Synchronisation

Lorsque plusieurs processus coopèrent, ils doivent souvent intéragir entre eux, ils doivent parfois attendre qu'une opération soit effectuée par un autre processus pour travailler.

Il faut donc avoir des mécanismes qui permettent d'envoyer des événements aux processus (un processus doit pouvoir attendre l'évènement).

Types de synchronisation

Sous UNIX, les mécanismes suivants sont mis en oeuvre pour la synchronisation :

- Les signaux
- Les sémaphores

On parlera de **point de synchronisation** lorsqu'un processus attend un autre.

Les signaux

Un signal est un événement capturé par un processus, c'est aussi un mécanisme simple utilisé par le système d'exploitation pour signaler aux processus une erreur (SIGILL, SIGFPE, SIGUSR1, SIGUSR2, etc).

Exemple

Voici par exemple un programme dont la fonction `sighandler` est appellée lorsque le signal SIGUSR1 est déclenché :

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void sighandler(int signum);

/*
  Ce programme va lier la fonction sighandler au signal SIGUSR1
  Ce qui signifie que lorsque l'on lance le programme (qui contient une boucle infinie), lorsque l'on lance le signal
```

```
via "pkill -SIGUSR1 a.out" (par exemple)
  La fonction sighandler va être appellée et "SIGUSR1 reçu" va donc s'afficher à l'écran.
*/
int main(void) {
  /* Si on remplace ici SIGUSR1 par SIGINT et que l'on fait CTRL+C, on va appeller la commande sighandler */
  if(signal(SIGUSR1, sighandler) == SIG_ERR) {
    printf("Erreur sur la gestion du signal\n");
    exit(-1);
  }

  while(1) {
    sleep(1);
    printf("Hello\n");
  }

  return EXIT_SUCCESS;
}

void sighandler(int signum) {
  printf("SIGUSR1 reçu\n");
}
```

Opérations

Il existe plusieurs opérations différentes sur les signaux :

- `signal` et `sigset` qui lient un signal à une fonction. `signal` la lie une seule fois, tandis que `sigset` la lie continuellement.
- `alarm` déclenche le signal SIGALARM au processus courrant.
- `pause` suspend le processus jusqu'a la réception d'un signal
- `kill` envoie un certain signal au processus dont le PID est donné.

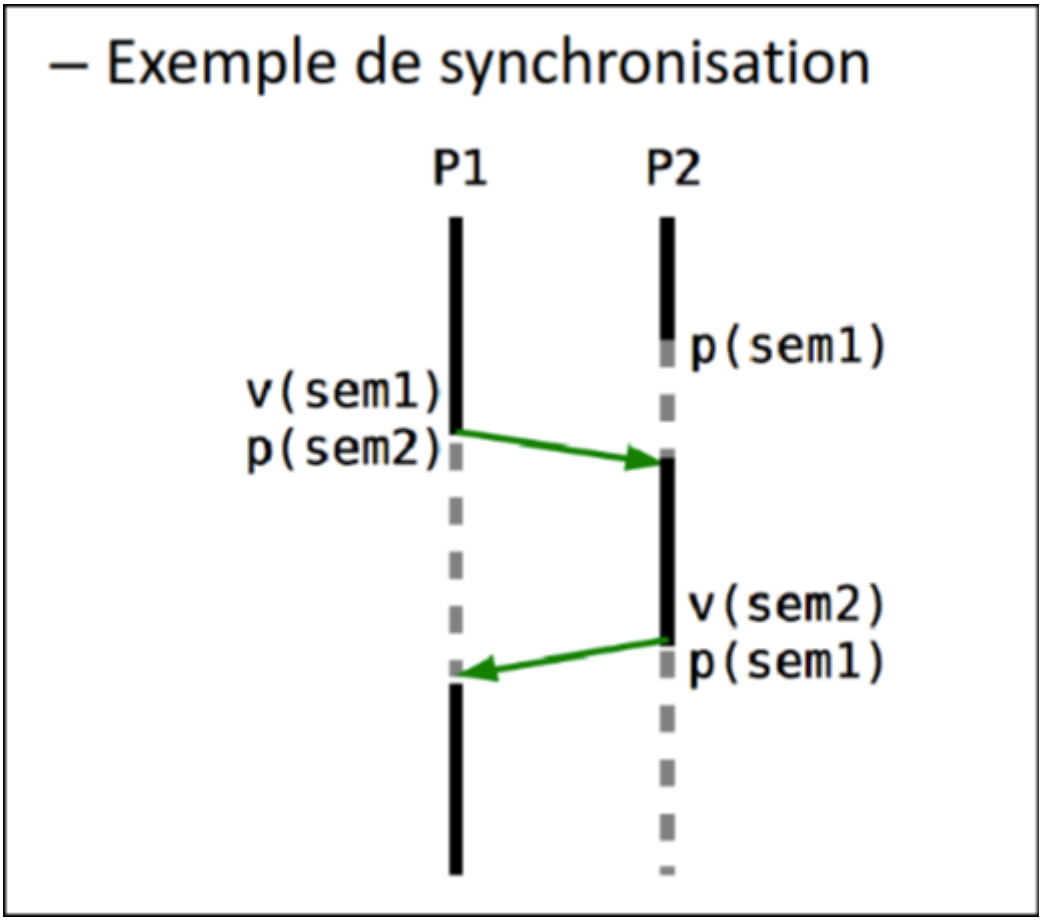
Les sémaphores

Un sémaphore est une variable entière en mémoire qui *excepté pour son initialisation* est accédée uniquement au moyen de fonction atomiques (ne pouvant pas être décomposée) `p()` et `v()`.

La fonction `p(sem)` va vérifier que la valeur est plus grande que zero, si c'est le cas, la variable est décrementée et l'exécution continue, si ce n'est pas le cas, alors il attend que ce soit le cas.

La fonction `v(sem)` va simplement incrémenter la variable de 1, et va ainsi réveiller tous les processus qui attendrait ce sémaphore.

Ces foncitons ne sont pas présente dans C de base il faut importer les fichiers `semadd.h` et `semadd.c` depuis l'espace de cours.



Exemple

Voici un exemple d'un programme qui communique avec un processus fils via 2 sémaphores. Il est intéressant de noter que généralement un processus ne va faire qu'une seule opération par sémaphore (par exemple que des `p()` sur sem1 et que des `v()` sur sem2 ou inversément)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "semadd.h"

#define SEM1 12345
#define SEM2 23456

/*
  Ce programme va créer 2 sémaphores et 2 processus (un père et un fils).
  Le fils et le père vont tous les deux exécuter une boucle sauf qu'a chaque itération ils vont s'attendre l'un l'autre.
```

Ainsi le père attends le sémaphore du fils (sem2) qui est émit lorsque le fils a fini son itération
De même le fils va ensuite attendre le sémaphore du père (sem1) qui est émit lorsque le père a fini son itération

Si on exécute ipcs -s lors de l'exécution du programme, on peut voir la liste des sémaphores créés.

Contrairement aux signaux, on peut créer nos propres sémaphores tandis que les signaux eux sont défini par le système d'exploitation.
*/

```
int main(void) {
  int sem1, sem2, i;

  /* Création des sémaphores */
  sem1=sem_transf(SEM1);
  sem2=sem_transf(SEM2);

  /* Création des deux processus */
  switch(fork()) {
    case -1:
      printf("Erreur fork()\n");
      exit(-1);

    /* Boucle du fils */
    case 0:
      printf("Je suis le fils %d\n", getpid());
      for(i=0;i<5;++i) {
        printf("[FILS] Valeur de i : %d\n",i);
        sleep(5);
        v(sem2); /* Envois du sémaphore (2) au père */
        p(sem1); /* Attente du sémaphore (1) du père */
      }

    /* Boucle du père */
    default:
      for(i=0;i<5;++i) {
        p(sem2); /* Attente du sémaphore (2) du fils */
        printf("[PERE] Je suis le père\n");
        sleep(5);
        v(sem1); /* Envois du sémaphore (1) au fils */
      }
```

```
  }

  return EXIT_SUCCESS;
}
```

Semget - Allocation de sémaphores

L'allocation se fait via `int semget(int key, int nb, int flag)`, où

- La valeur retournée est un descripteur "semid"
- La clé est la valuer qui identifie le sémaphore
- Les flags définissent les permissions, comme pour les mémoires partagées `IPC_CREAT` permet de demander la création des sémaphores

On peut aussi simplifier l'allocation à partir d'une clé en utilisant `int sem_transf(int key)`, cette fonction n'est **pas officielle** mais le fichier est disponible sur HELMo Learn.

Semctl - Gestion de sémaphores

On peut gérer les sémaphores (nottament pour libérer la mémoire) en utilisant `int semctl(int semid, int semnum, int cmd, union semun attr)` où

- semid est le descripteur du sémaphore
- semnum identifie le sémaphore (généralement c'est 0 si il n'y en a qu'un)
- cmd identifie la commande (IPC_{SET}, GETVAL, SETVAL, IPC_{RMID} ou IPC_{STAT}).
- `union semun attr` est une "union" (un type de structure où chaqun des éléments partagent la même zone mémoire, ainsi ce ne peut être qu'un seul élément à la fois, un peu comme une enum en Rust). Il faut généralement définir cette structure soi-même en revanche.

Semop - Faire les opérations sur les sémaphores

`int semop(int semid, struct sembuf* sops, unsigned nsops)` est la fonction qui est derrière les fonctions `p()` et `v()`.

- semid est le descripteur du sémaphore
- sops est un tableau de structures semfus (contenant l'opération)
- nsops est le nombre d'éléménts du tableau sops

Revision #5
Created 2 January 2024 10:40:58 by SnowCode
Updated 2 January 2024 14:32:58 by SnowCode