Programmation Système sous Linux

Pr. Karima AISSAOUI

Plan de la séance

- Rappel
- Terminaison des threads
- Synchronisation des threads

Synchronisation: rappel

- Dans la programmation concurrente, le terme de synchronisation se réfère à d eux concepts distincts (mais liés) :
  - La synchronisation de processus ou tâche : mécanisme qui vise à bloquer l'exécution des différents processus à des points précis de leur programme de manière à ce que tous les processus passent les étapes bloquantes au moment prévu par le programmeur.
  - La synchronisation de données : mécanisme qui vise à conserver la cohérence entre différentes données dans un environnement multitâche.
- Les problèmes liés à la synchronisation rendent toujours la programmation plus difficile.

Terminaison des threads

- Causes de terminaison des threads
  - Le thread termine sa fonction initiale
  - Le thread appelle la routine pthread_exit(): L'appel de « exit() » par un thread quelconque (principal ou fils) entraine la fin du processus et par conséquent la terminaison de tous les threads du processus.
  - Le thread est tue par un autre thread appelant pthread_cancel()
  - Tout le processus se termine à cause d'un exit, exec ou return du main(): La terminaison de l'activité initiale d'un processus (fonction «main()») entraine la terminaison du processus et par conséquent la terminaison des autres threads encore actifs dans le cadre du processus

pthread_exit()

- La fonction «pthread_exit() » permet de terminer le thread qui l'appelle.
- Syntaxe:

#include <pthread.h>

*void pthread_exit (void *p_status);*

- *p_status:* valeur de retour du thread (optionnel)
- Pour récupérer le code de retour, un autre thread doit utiliser *pthread_join()*
- « pthread_exit(NULL) » (pthread_exit(), pthread_exit(0)) est automatiquement exécuté en cas de terminaison du thread sans appel de «pthread_exit()».

Synchronisation: Attendre la fin d'un thread

- **pthread_join()**
- Un thread peut suspendre son exécution jusqu'à la terminaison d'un autre thread en appelant la fonction « pthread_join() ».
- Le thread A joint le thread B : A bloque jusqu'a la fin de B
- Similaire au wait() apres fork() (mais pas besoin d'être le créateur pour joindre un thread)
- Syntaxe: *int pthread_join (pthread_t thrd, void **code_retour);*
- - « thrd » : désigne l'identifiant du thread qu'on attend sa terminaison (le thread qui l'appelle attend la fin du thread « thrd »)
- - « code_retour »: un pointeur vers une variable « void* » qui recevra la valeur de retour du thread qui s'est terminé et qui est renvoyé par «pthread_exit()»

Synchronisation: Attendre la fin d'un thread

- Exemple: le thread principal (la fonction main()») doit attendre la terminaison de tous les threads qu'il a créé avant de se terminer.
- Si le thread attendu se termine anormalement, la valeur retournée dans « code_retour » est -1.
- - Si on n'a pas besoin de la valeur de retour du thread, on passe « NULL » au second argument.
- - Au plus un thread peut attendre la fin d'un thread donné.

Exemple

```
# include <pthread.h>
# include <stdio.h>
void* affiche_a (void *v) {
    int *cp=(int  *)v;
    int i;
    for (i=1;i<=*cp;i++)
        fputc ('a', stderr );
}

void* affiche_b (void *v) {
    int *cp=(int  *)v;
    int i;
    for (i=1; i<=*cp; i++)
        fputc ('b', stderr );
}
```

```
/* Le programme principal . */
int main () {
    int n=100, m=120;
    pthread_t thread_id1, thread_id2 ;
    / * créer le premier thread pour exécuter la fonction «affiche_a» */
    pthread_create (&thread_id1, NULL, affiche_a, (void *)n );
     /*créer le deuxième thread pour exécuter la fonction «affiche_a»*/
    pthread_create (& thread_id2, NULL, affiche_b, (void *)m);

    /* S'assurer que le premier thread « thread_id1 »  est terminé . */
    pthread_join ( thread_id1 , NULL );

    /* S'assurer que le second thread « thread_id2 » est terminé . */
    pthread_join ( thread_id2 , NULL );
    return 0;
}
```

Valeurs de retour des threads

- Si le second argument de « pthread_join() » n'est pas NULL, la valeur de retour du thread sera stockée à l'emplacement pointé par cet argument.
- Exemple: Un thread lit une valeur au clavier puis la transmet au thread pricipal.

```
include <pthread.h>
#include <stdio.h>
# define N 3
void *fonc_thread(void *v) {
    int k;
    scanf("%d",&k);
    pthread_exit((void *)k);
}
```

Valeurs de retour des threads

```
# include <pthread.h>
# include <stdio.h>
void *fonc_thread(void *v) {
    int k;
    scanf("%d",&k);
    pthread_exit((void *)k);
}
int main(){
   int donnee;
   void * res;
   pthread_t  thread_id;
   pthread_create(&thread_id, NULL, fonc_thread, NULL);
   pthread_join(thread_id, &res );
  donnee=(int) res;
  printf(" donnee = %d  \n", donnee);
  return 0;
}
```

- - « pthread_join() »: Suspend l'execution du thread appelant jusqu'à la terminaison du thread indiqué en argument.
- - « pthread_join() »: Rempli le pointeur passé en second argument avec la valeur de retour du thread.

Détacher un thread

- ***int pthread_detach (pthread t)***
- Détacher un thread revient a dire que personne ne le joindra à sa fin
- Le système libère ses ressources au plus vite
- Utile quand on ne veut ni synchronisation ni code retour
- I On ne peut pas ré-attacher un thread_detach

Synchronisation

- Les threads au sein du même processus sont concurrents et peuvent avoir un accès simultané à des ressources partagées (variables globales, descripteurs de fichiers, etc.).
- Si leur accès n'est pas contrôlé, le résultat de l'exécution du programme pourra dépendre de l'ordre d'entrelacement de l'exécution des instructions.
  - leur synchronisation est indispensable afin d'assurer une utilisation cohérente de ces données.
- Le problème de synchronisation se pose essentiellement lorsque:
  - Deux threads concurrents veulent accéder en écriture à une variable globale.
  - Un thread modifie une variable globale tandis qu'un autre thread essaye de la lire.

Synchronisation

- Des mécanismes sont fournis pour permettre la synchronisation des différents threads au sein de la même tâche:
  - La primitive « pthread_join() » (synchronisation sur terminaison): permet à un thread d'attendre la terminaison d'un autre.
  - Synchronisation avec une attente active (solution algorithmique).
  - Les « mutex » (sémaphores d'exclusion mutuelle): C'est un mécanisme qui permet de résoudre le problème de l'exclusion mutuelle des threads.
  - Les conditions (évènements).

Exemple

- Soient «thread1» et «thread2» deux threads concurrents. Supposons que le thread1 exécute la séquence d'instructions «A1» et le thread2 exécute la séquence d'instructions «A2».
- Les deux séquences d'instructions consistent à décrémenter une variable globale « V » initialisé à « v0 » (x variable locale).
- Thread 1 exécute la séquence d'instructions «A1» suivante
  - x=V-1;
  - V=x;
- Thread 2 exécute la séquence d'instructions «A2» suivante
  - x=V-1;
  - V=x;
- Les deux threads partagent la même variable globale «V». Le résultat de l'exécution concurrente de «A1» et «A2» dépend de l'ordre de leur entrelacement (ordre des commutations d'exécution des deux threads).

Premier scénario

- Supposons que c'est le « thread1 » qui commence l'exécution et que l'ordonnanceur ne commute les threads et alloue le processeur au « thread2 » qu'après la fin d'exécution de « A1 ».
- Le premier thread (exécution de « A1»)
  - lit la valeur initiale « v0 » dans un registre du processeur.
  - décrémente la valeur de « V » d'une unité (exécute l'instruction « x=V- 1 »).
  - écrit la nouvelle valeur dans la variable « V » (instruction « V=x »). La nouvelle valeur de « V » est « v0-1 »
- Ensuite l'ordonnanceur commute les tâches et alloue le processeur au deuxième thread.
- Le deuxième thread (exécution de « A2 »)
  - lit la valeur de « V » qui est « v0-1 ».
  - décrémente la valeur de « V » d'une unité (exécute l'instruction « x=V-1 »).
  - écrit la nouvelle valeur dans la variable « V » (instruction « V=x »). La nouvelle valeur de « V » est « v0-2 »;
- Donc après la fin d'exécution des deux threads, la valeur finale de « V » est égale à « v0-2 » ce qui est attendu.

Deuxième scénario

- On suppose que l'ordonnanceur commute les deux threads et alloue le processeur entre les deux threads avant la fin de leur d'exécution. Supposons que c'est le « thread1 » qui commence l'exécution.
- - Le « thread1 » lit la valeur initiale « v0 » puis effectue l'opération « x=V-1 ».
- - Avant que le « thread1 » écrit la nouvelle valeur dans la variable « V » (instruction « V=x »), l'ordonnanceur commute les threads et alloue le processeur au « thread2 ».
- - Le « thread2 » lit la valeur initiale de « V » (soit v0) et effectue les opérations « x=V-1 » et « V=x ». La nouvelle valeur de « V » devienne « v0-1 ».
- - L'ordonnanceur réactive le premier thread qui continue son exécution aupoint où il était arrêté, c'est-à-dire effectue l'opération « V=x », avec la valeur de x qui est « v0-1 ».
- Les opérations dans les threads sont effectuées dans l'ordre suivant:
  - thread1.1; thread2.1; thread2.2; thread1.2.
- Donc, après l'exécution des instruction dans cet ordre, la valeur finale de « V » est égale à « v0-1 » au lieu de « v0-2 » (valeur attendue).

Exemple

- Soit « tab » un tableau global d'entiers. Soient « thread1 » et « thread2 » deux threads concurrents: l'un remplit le tableau « tab » (modifie les éléments du tableau) et l'autre affiche le contenu du tableau après modification.

Thread1 thread2

pour (i=0; i<N; i++) pour (i=0;i<N,i++)

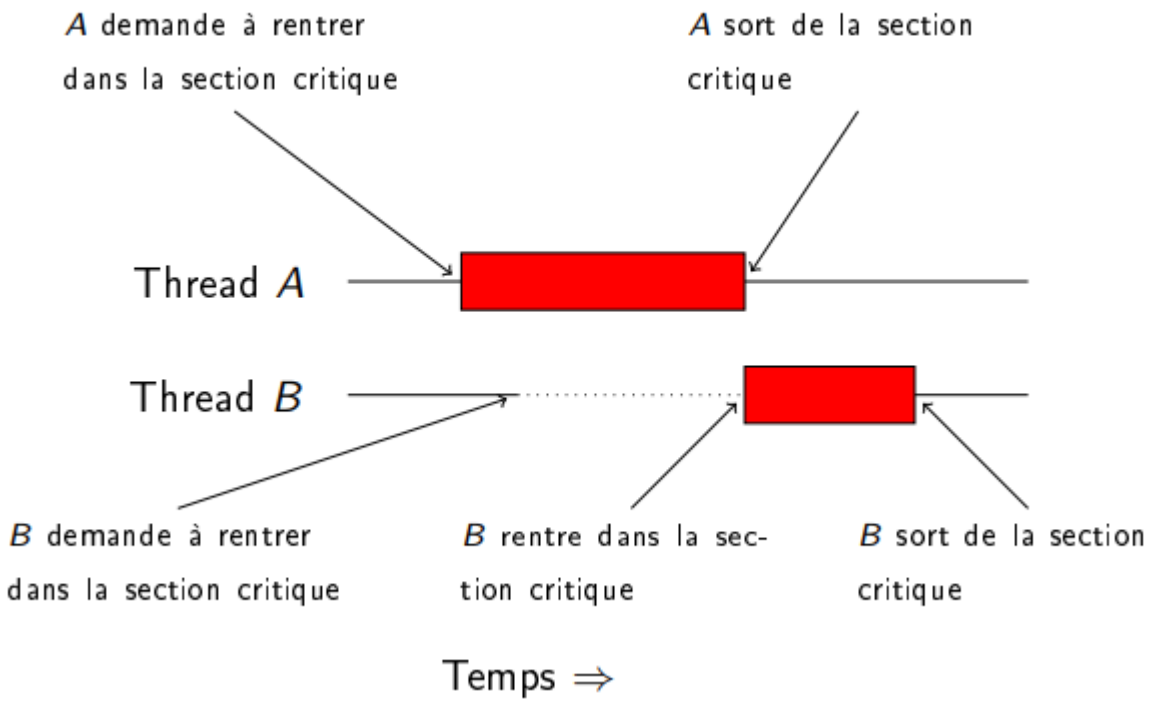
tab[i]=2*i; afficher (tab[i]);

- Puisque l'ordonnanceur commute entre les deux threads: il est possible que le « thread2 » commence son exécution avant la fin du remplissage du tableau (résultats erronés)
  - Pour avoir un affichage cohérent, on doit appliquer un mécanisme de synchronisation.
  - S'assurer que l'affichage ne peut avoir lieu qu'après la fin de remplissage du tableau.

Section critique

- Une section critique est une suite d’instructions dans un programme dans laquelle se font des accès concurrents à une ressource partagée, et qui peuvent produire des résultats non cohérents et imprévisibles lorsqu'elles sont exécutées simultanément par des threads différents.
- Une section critique est une section du code où il n'y a au maximum qu'un thread à la fois
- Si un thread B veut rentrer dans une section critique, et qu'un autre thread A est déjà dans la section critique, on doit faire attendre le thread B jusqu'à ce que le thread A sorte de la section critique.

Section critique



Avantage

- on ne bloque pas les autres threads qui ne travaillent pas sur la section critique
- il peut y avoir plusieurs sections critiques différentes, qui n'interfèrent pas entre elles.

Exemple

- Reprenons l'exemple 1.
- Puisque les deux threads accèdent de manière concurrente à la même variable partagée « V », alors la valeur finale dépend de la façon d'exécution des deux séquences d'instructions « A1 » et « A2 » par les deux threads.
  - Les séquences d'instructions « A1 » et « A2 » sont des sections critiques. Elles opèrent sur une variable partagée qui est «V».
- Pour éviter le problème de concurrence, on doit synchroniser les deux threads, c'est-à-dire s'assurer que l'ensemble des opérations sur la variable partagée (accès et mise à jour) est exécuté de manière atomique (indivisible).
  - Si les exécutions de « A1 » et « A2 » sont atomiques, alors le résultat de l'exécution de « A1 » et « A2 » ne peut être que celui de « A1 » suivie de « A2 » ou de « A2 » suivie de « A1 ».

Mécanisme de synchronisation sur attente

L'attente avant d'entrer dans une section critique peut être :

- active (spinlock) : le thread continue de tourner jusqu'à ce qu'il a le droit d'entrer dans la section critique
- passive : le thread est mis en pause jusqu'à ce qu'il a la possibilité de rentrer dans la section critique. Dans ce cas, il faut interférer avec l'ordonnanceur
  - Avantage : on laisse le temps CPU aux autres threads qui peuvent faire des choses plus constructives

Attente active

- Reprenons l'exemple2: modification et affichage des éléments du tableau.
- La synchronisation sur attente active est une solution algorithmique qui permet, par exemple, de bloquer l'exécution des instructions d'affichage du tableau (dans « thread2 ») jusqu'à la fin d'exécution des instructions des mises à jour du tableau (dans « thread1 »).

Attente active

- De manière générale, le prototype de l'attente active est le suivant (ressource est globale):

Thread 1	Thread 2
ressource occupée = true; utiliser ressource; /* Section critique : exécution de la partie qui nécessite la synchronisation: */ ressource occupée = false;	while (ressource occupée) {}; /* Rester bouclé jusqu'à ce que ressource occupée devienne false */ ressource occupée = true;

- Méthode très peu économique: « thread2 » occupe le processeur pour une boucle vide.

Solution avec une attente active

```
include <pthread.h>
#include <stdio.h>
# define N 300
pthread_t thread1,thread2;

int  ressource=0;  /* variable globale */

void *fonc_thread1(void *n);
void *fonc_thread2(void *n);
main(){
    pthread_create(&thread1, NULL, fonc_thread1, NULL );
    pthread_create(&thread2, NULL, fonc_thread2, NULL);
    pthread_join(thread1,NULL);
    pthread_join(thread2, NULL);
}
```

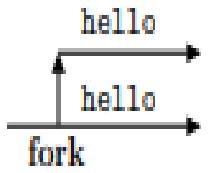
Solution avec une attente active - suite

```
void *fonc_thread1(void *n) {
   int i;
   for(i=0;i<N;i++)
         tab[i]=2*i;
   ressource=1;
/* modification de la variable « ressource » après avoir mis à jour
le tableau, c'est-à-dire après avoir été sortie de la section critique*/
}
void *fonc_thread2(void *n) {
    int i;
     while (ressource==0);  /* attend que la valeur de « ressource »
                               soit ≠ de 0 (boucle vide) avant d'entrer
                               dans la section critique */
     for(i=0;i<N;i++)
          printf("%d",tab[i]);
}
```

Exercice

- On considère les deux programmes suivants et leur schéma d'exécution:

```
1 int main() {
2    fork();
3    printf("hello!\n");
4    exit(0);
5 }
```



- Illustrer l'exécution de ces programmes:

```
int main() {
fork();
fork();
printf("hello!\n");
exit(0);
 }
```

```
int main() {
fork();
fork();
fork();
printf("hello!\n");
exit(0);
 }
```