Gestion des interblocages

Conditions pour avoir un interblocage

Un interblocage arrive quand les 4 conditions suivantes sont vérifiées en même temps : [Coffman 1971]

- exclusion mutuelle : la/les ressource(s) n'est pas partageable
- "hold and wait" : le(s) thread(s) a déjà bloqué une ressource, et en demande une autre
- non-préemption : c'est le thread qui libère par lui même les ressources
- attente circulaire : il existe une chaîne de processus $P_1 \ldots P_k$ telle que chaque processus P_i bloque une ressource R_i et chaque processus P_i demande la ressource $R_{(i+1)\%k}$.

Exemple : les 5 philosophes ont chacun la fourchette de gauche, et attendent la fourchette de droite.

Prévenir et éviter les interblocages

Briser une des 4 conditions :

- Enlever l'exclusion mutuelle : par exemple
  - remplacer par des opérations atomiques.
  - algorithmes "sans blocages" : utilisation d'opérations atomiques "lecture-modification-ecriture"
- Enlever "hold and wait" : Exemple :
  - s'imposer de demander au plus 1 ressource à la fois.
  - bloquer plusieurs mutex atomiquement en même temps
  - si on demande plusieurs mutex, on fait attention à ne pas bloquer si on tient déjà un mutex
- Préemption : Difficile à faire... faudrait que cela soit prévu par les primitives et le programme

Prévenir et éviter les interblocages

Contre l'attente circulaire : ne pas créer de cycles dans le graphe

- Par exemple : Solution de Dijkstra pour les 5 philosophes
- Solution générale possible : mettre un ordre total sur toutes les ressources, et demander les ressources dans l'ordre
- Le graphe de dépendance sera toujours un sous graphe d'un graphe acyclique
- Exemple : 5 philosophes asymétriques : un des philosophes prend les fourchettes dans l'autre sens.
- Une solution simple dans le cas général : adresse mémoire du mutex

Prévenir et éviter les interblocages

Si on a (en plus) les informations de quelles ressources, ou combinaisons de ressources, peuvent être demandés, il est possible de prévenir dynamiquement les cycles.

Idée : On connaît le graphe des dépendances possibles, et on connaît le sous graphe des dépendances actuelles.
Il suffit de bloquer l'accès à une ressource qui pourrait créer un cycle. (Rester dans des états "sains")

Exemple : algorithme du Banquier de Dijkstra

Prévenir et éviter les interblocages

Exemple (5 philosophes)

- P_0 prend f_0, P_1 prend f_1, P_2 prend f_2, P_3 prend f_3
- P_4 demande f_4. Lui donner ? Non !
- P_0 possède f_0 : l'arc $f_0 \to f_1$ est possible.
- Etc. $f_0 \to f_1 \to f_2 \to f_3 \to f_4$ est possible
- Donner f_4 à P_4 créerait un cycle : un interblocage est possible.
- La demande de P_4 pour f_4 bloque jusqu'à ce que cette possibilité soit écartée

Détecter les interblocages

Il est théoriquement possible (pour l'OS) de détecter les interblocages.

Par exemple, si on a que des mutex (exclusion mutuelle, et pas de préemption), il suffit de construire le graphe de dépendance :

- Pour tout thread t bloqué dans un lock(R_t), tous les arcs entre $x \rightarrow R_t$, pour tout les x bloqués par t.

Et de tester ce graphe contient un cycle. Si oui : il y a un interblocage...

Mais que peut-on faire si on détecte un interblocage ?

Sous Linux ?

Le noyau Linux :

- S'occupe qu'il n'y ait pas d'interblocage dans le noyau
- Mais ne détecte/résout pas les interblocages des processus.

Pourquoi ?

- Quand on est dans une situation d'interblocage, on ne peut pas débloquer en respectant l'exclusion mutuelle
  - à part "tuer" quelque chose...
- On ne peut pas savoir à l'avance si on va arriver à une situation d'interblocage :
  Il faudrait analyser le code pour savoir les dépendances possibles...
  On frise avec des problèmes indécidables

Il faudrait des primitives de verrouillage beaucoup plus compliquées.
Cela en vaut il la peine ?

Livelock

Un autre type d'interblocage : le livelock

Aucun thread n'est bloqué, mais aucun thread n'avance (le code exécuté ne sert qu'à la gestion de concurrence)

Exemple : excès de politesse. Un thread veut laisser sa place pour une ressource à un autre thread si il le demande. Chaque thread se passe la main.

Concurrence : Ordonnancement

Ordonnancement : rappels

(Ici thread = thread ou processus non multi-threadé)

L'ordonnancement est préemptif.

Travail de l'ordonnanceur :

- Choisir à quel thread (prêt) il doit donner la main, et
- (pour les ordonanceurs préemptifs) combien de temps il lui donne.

État des threads

États des threads considérés par l'ordonnanceur :

- exécution : le thread est en exécution (sur un des coeurs)
- prêt : le thread attend que l'ordonnanceur lui donne la main
- en attente : le thread ne peut/doit pas être exécuté pour le moment (en attente d'une IO ou d'un mutex, sleep...)
- terminé

Changement d'état

Les threads "vivants" changent constamment d'état (exécution, prêt, attente)

- exécution $\rightarrow$ attente : appel système sur une ressource bloquante
- attente $\rightarrow$ prêt : la ressource se libère
- prêt $\rightarrow$ exécution : l'ordonnanceur donne la main au thread (dispatch)
- exécution $\rightarrow$ prêt :
  - le thread décide par lui même de rendre la main (POSIX : `sched_yield()`),ou
  - (ordo préemptif) le temps accordé au thread est dépassé (interruption)

Les différents temps

- temps total ("réel") : temps total d'un processus (de son création à sa terminaison)
- temps user : temps passé en mode utilisateur (temps passé en "exécution")
- temps système : temps passé en mode système (dans les appels systèmes)
- temps d'attente : temps passé en état "prêt"

(Note : pour voir les temps réels, user et sys : `time` *`commande`*).

Objectifs d'un ordonnanceur

Un ordonnanceur doit avoir plusieurs objectifs :

- Utiliser le(s) CPU à 100%
- Respecter l'équité
- Respecter les priorités ("nice")
- Minimiser le temps total d'une tache courte. (Réactivité. Par exemple : une commande.)
- Minimiser le temps total pour un long travail
- Éviter de faire trop de context-switch (par exemple, accorder des temps trop courts)
- Éviter de passer trop de temps dans l'ordonnanceur
- ...

Algorithmes l'ordonnancement

- Problème difficile
- Il y en a plusieurs possibles, en fonctions des objectifs principaux
- Cela pourrait être le sujet de tout un cours...
- Les algorithmes simples ont souvent des problèmes.

Stratégies "typiques" :

- First-Come, First-Served (FCFS) : (coopératif). Algo "minimal", on ne réfléchit pas. Peu de context switch. Problème : le temps de réponse peut être long (infini). Pas de gestion des priorités.
- Shortest-Job-First (SJF) : résout le problème de la réactivité. Problème : il faut connaître (ou prédire) le temps d'un processus a priori.

Algorithmes d'ordonnancement : Round-Robin

Round-Robin (RR) : (ordo préemptif)

- L'ordonnanceur a une liste (ou queue) L de threads "prêt".
- L'ordonnanceur prend (et retire) le premier thread (T) de la liste
- L'ordonnanceur exécute T, avec une limite de temps déterminé (ex : 0.1 seconde).
- Quand T rend la main (ex : IO bloquant), ou a épuisé son temps autorisé, l'ordonnanceur le rajoute à la fin de L.

Problèmes :

- Pas de gestion de la priorité.
- Si T fait souvent des appels à des I/O bloquantes, il est laissé.

Avec les priorités ?

Solutions :

- Avoir plusieurs listes (une par priorité)
  Si beacoup de priorités, un peu lourd...
  Décisions à prendre : quelle liste dispatcher ?

- Utilisation de files de priorités (plutôt que des listes/queues)
  (permet aussi de dépénaliser les threads qui font beaucoup d'I/O bloquantes)

Et avec plusieurs processeurs/coeurs ?

Un thread est généralement associé à un coeur
Pourquoi ?

- histoire de cache
- histoire de mémoire...

Mais si un coeur est moins utilisé qu'un autre, l'ordonnanceur peut décider de déplacer un thread.

L'ordonnanceur doit choisir quel coeur associer à un thread
Et décider quand le changer de coeur (load balance)

Parenthèse : UMA / NUMA

Architecture UMA (uniform memory access)

- une mémoire centrale partagée par plusieurs processeurs/coeurs

Exemples : vos machines (Intel Core ix, smartphones...)

Problème :

- si il y a beaucoup de coeurs, goulot d'étranglement pour l'accès mémoire
- les contrôleurs de mémoire sont (maintenant) souvent intégrés aux processeurs, et il difficile de concevoir des processeurs avec beaucoup de coeurs (plus de pertes, problème de dissipation thermique)

Parenthèse : UMA / NUMA

Architecture NUMA (non-uniform memory access)

- Chaque processeur (ou groupe de coeurs) dispose de sa mémoire (et forme un noeud)
- Mais chaque noeud peut accéder à toute la mémoire de la machine.
- Si c'est une mémoire d'un autre noeud, il faut passer par un bus qui inter-connecte les différents noeud (bus très rapide, mais quand l'accès est quand même plus lent)

Exemples : Systèmes multiprocesseurs courants (Intel Xeon, AMD Opterons...)

- Optimalement, il faudrait qu'un thread soit exécute dans le noeud où est sa mémoire

Contrainte supplémentaire pour l'ordonnanceur...

- Il faut interférer avec l'ordonnanceur mémoire
- déplacer un thread d'un noeud à un autre est plus contraignant

Ordonnanceur(s) de Linux

Plusieurs ordonanceurs au cours de l'histoire de Linux

- O(n) scheduler (Linux 2.4 : 2001-2011)
  - le temps est divisé en "epoch". À chaque epoch, chaque thread a droit à un certain nombre de temps CPU (basé sur la priorité)
  - Si un thread n'a pas épuise tout son temps CPU au cours d'une epoch, il rajoute la moitié du temps restant à son temps à l'epoch suivante.
  - Problème : temps linéaire en le nombre de processus entre chaque epoch

- O(1) scheduler (Linux 2.6, avant 2.6.23 : 2003-2007)
  - 140 niveaux de priorités (0-99 pour le système et temps réel, 100-139 pour les utilisateurs)
  - 2 queues par priorité : actif/inactif
  - pénalité si temps écoulé, récompense si I/O bloquante

Ordonnanceur(s) de Linux

Actuellement : Completely Fair Scheduler

- une file de priorité : arbre rouge-noir, indexé par le temps utilisé par le processus
- temps accordé : le temps que le thread a attendu $\times$ le ratio du temps qu'il aurait utilisé sur un processeur "idéal"
- l'ordonnanceur donne la main au thread qui a moins utilisé son temps (le plus petit dans la liste)
- quand le thread rend la main, l'ordonnanceur réinsère le thread dans l'arbre rouge-noir, avec son nouveau temps
- les processus avec beaucoup d'attente sont naturellement "récompensés"

Politiques et priorités POSIX

(Voir `man sched`)

On peut interférer avec l'ordonnanceur pour changer les politiques ou les priorités

(Généralement, plutôt utile pour les applications temps réel.)

Déjà vu : `nice()` pour un processus

Il y a aussi `getpriority()` / `setpriority()`. (Priorité d'un processus, d'un groupe de processus, et d'un utilisateur.)

Différentes politiques POSIX

Différentes politiques d'ordonnancement sont prévues dans POSIX (pour les processus et threads) :

- SCHED_OTHER : politique "standard"

Des politiques "temps réel" :

- SCHED_FIFO : (First-in-First-Out)
- SCHED_RR : (Round-Robin)

Dans ce cas, il y a une priorité supplémentaire (généralement entre 0 et 99) pour le thread
(Et quelques autres politiques, apparues plus récemment)

Pour changer la politique d'un processus : `sched_setscheduler()`

Pour changer la politique d'un thread :
`pthread_setschedparam()`

Affinités

On peur vouloir contrôler sur quel noeud un processus tourne, ou répartir ses threads sur différents noeuds, pour avoir de meilleurs performances.

Il est possible de choisir (sous Linux) sur quel(s) coeur(s) un thread peut tourner, avec `sched_setaffinity`.

Il est aussi possible de choisir des affinités mémoires. Voir `man numa`

Problème : Inversion de priorité

- A de forte priorité
- B de priorité moyenne
- C de faible priorité

Scénario :

- C bloque une ressource R
- A demande (et bloque) sur la ressource R
- B arrive
- B prend 100% CPU, car il a la priorité sur C
- C ne relâche pas R
- A ne peut pas être exécuté

B bloque A (par l'intermédiaire de C).
(problème réel : Mars pathfinder)

Problème : Inversion de priorité

Solutions :

- ne pas trop prioriser les threads de priorité supérieure. (Pas valable en temps réel)
- aléatoirement, donner du temps CPU à des taches de priorité basse (bricolage...)
- (priority ceiling) donner une priorité à un mutex. Si un thread bloque le mutex, il hérite (temporairement) de la priorité du mutex (si elle est plus haute)
- (priority inherance) si un thread C bloque le mutex, et un thread A de plus haute priorité demande le mutex, C hérite (temporairement) de la priorité de A.

Problème : Inversion de priorité

Dans pthread :

```
int pthread_mutexattr_getprotocol(
    const pthread_mutexattr_t *attr,
    int *protocol);
int pthread_mutexattr_setprotocol(
    pthread_mutexattr_t *attr,
    int protocol);
int pthread_mutex_setprioceiling(
    pthread_mutex_t* mutex,
    int prioceiling, int* old_ceiling );
```

`protocol` :

- PTHREAD_PRIO_NONE : le fait de bloquer le mutex n'affecte pas la priorité
- PTHREAD_PRIO_PROTECT : (priority ceiling) bloquer le mutex donne au thread la priorité du mutex (s'il est supérieur)
- PTHREAD_PRIO_INHERIT : (priority inherance) bloquer le mutex donne au thread le maximum des priorités des threads demandant le même mutex

Partage de mémoire entre processus (POSIX)

Mémoire partagée inter-processus

Plusieurs processus peuvent partager leur mémoire : c'est un IPC très rapide

Attention : comme pour les threads, il faut gérer la concurrence, soit :

- avec des sémaphores (pshared=1)
- avec des mutex partagés (attribut pshared)

Mémoire partagée inter-processus : cas simple

Entre père et fils :

```
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS,0,0);
if(fork()==0) {
   /* fils */
   strcpy(x,"COUCOU");
   return 0;
}
/* pere */
sleep(1);
printf("%s\n",x);
```

Affiche : COUCOU

Sans lien de parenté :

```
int fd=open("partage",O_RDWR|O_CREAT,0644);
ftruncate(fd,65536);
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,
    MAP_SHARED,fd,0);
```

Processus 1 :

```
while(1) {
 snprintf(x,999,"COUCOU␣%d",rand());
 sleep(1);
}
```

Processus 2 :

```
while(1) {
 sleep(1);
 printf("%s\n",x);
}
```

- Cela marche, mais on utilise le disque (plus lent que la RAM)
- On voudrait mimer ce comportement, sans passer par le disque

Mémoire partagée inter-processus : `shm_open`

Solution : objets mémoire partagé (`man shm_overview`)

```
int fd=shm_open("/partage",O_RDWR|O_CREAT,0644);
ftruncate(fd,65536);
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,
    MAP_SHARED,fd,0);
```

(Processus 1 et 2 comme précédemment)

- OK !
- En fait `/partage` sera dans un système de fichier en RAM, dans `/dev/shm/`

Objet mémoire POSIX

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

int shm_open(const char *name, int oflag,
    mode_t mode);
int shm_unlink(const char *name);
```

(compiler avec `-lrt`)

Renvoie un descripteur de fichier, sur lequel on peut faire les opérations qu'on a déjà vues :
`ftruncate`, `mmap`, `munmap`, `close`...

Sémaphore partagé entre processus

(Voir `man shm_overview`)

Deux façons de créer un sémaphore partagé :

- Sémaphore anonyme
- Sémaphore nommé

Sémaphore anonyme : le sémaphore doit être crée avec `sem_init`, avec `pshared=1`, dans une zone mémoire partagée (via `mmap`)

Sémaphore partagé entre processus

Sémaphores nommés (mécanisme similaire à `shm_open`) :

```
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
    mode_t mode, unsigned int value);
```

(compiler avec `-pthread`)

Mutex partagé entre processus

```
pthread_mutexattr_t mutexattr;
pthread_mutexattr_init(&mutexattr);
pthread_mutexattr_setpshared(&mutexattr,
    PTHREAD_PROCESS_SHARED);

pthread_mutex_init(pmutex,&mutexattr);
//...
```

`pmutex` doit être dans une zone mémoire partagée

Similairement, les variables de condition, rwlocks, barrières...
peuvent aussi être partagés entre processus.