TD 4 - ASR7 Programmation Concurrente

—

Ordonnancement

Matthieu Moy + l'équipe ASR7 (cf. page du cours)

Printemps 2020

I Ordonancement

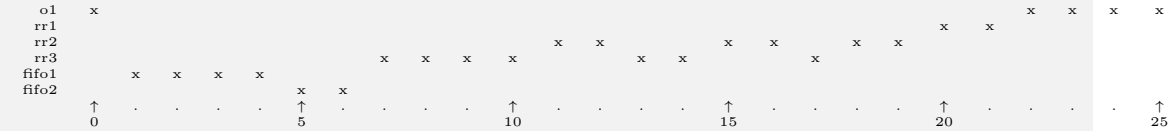
Nous allons utiliser l'implantation POSIX 1003b sur Linux. Il existe 100 niveaux de priorité :

- Le niveau 0 est réservé à SCHED_OTHER et les niveaux de priorité 1 à 99 aux politiques SCHED_FIFO et SCHED_RR.
- Les tâches de priorité 99 sont les tâches de plus forte priorité.
- SCHED_OTHER est dédié à l'ordonnanceur temps partagé.
- Le quantum utilisé par la politique SCHED_RR est de 2 unités de temps.
- Pour SCHED_FIFO et SCHED_RR, lorsqu'une nouvelle tâche arrive, elle est placée en queue (i.e. les tâches sont exécutées dans l'ordre d'arrivée).
- L'ordonnancement est préemptif.

Soit le jeu de tâches apériodiques suivant :

Tâche	Date d'arrivée	Priorité	Durée	Politique
o1	0	0	5	SCHED_OTHER
rr1	7	5	2	SCHED_RR
rr2	10	10	6	SCHED_RR
rr3	6	10	7	SCHED_RR
fifo1	1	10	4	SCHED_FIFO
fifo2	3	10	2	SCHED_FIFO

Q.I.1) - On suppose qu'une fois arrivées, les tâches sont toujours prêtes. Dessinez de l'instant 0 à l'instant 26, l'ordonnancement généré par l'ordonnanceur.



Q.I.2) - Donner le temps réponse de chaque tâche.

o1 : 25 - 0 + 1 = 26
rr1 : 21 - 7 + 1 = 15
rr2 : 19 - 10 + 1 = 10
rr3 : 17 - 6 + 1 = 12
fifo1 : 4 - 1 + 1 = 4
fifo2 : 6 - 3 + 1 = 4

II Choix d'ordonancement

Sur le noyau Linux (après le 2.4.4) une option est apparue pour l'ordonnanceur : il s'agit de `child_run_first`. Cette option concerne l'ordonnancement lorsqu'un processus fait un **`fork`**. Comme son nom l'indique, cette option signifie qu'à la suite du **`fork`**, c'est le processus fils qui prend la main.

Q.II.1) - Rappelez le comportement des **`fork`** vis-à-vis de la mémoire et la façon dont le système le gère.

Normalement, le fils copie la mémoire du père. Cela peut être coûteux surtout pour un **`fork`** suivit immédiatement d'un **`exec`** (écrasement de la mémoire pour faire autre chose).

Le système utilise le `copy-on-write` c'est à dire qu'il ne copie que la structure de données de gestion de la mémoire, de manière à ce que les processus partagent la même mémoire physique. Mais celle-ci est marquée Read-Only. Si l'un des processus fait une modification, la page est alors réellement copiée ailleurs.

Q.II.2) - Expliquez l'intérêt de l'option `child_run_first` en considérant le programme suivant.

```
1   int pid_t pid_com;
2   int nb_fils = 0;
3   ...
4   pid_com = fork();
5   if (pid_com < 0) {
6     perror("Il y a eu un problème durant le fork()");
7     exit(1);
8   }
9   if (pid_com == 0) {
10    /* Dans le fils */
11    execvp(arg_com1[0], arg_com1);
12    exit(2);
13  } else {
14    nb_fils++;
15    printf("Fils %d de pid %d lancé\n", nb_fils, pid_com);
16    ...
17  }
```

Le père change des données juste après le **`fork`** imposant un `copy-on-write` inutile puisque le fils n'en a pas besoin (il ne se sert pas de l'ancien contenu de la mémoire vu qu'il fait un `execvp` tout de suite). Lancer le fils en premier résout le problème.

Q.II.3) - Pourtant depuis le noyau Linux 2.6.32 cette option est désactivée par défaut. Proposez une raison.

À cause de l'évolution des processeurs le gain apporté par cette politique n'est plus avantageux par rapport au coût :

- Gain : seulement quelques pages par le principe de localité. Le nombre de pages potentiellement changée par le père est faible puisque un processus ne reste que peu de temps dans le processeur et donc n'exécute que peu d'instructions. De plus (pp de localité) pendant un temps court, les adresses mémoire utilisées sont proches les unes des autres (donc sur les mêmes pages mémoire)
- Coût : un changement de contexte : le fait de changer de processus n'est pas gratuit (En plus, les processeurs deviennent de plus en plus complexes et cela coûte de plus en plus cher) :

— le pipeline instruction (les instructions en préparation) est vidé or il est de plus en plus important

— le TLB est invalidé (celui qui permet de lire une case mémoire sans avoir à lire la table des pages) or ce dernier est de plus en plus gros

— cela stop tout un tas de petites optimisations fait à la volée par le processeur (sauvegarde du microcode, réordonancement des instructions, exécution en avance ...)

Des mesures ont montré que le coût est supérieur au gain.

On peut supposer que comme le `copy-on-write` est très utilisé, il y a eu un investissement matériel pour l'améliorer. De plus, à cause de l'apparition des multi-coeurs, il y a une bonne chance pour que le fils et le père soit de toute façon exécutés immédiatement et simultanément.

III Problème de mutex (si le temps le permet)

Un utilisateur exécute la fonction suivante plusieurs fois depuis plusieurs threads différents :

```
1  std::mutex m;
2  int get_tache(lt *l)
3  {
4    m.lock();
5    if (l->debut == l->nbtaches) {
6      return -1;
7    }
8    int res = l->tab[l->debut];
9    l->debut++;
10
11   m.unlock();
12   return res;
13 }
```

Après un certain temps, le programme se bloque. Expliquez le problème. Comment le corriger ?

Il manque un `m.unlock()` avant le `return -1`, donc la fonction peut retourner en gardant le mutex verrouillé. Les appels suivants seront bloqués. La seule manière de déverrouiller étant de passer par `m.lock()`, le mutex ne sera jamais déverrouillé. Solution : ajouter le unlock() manquant.