Systèmes d'exploitation — Exercices
Concurrence

Seweryn Dynerowicz (Seweryn.DYNEROWICZ@umons.ac.be)

1 Syncronisation

En présence d'entités d'exécution indépendantes (*i.e.* processus et/ou *threads*) qui manipulent des objets partagés, il est nécessaire de s'assurer qu'aucune *race condition* résultant des accès concurrents à ces objets ne peut émerger. À cet effet, les mécanismes classiques de sémaphores et de *mutex* sont disponibles sous Linux [1] pour réglementer les passages en section critique. Les sémaphores peuvent se décliner en deux types différents ; anonyme ou nommé.

```
1  #include <stdlib.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4
5  #define SEM_PRIVATE 0
6  #define SEM_SHARED  1
7
8  sem_t count;
9  int main(int argc, char* argv[]) {
10   sem_init( &count , SEM_PRIVATE , 0);
11   // Traitement multi-threads
12   sem_destroy( &count );
13
14   return EXIT_SUCCESS;
15 }
16
17 void* behavior(void* unused) {
18   sem_wait( &count );
19   // Section critique
20   sem_post( &count );
21 }
```

FIGURE 1 – Utilisation d'un sémaphore anonyme partagé entre plusieurs *threads* d'un processus.

Un sémaphore anonyme est simplement identifié par son emplacement en mémoire. Si un processus ou un *thread* dispose de l'accès vers cette portion de mémoire, celui-ci est autorisé à appliquer les fonctions propres à ce sémaphore (*i.e.* `wait` et `post`). Le *listing* ci-dessus présente les éléments essentiels de l'utilisation d'un sémaphore anonyme.

Le sémaphore anonyme est présent sous la forme d'une variable globale (*cfr.* ligne 7) pour permettre à tous les *threads* d'y accéder. Notez qu'il est possible de partager un sémaphore anonyme entre plusieurs processus en plaçant celui-ci dans un segment de mémoire partagée (*cfr.* Communication).

L'initialisation du sémaphore est réalisée à l'aide de la fonction `sem_init(..)` (*cfr.* ligne 10) à laquelle sont passés, dans cet ordre, la référence à la variable stockant le sémaphore (*viz.* `&count`), une valeur stipulant si le sémaphore est privé à un processus (*viz.* `SEM_PRIVATE`) ou bien partagé entre plusieurs processus (*viz.* `SEM_SHARED`). Enfin, la valeur initiale du sémaphore (*viz.* `0`) est fournie comme dernier paramètre. Au terme de l'utilisation du sémaphore, la fonction `sem_destroy(..)` (*cfr.* ligne 12) doit être utilisée pour clôturer celui-ci et libérer les ressources associées.

La manipulation du sémaphore peut être réalisée à l'aide des fonctions [2] `sem_wait(..)` (*cfr.* ligne 18) et `sem_post(..)` (*cfr.* ligne 20). La portion de traitement *multi-threads* (*cfr.* ligne 11) représente la création et l'attente de terminaison d'un ensemble de *threads* qui exécute la fonction `behavior(..)` contenant une section critique nécessitant le sémaphore.

1. Pour compiler un programme les utilisant, la commande `$ gcc -pthread -lrt sourcecode.c` doit être invoquée.
2. Il existe également une fonction `sem_trywait(..)` qui permet de récupérer la valeur du sémaphore de manière atomique et non-bloquante.

```
1  #include <fcntl.h>
2  #include <stdlib.h>
3  #include <semaphore.h>
4
5  int main(int argc, char* argv[]) {
6    sem_t* count = sem_open( "/application-semaphore", O_CREAT, 0600, 1);
7
8    sem_wait( count );
9    // Section critique
10   sem_post( count );
11
12   sem_unlink( "/application-semaphore" );
13
14   return EXIT_SUCCESS;
15 }
```

FIGURE 2 – Utilisation d'un sémaphore nommé qui est partagé entre plusieurs processus.

Un sémaphore nommé est identifié par une chaîne de caractères (*i.e.* son nom) qui doit commencer par un caractère '/'. Si un processus ou un *thread* connaît le nom du sémaphore, il est en mesure d'obtenir un accès dessus pour y appliquer les fonctions propres (*i.e.* `wait` et `post`). Le *listing* ci-dessus présente les éléments essentiels des sémaphores nommés.

Le sémaphore peut être présent sous la forme d'une variable locale (*cfr.* ligne 4) étant donné que lorsque son nom est connu, l'accès peut être obtenu à n'importe quel endroit du code (y compris dans le corps d'un *thread*). La fonction `sem_open(..)` prend comme paramètres, dans cet ordre, le nom du sémaphore dont on demande l'ouverture, les paramètres de l'ouverture (*cfr.* `open(..)`), les modalités d'accès (*i.e.* lecture et/ou écriture) ainsi qu'une valeur initiale à utiliser si le sémaphore est créer par cet appel. Cette ouverture fournit comme valeur de retour une adresse à laquelle le sémaphore est accessible. Au terme de son utilisation, la fonction `sem_unlink(..)` (*cfr.* ligne 11) doit être utilisée pour clôturer l'accès à ce dernier et libérer les ressources allouées.

La manipulation du sémaphore peut être réalisée à l'aide des fonctions `sem_wait(..)` (*cfr.* ligne 7) et `sem_post(..)` (*cfr.* ligne 9). Le traitement principal (*cfr.* lignes 6 à 9) peut être plus complexe (*e.g.* boucle) voire organisé sous forme d'un traitement *multi-threads*.

```
1  #include <pthread.h>
2
3  pthread_mutex_t mutex;
4  int main(int argc, char* argv[]) {
5    pthread_mutex_init( &mutex, NULL );
6
7    // Traitement principal
8    pthread_mutex_lock( &mutex );
9    // Section critique
10   pthread_mutex_unlock( &mutex );
11
12   pthread_mutex_destroy( &mutex );
13
14   return EXIT_SUCCESS;
15 }
```

FIGURE 3 – Utilisation d'un mutex utilisé par plusieurs *threads*.

Un *mutex* est identifié par son emplacement en mémoire de façon similaire à un sémaphore anonyme. Celui-ci peut se présenter sous la forme d'une variable globale (*cfr.* ligne 3) ou résider dans un segment de mémoire partagée. L'initialisation est réalisée à l'aide de la fonction `pthread_mutex_init(..)` (*cfr.* ligne 5) à laquelle il faut passer une référence à l'emplacement (*i.e.* `&mutex`) ainsi que la référence vers un descripteur d'attributs (*cfr.* `pthread_mutexattr_t` pour plus de détails) pouvant être utilisés pour configurer le fonctionnement spécifique du *mutex*. Le passage d'une valeur `NULL` permet de stipuler que le *mutex* doit se comporter suivant l'implémentation classique.

La manipulation du *mutex* se fait à l'aide des fonctions `pthread_mutex_lock(..)` et `pthread_mutex_unlock(..)` qui permettent, respectivement, d'obtenir et de restituer l'accès à la section critique. Le fonctionnement classique de blocage/déblocage avec file d'attente des *threads* bloqués est utilisée derrière ces fonctions.

2 Communication

Afin de pouvoir transférer des données entre processus indépendants [3], le mécanisme de communication des *message queues* est adapté lorsque le volume de données à échanger est faible, étant donné qu'il repose sur des opérations de copies.

Le processus qui recevra des données via la *message queue* va se charger de créer cette dernière (*cfr.* ligne 11) en stipulant les arguments `O_CREAT | O_RDONLY` pour une création avec ouverture en lecture seule. Celui-ci se chargera également de détruire cette dernière au terme de son exécution avec la fonction `mq_unlink(..)` (*cfr.* ligne 18). La réception à proprement parler (*cfr.* lignes 14 et 15) repose sur la fonction bloquante `mq_receive(..)` à laquelle doit être passé un *buffer* où placer une copie du message reçu, la taille maximale (*viz.* `1024`) et une variable pour inscrire la priorité du message reçu.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <mqueue.h>
5  #include <sys/stat.h>
6
7  int priority = 0;
8  const char buffer[1024];
9
10 int main(int argc, char* argv[]) {
11   mqd_t queue = mq_open( "/message-queue", O_CREAT | O_RDONLY);
12   if(queue == -1) { perror("mq_open"); return EXIT_FAILURE; }
13
14   ssize_t amount = mq_receive( queue, buffer, 1024, &priority);
15   if(amount == -1) perror("mq_receive");
16
17   mq_close( queue );
18   mq_unlink("/message-queue");
19
20   return EXIT_SUCCESS;
21 }
```

FIGURE 4 – Réception depuis une *message queue*.

Le processus qui enverra des données via la *message queue* doit simplement ouvrir cette dernière (*cfr.* ligne 9) en stipulant l'argument `O_WRONLY` pour une ouverture en écriture uniquement. Au terme de son exécution, ce processus se limitera à clôturer la *message queue* en utilisant la fonction `mq_close(..)` (*cfr.* ligne 15). L'envoi d'un message (*cfr.* lignes 12 et 13) requiert de stipuler son contenu, sa taille ainsi que la priorité associée à cet envoi.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <mqueue.h>
5  #include <sys/stat.h>
6
7  const char* message = "Hello, process!";
8  int main(int argc, char* argv[]) {
9    mqd_t queue = mq_open( "/message-queue", O_WRONLY);
10   if(queue == -1) { perror("mq_open"); return EXIT_FAILURE; }
11
12   int status = mq_send(queue, message, strlen(message), 0);
13   if(status == -1) perror("mq_send");
14
15   mq_close( queue );
16
17   return EXIT_SUCCESS;
18 }
```

FIGURE 5 – Expédition vers une *message queue*.

3. *n.b.* Les *threads* d'un même processus ont, par définition, accès à l'intégralité de l'espace d'adressage et peuvent se passer des références.

Il existe également un mécanisme de communication, les segments de mémoire partagée [4], particulièrement adapté pour transférer des volumes de données plus importants entre processus.

La plupart des ressources associées aux mécanismes de communication requièrent l'utilisation d'une clé d'identification. Il est déconseillé de choisir directement une valeur de clé et de plutôt se reposer sur la fonction `ftok(..)` (*cfr.* ligne 10). Cette clé peut être utilisée pour récupérer un segment partagé en spécifiant sa taille ainsi que les paramètres pour demander sa création s'il n'existe pas (*cfr.* ligne 16). De la sorte, différents processus peuvent récupérer le même segment sur base de la clé unique dont il dispose.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/ipc.h>
4  #include <sys/shm.h>
5
6  #define SHM_RDWR 0
7
8  unsigned int* segment;
9  int main(int argc, char* argv[]) {
10   key_t ipc_key = ftok("application-shm", 4242);
11   if(ipc_key == -1) {
12     perror("ftok");
13     return EXIT_SUCCESS;
14   }
15
16   int shmid = shmget( ipc_key, 8192, IPC_CREAT | 0666 );
17   if(shmid == -1) {
18     perror("shmget");
19     return EXIT_FAILURE;
20   }
21
22   segment = shmat(shmid, NULL, 0);
23
24   if(segment == (void*) -1)
25     perror("shmat");
26   else
27     // Traitement principal sur le contenu du segment
28     printf("segment@%p\n", segment);
29
30   if (shmctl( shmid, IPC_RMID, NULL) == -1)
31     perror("shmctl");
32
33   shmdt( segment );
34
35   return EXIT_SUCCESS;
36 }
```

Figure 6 – Utilisation d'un segment de mémoire partagé.

Pour pouvoir effectivement utiliser le segment partagé, ce dernier doit être attaché dans l'espace d'adressage d'un processus à l'aide de la fonction `shmat(..)` (*cfr.* ligne 22). L'adresse obtenue représente celle où le segment est attaché et peut être utilisée comme n'importe quel pointeur afin de manipuler son contenu [5].

L'avantage de travailler avec ce mécanisme provient du fait que les processus qui produisent les données peuvent directement les inscrire dans un segment partagé. De ce fait, il n'est pas nécessaire de réaliser une copie lors du transfert ; tous les processus participants ont vue sur l'exemplaire unique des données. Ceci explique pourquoi les segments partagés sont préférables pour de grands volumes de données.

Une fois que le segment n'est plus utilisé par un processus, celui-ci doit être détaché de son espace d'adressage à l'aide de la fonction `shmdt(..)` en lui passant l'adresse où le segment est attaché. La fonction `shmctl(..)` (*cfr.* ligne 30) peut être utilisée que le segment soit détruit lorsque le dernier processus l'utilisant s'en détache (*viz.* `IPC_RMID`).

4. Vous pouvez lister l'ensemble des segments de mémoire partagée existants avec la commande `$ ipcs -a`
5. En prenant bien garde de ne pas dépasser l'adresse maximale déterminée par la taille demandée lors de la récupération du segment.

2.1 *Ring buffer*

Un *ring buffer* est une structure qui permet de transférer des données entre processus ou *threads* suivant une stratégie de file d'attente (*i.e. FIFO*). Cette structure est caractérisée par un espace de stockage de taille fixe pouvant contenir des éléments de taille fixe. Une telle structure dispose d'une tête de lecture ; marquant le prochain élément qui sera extrait, et d'une tête d'écriture ; marquant le prochain emplacement dans lequel une insertion se fera. Ces têtes ne peuvent jamais se dépasser ; à tout moment, la tête de lecture "précède" la tête d'écriture. Initialement, les deux têtes coïncident sur la position `0`. **Attention :** Lorsque les deux têtes coïncident, cela peut signifier soit que le *buffer* est vide, soit qu'il est rempli.

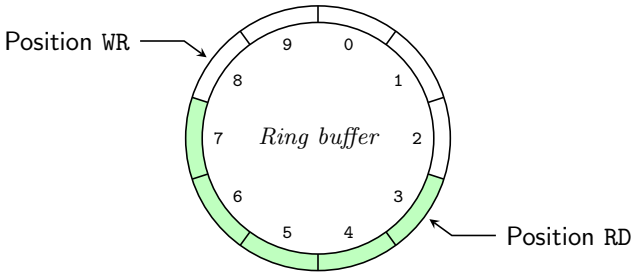

FIGURE 7 – Illustration d'un *ring buffer* d'une capacité de 10 éléments en contenant 5.

Il est possible de gérer l'insertion dans un *buffer* ne contenant pas de places libres suivant différentes politiques qui sont décrites ci-dessous.

Politique `DROP` : l'insertion dans un *buffer* plein n'est pas réalisée et l'élément est oublié. Le processus ou *thread* ayant tenté l'insertion est notifié du statut de l'opération.

Politique `OVERWRITE` : l'insertion dans un *buffer* plein a pour effet de remplacer l'élément qui se trouve à la position `WR` quoi qu'il advienne. L'élément remplacé est totalement oublié et son traitement n'arrivera jamais.

Politique `WAIT` : l'insertion dans un *buffer* plein a pour effet de bloquer le processus ou *thread* qui tente l'insertion jusqu'à ce qu'une position se libère.

3 Énoncés

Exercice 1

Implémentez un programme *multi-threads* qui utilise un *ring buffer* ayant une capacité de 50 éléments utilisant une politique `DROP`. Ce programme comporte 10 *threads* qui génèrent de façon répétitive chacun un nombre aléatoire après un temps aléatoire (compris entre 500ms et 5000ms) et le place dans le *ring buffer*. Un *thread* est chargé de lire ces valeurs et de comptabiliser combien de valeurs paires et impaires ont été fournies. Après avoir lu et testé une valeur, ce *thread* doit dormir pendant 250 ms.

Exercice 2

Implémentez un programme *multi-threads* qui utilise un *ring buffer* ayant une capacité de 50 éléments utilisant une politique `WAIT`. Ce programme comporte 10 *threads* qui génèrent de façon répétitive chacun un nombre aléatoire après un temps aléatoire (compris entre 500ms et 5000ms) et le place dans le *ring buffer*. Un *thread* est chargé de lire ces valeurs et de comptabiliser combien de valeurs paires et impaires ont été fournies. Après avoir lu et testé une valeur, ce *thread* doit dormir pendant 250 ms.

Exercice 3

Implémentez la logique de l'exercice précédent sous la forme de deux processus indépendants qui échangent les valeurs aléatoires à travers une *message queue*.

Exercice 4

Implémentez un programme composé de deux processus. Le processus enfant génère un tableau pouvant contenir un maximum de 32.768 entiers signés. Le processus parent se charge de trier le contenu de ce tableau. Une fois que le processus parent a terminé, le processus enfant vérifie que le tableau est bien trié et affiche un *feedback* dans le terminal stipulant la taille du tableau ainsi que son élément minimum.

Exercice 5

Implémentez un programme qui comporte deux *threads* manipulant une variable de type `long int` (*i.e.* 64 bits, initialisée à une puissance de deux). Ces deux *threads* appliquent, respectivement, une opération de multiplication par deux et une opération de division par deux un certain nombre de fois (*i.e.* `OPCOUNT`). Au terme de l'exécution de votre programme, la valeur de la variable partagée doit être égale à sa valeur initiale.

Exercice 6

Implémentez le dîner des philosophes en ajoutant une logique de détection de *deadlocks*. Dans cette approche, les baguettes doivent être prises individuellement (*cfr.* Troisième version, Chapitre 4, slide 31). Définissez la représentation de l'état de détention/attente ainsi que le critère qui permet de confirmer la manifestation d'un *deadlock*.

Exercice 7

Implémentez une solution au *sleeping barber problem*. Dans ce problème, un barbier dispose de $n > 0$ sièges dans une salle d'attente. Les règles suivantes doivent être respectées par votre implémentation.

- En l'absence de clients dans la salle d'attente, le barbier s'endort
- Lorsqu'un client arrive et que le barbier est occupé avec un autre client, l'arrivant va dans la salle d'attente. Si tous les sièges sont occupés, l'arrivant quitte la magasin
- Si le barbier est endormi, l'arrivée d'un client le réveille
- Lorsque le barbier termine avec un client, il va chercher le client suivant dans la salle d'attente

Travail de groupe

Implémentez le dîner des philosophes en utilisant une approche de *two-phase locking*. Dans cette approche, un tableau est utilisé pour représenter la disponibilité de chaque baguette. Lorsqu'un philosophe termine de `penser()` et s'apprête à `manger()`, celui-ci tente d'acquérir les deux baguettes dont il a besoin en une seule fois. Si au moins une des baguettes n'est pas disponible (*i.e.* détenue par un de ses voisins), le philosophe n'en prend aucune et retentera l'acquisition ultérieurement. Trouvez une solution pour éliminer l'attente active résultante.
Astuce : vous pouvez partir du code de la "Première version" (*cfr.* Chapitre 4, slide 27).