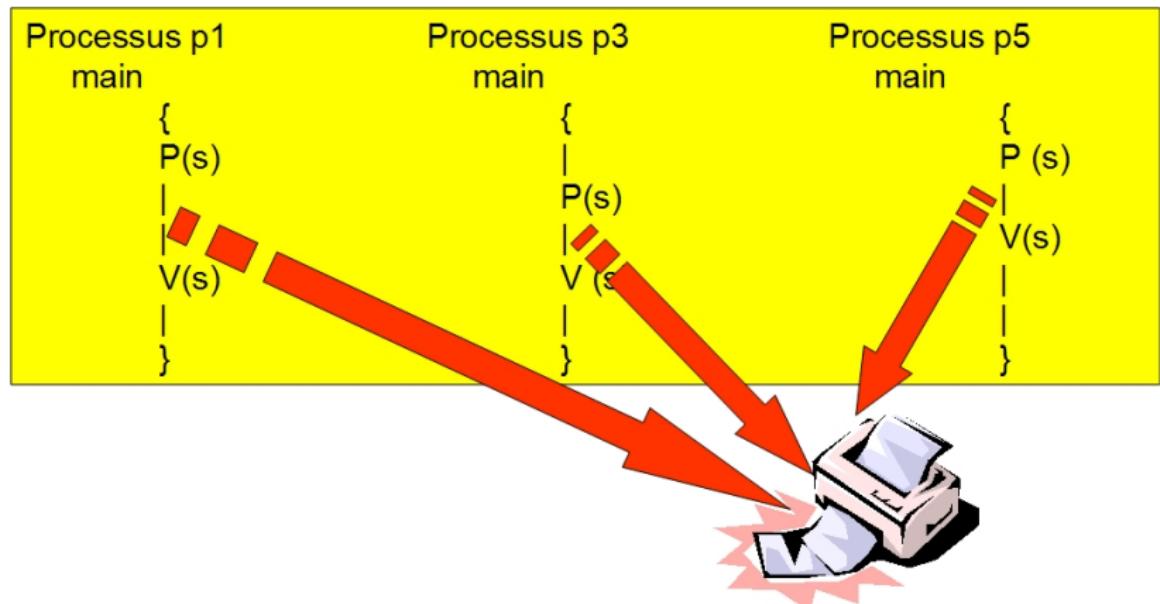
sortir un processus de f(s)

soit q, état(q) = éligible

# Synchronisation des processus

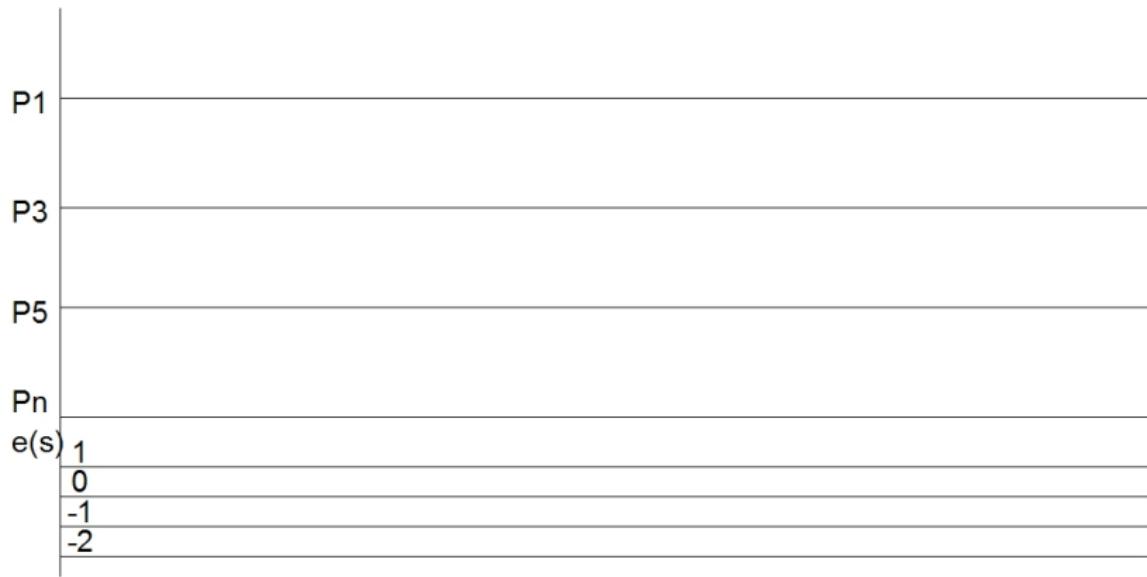
## Exemple de sémaphore d'exclusion mutuelle

- 3 tâches, de priorité 1, 3, 5, qui désirent utiliser une ressource commune (une imprimante)



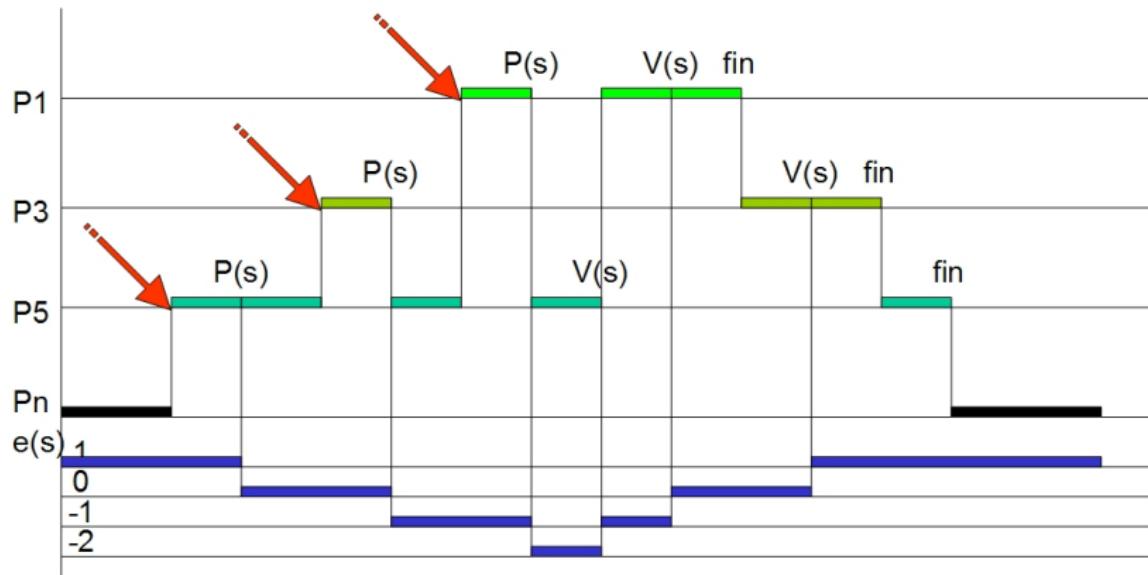
# Synchronisation des processus

## Exemple de sémaphore d'exclusion mutuelle



# Synchronisation des processus

## Exemple de sémaphore d'exclusion mutuelle



# Synchronisation des processus

## Interblocage

- Soit deux processus P0 et P1 (P0 plus prioritaire que P1). P1 s'exécute et s'approprie la ressource protégé par s2. P1 est interrompu par P0 qui exécute P(s) sur s1 puis P(s) sur s2. La ressource étant occupée c'est P1 qui reprend la main et exécute P(s) sur s1. La ressource est occupée, il est mis en attente.

P0  
begin  
    Wait (s1);  
    Wait (s2);  
    .  
    .  
    Signal (s2);  
    Signal (s1);  
end



P1  
begin  
    Wait (s2);  
    Wait (s1);  
    .  
    .  
    Signal (s1);  
    Signal (s2);  
end

# Synchronisation des processus

## Ordonnancement non prévisible

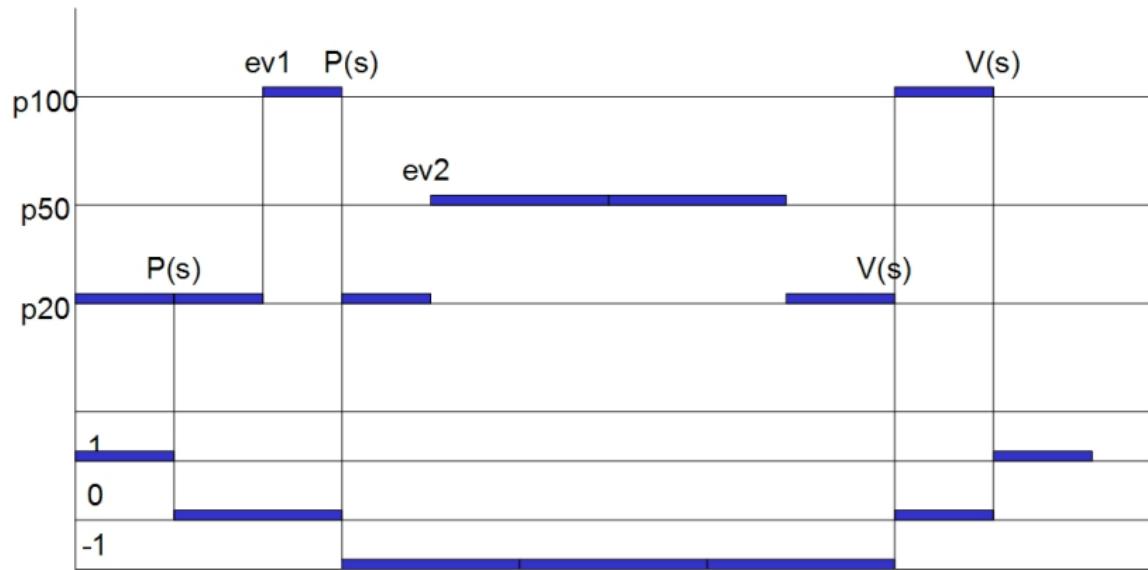
```
tâche a: prio 100    tâche b: prio 50    tâche c: prio 20
attendre(ev1);        while (1){          attendre(ev3);
P(s);                  attendre(ev2);      P(s);
calcul(1ms);          calculer(1s);     calcul(10ms);
V(s);                  }                   V(s);
reponse();
```

Priorité: a > b > c

- Au départ toutes les tâches sont en attente
- 0 ms ev3 réveille c, elle prend s
- 2 ms ev1 réveille a, a est bloquée par s
- 4 ms ev2 réveille b, c est repoussée (prio!)
- 1004 ms b est bloquée, c calcule les 6 ms
- 1010 ms c exécute V, a est réveillée
- 1011 ms a donné la réponse (trop tard)

# Synchronisation des processus

## Ordonnancement non prévisible



# Synchronisation des processus

## Exemple

- un ordonnanceur contrôle l'ensemble des tâches suivantes en fonction de leur priorité  $T_1 > T_2 > T_3 > T_4$  :
  - $T_1$  : durée d'exécution 100 ms
  - $T_2$  : durée d'exécution 200 ms
  - $T_3$  : durée d'exécution 300 ms
  - $T_4$  : durée d'exécution 400 ms
- ces tâches utilisent une ressource R commune et non partageable protégée par un sémaphore S. Les tâches sont rendues éligibles à 50 ms d'intervalle dans l'ordre  $T_4$ , puis  $T_3$ , puis  $T_2$  et enfin  $T_1$  et :
  - $T_1$  prend la ressource après 30 ms d'exécution pendant 40 ms
  - $T_2$  prend la ressource après 80 ms d'exécution pendant 100 ms
  - $T_3$  prend la ressource après 40 ms d'exécution pendant 120 ms

# Synchronisation des processus

## Sémaphore privé :

- fonctionnement identique au précédent
- pas de file d'attente
- $e(s)$  est initialisé à zéro
- seul un processus effectue  $P(s)$ , soit  $q$
- les autres ne peuvent effectuer que  $V(s)$
- ceci représente un capital d'activation pour  $q$

# Synchronisation des processus

## Sémaphore privé :

- à chaque fois d'un processus exécutera  $V(s)$ , la variable  $e(s)$  sera incrémentée de 1 et représentera un capital d'activation pour le processus qui effectue  $P(s)$ .

Processus p  
main

```
{  
|  
|  
V (s)  
|  
|  
}
```

Processus q  
main

```
{  
|  
|  
V (s)  
|  
|  
}
```

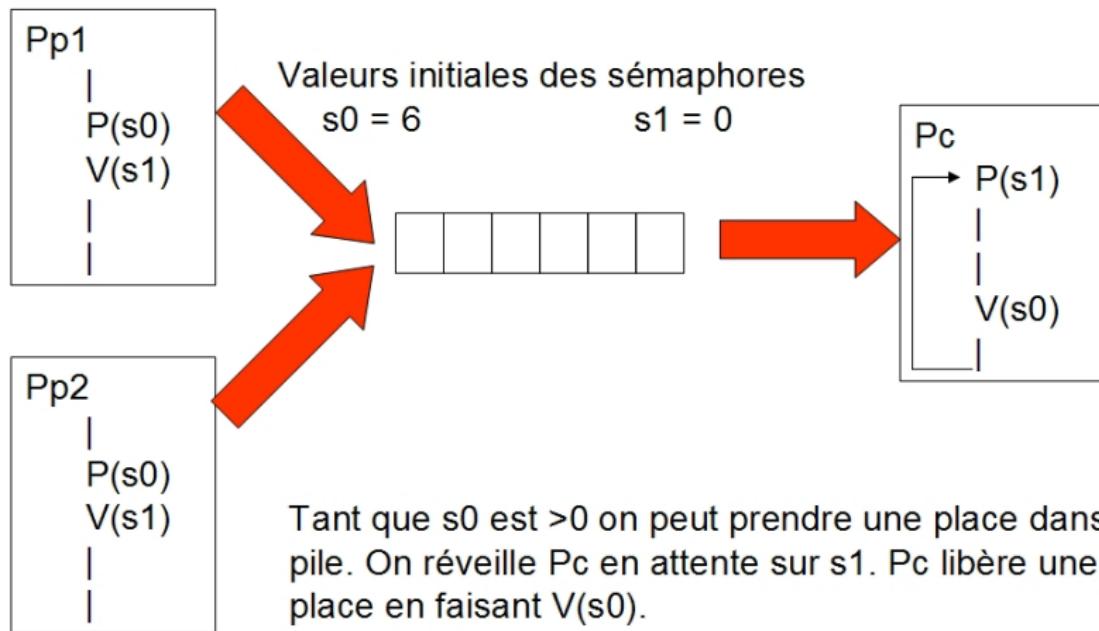
Processus r  
main

```
{ while (;;) {  
| P (s)  
|  
|  
}  
|  
|  
}
```

# Synchronisation des processus

## Sémaphore à compte :

- soit un processus consommateur et plusieurs processus producteurs



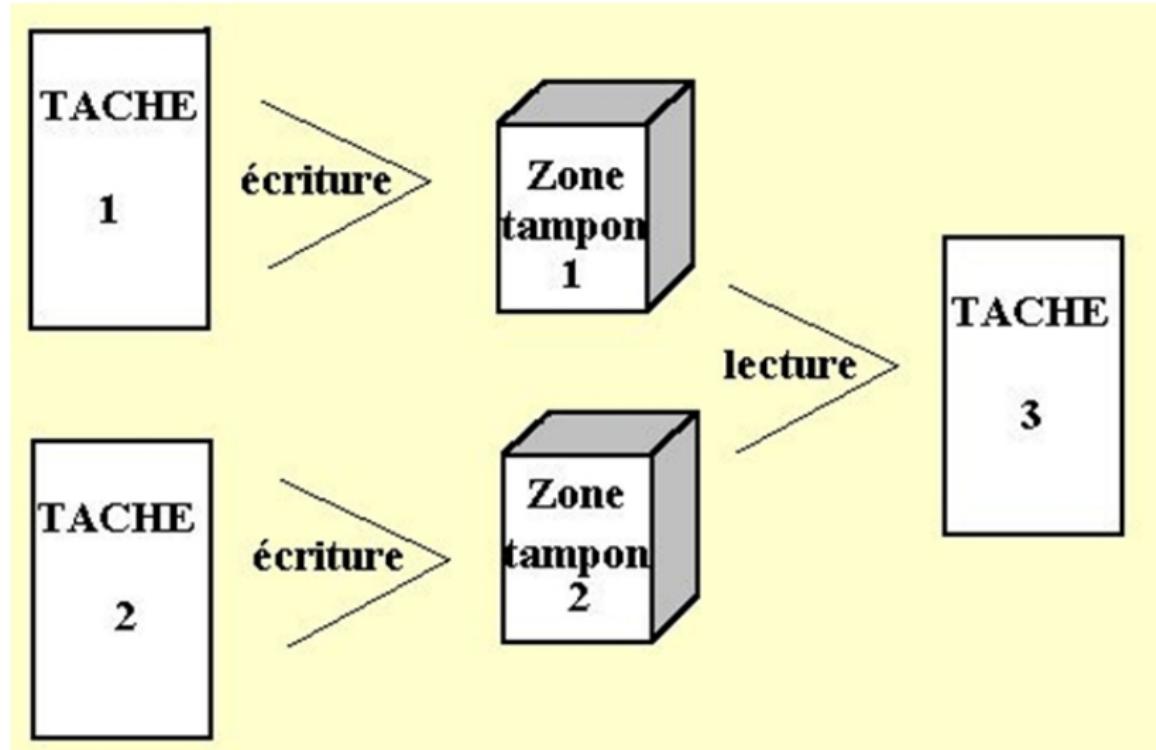
# Synchronisation des processus

## Exemple

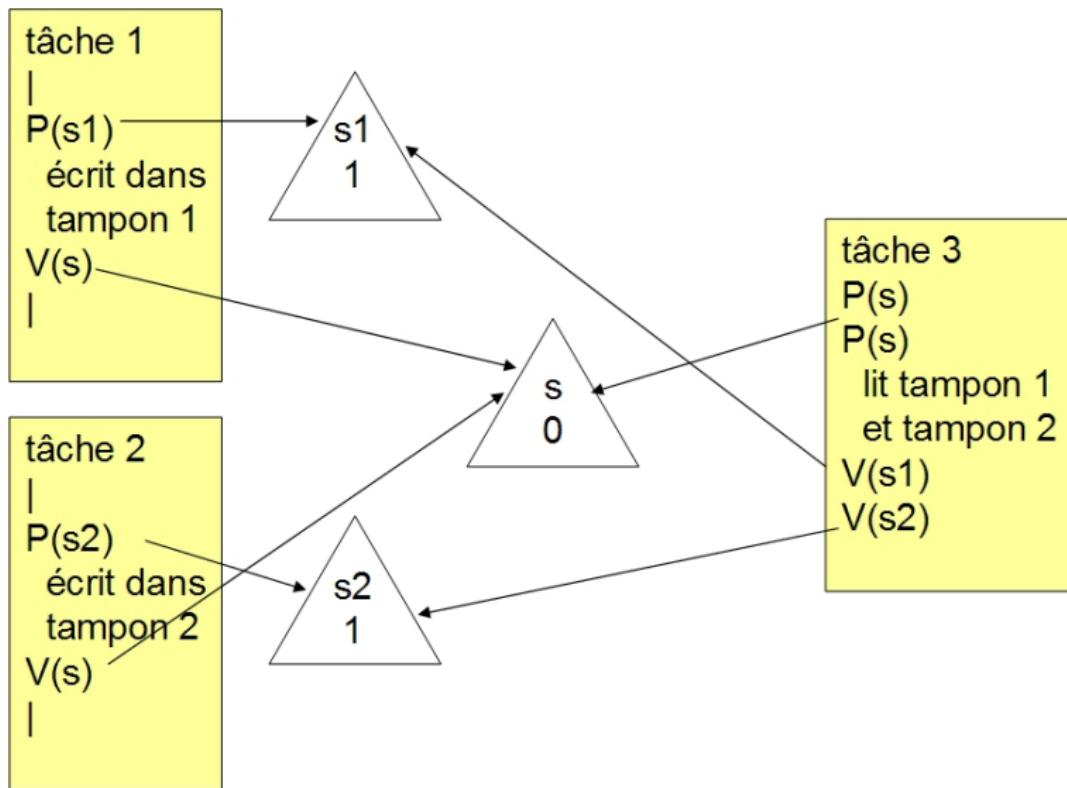
- la tâche 1 et la tâche 2 écrivent dans une zone tampon appelée zone tampon 1 et 2
- la tâche 3 lit ces zones lorsqu'elles sont toutes deux remplies
- la tâche 3 ne peut pas lire si la tâche 1 ou la tâche 2 écrit
- la tâche 1 et la tâche 2 ne peuvent pas écrire tant que leur zone tampon respective n'a pas été lue

# Synchronisation des processus

## Exemple : schéma



## Exemple : solution



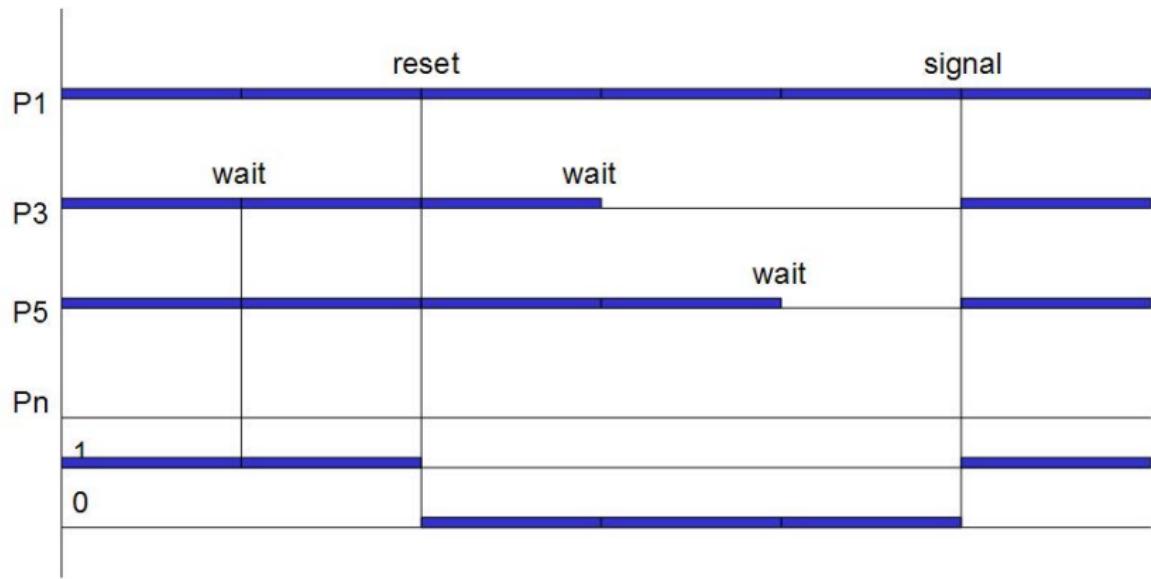
# Synchronisation des processus

## Les événements :

- notion d'événements désignant un signal logiciel asynchrone par rapport au processus récepteur
- deux primitives :
  - `wait(evt)` : attend que l'événement passe à 1
  - `signal(evt)` : signale que l'événement est arrivé
- si l'événement survient alors qu'il n'est pas attendu :
  - l'événement est mémorisé. Lors de l'exécution de la primitive `wait` remet (ou non) la bascule à zéro et la tâche se poursuit. S'il n'y a pas de remise à zéro, il y a nécessité d'avoir une primitive de type `reset(evt)`
  - l'événement n'est pas mémorisé. Le signal est alors perdu et la tâche se bloque lors de l'exécution de la primitive `wait`

# Synchronisation des processus

## Exemple de synchronisation par événements



# Interruption

- une interruption est un signal provenant de l'extérieur du programme et arrivant de manière asynchrone au déroulement du programme en cours
- le programme en cours d'exécution est suspendu dans son activité
- il y a exécution du programme prédéfini associé au traitement de cette interruption
- les causes d'interruptions sont multiples et il faut être capable de les traiter

# Interruption

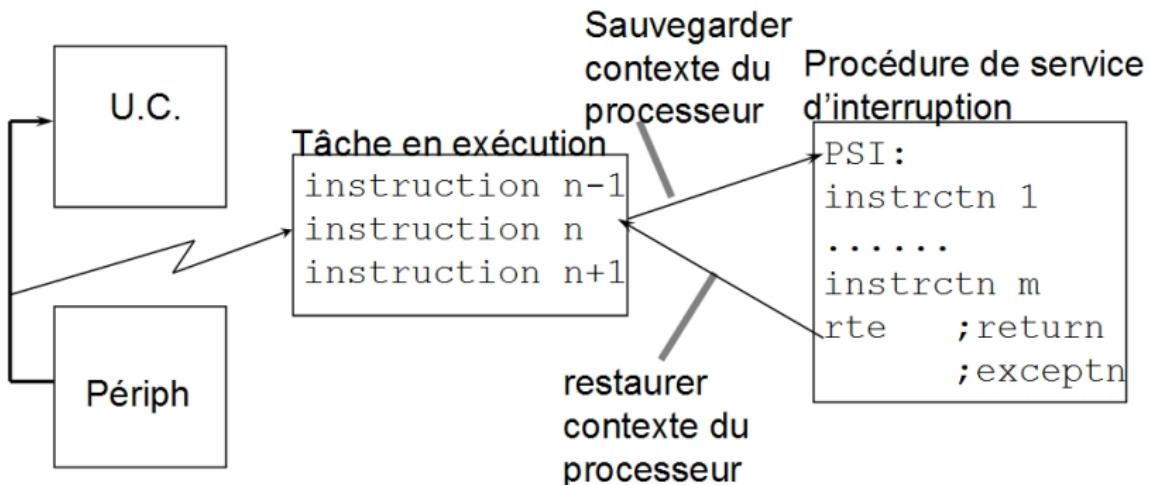
Diverses méthodes possibles :

- un indicateur unique utilisé pour toutes les interruptions. Il doit exister un indicateur supplémentaire, stocké en mémoire permettant de distinguer les causes pour appeler la procédure appropriée.
- un indicateur distinct attaché à chaque cause. Chaque cause est attachée à un niveau d'interruption pour lequel un programme de traitement est associé.
- les deux systèmes précédents peuvent cohabiter. On distinguera alors des sous-niveaux d'interruptions.

# Interruption

Principe :

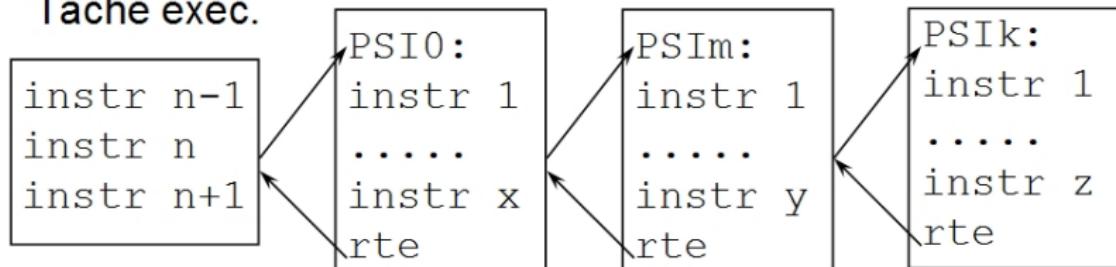
- une interruption est un signal électrique (fourni par la périphérie), qui arrête le processus en cours pour forcer l'exécution d'un autre processus.



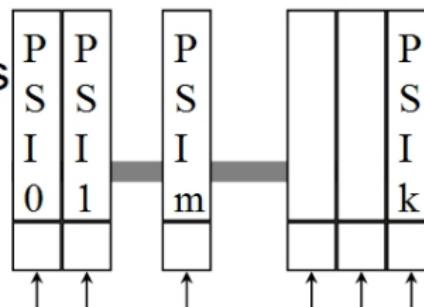
# Interruption

Gestion de plusieurs interruptions :

Tâche exéc.



Adresses



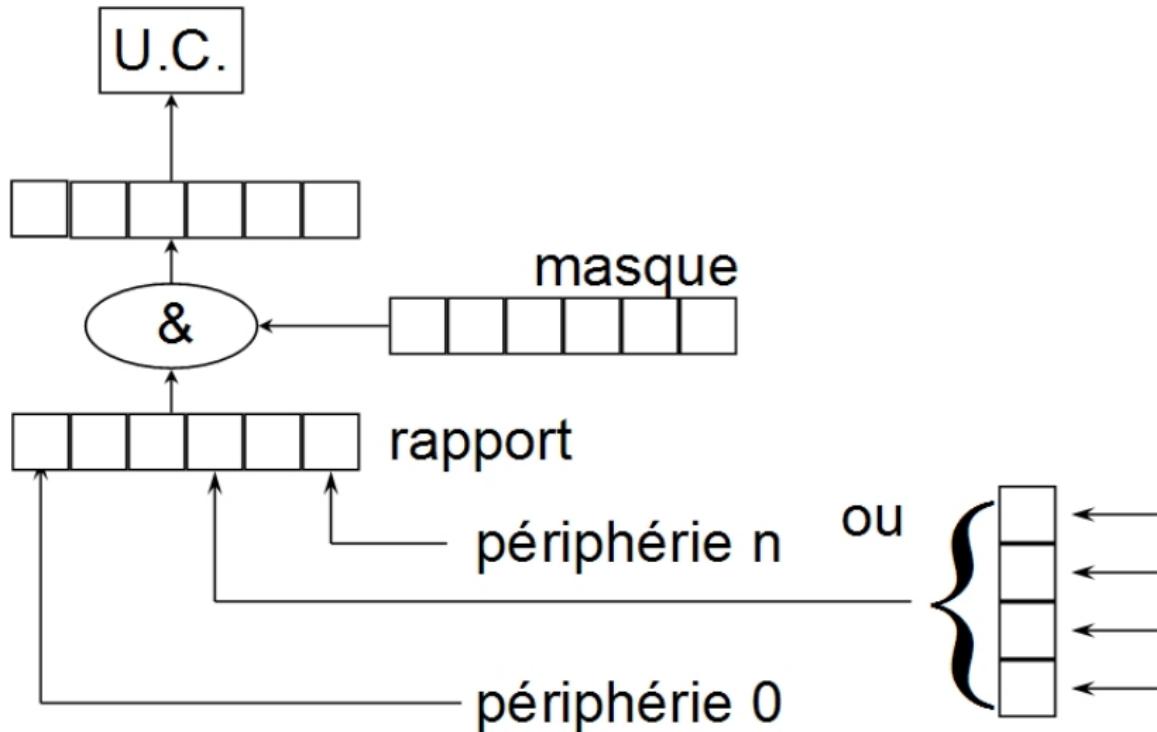
Priorité

Interruptions

I<sub>0</sub> I<sub>1</sub>..... I<sub>k</sub>

# Interruption

Masquage :



# Interruption

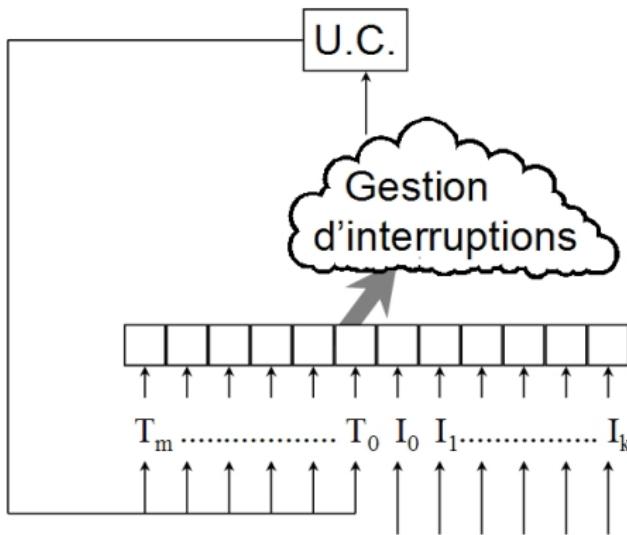
Différents types d'interruption :

Mécanismes	Cause	Utilisation
<b>Interruption</b>	extérieure au déroulement de l'instruction en cours	réaction à un événement extérieur asynchrone
<b>Déroutement (trap)</b>	liée au déroulement de l'instruction en cours	traitement d'une erreur ou situation exceptionnelle
<b>Appel au superviseur</b>	liée au déroulement de l'instruction en cours	utilisation d'une fonction du noyau

# Interruption

Trap (déroutement) :

- un trap (piège) est une interruption forcée par l'UC
- principe :



# Interruption

## Trap : application

- traitement d'erreurs
  - erreur de lecture/écriture en mémoire
  - division par zéro
  - instruction non implémentée
  - tentative d'exécution d'une opération interdite par un dispositif de protection mémoire
- appel au système d'exploitation
  - fin de tâche
  - fin de procédure de service d'interruption
  - communication intertâche