CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 – NANCY/METZ

EXAMEN DE RATTRAPAGE – SEPTEMBRE 2012

Année 2011 – 2012, deuxième semestre
Coefficient : 1 (100 % de la note)

Correcteur : Emmanuel DESVIGNE

Documents autorisés : NON

Exercice n°1 : synchronisation *(3 questions pour un total de 7 points)*

Un programme de calcul de prévisions météorologiques est composé de trois routines :

- lecture d'un enregistrement de données,
- traitement de ces données (simulation)
- et sauvegarde des résultats.

Ceci est inclus dans une boucle infinie. L'ensemble peut s'écrire sous la forme du pseudo code suivant :

```
tant_que(vrai) faire
      lecture_donnees()
      simulation()
      sauve_resultats()
fin_tant_que
```

Afin d'accélérer le traitement (utilisation du CPU pour le traitement des données en même temps que se font les entrées/sorties liées aux lectures des données et des sauvegardes des résultats), l'idée est de faire trouver ces trois routines en parallèle (un processus par routine) :

```
// processus p1
tant_que(vrai) faire
      lecture_donnees()
fin_tant_que
```

```
// processus p2
tant_que(vrai) faire
      simulation()
fin_tant_que
```

```
// processus p3
tant_que(vrai) faire
      sauve_resultats()
fin_tant_que
```

L'utilisation d'une zone de mémoire partagée assure la communication entre les trois processus, via une gestion de files d'attente de données (vous n'avez pas à gérer la programmation de ces files d'attente). Il va de soit que ces trois processus doivent être synchronisés : le lancement de la simulation ne peut avoir lieu qu'une fois les données lues, et la sauvegarde du résultat du traitement ne peut se faire que lorsque la simulation est terminée.

Question 1.a) [*1 point*] : Rappelez ce qu'est un sémaphore, et à quoi servent les trois fonctions de manipulation des sémaphores : `Init()`, `P()`, et `V()`.

Question 1.b) [*2,5 points*] : Nous faisons l'hypothèse que la mémoire de l'ordinateur est infinie (ou, de façon plus pragmatique, nous faisons l'hypothèse que même si le processus p1 boucle plus rapidement que p2 ne traite les données lues, et que même si p2 génère ses résultats plus vite que p3 ne les sauve, ça ne suffira pas à saturer la zone de mémoire partagée).

Dans ce cas, modifier le code (uniquement en utilisant les fonctions `Init()`, `P()`, et `V()`) des processus p1, p2, et p3 pour que la contrainte « *le lancement de la simulation ne peut avoir lieu qu'une fois les données lues, et la sauvegarde du résultat du traitement ne peut se faire que lorsque la simulation est terminée* » soit respectée.

Question 1.c) [3,5 *points*] : Même question que précédemment, mais considérant cette fois-ci que la mémoire n'est plus infinie : la file d'attente stockant les données lues par p1 à destination de p2 ne contient au plus que deux éléments, et la file d'attente stockant les résultats générés par p2 afin qu'ils soient sauvés par p3 ne contient elle aussi au plus que deux éléments.

* * * * * *

Exercice n°2 : les signaux *(4 questions pour un total de 7 points)*

Soit le code C suivant :

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

void   sg(int sig)
{
    printf("Signal SIGUSR1 reçu\n");
    signal(sig, sg);
}

void   main()
{
    signal(SIGUSR1, sg);
    while(1);
}
```

Question 2.a) [*1 point*] : Qu'est ce qu'un signal ?

Question 2.b) [*1 point*] : Que fait ce programme ?

Question 2.c) [*2,5 points*] : Si vous aviez à écrire un programme dont le rôle serait d'envoyer un signal SIGUSR1 au programme ci-dessus, quelle serait votre problématique ? Donner au moins deux solutions pour contourner cette problématique.

Question 2.d) [*2,5 points*] : Corrigez le programme ci-dessus pour qu'il n'affiche plus la réception de chaque signal SIGUSR1, mais pour :

- qu'il comptabilise le nombre de fois où SIGUSR1 est reçu,
- et pour que ce compteur soit affiché à chaque réception d'un signal SIGUSR2.

Êtes-vous sûr d'être exhaustif ? Pourquoi, et comment éviter ce problème ?

* * * * * *

Exercice n°3 : ordonnancement *(4 questions pour un total de 6 points)*

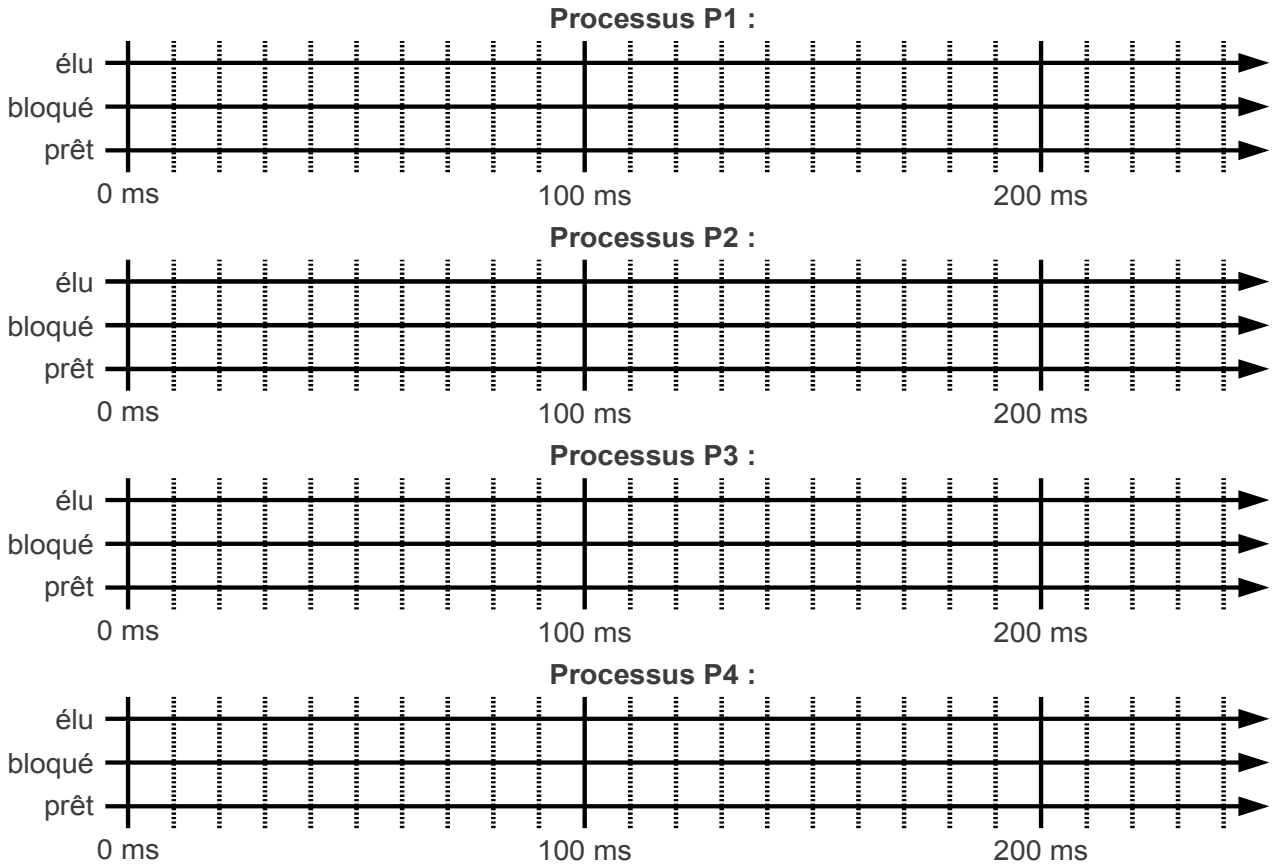
Considérons les quatre processus suivants :

- processus P1 : date de départ = 0 ms., durée = 30 ms., puis appel d'une primitive système bloquante durant 30 ms., puis calcul pendant 40 ms., priorité = 4 ,
- processus P2 : date de départ = 10 ms., durée = 20 ms., priorité = 3 ,
- processus P3 : date de départ = 40 ms., durée = 40 ms., priorité = 2 ,
- processus P4 : date de départ = 100 ms., durée = 40 ms., priorité = 4 .

Question 3.a) [*1 point*] : Dans un système d'exploitation multiprogrammé, qu'est-ce qu'un ordonnanceur ? Quand intervient-t-il ?

Question 3.b) [*1 point*] : Qu'est ce qu'un système d'exploitation « temps-réel » ?

Question 3.c) [*2 points*] : Dessinez le schéma d'ordonnancement (chronogramme) dans le cas où ces processus tournent sur un système d'exploitation **non préemptif** utilisant un algorithme d'ordonnancement par priorité (par convention, le processus ayant le **plus petit** numéro de priorité sera le **plus prioritaire**). Exemple de chronogramme **à recopier** et à remplir **sur votre copie** (ne pas écrire sur le présent sujet) :



Question 3.d) [*2 points*] : Dessinez le schéma d'ordonnancement (même type de chronogramme) dans le cas où ces processus tournent sur un système d'exploitation **préemptif**, avec un ordonnanceur basé sur l'algorithme du tourniquet avec gestion des priorités (un processus n'est élu que lorsqu'il n'existe pas de processus plus prioritaire à l'état « prêt »), et un « tick » d'horloge de 10 ms. (même convention que pour la question 1.c : le processus ayant le **plus petit** numéro de priorité sera le **plus prioritaire**).

Remarque : dans ces chronogrammes, le temps de CPU utilisé par l'ordonnanceur pour calculer ses algorithmes sera considéré comme négligeable.

* * * * * *

Annexe : les appels systèmes utiles (extrais des « man » Linux)

- `int socket(int domain, int type, int protocol)` : crée un point de communication, et renvoie un descripteur (ou –1 si erreur). Ici, **domain** doit être `PF_INET`, le **type** `SOCK_DGRAM` (protocole UDP), le protocole à 0.
- `int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen)` : fournit à la socket **sockfd**, l'adresse **my_addr**, adresse de longueur **addrlen** octets. Retourne 0 si OK, -1 sinon. Avec : struct sockaddr_in {

```
        short sin_family;        /* AF_INET */
        u_short sin_port;        /* Num. de port en endian Internet */
        struct in_addr sin_addr; /* Adresse IP (en endian Internet) */
        char sin_zéro [8];       /* Bourrage */
};
struct in_addr {
      u_long s_addr;
};
```

- `int recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, int *fromlen)` : utilisé pour recevoir des messages depuis un socket. Avec **s** le descripteur de socket, **buf** une zone mémoire de taille minimum **len**, **len** le nombre d'octets demandés, **from** est remplis par l'adresse source, et **fromlen** la taille (en octets) de cette adresse source. Retourne le nombre d'octets reçus, -1 en cas d'erreur.
- `int sendto(int s, const void *buf, int len, int flags, const struct sockaddr *to, int tolen)` : envoie dans le socket **s** les **len** octets pointés par **buf**, à destination de l'adresse **to** de longueur **tolen**. Ici, **flags** vaut 0. Retourne le nombre d'octets écrits, -1 en cas d'erreur.
- `bzero(void *ptr, int n)` : mets à 0 les **n** octets pointés par **ptr**.
- `memcpy(void *dest, void *source, int n)` : copie les **n** octets pointés par **source** vers **dest**.
- `memcmp(void *ad1, void *ad2, int n)` : retourne 0 si les **n** octets pointés par **ad1** sont égaux à ceux pointés par **ad2**, et un nombre non nul sinon.
- `int htons(int n)` : normalise l'entier **n** au format réseau.
- `int htonl(long l)` : normalise l'entier long **l** au format réseau.
- `int ntohs(int n)` : normalise l'entier **n** au format local.
- `int ntohl(long l)` : normalise l'entier long **l** au format local.
- `struct hostent *gethostbyname(const char *name)` : retourne l'adresse IP de la machine de nom **name**, dans une structure `struct hostent {`

```
        char    *h_name;       /* Nom officiel de l'hôte. */
        char   **h_aliases;    /* Liste d'alias. */
        int      h_addrtype;   /* Type d'adresse de l'hôte. */
        int      h_length;     /* Longueur de l'adresse. */
        char   **h_addr_list;  /* Liste d'adresses. */
    }
    #define h_addr  h_addr_list[0] /* pour compatibilité. */
```

- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)` : permet à un programme de surveiller des descripteurs de fichiers contenus dans les ensembles **readfds** (ensemble de descripteurs à surveiller pour voir si des octets sont disponibles en lecture), **writefds** (ensemble de descripteurs à surveiller pour voir si des octets sont disponibles en écriture), **exceptfds** (recherche des conditions exceptionnelles). **nfds** est égal à 1 + numéro du descripteur le plus grand des 3 ensembles **readfds**, **writefds**, et **exceptfds**. **timeout** permet de gérer un timer, et est de type : `struct timeval {`

```
        long    tv_sec;         /* secondes      */
        long    tv_usec;        /* microsecondes */
    };
```

La fonction retourne le nombre de descripteurs ayant une action possible, 0 en cas de timeout. Quatre macros sont disponibles pour la manipulation ensembles : « `void FD_ZERO(fd_set *`**`set`**`);` » efface un ensemble. « `void FD_SET(int` **`fd`**`, fd_set *`**`set`**`);` » et « `void FD_CLR(int` **`fd`**`, fd_set *`**`set`**`);` » ajoutent et suppriment un descripteur dans un ensemble. « `int FD_ISSET(int` **`fd`**`, fd_set *`**`set`**`);` » vérifie si un descripteur est contenu dans un ensemble.

- `int pipe(int` **`filedes`**`[2])` : crée une paire de descripteurs de tube, et les place dans un tableau **`filedes`** (**`filedes`****[0]** est utilisé pour la lecture, et **`filedes`****[1]** pour l'écriture). Retourne –1 si erreur, 0 sinon.
- `int read(int` **`fd`**`, void *`**`buf`**`, size_t` **`count`**`)` : lit count octets depuis le fichier **`fd`** et place les octets lus dans **`buf`**. Retourne le nombre d'octets lus, -1 sinon.
- `int write(int` **`fd`**`, const void *`**`buf`**`, size_t` **`count`**`)` : écrit dans le fichier **`fd`** **`count`** octets pointés par **`buf`**. Retourne le nombre d'octets écrits, -1 sinon.
- `close(int` **`fd`**`)` : ferme proprement le descripteur de fichier **`fd`**.
- `void (*signal(int` **`signum`**`, void (*`**`handler`**`)(int)))(int)` : positionne la fonction **`handler`** comme fonction appelée en cas de réception du signal **`signum`**.
- `int fork()` : crée un processus fils en copiant le processus appelant. Retourne une valeur négative en cas d'erreur, 0 au fils, et le PID du processus nouvellement créé au père.
- `int wait(int *`**`status`**`)` : attend la fin d'un enfant. Si **`status`** est non NULL, met dans ***`status`** le code retourné par cet enfant. Retourne un nombre négatif en cas d'erreur, ou le PID du fils qui vient de se terminer. La lecture du statut supprime l'entrée du processus fils de la talbe des processus du système.
- `int pause()` : blocage d'un processus (sans consommation de CPU) dans l'attente d'un signal. Cette fonction renvoie toujours une valeur -1.
- `int kill(pid_t` **`pid`**`, int` **`sig`**`)` : envoie le signal **`sig`** au processus numéro **`pid`**. Retourne un noumbre négatif en cas d'erreur.
- `int getpid()` : permet à un processus de connaître sont propre PID. Retourne un nombre négatif en cas d'erreur.
- `int getppid()` : permet à un processus de connaître le PID de son père. Retourne un nombre négatif en cas d'erreur.
- `int msgget(key_t` **`key`**`, int` **`msgflg`**`)` : renvoie l'identifiant de la file de messages associée à la clé **`key`**. Dans les problèmes énnoncés ici, **`msgflg`** vaudra : **`IPC_CREAT|S_IRUSR|S_IWUSR`**
- `msgctl(`**`msqid`**`, IPC_RMID, NULL)` **:** permet de détruire la file de message d'identifiant **`msqid`**.
- `int msgsnd(int` **`msqid`**`, const void *`**`msgp`**`, size_t` **`msgsz`**`, int` **`msgflg`**`)` : envoie sur la file ayant comme identifiant **`msqid`** un message pointé par **`msgp`**. Ce message doit avoir la forme d'une structure du type :

```
struct msgbuf {
    long        mtype;     /* type de message ( > 0 ) */
    mon_message message;   /* contenu du message */
};
```

`msgsz` est la taille du type "`mon_message`", et **`msgflg`** vaudra 0 dans notre cas. La fonction retourne 0 si tout va bien, -1 sinon.

- `int msgrcv(int` **`msqid`**`, const void *`**`msgp`**`, size_t` **`msgsz`**`, long` **`msgtyp`**`, int` **`msgflg`**`)` : reste bloqué sur la file ayant comme identifiant **`msqid`** jusqu'à l'arrivée d'un message pointé par **`msgp`**. Ce message doit avoir la forme d'une structure du type :

```
struct msgbuf {
    long        mtype;      /* type de message ( > 0 ) */
    mon_message message;    /* contenu du message */
};
```

  `msgsz` est la taille du type "`mon_message`", **`msgflg`** vaudra 0 dans notre cas. Si l'argument **`msgtyp`** vaut 0, le premier message est lu (quel que soit son type). Si **`msgtyp`** est supérieur à 0, alors seul le premier message de type **`msgtyp`** est extrait de la file. La fonction retourne -1 en cas d'erreur, ou le nombre d'octets écrits dans le champ **`"msgp.message".`**
- `int semget(key_t` **`key`**`, int` **`nsems`**`, int` **`semflg`**`)` : Cette fonction retourne l'identifiant de l'ensemble de sémaphores associé à la valeur de clé **`key`**. La valeur de **`nsems`** indique le nombre de sémaphores que doit contenir cet ensemble. Dans les problèmes énnoncés ici, sem**`flg`** vaudra : **`IPC_CREAT|S_IRUSR|S_IWUSR`**
- `semctl(int` **`sem_id`**`, int` **`sem_num`**`,` **`SETVAL`**`, int` **`v0`**`)` : initialise le sémaphore numéro **`sem_num`** de l'ensemble **`sem_id`** à la valeur **`v0`**.
- `int semop(int` **`semid`**`, struct sembuf *`**`sops`**`, unsigned` **`nsops`**`)` : fonction qui effectue des opérations sur les membres de l'ensemble de sémaphores identifié par **semid**. Chacun des **nsops** éléments dans le tableau pointé par **sops** indique une opération à effectuer sur un sémaphore en utilisant une structure **`struct sembuf`** contenant les membres suivants :

```
unsigned short sem_num; /* Numéro du sémaphore    */
short     sem_op; /* Opération sur le sémaphore */
short     sem_flg; /* Options pour l'opération  */
```

  Les options possibles pour **`sem_flg`** sont **`IPC_NOWAIT`** (ne sera pas utile ici) et **`SEM_UNDO`**. Si une opération indique l'option **`SEM_UNDO`**, elle sera annulée lorsque le processus se terminera. L'ensemble des opérations contenues dans le tableau **`sops`** est effectué atomiquement. Les opérations sont toutes réalisées en même temps, et seulement si elle peuvent toutes être effectuées.
  Chaque opération est effectuée sur le **`sem_num`**-ième sémaphore de l'ensemble. Le premier sémaphore est le numéro 0. Pour chaque sémaphore, l'opération est l'une des 3 décrites ci-dessous :
  - Si l'argument **`sem_op`** est un entier positif, la fonction ajoute cette valeur au **`sem_num`**-ième sémaphore. Cette opération n'est jamais bloquante.
  - Si **`sem_op`** vaut 0, le processus attend que semval soit nul ; si semval vaut zéro, l'appel système continue immédiatement.
  - Si **`sem_op`** est inférieur à 0, si **`semval`** est supérieur ou égal à la valeur absolue de **`sem_op`**, la valeur absolue de **`sem_op`** est soustraite du **`sem_num`**-ième sémaphore. Sinon, l'appel est blocant.

  En cas de réussite, `semop()` renvoit 0, et -1 sinon.
- `semctl(`**`sem_id`**`, 0,` **`IPC_RMID`**`, 0)` : permet de détruire l'ensemble de sémaphores identifié par **`sem_id`**.
- `int pthread_create(pthread_t *`***`thread`***`, pthread_attr_t *`***`attr`***`, void *(*`***`start_routine`***`)(void *), void *`***`arg`***`)` : permet la création d'un nouveau thread, identifié par l'identificateur `*thread`, et attaché à l'exécution de la routine (`*start_routine`). Le fil d'exécution démarre son exécution au début de la fonction `start_routine` spécifiée et disparaît à Ia fin de l'exécution de celle-ci. L'argument `*attr` sera mis à `NULL` dans les exercices ici. Enfin, `*arg` correspond à un argument passé au *thread* pour l'exécution de la fonction (`*start_routine`) (peut valoir `NULL` si aucun argument est passé en paramètre). La primitive renvoie la valeur 0 en cas de succès, et une valeur négative sinon. Pour utiliser les thread, il conviendra de placer un « `#include <pthread.h>` » au début du programme.

- `pthread_exit(void *ret)` met fin au *thread* qui l'exécute, en retournant la valeur `*ret`.
- `pthread_join(pthread_t` ***thread***`, void **`***retour***`)` : permet à un thread d'obtenir la valeur de retour envoyée par un autre thread via la fonction « `pthread_exit()` ». Le paramètre ***thread*** correspond à l'identifiant du *thread* attendu, et ****retour*** cotiendra la valeur retournée par l'autre thread. Important : l'exécution du processus qui effectue un appel à la fonction « `pthread_join()` » est suspendue jusqu'à ce que le *thread* attendu se soit achevé.
- Pour créer un mutex, il suffit de créer une variable de type `pthread_mutex_t`, et de l'initialiser avec la constante `PTHREAD_MUTEX_INITIALIZER`. Exemple :
```
pthread_mutex_t  pMutex = PTHREAD_MUTEX_INITIALIZER;
```
- `pthread_mutex_destroy(pthread_mutex_t` ****pMutex***`)` : permet de détruire proprement le mutex ***pMutex***.
- `int pthread_mutex_lock(pthread_mutex_t *`***pMutex***`)` : verrouille le mutex ***pMutex***. Retourne 0 en cas de succès.
- `int pthread_mutex_unlock(pthread_mutex_t` ****pMutex***`)` : déverrouille le mutex ***pMutex***. Retourne 0 en cas de succès.
- Pour créer une variable condition, il suffit de créer une variable de type `pthread_cond_t` et de l'initialiser avec la constante `PTHREAD_COND_INITIALIZER`. Exemple :
```
pthread_cond_t   condition = PTHREAD_COND_INITIALIZER;
```
- `pthread_cond_wait(pthread_cond_t` ****condition***`, pthread_mutex_t` ****mutex***`)` : permet à un thread de se mettre en attente sur la variable condition ***condition***. Le mutex ***mutex*** devra être verrouillé avant l'appel de cette fonction, et libéré juste après.
- `pthread_cond_signal(pthread_cond_t` ****condition***`)` : signale que la condition ***condition*** est remplie. L'appel à cette fonction doit être protégé par le même mutex que celui protégeant l'appel de la fonction `pthread_cond_wait()`.
- `pthread_cond_destroy(pthread_cond_t` ****condition***`)` : détruit la variable condition ***condition***.