

Programmation système

Ressource R3.05 - Interruption - ordonnancement

monnerat@u-pec.fr 

IUT de Fontainebleau

1. Interruption

2. Ordonnancement

- Généralités
- Algorithmes non préemptifs
- Algorithmes préemptifs
- Linux

Interruption

Interruption d'un processus

Une **interruption** permet à la CPU d'être prévenue lorsqu'un événement nécessite son attention, sans **attente active**.

Interruption

Opération atomique effectué par un cpu qui change le CO et le mode.

Chaque interruption a une cause identifiée par un numéro. Le vecteur d'interruption est une table qui stocke les adresses de la routine associée.

De manière schématique, la prise en compte d'un interruption k :

traitement

```
push co,mode  
mode := maitre  
co := vi[k]
```

reprise

```
pop mode  
pop co
```

- Il y a un peu plus de contexte d'interruption qui est empilé :
RIP, CS, RFLAGS, RSP, SS ...
- ... sur la pile noyau de la tâche, que la cpu peut retrouver grâce à un registre particulier (TR).

code utilisateur

123	LOAD R1,
124	#10
125	INC R1
126	ADD R1,R2

code système

100	
101	
102	
103	RTI

vecteur d'interruptions

0	
1	
2	100
3	

co	mode	sp	r1	r2
123	esclave	200	-	34

load R1,#10

co	mode	sp	r1	r2
125	esclave	200	10	34

interruption n°2

co	mode	sp	r1	r2
100	maître	201	10	34

⋮

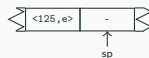
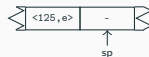
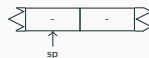
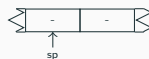
co	mode	sp	r1	r2
103	maître	201	10	34

RTI

co	mode	sp	r1	r2
125	esclave	200	10	34

INC R1

co	mode	sp	r1	r2
126	esclave	200	11	34



Handler d'interruption

C'est la routine qui traite l'interruption.

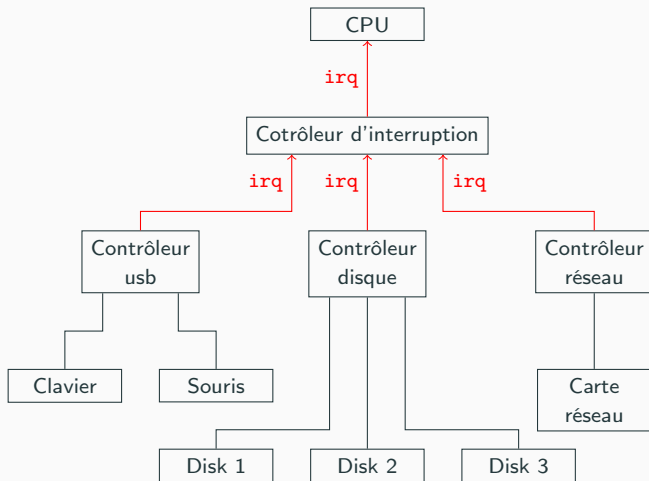
Sa structure est la suivante :

1. sauvegarde du contexte ;
2. traitement de la cause ;
3. restauration du contexte ;
4. retour au processus interrompu (instruction IRETQ).

3 types

- Interruption matérielle : réaction aux événement extérieurs ;
- Appel au superviseur : appel explicite d'une routine système ;
- Déroutement : traitement des erreurs et des situations anormales.

Interruption matérielle



- De manière simplifiée, un signal électrique est émis vers le processeur qui provoque une interruption ;
- Il y a branchement vers un handler avec changement de mode.

Appels système

Intrusion qui provoque une entrée dans le noyau. Le processeur passe alors du mode utilisateur (ring 3) au mode noyau (ring 0).

Sous linux : interruption logicielle `int $0x80`, `syscall` plus moderne.

Avantages

- **Sécurité** : le programme utilisateur ne peut pas accéder directement aux ressources sensibles (mémoire, périphériques, etc.).
- **Protection du système** : le noyau contrôle les opérations demandées et vérifie les droits.
- **Interface uniforme** : les programmes utilisent une interface standard (`read`, `write`, `open`, `fork`...) indépendamment du matériel.
- **Simplicité pour le programmeur** : pas besoin de connaître le fonctionnement détaillé du périphérique.
- **Isolement** : une application ne peut pas directement perturber les autres applications ou le noyau.

L'objectif des déroutements est double :

- traitement synchrone systématique des erreurs ou des situations anormales (défaut de page),
- protection et bonne utilisation de la machine (les processus sont surveillés).

Si l'exécution d'une instruction produit une erreur alors, il y a interruption et branchement vers la routine du système qui traite les erreurs. Les causes possibles sont :

Erreurs

- données incorrectes (division par zéro),
- opération interdite (instruction privilégiée),
- instruction inconnue,
- accès à une zone mémoire interdite,
- erreur de bus mémoire.

Division par zéro

Soit le code C

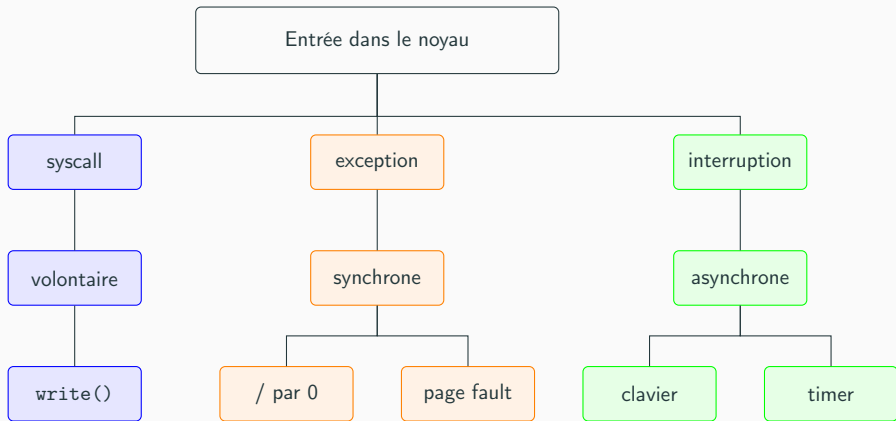
```
int a = 10;  
int b = 0;  
int c = a/b;
```

et l'assembleur correspondant

```
movl $10, -12(%rbp)  
movl $0, -8(%rbp)  
movl -12(%rbp), %eax  
cld  
idivl -8(%rbp)
```

Le cpu ne va pas exécuter la division.

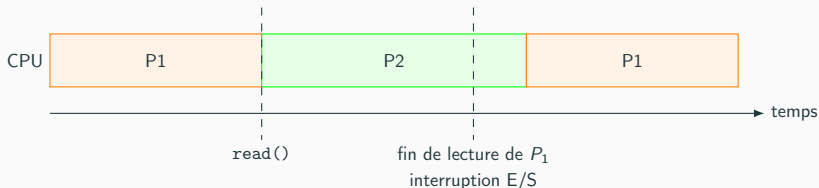
- Intel définit l'exception #DE, associé au vecteur d'exception 0.
- Le noyau dispose d'un gestionnaire pour cette exception.
- En général, linux transforme cette exception en un signal SIGFPE envoyé au processus fautif.



Interruptions pour les E/S

Il n'y a pas d'attente active.

- le processus utilisateur P_1 réalise un appel système
- P_1 est mis en attente sur l'évènement de fin de lecture
- la fin de la lecture déclenche une interruption d'entrée/sortie
- le traitement de l'interruption place P_1 parmi les processus "prêt"
- P_1 reprendra ses traitements quand le processeur lui sera alloué



Remarque : toutes les appels systèmes à `read` ne se termine pas par une interruption !

Ordonnancement

Ordonnancement

Généralités

- Faculté du système à exécuter plusieurs processus "à la fois".
- Pseudo parallélisme et entrelacement

CPU time sharing

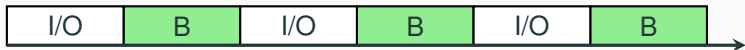
CPU



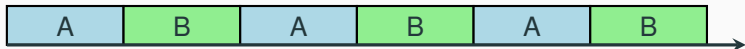
Remarque : l'entrelacement doit être "assez fin" pour ne pas se "voir".

Multitâche

- Empiriquement, un processus alterne des phases de calculs (**CPU burst**) et des phases d'E/S (**I/O burst**).



CPU



- Evidemment, on peut avoir des processus déséquilibrés (plus de calculs ou d'E/S).

E/S

- latence d'accès au matériel : disque, réseau...
- lenteur de l'utilisateur d'un programme interactif
- synchronisation avec d'autres programmes

Mauvaise solution : attente active (polling)

- difficile pour le programmeur d'application
- temps processeur gâché à attendre

Bonne solution : attente passive

- plus facile pour le développeur
- meilleur taux d'utilisation du CPU
- masquage (= recouvrement) des latences

besoin d'un mécanisme pour se partager le(s) processeur(s)

Commutation de Contexte

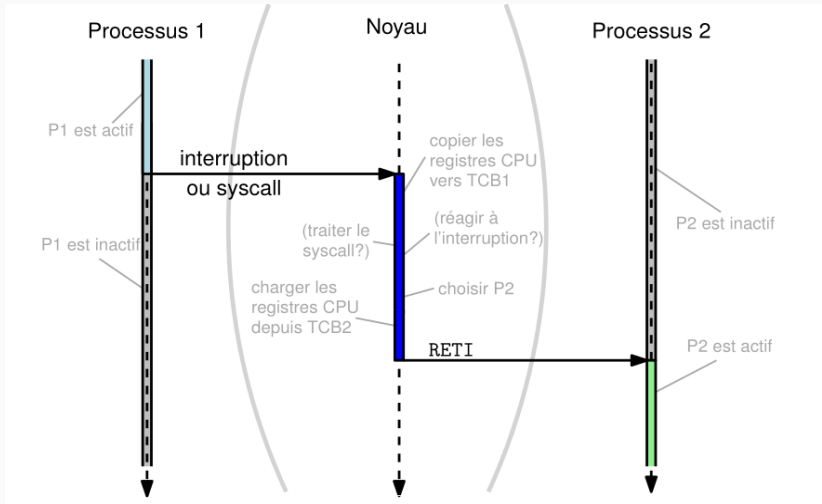


Figure 1: Commutation de contexte

Le remplacement du processus en exécution a un coût !

Commutation de contexte

- Exécution de la routine d'ordonnancement
- Sauvegarde du contexte (registres + PC)
- Chargement d'un nouveau contexte

Structures du noyau

- **ProcessCB** : identité, mémoire, fichiers ouverts, statistiques, etc.
- **ThreadCB** : fil d'exécution lié à un processus : registres, PC, SR, pile, etc

Les objectifs d'un ordonnanceur d'un système multi-utilisateur sont, entre autres :

- S'assurer que chaque processus en attente d'exécution reçoive sa part de temps processeur.
- Minimiser le temps de réponse.
- Utiliser le processeur à 100%.
- Utilisation équilibrée des ressources.
- Prendre en compte des priorités.
- Être prédictible.

Les objectifs ne sont pas les mêmes selon qu le système est interactif, temps réel, traitement par lots, etc.

- **CPU utilization rate** : fraction du temps où le CPU est productif i.e. occupé à exécuter du code applicatif (vs noyau, ou CPU idle)
- **Throughput = débit** : nombre de tâches terminées par unité de temps.
attention : n'a de sens que si les «tâches» peuvent «terminer»
- **Non-Starvation** : absence garantie de risque de famine
- **Turnaround time (tps de séjour)** : durée entre arrivée et terminaison
n'a de sens que si les «tâches» peuvent «terminer»
- **Waiting time (temps d'attente)** : durée passée dans la ready queue
- **Response time** : durée entre arrivée et «réponse»

Formulation

- étant donnés K processus prêts,
- étant donnés N processeurs disponibles,

décider quel processus exécuter sur chaque processeur.

Remarque : quand ordonnancer ?

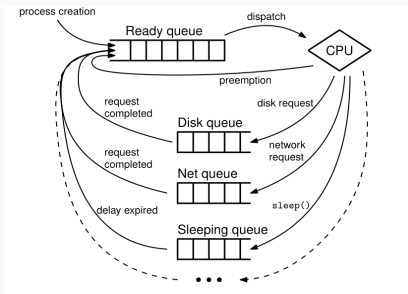
- le processus qui s'exécute se termine ou se bloque.
- un processus passe dans l'état prêt.
- le processus en exécution a eu le processeur "trop longtemps".

Principes généraux

Préemptif/Coopératif

- Préemptif : le noyau peut remplacer un processus en exécution par un autre processus prêt.
 - interruption de timer régulière pour assurer la préemption.
 - le noyau garde le "contrôle de la machine".
 - coûteux.
- Coopératif : les processus rendent explicitement la main au noyau.

File d'attentes/priorités



- Les PCB des processus prêts forment la Ready Queue
- Rôle du scheduler : choisir un PCB parmi la Ready Queue
- Processus bloqués : transférés dans une autre file d'attente
 - une Device Queue par périphérique d'entrées-sorties
 - une file d'attente pour les processus endormis
 - ... une file pour chaque autre raison d'être Blocked
- Priorité
 - Type de processus (système ou utilisateur, E/S ou calcul)
 - Priorité fixée par l'utilisateur
 - Variable au cours de l'exécution (règles de l'OS)

Ordonnancement

Algorithmes non préemptifs

FCFS

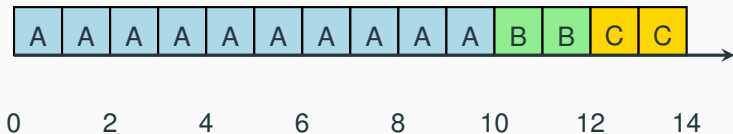
First Come, First Served

Principe

Premier arrivé, premier servi

Proc	Date	Durée
A	0	10
B	1	2
C	1	2

CPU



SJF

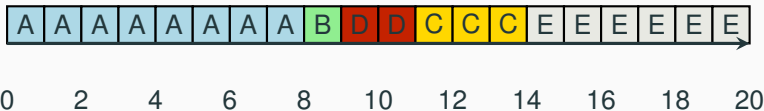
Shortest Job First

Principe

On exécute le plus court d'abord (version non préemptive)

Proc	Date	Durée
A	0	8
B	1	1
C	2	3
D	3	2
E	4	6

CPU



Ordonnancement

Algorithmes préemptifs

SRTF(Q)

Shortest Remaining Time First

Principe

C'est la version préemptive de SJF. Tous Q quantum de temps, on élit la tâche la plus courte.

- On choisit le processus dans la file d'attente selon le plus court SRT.
- Le processus est exécuté au plus pendant un quantum de temps.
- Si, après ce quantum, il n'est pas terminé, il est ajouté à la file d'attente, et son nouveaux SRT' est calculé selon la formule : $SRT' = SRT - \text{quantum}$

Paramètre : le quantum du temps.

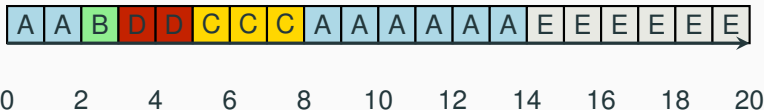
Analogue du SJF, algorithme théorique car les temps d'exécution n'est pas en général connu.

Exemple

Avec $Q=2$

Proc	Date	Durée
A	0	8
B	1	1
C	2	3
D	3	2
E	4	6

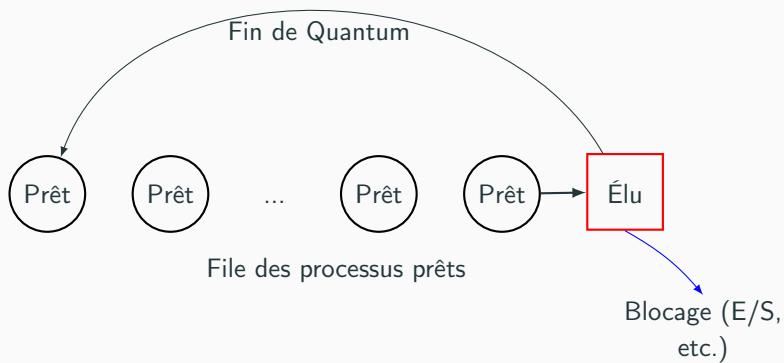
CPU



- **Quantum** = durée maximale d'activité continue.
- **Préemption** du processeur au profit d'un autre processeurs :
 - soit en fin de quantum ;
 - soit sur blocage du processus actif.
- Adaptation possible de la durée du quantum au profil d'exécution (calcul ou E/S).
- À l'expiration du quantum, mise en fin de file des prêts.

Choix du Quantum ?

- **bref** : overhead
- **long** : temps de réponse important



Exemple

Avec $Q=2$

Proc	Date	Durée	Déb	Fin	Tps Sej	Att	Pénal
A	0	8					
B	1	1					
C	2	3					
D	3	2					
E	4	6					

Exemple

Avec $Q=2$

Proc	Date	Durée	Déb	Fin	Tps Sej	Att	Pénal
A	0	8	0	18	18	10	2.25
B	1	1	2	3	2	1	2.00
C	2	3	3	12	10	7	3.33
D	3	2	7	9	6	4	3.00
E	4	6	9	20	16	10	2.67

CPU



Ordonnancement

Linux

Classes d'ordonnement

```
$ chrt -m  
propriété min/max SCHED_OTHER   : 0/0  
propriété min/max SCHED_FIFO    : 1/99  
propriété min/max SCHED_RR      : 1/99  
propriété min/max SCHED_BATCH    : 0/0  
propriété min/max SCHED_IDLE     : 0/0  
propriété min/max SCHED_DEADLINE : 0/0
```

- Stratégies temps réel
 - SCHED_FIFO (FIFO real-time)
 - SCHED_RR (Round-Robin real-time)
- Stratégies normales
 - SCHED_OTHER
 - SCHED_BATCH
 - SCHED_IDLE

Stratégies temps réels

Les processus en temps réel sont ordonnancés en premier et les processus normaux sont ordonnancés une fois que tous les threads en temps réel ont été ordonnancés.

Les stratégies en temps réel sont utilisées pour des tâches pour lesquelles le temps est critique, des tâches devant être effectuées sans interruption.

- `SCHED_FIFO` : le processus avec la plus grande priorité (1-99) est exécuté jusqu'à ce qu'il se termine, se bloque, ou soit preempté par un processus de plus haute priorité.
- `SCHED_RR` : version round-robin du précédent. Chaque priorité (1-99) est organisé en file FIFO.
- Les processus "temps réels" sont prioritaires sur tous les autres.
- Un mécanisme permet de limiter la monopolisation du cpu par le temps réel : `/proc/sys/kernel/sched_rt_runtime_us` et `/proc/sys/kernel/sched_rt_period_us`

Principalement SCHED_OTHER "Historiquement" (noyau 2.6) :

- chaque processus a un timeslice de temps d'exécution
- deux listes de processus actifs (timeslice > 0) / expirés (timeslice = 0) par priorité (1-140).
- lorsque qu'un processus expire son timeslice, celui-ci est recalculé, et la processus intègre la file des expirés.
- quand le file des actifs est vide, on switch les deux files.
- L'ordonnancement est basé sur une **priorité dynamique** : calcul qui fait intervenir la priorité statique (nice entre -20 et 10, 0 par défaut) et une pénalité/bonus si le processus est considéré comme interactif (dort souvent) ou non.
- Calcul du timeslice : initialement, la moitié de son père, sinon fonction de la priorité dynamique.

Après le noyau 2.6 : Completely Fair Scheduler (CFS). Idées :

- Quand un processus est exécuté, il augmente son `vruntime` : temps d'exécution pondéré par sa priorité.
- L'ordonnanceur utilise un arbre binaire de recherche coloré (red-black tree) pour trier les processus suivant leur `vruntime`

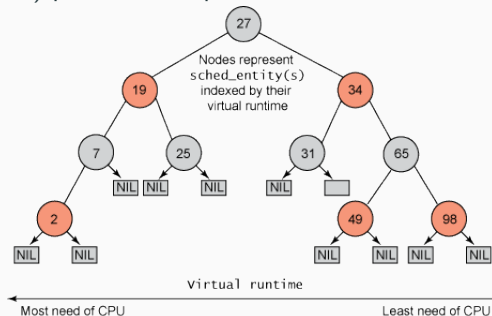


Figure 2: CFS (scheduler)

- On choisit toujours le noeud le plus à gauche (le plus petit `vruntime`)

Le temps d'exécution est pondéré par la priorité du processus (nice)

Une priorité faible accélère le temps, une priorité forte le ralentit.

- I/O bound processus n'ont pas besoin de beaucoup de temps processeur. Leur `vruntime` sera faible $\Rightarrow$ on leur attribue une priorité importante.
- CPU bound processus au contraire ont besoin de beaucoup de temps processeurs $\Rightarrow$ on leur attribue une priorité faible.

```
int nice(int inc);
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_getscheduler(pid_t pid);
int sched_setscheduler(pid_t pid, int policy,
    const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_yield(void);
int sched_rr_get_interval(pid_t pid, struct timespec *interval);
int sched_getcpu(void);
int sched_setaffinity(pid_t pid, size_t cpusetsize,
    const cpu_set_t *mask);
int sched_getaffinity(pid_t pid, size_t cpusetsize,
    cpu_set_t *mask);
```

`chrt` - manipulate the real-time attributes of a process

`nice` - run a program with modified scheduling priority

`taskset` - set or retrieve a process's CPU affinity