CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 - NANCY/METZ

EXAMEN – 28 JUIN 2011

Année 2010 – 2011, deuxième semestre
Coefficient : 2/3

Documents autorisés : NON

CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

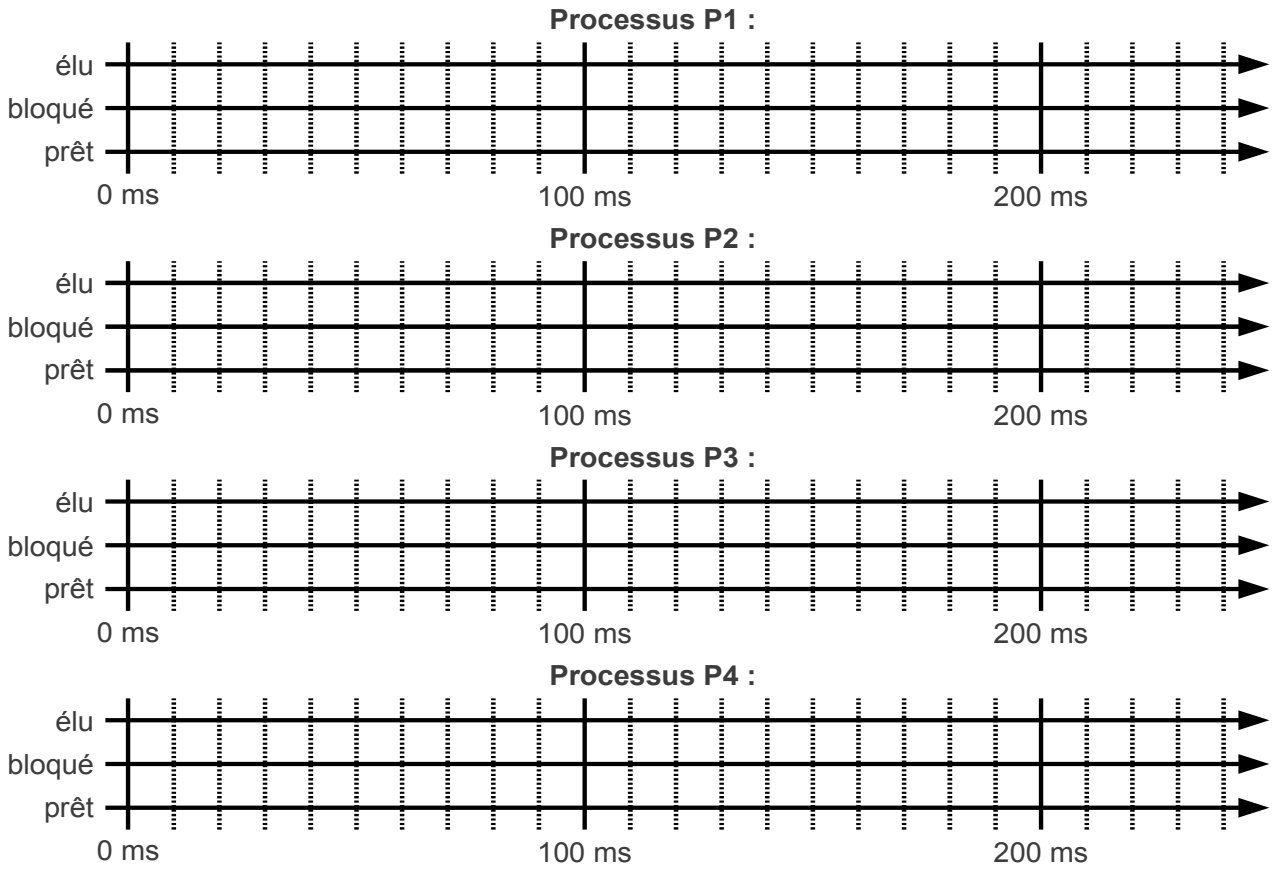
Exercice n°1 : ordonnancement *(3 questions pour un total de 5 points)*

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

Question 1.b) [*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 1.c) [*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.b : 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.

* * * * * *

Exercice n°2 : contrôle d'un capteur de pression d'une chaîne de montage *(4 questions pour un total de 8 pts)*

Une chaîne de montage utilise de l'air comprimé pour fonctionner. Un capteur est utilisé pour mesurer la pression fournie par une pompe. Lors d'une utilisation normale, l'ensemble des circuits fonctionne avec une pression inférieure à 15 bar. Néanmoins, il arrive par accident que la pression dépasse ces 15 bar. Un dispositif de contrôle arrête alors la chaîne de montage, qui se met alors en alarme. Pour des raisons de sécurité, il est obligatoire que tout capteur ayant subi trois fois une pression supérieure à 15 bar soit changé. Néanmoins, lors d'une révision de la machine, si le technicien effectue un nettoyage du capteur, et s'il constate que celui-ci ne présente pas de défaut, il peut considérer que l'élément de mesure de pression pourra subir une fois de plus une pression supérieure à 15 bar sans être changé.

Une équipe de programmeurs a pour mission de développer le mécanisme de surveillance de la chaîne de montage sur un ordinateur sous Unix/Linux. Lorsqu'un capteur neuf est installé sur la chaîne de montage, celui-ci a un potentiel de 3 mises en défaut de chaîne avant son remplacement. Dès qu'un incident fait atteindre une pression de 15 bar, l'alerte est activée, et ce « nombre de détection de surpression avant changement » diminue de 1. Inversement, lors d'une révision ne laissant pas apparaître de défaut, ce nombre augmente de 1.

Pour ce, les programmeurs décident que le programme principal doit créer un processus fils, dont le rôle sera de toujours connaître la pression relevée par le capteur.

Dès que ce processus fils détecte une pression supérieure à 15 bar, il envoie un signal « `SIG_USR1` » au processus père (qui est le programme de contrôle principal). A chaque réception de ce signal, le processus père exécutera une fonction `detection_surpression()`, dont le rôle sera :

- d'appeler une fonction `declenche_alerte()` ce qui aura pour effet d'arrêter la chaîne de production ;
- puis d'appeler une fonction `change_nb_utilisation_capteur(-1)`, ce qui aura comme effet de décrémenter une variable « `nb_utilisation_capteur` », qui est une variable entière globale du programme principal, qui contient le nombre de fois où le capteur de pression peut atteindre 15 bar avant d'être remplacé. La valeur retournée par cette fonction « `change_nb_utilisation_capteur()` » est le nombre de fois où le capteur peut déclencher une alerte avant son remplacement. Si ce nombre est égal à zéro, il faut alors appeler la fonction `remplace_capteur()` (qui informe les techniciens que le capteur doit être changé). Sinon, une fois la panne réparée, la chaîne est remise en production, jusqu'à ce que le signal `SIG_USR1` soit reçu à nouveau.

Lors d'une révision de maintenance, si le technicien nettoie le capteur et considère que celui-ci est en bon état, il appuie sur un bouton qui enverra cette fois-ci un signal `SIG_USR2` au processus père. A la réception du signal `SIG_USR2`, celui-ci exécutera une fonction `maintenance_ok()`, qui réalise les actions suivantes :

- appel d'une fonction `acquitte_maintenance()`, dont le rôle est de faire clignoter un voyant quelques secondes sur le capteur, pour que le technicien sache que l'appui sur le bouton a bien été pris en compte,
- et la fonction `change_nb_utilisation_capteur()` est appelée avec la valeur « `1` » comme paramètre, afin d'incrémenter la variable globale `nb_utilisation_capteur`.

Sans surprise, pour armer ce mécanisme, nous trouvons dans les première lignes de la fonction `main()` du programme principal les deux lignes suivante :

```
signal(SIG_USR1, detection_surpression);
signal(SIG_USR2, maintenance_ok );
```

Question 2.a) [*1 point*] : Écrire la fonction « `detection_surpression()` ».

Question 2.b) [*2 points*] : En supposant que le technicien double clique très rapidement par accident sur le bouton censé indiquer le nettoyage d'un capteur sans défaut, le programme de contrôle réagira-t-il toujours de la même façon (justifiez votre réponse) ?

Question 2.c) [*1 point*] : Dans le programme principal, une multitude de fonctions exécutées dans différents threads ont besoin de savoir régulièrement combien la durée de vie potentielle de chaque capteur (pour l'affichage, pour l'intelligence artificielle qui évalue quand doit être effectuée la prochaine maintenance, etc). Ces routines font alors appel à une fonction « `lit_nb_utilisation_capteur()` », qui retourne le nombre de fois où le capteur de pression pourra subir 15 bar avant d'être remplacé.

Quel est le risque lié aux fonctions `change_nb_utilisation_capteur()` et `lit_nb_utilisation_capteur()` ? Indiquez le nom de la solution la plus efficace (vue en cours) pour pallier à ce risque.

Question 2.d) [*4 points*] : Programmez (en C ou en pseudo-code faisant appel aux appels système Linux) ces deux fonctions `change_nb_utilisation_capteur()` et `lit_nb_utilisation_capteur()` mettant en œuvre la solution proposée dans la question précédente. **Remarque** : pour simplifier, le candidat n'a pas à traiter les codes retour (indiquant des erreurs) des appels système.

* * * * * *

Exercice n°3 : les threads *(3 questions pour un total de 7 points)*

Question 3.a) [*3 points*] : Q'est-ce qu'un processus ? Faites un schéma qui illustre les trois principaux états d'un processus, et les raisons qui font passer un processus d'un état à un autre.

Qu'est-ce qu'un thread ? Quel est l'avantage d'un thread par rapport à un processus ?

Question 3.b) [*1 point*] : Quels sont les deux types de thread rencontrés dans les systèmes d'exploitation multiprogrammés ? Pour chacun d'eux, précisez les avantages/inconvénients.

Question 3.c) [*3 points*] : Écrivez (en C ou en pseudo code) le programme suivant :

Un processus initialement composé d'un seul thread (qu'on appellera par la suite « *premier thread* ») crée un second thread, puis il appelle une fonction externe `gros_calcul()`. De son coté, le second thread nouvellement créé va appeler une fonction externe `petit_calcul()`, avant de se mettre en attente d'une variable condition. Lorsque la fonction `gros_calcul()` se termine, le premier thread va signaler au second thread que la condition attendue est réalisée. Il attend alors la fin du second thread (à l'aide de la primitive `pthread_joint()`), avant de se terminer proprement.

Le candidat ne s'occupera pas des fonctions `petit_calcul()` et `gros_calcul()`, qui sont supposées existantes, et déclarées ainsi :

```
extern void      petit_calcul(), gros_calcul();
```

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***.