Cours de système

Ordonnancement des processus

Sébastien Paumier/Sylvain Cherrier

Le multi-tâche

- 2 contraintes:
  - on ne doit rien changer aux habitudes de programmation
  - un processus ne doit pas pouvoir squatter le processeur
- conséquences:
  - hors de question de passer par une participation active des processus (cooperatif)
  - on doit avoir un support matériel, comme pour la mémoire

The holy clock

- solution: interruptions régulières du processeur par un mécanisme externe, l'horloge
- permet au noyau de reprendre le contrôle du processeur pour faire ce qu'il a à faire, comme changer de processus en cours d'exécution
- L'horloge est hyper-prioritaire

Les interruptions

- interruption=alerte que le processeur peut recevoir n'importe quand et qui suspend son activité courante pour être traitée par un code spécial, le *gestionnaire d'interruption*
- vecteurs d'interruption=numéros entre 0 et 255 de toutes les interruptions que le processeur peut recevoir

`arch/x86/kernel/traps.c: trap_init`

Chemins de contrôle

- chemin de contrôle=séquence d'instructions exécutée par le noyau pour gérer une interruption ou une exception
- ces routines doivent être:
  - rapides
  - protégées contre les problèmes de réentrance
  - protégées contre les problèmes d'accès concurrents sur les systèmes multi-processeurs

Les interruptions matérielles

- émises par le matériel:
  - horloge, contrôleurs de disque, clavier, port série, etc
- traitées par le contrôleur d'interruption:
  - conversion du signal physique émis par un périphérique en signal vers le processeur pour le prévenir qu'une interruption a eu lieu
  - attend un acquittement de celui-ci

Les interruptions matérielles

- 2 catégories:
  - masquables: peuvent être temporairement désactivées, pour éviter des problèmes de réentrance dans les gestionnaires d'interruptions
  - non masquables: réservées à des événements très critiques comme des défaillances du matériel; utilisées entre autres pour programmer des *watchdog timers*

Les interruptions matérielles

- E/S indépendantes du processus courant, pouvant être émises n'importe quand
- règle d'or sous Linux: pas de commutation de processus pendant un chemin de contrôle lié à une interruption matérielle !
- mais, ils peuvent être imbriqués
  - exemple: acquitter une interruption pendant le traitement d'une autre

Les interruptions logicielles

- programmables, pouvant être déclenchées par du code
- utilisées pour implémenter des appels système, car ils doivent provoquer un changement de contexte (utilisateur → noyau)
- utilisées aussi pour des vérifications de débordement et du débogage

Les exceptions

- interruptions produites quand le processeur détecte une anomalie à l'exécution d'une instruction
  - faute: erreur pour laquelle on pourra ré-exécuter l'instruction fautive (exemple: faute de page)
  - trappe: l'instruction ne sera pas relancée (exemples: interruption de débogage, division par zéro)
  - abandon: erreur interne de l'UC

Les exceptions

- relatives au processus courant
- sauf bug du noyau, la seule exception possible en mode noyau est la faute de page
  - son gestionnaire est soigneusement écrit pour ne générer aucune exception
- toutes les autres se produisent en mode utilisateur

Synchronisation

- certaines portions de chemins de contrôle ne doivent pas être interrompues sous peine de corrompre des données
- pour éviter ça, 4 techniques:
  - non-préemptibilité des processus en mode noyau
  - opérations atomiques
  - désactivation des interruptions
  - verrouillage du noyau

Non-préemptibilité

- aucun processus ne peut être préempté quand il est en mode noyau
- un processus en mode noyau peut être interrompu par un chemin de contrôle, mais il reprend son exécution ensuite
- cela garantit qu'un appel système non bloquant ne peut pas être interrompu par un autre appel système pour le compte d'un autre processus
- pas vrai dans un noyau temps-réel (inversion de priorité)

Atomicité

- pour se protéger avec des verrous, il faut que la pose d'un verrou puisse se faire de façon sécurisée
- opérations atomiques des processeurs:
  - atomic_read(v)
  - atomic_set(v,i)
  - atomic_add(v,i)
  - atomic_dec_and_test(v)
  - etc

Bloquer les interruptions

- utile quand on ne peut pas se contenter d'une instruction atomique
- à manipuler avec précaution:
  - si le processeur exécute une instruction bloquante comme une E/S sur disque, il ne sera jamais réveillé !
- solution qui ne convient qu'à de petites portions de code

Verrous

- principe: pour accéder à une portion de code, il faut acquérir un verrou
- tant que le verrou est posé, les autres processus qui en ont besoin attendront
- à la libération du verrou, l'un d'eux pourra acquérir le verrou à son tour

Big Kernel Lock

- Par souci de sécurité, Linux proposait et utilisait un verrou de noyau complet (Big Kernel Lock : BKL)
- Simple et sécurisé, mais pas très efficace !!
- Devenu l'ennemi public n°1 ('Kill the BKL'), il a completement disparu depuis le 2.6.39
- Les locks sont faits au plus proche.

Les spinlocks

- attente **active** sur une valeur testée et modifiée de façon atomique
- utiles quand il y a plusieurs processeurs
- pour des raisons d'efficacité, il faut essayer d'avoir des verrous les plus fins possibles:
  - les uns en lecture, les autres en écriture
  - verrous différents pour chaque type de données à protéger
- coûteux, donc faire bien attention !

Les sémaphores

- verrous possédant:
  - un compteur pour savoir combien de chemins de contrôle attendent la ressource
  - une file d'attente des processus endormis en attente de la ressource
  - un spinlock permettant de garantir qu'un seul processus va acquérir la ressource
- les processus en attente sont suspendus (plus léger qu'un spinlock)

Etats d'un processus(simple)

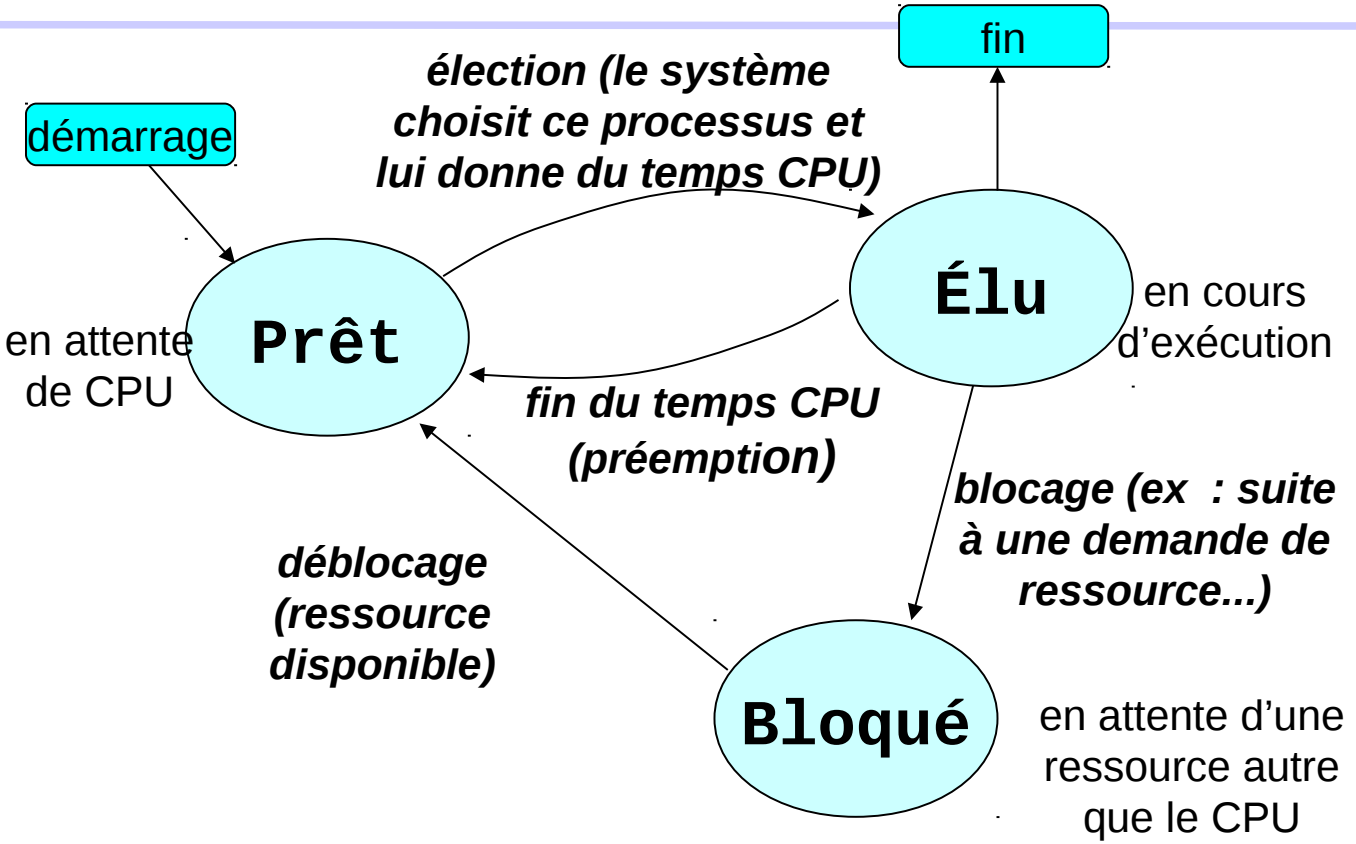
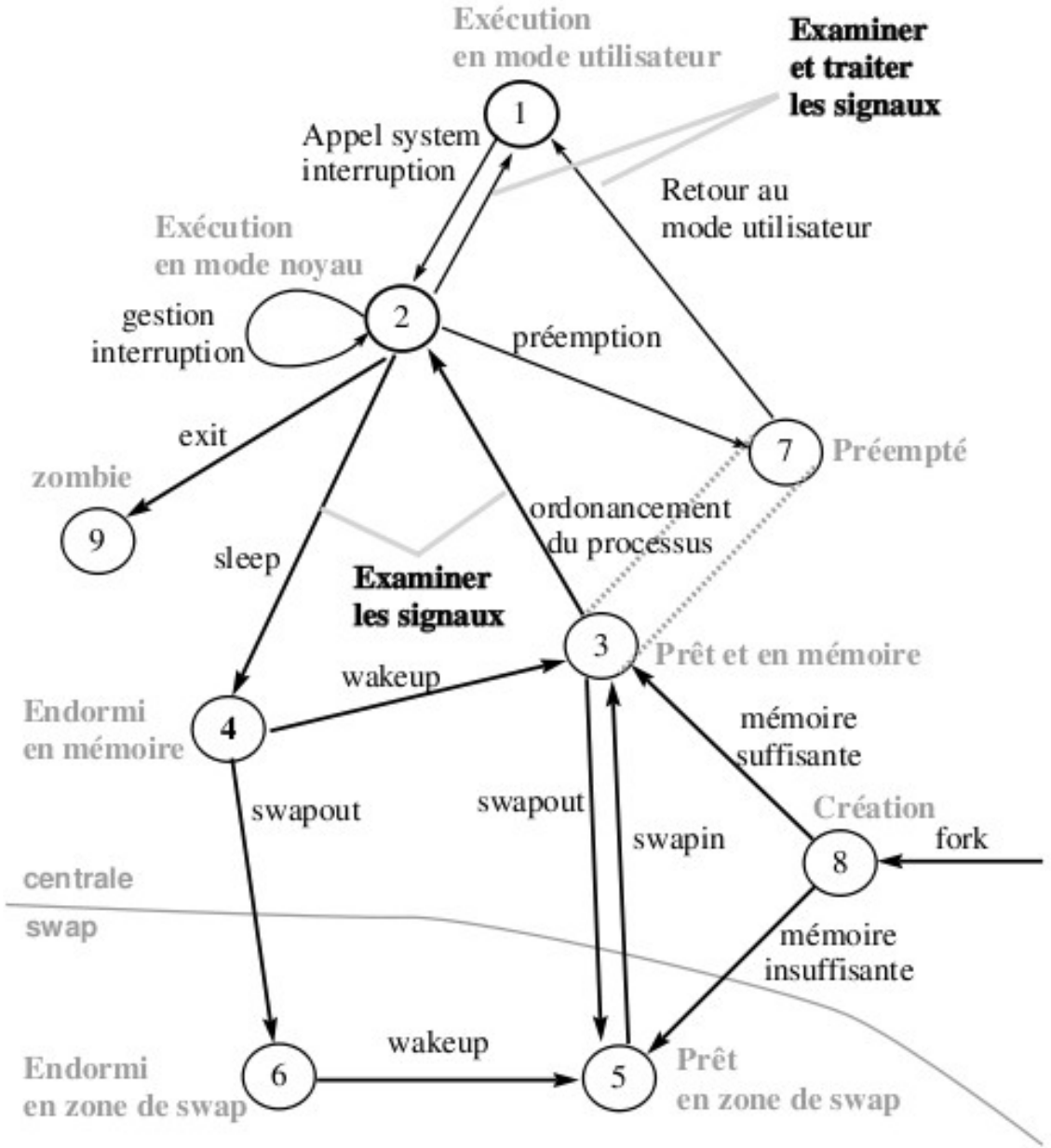


Diagramme d'état d'un processus Unix/Linux

Voir cours d'Emmanuel Desvigne (NSY103)

Etats d'un processus



`include/linux/sched.h`

Ordonnancement

- ordonnancement=sélection du prochain processus qui va s'exécuter
- ordonnancement non préemptif:
  - on attend qu'un processus termine ou fasse un appel bloquant
  - très mauvais (boucle infinie ?)
- ordonnancement préemptif:
  - grâce aux interruptions, le noyau peut en plus reprendre régulièrement le contrôle au processus qui s'exécutait

Ordonnancement

- l'ordonnanceur doit viser plusieurs objectifs:
  - éviter les famines
  - gérer des priorités d'exécution
  - garantir un temps de réponse raisonnable aux processus interactifs
  - occuper le CPU au maximum
  - éviter autant que possible le *cache trashing*

Cache Trashing

- Défaut d'un système qui passe son temps à échanger des pages mémoires jusqu'à écroulement du système complet
- Lié à une famine de mémoire, au nombre de processus présents, et à la qualité du code

BAD

```
int m[256][256];
 for(i=0; i<=255; i++){
  for(k=0; k<=255; k++){
    m[k][i] = biniou();
  }
}
```

GOOD !!

```
int m[256][256];
 for(i=0; i<=255; i++){
  for(k=0; k<=255; k++){
    m[i][k] = biniou();
  }
}
```

Mesures

- on mesure ses performances avec (entre autres critères):
  - le débit: nombre moyen de processus exécutés en un temps donné
  - le taux utile: proportion de temps utilisé pour les processus utilisateurs
  - le temps de réponse (ex: délai entre la frappe d'une touche et l'affichage)
- la préemption permet d'entrelacer des processus pour optimiser ces valeurs

Types de processus

- 2 grandes catégories de processus:
  - processus de calcul: utilisent intensivement le CPU
  - processus interactifs:
    - attendent souvent l'utilisateur avec des appels d'E/S bloquants
    - peuvent être marqués comme prioritaires car ils travaillent peu et rendent vite la main
- l'ordonnanceur essaie de deviner à quel type appartient chaque processus

Quantum de temps

- chaque processus se voit attribué un quantum de temps, qui peut éventuellement être différent d'un processus à l'autre
- quand un processus est élu, il peut:
  - s'exécuter pendant ce quantum de temps avant d'être préempté par le noyau
  - rendre la main à cause d'un appel système bloquant

Round robin

- algorithme du tourniquet:
  - tout processus éligible est placé en fin de liste dès qu'il a rendu la main
  - on double les processus endormis
  - garantit l'absence de famine
- problème: bien choisir le quantum
  - trop petit: le taux utile chute
  - trop grand: algo équivalent à FCFS (First Come First Served)

Priorités

- amélioration avec des priorités:
  - à chaque itération, on élit la tâche la plus prioritaire
  - quand elle a consommé son quantum de temps, on lui donne une priorité basse
  - la priorité augmente avec le temps d'attente, pour éviter les famines
- prioritée basée sur la valeur nice et évoluant en fonction de l'utilisation du CPU

`kernel/sched.c: schedule`

Classes de priorités

- le système doit gérer les différentes priorités de façon intelligente
- le processus le plus prioritaire doit toujours être le swappeur:
  - la faute de page est la seule exception pouvant survenir en mode noyau
  - on doit pouvoir swapper pour la résoudre
  - le swappeur ne doit jamais être gêné

Queues multiples

- on peut affiner la gestion en séparant les processus en plusieurs queues
- chaque queue peut être gérée avec son propre ordonnanceur
- exemple:
  - round robin pour les tâches interactives
  - FCFS pour les calculs en tâche de fond

CFS

- Completely Fair Scheduler
- scheduler actuel de Linux, basé sur une implémentation par arbres rouge-noir
- la priorité décroît en fonction de l'utilisation réelle du CPU
- permet de gérer de l'ordonnancement par groupes de processus (par exemple, par utilisateur)

```
kernel/sched_fair.c
Documentation/scheduler/sched-design-CFS.txt
```

Load balancing

- sur les systèmes multi-processeurs (SMP), l'ordonnanceur doit choisir le processeur qui va exécuter le processus
  - à la création (fork balancing)
  - à l'élection
- permet parfois de garder des choses en cache pour améliorer les performances

`kernel/sched.c: set_task_cpu`

Appels système

- on peut modifier les paramètres utilisés par l'ordonnanceur avec:
- **`nice`**/**`getpriority`**/**`setpriority`**
- **`sched_yield`**: rendre volontairement le processeur sans bloquer le processus

sched_no_yield.cpp

sched_yield.cpp

Appels système

- **`sched_setscheduler`**/**`sched_getscheduler`**: changer la politique d'ordonnancement du processus courant
  - SCHED_OTHER: round robin
  - SCHED_BATCH: traitement par lot (exécution moins fréquente mais plus longue)
  - SCHED_IDLE: priorité encore plus faible que nice 19
  - SCHED_FIFO: FCFS

Le temps

- pour gérer le temps, le système dispose de plusieurs horloges:
- horloge temps réel (RTC):
  - sur pile
  - indépendante de tout autre circuit
  - utilisée pour initialiser la date et l'heure au boot
  - émet des interruptions n°8 périodiquement
  - ordre de grandeur=milliseconde

Le temps

- le compteur d'estampilles temporelles (TSC):
  - sur les x86, registre mis à jour régulièrement, lisible par l'instruction assembleur **`rdtsc`**
  - ordre de grandeur=nanoseconde
  - mais, des correctifs à faire car la mise à jour peut être instable

`arch/x86/kernel/tsc.c`

Le temps

- le timer périodique programmable (PIT):
  - émet régulièrement des interruptions n°0
  - intervalle=*tick*
  - ordre de grandeur=milliseconde
  - utilisé pour l'ordonnancement
  - tient à jour l'horloge du système:
    - utilise le TSC pour mettre à jour la RTC
    - utile pour synchroniser les horloges de machines en réseau

Les timers

- **`unsigned int alarm(unsigned int nb_sec);`**
- envoie au processus en cours un signal SIGALRM au bout d'une durée fixée en secondes (peu précis)
- pour un timer plus précis, il faut les itimers:
- **`int setitimer(int which, const struct itimerval *value,struct itimerval *ovalue);`**

Les timers

- **`which`** définit le style de timer:
  - ITIMER_REAL: envoi de SIGALRM à l'expiration du délai
  - ITIMER_VIRTUAL: envoi de SIGALRM quand le processus a effectivement consommé le temps indiqué en mode utilisateur
  - ITIMER_PROF: envoi de SIGPROF quand le processus a consommé tout le temps imparti, modes noyau et utilisateur cumulés

Les timers

- **`value`** donne la précision:

```
struct itimerval {
   struct timeval it_interval; /* valeur suivante */
   struct timeval it_value;    /* valeur actuelle */
};

struct timeval {
   long tv_sec;                /* secondes        */
   long tv_usec;               /* micro secondes  */
};
```

- mais, on n'a pas une précision parfaite...

itimer.cpp

Temps réel

- pour une précision encore meilleure, il faut faire du temps réel (mou vs dur)
- pour ça, il faut un noyau capable de se préempter lui-même, ce qui est compliqué à gérer à cause des sections critiques des chemins de contrôle
- RT mou supporté par Linux depuis le noyau 2.6

`kernel/sched.c: preempt_schedule`

Temps réel dur

- le RT mou n'est fiable qu'à 99,XX %
- pour une fiabilité totale, il faut un vrai ordonnanceur temps réel
- principe du double-noyau pour avoir du RT dur
- le noyau normal est vu comme une tâche de fond par l'ordonnanceur RT
- ne travaille que s'il n'y a aucune tâche temps réel en cours