

# Advanced Systems

Ordonnancement/ Ordonnancement temps-réel 1/2

Laure Gonnord

University of Lyon/ LIP

MIF18 - Avril 2020

# Plan

## Introduction

## Ordonnancement dans les systèmes classiques

Algorithmes principaux

Implémentation Posix

# Les systèmes généralistes à temps partagé 1/2

Objectifs de tels systèmes :

- Faciliter l'accès aux ressources.
- Masquer les ressources (process, mémoire, disques).
- Recherche de l'équité, maximisation du débit global.

# Les systèmes généralistes à temps partagé 2/2

Deux styles de programmation :

- Mono-programmation : interaction synchrone entre le couple processeur/mémoire et les périphériques.
- Multi-programmation : interaction asynchrone. Le processeur profite du temps ainsi libéré pour effectuer d'autres traitements.

# Plan

Introduction

Ordonnancement dans les systèmes classiques

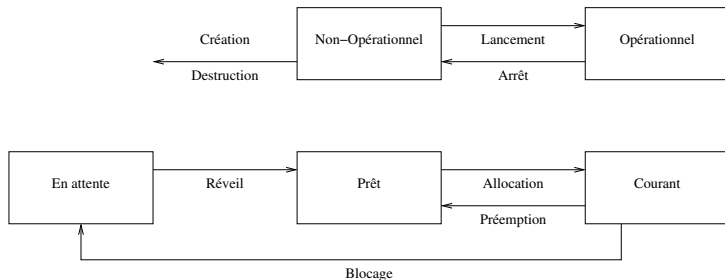
Algorithmes principaux

Implémentation Posix

# Généralités

Il n'existe pas d'ordonnanceur optimum pour tous les cas d'utilisation d'un système d'exploitation.

Schéma général des tâches système :



# Plan

Introduction

Ordonnancement dans les systèmes classiques

Algorithmes principaux

Implémentation Posix

# Un ordonnanceur, pourquoi ?

C'est l'ordonnanceur qui gère l'allocation et la suspension des tâches. L'ordonnanceur peut être :

- “hors-ligne” ou “en ligne”.
- Préemptif ou non-préemptif.

Ordonnanceurs classiques :

- PAPS (FIFO) Premier arrivé, premier servi.
- Tourniquet (Round Robin) à tour de rôle avec un *quantum* de temps.
- Priorité : plus prioritaire d'abord.



# Les priorités Unix

Priorités  $\neq$  respect des échéances :

- Les priorités UNIX, laissent la main aux tâches moins prioritaires afin qu'elles avancent un peu.
- Si les tâches faiblement prioritaires n'avançaient jamais cela pourrait introduire des *dead-lock*.

► Mais les priorités ne permettent pas forcément le respect des échéances.

# Algo d'ordo PAPS (FIFO)

Principe : premier arrivé premier servi.

Algo PAPS, durée : A=4, B=6, C=2														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	A	A	A	B	B	B	B	B	B	C	C		D
Démarrage	A	B	C											D

# Algo d'ordo Tourniquet (Round Robin)

Principe : donner à manger à tout le monde "en tournant".

Algo tourniquet, durée : A=4, B=6, C=2														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	B	C	A	B	C	A	B	A	B	B	B		D
Démarrage	A	B	C											D

# Algo d'ordo PAPS + prio

Principe : idem PAPS, mais en mettant des priorités (choix du processus à ordonnancer si deux sont ordonnançables).

PAPS + Priorités : A=+, B=++, C=+++														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	B	B	B	B	B	B	C	C	A	A	A		D
Démarrage	A	B	C											D

# Algo d'ordo Tourniquet + prio

Principe : idem tourniquet + priorités.

Tourniquet + Priorités : A=+, B=++, C=++														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	B	C	B	C	B	B	B	B	A	A	A		D
Démarrage	A	B	C											D

# Inconvénients des ordonnanceurs classiques

- Le processeur est toujours utilisé et ne peut se mettre en veille. Le passage à l'état de veille du processeur est souvent long.
- Cela fait perdre du temps CPU. (très gênant quand on fait tourner des machines virtuelles.)
- Le *quantum* de temps doit plutôt être choisi en fonction de la charge de la machine.

**Solution :** au lieu d'être lancé périodiquement, l'ordonnanceur indique dans combien de temps il doit être réveillé.  
Évidemment, il peut être réveillé par une tâche prioritaire qui devient prête.

# Plan

Introduction

Ordonnancement dans les systèmes classiques

Algorithmes principaux

Implémentation Posix

# Et si on veut plus de garanties ?

Les systèmes généralistes ont quelques soucis :

- Ils ne sont pas déterministes : matériel / logiciel.
- L'ordonnanceur temps partagé n'offre aucune garantie.
- (linux) le noyau (les processus noyaux) est non préemptif (donc si un autre processus plus prioritaire a besoin du processeur ça foire.)

► POSIX 1003.1b sous Linux



# Ordonnanceur

Principe :

- On ajoute une classe de processus “temps réel”.
  - Ceux-ci interrompent les autres tâches (sans leur laisser le temps de finir leur quantum) quand ils sont prêts.
- Raccourcir le temps de démarrage des processus “temps-réel”.

# Préemptivité

Rappel : en Unix classique, un appel système (noyau) n'est pas préemptible.

Solutions :

- insérer le “bon nombre” de points de préemptions.
- OU rendre le noyau complètement préemptif (en protégeant toutes les structures de données par des sémaphores).

**Remarque** Ce problème est le même en multi-processeurs quand on veut que plusieurs processeurs exécutent le code du noyau.

# La norme POSIX

**Objectif** Définir une **interface standard des services offerts** par Unix (portabilité). Publié conjointement ISO/ANSI. Remarques :

- portabilité difficile (diversité des systèmes)
- tout ne peut être normalisé
- quelques divergences dans l'implémentation (ex threads Unix).

# Norme POSIX, organisation

La norme est découpée en chapitres, sous-chapitres. L'implémentation de certains (sous-) chapitres sont obligatoires, mais pas les autres.

Exemples de chapitres :

- 1003.1 : services de base (ex :fork, exec, ...)
- 1003.2 : commandes shell (ex : sh)
- 1003.5 : POSIX en ADA

# POSIX, ce qui nous intéresse dans la suite

Chapitre :

- Extensions temps-réel 1003.1b

# Résumé rapide

Les extensions “temps-réel” POSIX et leurs préfixes :

- ordonnancement sched\_
- synchronisation (sémaphores, signaux) sem\_ intr\_
- messages
- gestion mémoire shm\_ mémoire partagée
- gestion du temps (horloges clock\_, timers timer\_ )
- entrées/sorties asynchrones aio\_

# Services POSIX pour l'ordonnancement

- ils doivent s'appliquer au niveau thread et/ou niveau processus ;
- doivent implémenter  $\geq 32$  niveaux de priorités (fixes) ;
- une file d'attente par niveau de priorité ;
- des politiques différentes de gestion de file : SCHED\_FIFO, SCHED\_RR, SCHED\_OTHERS


Un peu de doc ? `man 7 sched` :

SCHED\_FIFO: First in-first out scheduling

SCHED\_FIFO can be used only with static priorities higher than 0 [...]

SCHED\_RR: Round-robin scheduling

# Niveaux de priorité

`sched_get_priority_min()` → 


Niveau de priorité le plus faible

... → 

n → 

n+1 → 

... → 

`sched_get_priority_max()` → 

Niveau de priorité le plus fort



# Listing des fonctions principales

<i>sched_get_priority_max</i>	Consulte la valeur de la priorité maximale.
<i>sched_get_priority_min</i>	Consulte la valeur de la priorité minimale.
<i>sched_rr_get_interval</i>	Consulte la valeur du quantum.
<i>sched_yield</i>	Libère le processeur.
<i>sched_setscheduler</i>	Positionne la politique d'ordonnancement.
<i>sched_getscheduler</i>	Consulte la politique d'ordonnancement.
<i>sched_setparam</i>	Positionne la priorité.
<i>sched_getparam</i>	Consulte la priorité.
<i>pthread_setschedparam</i>	Positionne la priorité.
<i>pthread_getschedparam</i>	Consulte la priorité.

Voir le man pour les détails.

## Exemple : changement de paramètres de 2 processus

```
struct sched_param parm;  
int res=-1;  
  
...  
/* Tache T1 ; P1=10 */  
parm.sched_priority=15;  
res=sched_setscheduler(pid_T1,SCHED_FIFO,&parm)  
if(res<0)  
    perror("sched_setscheduler tache T1");  
  
/* Tache T2 ; P2=30 */  
parm.sched_priority=10;  
res=sched_setscheduler(pid_T2,SCHED_FIFO,&parm)  
if(res<0)  
    perror("sched_setscheduler tache T2");
```

# Extensions POSIX 1001b pour la manipulation du temps

Entre autres :

- Support de plusieurs horloges.
- Impose la présence d'au moins une horloge : `CLOCK_REALTIME` (précision d'au moins 20 ms).
- Structure `timespec`.
- Services : consultation et modifs des horloges, mise en sommeil d'une tâche, **timer périodique couplé avec les signaux UNIX**.

Et des signaux, ...

# Conclusion

Take away message :

- Linux n'est pas un système temps-réel et les ordonnanceurs n'offrent pas de garantie.
- On peut faire des choses sympa avec la norme Posix1001b.