Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TD n°4 : Synchronisation, signaux

L'objectif de ce TD [1] est de compléter certaines notions sur la synchronisation des processus et sur l'emploi des signaux pour réaliser cette synchronisation.

1 Synchronisation entre deux processus et section critique

Lorsqu'un processus père et son fils coopèrent, il est souvent nécessaire de les synchroniser. Plus précisément, chacun de ces deux processus a généralement une tâche distincte à accomplir et doit à une étape donnée informer l'autre qu'il est prêt à passer à l'étape suivante et/ou attendre la notification de l'autre. Par exemple, le père pourrait avoir besoin de consulter/modifier le contenu de fichiers que le fils est censé créer.

Une autre manière de voir la synchronisation et de considérer l'accès à des ressources partagées. Ainsi, si deux processus (p.ex père et fils) doivent travailler sur le même fichier, il faut s'assurer que le contenu du fichier reste cohérent. Pour cela, les accès concurrents du père et du fils au fichier doivent être contrôlés. Si les deux effectuent des lectures, alors les accès pourront se faire de manière concurrente. Si l'un des deux effectue une écriture, son accès doit être garanti *exclusif*.

La structure de base utilisée pour mettre en place de la synchronisation entre activités concurrentes s'appelle une *section critique*. La section critique peut etre vue comme une portion de code qui ne peut être exécutée que de manière exclusive. pour garantir une solution correcte, une implémentation de section critique doit garantir, sans faire des hypothèses sur les vitesses relatives d'exécution des différents processus, les propriétés suivantes :

- Exclusion mutuelle : deux processus ne peuvent accéder au même temps à la setion critique
- Absence de blocage : il ne doit pas y avoir de processus qui s'attendent mutuellement.
- Progrès : si il y a des processus qui demandent l'accès à la section critique et que aucun processus ne l'occupe, le prochain processus à entrer en section critique doit être choisi en temps borné.
- Attente bornée : un processus qui en fait la demande doit accéder à la section critique au bout d'un temps fini.

2 Réalisation d'une section critique par attente active

Reprenons les exemples vus en cours au sujet du problème de l'exclusion mutuelle par attente active avec deux processus. Expliquer la faille des deux propositions erronées

1. Plusieurs exemples sont empruntés à W.R Stevens, S. A. Rago. *Advanced Programming in the Unix Environment (2nd Edition)*, Addison-Wesley, 2005.

et démontrer que la solution de Peterson respecte les critères d'atomicité de la section critique, de progrès et d'attente bornée.

Proposition (incorrecte) n°1

`c` est initialisée avec la valeur 0.

Processus P1 :

```
while (1) {
    while (c == 1);
    c = 1;
    <section critique>
    c = 0;
}
```

Processus P2 :

```
while (1) {
    while (c == 1);
    c = 1;
    <section critique>
    c = 0;
}
```

Proposition (incorrecte) n°2

`c1 et c2` sont initialisées avec la valeur 0.

Processus P1 :

```
while (1) {
    c1 = 1;
    while (c2 == 1);
    <section critique>
    c1 = 0;
}
```

Processus P2 :

```
while (1) {
    c2 = 1;
    while (c1 == 1);
    <section critique>
    c2 = 0;
}
```

Solution de Peterson

`c1 et c2` sont initialisées avec la valeur 0, `t` est initialisée avec la valeur 1.

Processus P1 :

```
while (1) {
    c1 = 1;
    t = 2;
    while ((c2 == 1) && (t == 2));
    <section critique>
    c1 = 0;
}
```

Processus P2 :

```
while (1) {
    c2 = 1;
    t = 1;
    while ((c1 == 1) && (t == 1));
    <section critique>
    c2 = 0;
}
```

3 Synchronisation entre processus père et fils à base de signaux

On considère l'exemple suivant avec deux processus ayant chacun un message à afficher. La primitive `slow_write` est utilisée dans cet exemple pour simuler un périphérique de sortie très lent. Cette primitive reçoit une chaîne de caractères en paramètre d'entrée et affiche les caractères un par un [2], avec un petit délai entre chaque caractère.

```
#include "csapp.h"

void slow_write(char *str)
{
    char *ptr;
    char c;
    int i;

    ptr = str;
    setbuf(stdout, NULL); // unbuffered mode
    while (*ptr != 0) {
        c = *ptr;
        putc((int)c, stdout); // print the character on the standard output
        ptr++;
        for (i=0; i<50000; i++); // delay before the next character
    }
}


int main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        unix_error("fork error");
    } else if (pid == 0) {
        slow_write("output from child\n");
    } else {
        slow_write("output from parent\n");
    }
    exit(0);
}
```

2. L'écriture de chaque caractère est réalisée par un appel à la primitive `putc` en mode non tamponné (`unbuferred`). Dans cette configuration, chaque appel à `putc` déclenche un appel système pour envoyer un caractère sur la sortie.

3.1 Question

L'exécution de ce programme peut poser un problème. Lequel ?

3.2 Question

On se propose d'améliorer le programme précédent en utilisant une bibliothèque de primitives de synchronisation développée à cet effet :

- `wait_parent()` : attendre la notification du processus père du processus appelant ;
- `wait_child()` : attendre la notification d'un processus fils du processus appelant ;
- `tell_parent()` : notifier le processus père du processus appelant ;
- `tell_child(c)` : notifier le processus fils (dont le PID est `c`) du processus appelant ;
- `init_synchro()` : effectuer les initialisations nécessaires pour les primitives décrites ci-dessus.

Écrire une nouvelle version du programme qui répond au problème discuté à la question précédente en tirant parti d'une ou plusieurs des primitives présentées ci-dessus. Les messages s'affichent-ils dans un ordre déterministe ? Le cas échéant, préciser cet ordre et indiquer s'il est possible de le modifier. Donner éventuellement un diagramme temporel pour expliquer le raisonnement.

3.3 Question

Un problème peut éventuellement apparaître avec la solution précédente si le programme est lancé plusieurs fois d'affilée [3]. Lequel ? Expliquer la cause de ce problème et comment y remédier. Indice : ce problème est lié à l'ordre dans lequel les processus effectuent leurs affichages respectifs.
Donner éventuellement un diagramme temporel pour expliquer le raisonnement.

3.4 Question

On considère maintenant un autre exemple dans lequel une tâche (par exemple un calcul ou une analyse de données) est parallélisée grâce à l'utilisation de deux processus (un père et son fils). Toutes les données manipulées par le programme (paramètres d'entrée, résultats intermédiaires et résultat final) sont stockées dans des fichiers connus et accessibles par chacun des processus. Le calcul est décomposé en trois étapes : pour chaque étape, les données d'entrée sont découpées en deux partitions (l'une étant attribuée au père et l'autre au fils). Lorsque le père a terminé le traitement des données qui lui étaient assignées pour une étape donnée, il doit attendre que le fils ait également terminé son propre traitement pour l'étape considérée afin de fusionner les deux résultats partiels et d'afficher un résultat intermédiaire (procédure `merge`). En revanche, on suppose que le fils n'a pas besoin d'attendre la fin de la fusion avant de commencer l'étape suivante.

a) Compléter les squelette ci-dessous pour ajouter les appels aux primitives `tell`/`wait` nécessaires pour la synchronisation des deux processus.

3. par exemple, si le fichier exécutable s'appelle `prog`, via la ligne de commandes `prog; prog; prog`

```
int main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        unix_error("fork error");
    } else if (pid == 0) {    /* child */
        step1(A2);
        step2(B2);
        step3(C2);
    } else {    /* parent */
        step1(A1);
        merge(A1, A2);
        step2(B1);
        merge(B1, B2);
        step3(C1);
        merge(C1, C2);
    }
    exit(0);
}
```

b) La bibliothèque de primitives de synchronisation `tell`/`wait` est basée sur les mécanismes de signaux. L'envoi d'une notification d'un fils à son père (respectivement d'un père à son fils) correspond à l'émission d'un signal `SIGUSR1` (resp. `SIGUSR2`). Compte tenu de cette précision, la solution proposée à la question précédente présente-t-elle des problèmes potentiels en matière de synchronisation ? Le cas échéant, quelle précaution faut-il prendre pour assurer (de façon déterministe) le bon fonctionnement du programme ?

4 Consignes pour l'écriture d'un traitant de signaux

Cet exercice s'intéresse à des précautions importantes à prendre vis à vis des traitants de signaux. Pour chacun des exemples suivants, discuter des problèmes potentiels qui peuvent survenir et proposer des règles de programmation pour les éviter.

4.1 Accès concurrents à des données

```
montypeentier mavariable = valeur1;

void handler (int sig) {
    mavariable = valeur2;
}

int main(int agc, char *argv[]) {
    Signal(SIGINT, handler);
    ...
```

```
    printf("mavariable == %llx\n", (unsigned long long)mavariable);
    ...
}
```

4.2 Appels de fonctions

```
void handler (int sig) {
    int res;
    ...
    res = fonction();
    ...
}

int main(int agc, char *argv[]) {
    int r;
    Signal(SIGINT, handler);
    ...
    r = fonction();
    ...
}
```