Programmation temps-réel : threads POSIX, mutexes et sémaphores

Aline Huf. <alinehuf@ai.univ-paris8.fr>

2016-2017

Table des matières

1 Processus légers : les threads POSIX

Parenthèse : c'est quoi POSIX ?
"POSIX est une famille de normes techniques définie depuis 1988 par l'Institute of Electrical and Electronics Engineers (IEEE), et formellement désignée par IEEE 1003. Ces normes ont émergé d'un projet de standardisation des interfaces de programmation des logiciels (API pour Application Programming Interface) destinés à fonctionner sur les variantes du système d'exploitation UNIX." (wikipédia)
1960-1965 : la notion de processus léger et de thread apparait

1995 : création d'un standard d'interface pour l'utilisation des processus légers
- pthread (ou POSIX thread).
- standard POSIX 1003.1c-1995 défini par l'IEEE PASC
  (Institute of Electrical and Electronics Engineers Portable Application Standars Commitee)

Avertissement de POSIX : ne pas combiner thread et fork
Une explication simple des problèmes possibles :
http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them

```
$ man pthreads
```

1.1 Créer un thread

```
int pthread_create(pthread_t* thread, const pthread_attr_t* attr,
                   void* (*start_routine)(void*), void* arg);
```

- thread : identifiant préalablement alloué (`pthread_t`, structure obscure variable selon l'implémentation).
- attr : attributs du thread créé
- start routine : fonction définissant la séquence d'instructions associée
- arg : arguments de start routine

thread_create.c

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

// $ gcc -Wall thread_create.c -lpthread
// Discussion : Processus et thread
// Discussion : thread principal et thread secondaire
// ps maux     (dans une autre console)
// Ctrl+C pour arrêter le processus en force

void* f(void* p) {
  while(1) {
    puts("thread secondaire");
    sleep(1);
  }
  return NULL;
}

int main() {
    pthread_t thr;
    if (pthread_create(&thr, NULL, f, 0) != 0) {
      fprintf(stderr, "Erreur dans pthread_create\n");
      exit(EXIT_FAILURE);
    }
    while (1) {
      puts("thread PRINCIPAL = thread main");
      sleep(1);
```

```
    }
    return 0;
}
```

ps aux : on ne voit que le processus lourd
ps maux : on voit le processus et le détail des threads (thread principal + threads secondaires)
ps -T maux : une colonne SPID affiche l'identifiant de thread Remarquez les statuts des threads (Sl = sleeping, R = Running, ...) qui sont indépendants.

PID indentique dans tous les threads (celui du processus). L'identifiant : `pthread_t` n'a pas de signification en dehors de l'espace d'adressage du processus.

Deux threads qui travaillent en parallèle et pas au même "rythme" : threads asynchrones (exemple de calcul synchrone : GPU).

1.2 Identifiant d'un thread

```
pthread_t pthread_self(void);
```

thread_ids.c

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

// Discussion : thread principal et thread secondaire
// Discussion : comprendre les ID

void* f(void* p) {
    printf("thread %d, my PID is %d\n", (int) pthread_self(), (int) getpid());
    // note : pthread_self() et  getpid() ne sont pas des int
    // utiliser pthread_equal() pour faire une comparaison entre des pthread_t

    return NULL;
}

int main() {
    printf("thread PRINCIPAL %d, my PID is %d\n",
           (int) pthread_self(), (int) getpid());
    f(0);

    pthread_t id[10];
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    int i = 0;
    for(i = 0; i < 10; i++) {
        if (pthread_create(&id[i], &attr, f, 0) != 0) {
            fprintf(stderr, "Erreur dans pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
    return 0;
}
```

1.3 Attributs d'un thread

```
typedef struct {
```

```
        int                 flags;        // detached ou joinable
        size_t              stacksize;    // taille de la pile
        void                *stackaddr;   // adresse de la pile
        void                (*exitfunc)(void* status);
        int                 policy;       // ordonnancement
        struct sched_param  param;
        unsigned            guardsize;
} pthread_attr_t;
```

- flags : booléen indiquant exécution detached ou joinable
  (i.e. exécution respectivement sans ou avec rendez-vous)
- exitfunc : fonction exécutée à la fin de la séquence d'instructions
- policy : temps réel SCHED_FIFO, SCHED_RR, pas temps réel SCHED_BATCH, ... SCHED_OTHER

Pour modifier les attributs du thread :

```
pthread_attr_t attr;

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

pthread_attr_getX   // getter pour obtenir la valeur de l'attribut X
pthread_attr_setX   // setter pour modifier la valeur de l'attribut X

// exemple :
pthread_attr_getdetachstate(const pthread_attr_t * attr, int * valeur);
pthread_attr_setdetachstate(const pthread_attr_t * attr, int * valeur);
```

`pthread_attr_t` n'est plus utile après la création du thread. La structure peut-être réutilisée ou bien un appel à `pthread_attr_destroy` permet de libérer les données dynamiques internes.

1.4 Possibilités d'ordonnancement

ordonnancement disponibles si `_POSIX_THREAD_PRIORITY_SCHEDULING` est définie dans `<unistd.h>`

support_thread_posix.c

```
/* support_thread_posix.c
 * programme testant les possibilites d'ordonnancement des threads posix
 * n 2006 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

int main(void) {
    printf("_POSIX_VERSION=%ld\n", _POSIX_VERSION);
#if _POSIX_VERSION < 199506L
    printf("sans thread posix\n");
#else
    printf("avec thread posix\n");
#ifdef _POSIX_PTHREAD_PRIORITY_SCHEDULING
    printf("avec ordonnancement des threads\n");
#else
    printf("sans ordonnancement des threads\n");
#endif /* _POSIX_PTHREAD_PRIORITY_SCHEDULING */
#endif /* _POSIX_VERSION < 199506L */
    return 0;
}
```

1.5 Thread détaché ou "joinable"

```
int pthread_detach(pthread_t thread);                    // thread_detach.c
int pthread_join(pthread_t thread, void** value_ptr);    // thread_joinable.c
```

Thread joinable par défaut : quand un thread se termine le code de retour est conservé, c.à.d. la pile du thread est conservée.
La mémoire utilisée par la pile n'est libérée qu'après l'appel à `pthread_join` qui permet de récupérer la valeur retournée.
Les piles sont dans l'espace d'adressage du processus, sans `pthread_join`, la mémoire est occupée par une pile devenue inutile...

Pour éviter de devoir attendre un thread on peut le "détacher".

thread_joinable_mem.c

```
/* creation d'un pthread attendu par defaut
 * le main attend la fin du pthread
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// discussion : libération de la mémoire

void* funcPth0(void* arg) {
    return NULL; // retourne de suite, la pile est conservée
}

int main(void) {
    pthread_t pth; // même variable écrasée à chaque fois
    int nb_start = 0;

    while (pthread_create(&pth, NULL, &funcPth0, NULL) == 0) {
        pthread_detach(pth); // pour que la pile soit libérée à la fin du thread
        nb_start++;
        if (nb_start % 5000 == 0) printf("%d\n", nb_start);
    }
    fprintf(stderr, "Echec de création après %d threads lancés\n", nb_start);

    // fin du processus : return met fin aux threads qui pourraient encore
    // être "vivants"=> libération de la mémoire
    return EXIT_SUCCESS;
}
```

Problème : si le processus parent termine avec `return` , il termine tous les threads avec lui...

thread_detach.c

```
/* creation d'un pthread detache
 * la fin du main met fin a tous les threads
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void* funcPth0(void* _av) {
    pthread_t id = pthread_self();
    int i;
```

```
    for (i = 0; i < 10; i++) {
        printf("thread %d : %d\n", (int) id, i);
        sleep(1);
    }
    printf("thread %d : Je termine\n", (int) id);
    // PTHREAD_CREATE_DETACHED :
    // les ressources qui étaient allouées sont automatiquement libérées
    return 0;
}

int main(void) {
    pthread_t pth;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    pthread_create(&pth, &attr, &funcPth0, NULL);

    pthread_attr_destroy(&attr); // attr inutile une fois le thread créé

    sleep(3);  // travail quelconque

    pthread_t id = pthread_self();
    printf("thread %d : je retourne\n", (int) id);

    //return 0; // le processus retourne et ses threads sont détruits
    pthread_exit(0); // le processus retourne mais ses threads restent en vie
}
```

thread_joinable.c

```
/* creation d'un pthread attendu
 * le main attend la fin du pthread
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/* thread recevant une valeur v
 * et retournant v+1 apres 3 secondes */
void* funcPth0(void* arg) {
    int v = *(int*)arg;
    pthread_t id = pthread_self();
    printf("thread %d\n", (int) id);
    v++;

    sleep(3);

    return (*(void**)&v); // convertion de v en void*
                          // (4 octets pour l'int, 4 octets indéterminés...)
}

int main(void) {
    int v = 3; /* valeur envoyee */
    void* ret; /* valeur recue */

    pthread_t pth;
    // autre manière de définir un thread détaché ou joinable :
```

```
    // le préciser dans ses attirbuts
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    // PTHREAD_CREATE_JOINABLE est la valeur par défaut
    // (pour un thread détaché : PTHREAD_CREATE_DETACHED)
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    pthread_create(&pth, &attr, &funcPth0, &v);

    printf("J'attend le thread\n");

    pthread_join(pth, &ret); // ret contient l'entier dans les 4 premiers octets
    // je récupère le retour du thread et seulement maintenant
    // les ressources allouées au thread sont libérées
    printf("envoyee:%d recue:%d\n", v, *(int*)&ret);

    return 0;
}
```

1.6 Transmettre, récupérer des données

Transmission de paramètres, récupération de valeur retournée. Plusieurs approches :

- Si on retourne une valeur de taille inférieure ou égale à un pointeur
  - ➤ On peut faire un cast sauvage pour faire passer cette valeur pour un `void *`
  - ➤ Dans le cas d'une valeur de taille inférieure : les octets restants contiennent des données indéterminées.
  - ➤ Attention, la variable utilisée pour récupérer la valeur dans le main() doit être assez grande pour contenir le void* retourné.
- Si on retourne une structure de donnée (tableau ou struct)
  - ➤ On peut allouer un espace mémoire pour stocker la structure de donnée et retourner un pointeur qui pointe dessus. Attention, il faut libérer la mémoire dans le main().
  - ➤ On peut aussi utiliser une structure définie globalement (un tableau contenant les données de tous les threads) et retourner juste un pointeur sur la partie du tableau concernée.
  - ➤ Note : on peut se passer de retourner une valeur si on utilise une structure de donnée définie globalement ou si on écrit le résultat dans la srtucture passée en entrée au thread, mais il faut quand même utiliser `pthread_join()` pour s'assurer que le thread a fini son travail avant de consulter le résultat.

thread_param_returnval1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Discussion : comprendre paramètres et retour

void* f(void* p) {
    int a = *(int*) p;
    int b = *(int*) (p + sizeof(int));
    printf("thread %d %d %d\n", (int)pthread_self(), a, b);
    a += b;
    return *((void**) &a);
}

int main() {
    pthread_t id;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    int v[2];
    v[0] = 1;
```

```
    v[1] = 2;
    pthread_create(&id, &attr, f, v);
    int ret;
    pthread_join(id, (void**) &ret); // ATTENTION !!!
                                     // écriture d'un pointeur dans l'espace d'un int...
    printf("ret %d\n", ret);

    //printf("sizeof(int)=%lu  sizeof(void*)=%lu\n", sizeof(int), sizeof(void*));
    return 0;
}
```

thread_param_returnval2.c

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

// Discussion : comprendre paramètres et retour

void* f(void* p) {
  int a = *(int*) p;
  int b = *(int*) (p + sizeof(int));
  printf("thread %d %d %d\n", (int)pthread_self(), a, b);

  int * t = malloc(2 * sizeof(int)); // j'alloue de la mémoire
  t[0] = 31 + a;
  t[1] = 32 + b;
  return (void*) t; // je retourne un pointeur sur la mémoire allouée
}

int main() {
  pthread_t id;
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  int v[2];
  v[0] = 1;
  v[1] = 2;
  pthread_create(&id, &attr, f, v);
  int * ret;
  pthread_join(id, (void**) &ret);
  printf("ret %d %d\n", ret[0], ret[1]);
  free(ret); // je libère la mémoire allouée dans le thread
  return 0;
}
```

1.7 Autre fonctions pour contrôler les threads

```
int pthread_equal(pthread_t t1, pthread_t t2);
int pthread_abort(pthread_t thread);
int pthread_kill(pthread_t thread, int sig);
void pthread_exit(void* value_ptr);
int pthread_cancel(pthread_t thread)
```

equal : pour comparer l'identifiant d'un thread avec un autre.

- exemple : un thread itère sur les identifiants pour effectuer une communication avec chaque autre thread et doit sauter l'identifiant qui correspond à lui-même.

abort : a ne jamais utiliser. Sauf dans les systèmes temps réel quand une erreur est détectée.

- exemple : plus d'information en provenance des capteurs, le thread semble planté.

- on tente cancel pour arrêter le thread, si rien ne se passe on utilise abort :
  - pas de nettoyage de la mémoire, pas de libération des verrous
  - le système de gestion d'erreur doit assurer le nettoyage, vérifier que les structures de données ne sont pas corrompues et relancer un nouveau thread

kill : permet d'envoyer un signal (interruption) à un thread donné.
exit : utilisé à la place de return à la fin d'un thread

- utile uniquement si on a défini des fonctions de nettoyage en cas d'arrêt brutal du thread

➥ permet de les lancer avant l'arrêt

Cancel : si une fonction de nettoyage est définie elle est exécutée avant de terminer le thread

thread_cancel.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/* thread affichant son tid toute les secondes */
void* funcPth0(void* p) {
    while(1) {
        printf("thread %d\n", (int) pthread_self());
        sleep(1);
    }
    return NULL;
}

int main(void) {
    pthread_t pth;
    pthread_create(&pth, NULL, funcPth0, NULL);

    printf("Attend 3 secondes et stoppe le thread\n");
    sleep(3);
    pthread_cancel(pth); // attention le thread est stoppé brutalement

    sleep(5); // on attend un peu pour vérifier que le thread n'affiche plus rien
    puts("Exit");
    return 0;
}
```

En cas de section critique, il faut désactiver la prise en compte de `cancel()` :

```c
pthread_setcanceltype(PTHREAD_CANCEL_ENABLE, NULL); // le thread est "cancelable"
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL); // peut être annulé à tout moment
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);  // par defaut, attend un "cancellation point",
               // voir "man pthreads" pour la liste des fonctions qui sont des "cancellation point"

int old_cancel_state;
/* Début de la section critique. */
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state);
/* Fin de la section critique. */
pthread_setcancelstate(old_cancel_state, NULL);
```

Autres fonctions utiles pour rendre privées des données globales du processus `pthread_key_create`, `pthread_key_delete`, `pthread_setspecific` et `pthread_getspecific`.

2 Partage des ressources et accès concurrents

Discussion : problème des écritures concurrentes, illusion que ça marche sans temps d'attente... Si on ajoute un délai simulant un traitement long, on peut voir le problème d'écriture concurrente.

ecritures-concurentes_probleme-visible.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h> // struct timespec

// Discussion : ecritures concurentes
// $ gcc -Wall ecritures-concurentes.c -lpthread

void traitementLong() {
    struct timespec ts;
    ts.tv_sec = 0;
    ts.tv_nsec = 10;
    nanosleep(&ts, NULL); // on simule un traitement long
    // note : usleep behavior inconsistent when it is interrupted
    //        by a signal among historical systems
    //        nanosleep returns immediately when any signal handler is executed
}

int VALEUR;

void* f0(void* _p) {
    int i;
    for(i = 0; i < 1000; i++) {
        int x = VALEUR;
        traitementLong(); // VALEUR est modifiée par l'autre thread pendant ce temps...
        VALEUR = x + 1;
    }
    puts("f0_fini");
    return NULL;
}

void* f1(void* _p) {
    int i;
    for(i = 0; i < 1000; i++) {
        int x = VALEUR;
        traitementLong();
        VALEUR = x + 2;
    }
    puts("f1_fini");
    return NULL;
}

int main() {
    pthread_t pid[2];

    pthread_create(&pid[0], NULL, f0, NULL); // créer un thread
    pthread_create(&pid[1], NULL, f1, NULL); // pid, attributs, fonction, argument

    pthread_join(pid[0], 0); // attendre la fin d'un thread
    pthread_join(pid[1], 0);

    printf("VALEUR=%d\n", VALEUR);

    return 0;
}
```

Tâche 1	Tâche 2
`read(A)`	`read(A)`
`A=A+10`	`A=A+20`
`write(A)`	`write(A)`

Exemple 1 : Écritures dans une même variables
=> contenu de A incertain
i = i + 1 ;

- lire i dans un registre
- incrémenter le registre
- écrire le registre dans i

➥ risque d'être interrompu entre 2 instructions

Tâche 1	UART	Tâche 2
Test (Tx == libre)		
	← return OK	
		Test (Tx == libre)
	return OK →	
write(Tx=out char1)		write(Tx=out char2)
	erreur = débordement	

Exemple 2 : Utilisation simultanée d'un UART par 2 tâches
=> émission de deux caractères par deux processus : plantage matériel

2.1 Séquence atomique

Suite d'opérations exécutées par un système qui permet de passer d'un état cohérent du système à un autre état cohérent sans possibilité d'interruption durant la séquence. Une interruption pourrait laisser le système dans un état incohérent.
Les problèmes d'atomicité concernent toutes les ressources du système :

- les zones mémoire ;
- les registres de périphériques ;
- les variables ;
- les groupes de variables ;
- les portions de code effectuant des accès non contrôlés par le système d'exploitation à une ressource effective.

C : `sig_atomic_t` est un type d'entier atomique
C++ : template `atomic` permet de rendre atomique une variable.

2.2 Code (non) réentrant / (non) thread-safe

Un code, qui accède de manière incontrôlée à des variables globales ou aux registres d'un périphérique, est dit non réentrant. Un tel code doit être considéré comme une ressource en soi.
relation entre réentrant et thread-safe :

- thread-safe : la manière dont les fonctions gèrent les ressources

➥ système de protection pour éviter d'accéder à la même ressource en même temps

- réentrant : la manière dont on fourni les données à la fonction

➥ passage de variables pour fournir les données à la fonction (plutôt que d'utiliser des globales)

2.3 Section critique

Section critique (= région critique) : séquence d'opérations qu'il faut effectuer de façon atomique
sections critiques utilisant une ressource commune :

- doivent être exécutées en exclusion mutuelle
- les séquences d'opérations ne sont pas entrelacées

goulot d'étranglement :

- exclusion mutuelle = les autres tâches ne peuvent pas accéder à la ressource
- risque d'un bouchon/file d'attente

Attentes interdites en section critique :
- on bloque toute autre tâche et on ne fait rien => blocage du système

2.4 Protection contre les accès concurrents : Mutex

- masquage matériel : on bloque tout pour finir
- masquage logiciel : on laisse les interruptions s'exécuter, mais pas les autres tâches
- Interdire l'exécution d'une tâche précise désignée : sémaphore

Sémaphore binaire d'exclusion mutuelle. Prévu pour protéger l'accès à une ressource ou/et délimiter une zone critique. Conçu pour être verrouillé puis libéré par la même tâche.
Contrairement aux sections critiques, il est autorisé d'avoir un appel bloquant de l'OS dans une section protégée par un mutex : risqué.

Tâche 1 : T_1	**Tâche 2**
`P(mutex);`	`P(mutex);`
`wait(event);`	`signal(`T_1`, event);`
`V(mutex);`	`V(mutex);`

=> si T1 s'exécute avant T2 : blocage !
=> deux processus s'attendent mutuellement : interblocage, deadlock, étreinte fatale.

Exemple d'exclusion mutuelle à bas ou haut niveau :
- accès à l'UART : garantir l'émission d'un seul caractère à la fois
- au niveau logiciel : garantir l'émission d'une phrase complète, éviter le mélange des caractères.

2.5 Manipuler un Mutex POSIX en C

Pour créer / détruire un mutex :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // initialisation statique
ou
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);

int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

Avec `attr = NULL` utilise les paramètres par défaut.
Détail des attributs : voir le manuel.

```
int pthread_mutex_getX( ... )
int pthread_mutex_setX( ... )
```

avec X le nom de l'attribut à régler, les paramètres diffèrent selon le paramètre.

type : indique le comportement du mutex quand on le verrouille 2 fois de suite : deadlock, retour d'un message d'erreur, ou verrouillage normal (il faudra le déverrouiller un nombre égal de fois).
Le comportement par défaut n'est pas le même d'un système à l'autre : il vaut donc mieux toujours le déverrouiller après l'avoir verrouillé si on ne précise pas cet attribut.
Implémentation par défaut : la plus rapide et efficace possible donc pas de test pour éviter les erreurs et interblocage en cas de mauvais usage du mutex.
Pour le déboggage, on peut changer le type : `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_ERRORCHECK`, `PTHREAD_MUTEX_RECURSIVE`, `PTHREAD_MUTEX_DEFAULT`.

pshared : mutex placé dans la mémoire partagée pour le partager entre plusieurs processus, ou non pour l'utiliser entre plusieurs threads.

prioceiling : indique le niveau minimum de priorité auquel est exécuté le code contenu dans la section critique protégée par le mutex. La valeur doit être dans l'intervalle définit par `SHED_FIFO`. Utile pour éviter le problème d'"inversion de priorité"
exemple : les processus L et H (basse et haute priorité) se partagent une ressource avec un mutex. L obtient le mutex et bloque H qui est pourtant plus urgent. M (de priorité moyenne) demande le processeur. L est préempté et mit en attente et ne libère pas le mutex. H se retrouve à attendre alors qu'il aurait dû avoir la priorité.
=> augmenter la priorité de la section critique permet de s'assurer que L finira vite ton travail et libèrera le mutex pour débloquer H

Pour verrouiller/déverrouiller le mutex :

```
int pthread_mutex_lock(pthread_mutex_t * mutex);   // vérouille
int pthread_mutex_unlock(pthread_mutex_t * mutex); // dé-vérouille
```

ecritures-concurentes-mutex.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h> // struct timespec

// Discussion : ecritures concurentes
// $ gcc -Wall ecritures-concurentes.c -lpthread

int VALEUR;
pthread_mutex_t mutex; // mutex pour protéger VALEUR

void traitementLong() {
    struct timespec ts;
    ts.tv_sec = 0;
    ts.tv_nsec = 10;
    nanosleep(&ts, NULL); // on simule un traitement long
    // note : usleep behavior inconsistent when it is interrupted
    //        by a signal among historical systems
    //        nanosleep returns immediately when any signal handler is executed
}

void* f0(void* _p) {
    int i;
    for(i = 0; i < 1000; i++) {
        pthread_mutex_lock(&mutex); //---- vérouillage
        int x = VALEUR;
        traitementLong();  // on ajoute un traitement long
        VALEUR = x + 1; // VALEUR n'est pas modifiée par l'autre thread pendant ce temps...
        pthread_mutex_unlock(&mutex); //---- dé-vérouillage
    }
    puts("f0_fini");
    return NULL;
}

void* f1(void* _p) {
    int i;
    for(i = 0; i < 1000; i++) {
      pthread_mutex_lock(&mutex); //---- vérouillage - attend tant qu'il est vérouillé par f0
        int x = VALEUR;
        traitementLong();  // on ajoute un traitement long
        VALEUR = x + 2;
        pthread_mutex_unlock(&mutex); //---- dé-vérouillage
    }
```

```
    puts("f1_fini");
    return NULL;
}

int main() {
    pthread_mutex_init(&mutex, NULL); //----  Initialiser le mutex !! Important !
    pthread_t pid[2];

    pthread_create(&pid[0], NULL, f0, NULL); // créer un thread
    pthread_create(&pid[1], NULL, f1, NULL); // pid, attributs, fonction, argument

    // les threads travaillent

    pthread_join(pid[0], 0); // attendre la fin d'un thread
    pthread_join(pid[1], 0);

    pthread_mutex_destroy(&mutex); //----  Détruire le mutex
    printf("VALEUR=%d\n", VALEUR);

    return 0;
}
```

Verrouillage sous condition :

```
int pthread_mutex_trylock(pthread_mutex_t * mutex);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict abs_timeout);    // #include <time.h>

struct timespec { // timeout sice the epoch 1970-01-01 00:00:00 +0000 (UTC)
    time_t tv_sec;      /* Seconds */
    long   tv_nsec;     /* Nanoseconds [0 .. 999999999] */
};
```

- trylock retourne un message d'erreur si le mutex est déjà verrouillé
- timedlock permet de ne pas se bloquer indéfiniment en attente du mutex
  - ➤ si un thread est planté, l'attente est stoppée pour ne pas bloquer les autres threads
  - ➤ le temps indiqué est un temps absolu à partir de 1970-01-01 00 :00 :00 +0000 (UTC)
  - ➤ si le mutex n'est pas obtenu avant la date il ne sera pas verrouillé du tout et une erreur est retournée

exemple-timedlock.c.c

```
#include <assert.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#define NB_PAQUET 5

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pileA = NB_PAQUET;
int pileB = 0;
int pileC = 0;

void* monk(void *arg) { // passe les paquets de la pile A à la pile B
    while (pileA > 0) {
```

```
        pthread_mutex_lock(&mutex);
        printf("Monk : Je passe un paquet de A à B, laisse moi bien le poser\n");
        pileA--;
        sleep(rand() % 5); // prend du temps à poser le paquet
        pileB++;
        printf("Monk : paquet posé !\n");
        pthread_mutex_unlock(&mutex);
        sleep(1); // pause
    }
    return NULL;
}

void* impatient(void *arg) { // passe les paquets de la pile B à la pile C
    int error;

    while (pileC < NB_PAQUET) {
        struct timespec absolute_time;

        clock_gettime(CLOCK_REALTIME, &absolute_time); // l'heure courante
        absolute_time.tv_sec += 2;                     // + 2 secondes

        // on attend jusqu'à l'échéance après on laisse tomber
        error = pthread_mutex_timedlock(&mutex, &absolute_time);
        assert(error == 0 || error == ETIMEDOUT);

        if (error == ETIMEDOUT) {
            puts("Impatient : T'es vraiment trop lent. J'vais prendre un café\n");
            sleep(1); // pause café !
            continue;
        }
        if (pileB > 0) {
            pileB--;
            pileC++;
            puts("J'ai passé un paquet de B à C\n");
        }
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    srand(time(NULL)); // initialisation de rand
    pthread_t tid[2];

    printf("Paquets sur les piles A=%d B=%d C=%d\n", pileA, pileB, pileC);

    pthread_create(&tid[0], NULL, impatient, NULL);
    pthread_create(&tid[1], NULL, monk, NULL);

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    printf("Paquets sur les piles A=%d B=%d C=%d\n", pileA, pileB, pileC);

    return 0;
}
```

2.6 Nettoyage et utilité de pthread_exit()

`pthread_exit` permet d'appeler des fonctions de nettoyage même quand le thread est stoppé brutalement avec `pthread_cancel` et de s'assurer que la mémoire est bien libérée et que les verrous sont bien relâchés.
Les appels aux fonctions de nettoyages sont empilés sur la pile du thread avec :

```
void pthread_cleanup_push(void (*routine)(void *), void *routine_arg)
```

Si le thread se termine correctement, les appels peuvent être dépilés (dans l'ordre inverse) avec :

```
void pthread_cleanup_pop(int execute)
```

`execute` indique si il faut exécuter la fonction de nettoyage à ce moment là (1) ou pas (0).

Que le thread se termine normalement ou soit stoppé brutalement, l'appel à `pthread_exit()` déclenche chaque fonction de nettoyage restant sur la pile en commençant par celle qui est sur le dessus.
En l'absence de fonctions de nettoyage, `pthread_exit()` n'est pas vraiment utile.

thread_exit_exemple.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t mutex;

/* fonction de nettoyage */
void clean_mutex(void * data) {
    pthread_mutex_unlock(&mutex);
    printf("Thread %d a libéré le mutex\n", (int) pthread_self());
}

/* fonction de nettoyage */
void clean_buffer(void * data) {
    int * buffer = (int*) data;
    free(buffer);
    printf("Thread %d a libéré la mémoire allouée\n", (int) pthread_self());
}

/* thread allouant de la mémoire, bloquant le mutex et mettant trop de temps à le libérer */
void* th0(void* p) {
    int * buffer = malloc(1024 * sizeof(int));
    printf("Thread %d a alloué de la mémoire et attend le mutex\n", (int) pthread_self());

    pthread_cleanup_push(clean_buffer, buffer); // empile une fonction de nettoyage
    //pthread_cleanup_push(clean_mutex, NULL); // empile une fonction de nettoyage

    pthread_mutex_lock(&mutex);
    printf("Thread %d a bloqué le mutex\n", (int) pthread_self());
    sleep(6); // simule un traitement trop long
    pthread_mutex_unlock(&mutex);
   printf("Thread %d se termine normalement après avoir libéré le mutex\n", (int) pthread_self())

    //pthread_cleanup_pop(0); // retire la fonction clean_mutex sans l'executer
    pthread_cleanup_pop(1); // retire la fonction clean_buffer et l'exécute
    pthread_exit((void*)3); // sortie theorique propre
    // appels implicites pthread_cleanup_pop eventuels
    // 3 sert de marqueur de sortie
    return NULL; // superflu
}
```

```c
/* thread bloquant le mutex et le libérant rapidement */
void* th1(void* p) {

    pthread_cleanup_push(clean_mutex, NULL); // empile une fonction de nettoyage

    printf("Thread %d attend le mutex\n", (int) pthread_self());
    pthread_mutex_lock(&mutex);
    printf("Thread %d a bloqué le mutex\n", (int) pthread_self());
    pthread_mutex_unlock(&mutex);
   printf("Thread %d se termine normalement après avoir libéré le mutex\n", (int) pthread_self())

    pthread_cleanup_pop(1);
    pthread_exit((void*)3); // sortie theorique propre
    // appels implicites pthread_cleanup_pop eventuels
    // 3 sert de marqueur de sortie
    return NULL; // superflu
}

int main(void) {
    pthread_t pth[3];
    void* ret;
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&pth[0], NULL, th0, NULL); // thread qui va passer
    pthread_create(&pth[1], NULL, th0, NULL); // thread qui va être intérrompu
    sleep(1);
    pthread_create(&pth[2], NULL, th1, NULL); // thread qui attend le mutex

    printf("Attend 10 secondes et stoppe les threads\n");
    sleep(10); // simule une attente avant de considérer les threads plantés
    pthread_cancel(pth[0]); // attention le thread est stoppé brutalement
    pthread_cancel(pth[1]); // attention le thread est stoppé brutalement

    printf("Les threads ont été priés de s'arrêter\n");

    pthread_join(pth[0], (void**)&ret);
    printf("retour du thread %d : %d\n", (int) pth[0], *((int*) &ret));
    pthread_join(pth[1], (void**)&ret);
    printf("retour du thread %d : %d\n", (int) pth[1], *((int*) &ret));
    pthread_join(pth[2], (void**)&ret);
    printf("retour du thread %d : %d\n", (int) pth[2], *((int*) &ret));

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Deux threads allouent de la mémoire et bloquent un mutex. Si on les stoppe brutalement sans fonction de nettoyage, on a une fuite de mémoire et le mutex n'est pas relâché pour le 3eme thread.

3 Synchronisation des processus

Discussion, exemple qui ne marche pas : problème de perte de cycles, accès concurrents à la même variable, problème de synchronisation.

synchro-valeur.c

```c
#include<stdio.h>
#include<stdlib.h>
```

```
#include<pthread.h>

// Discussion : perte de cycles
// $ gcc -Wall synchro-valeur.c -lpthread

char TURN;

void* f0(void* _p) {
    int i;
    for(i = 0; i < 100; i++) {
        if(TURN == 'A') {
            puts("A");
            TURN = 'B'; }
    }
    puts("f0_fini");
    return NULL;
}

void* f1(void* _p) {
    int i;
    for(i = 0; i < 100; i++) {
        if(TURN == 'B') {
            puts("B");
            TURN = 'A'; }
    }
    puts("f1_fini");
    return NULL;
}

int main() {
    pthread_t pid[2];
    TURN = 'B';
    pthread_create(&pid[0], NULL, f0, NULL); // pid, attributs, fonction, argument
    pthread_create(&pid[1], NULL, f1, NULL);
    pthread_join(pid[0], 0);
    pthread_join(pid[1], 0);

    return 0;
}
```

3.1 Assurer la synchronisation de processus : Sémaphore

Le Mutex est un sémaphore binaire : verrouillé ou pas.
Sémaphore : plus général, permet de laisser l'accès à plusieurs processus avant de se verrouiller

Entité logique associée à une ressource, il est constitué de :
- un compteur
- une liste de tâche en attente

On parle de Sémaphore à compte par opposition au Mutex si le compteur est initialisé avec une valeur supérieure à 1.

Deux primitives P() et V() permettent de manipuler le sémaphore. Inventé par Edsger Dijkstra : "Proberen" et "Verhogen" signifient "tester" et "incrémenter" en néerlandais.
- Entrée d'une tâche en section critique : elle invoque le P()
- Fin du traitement protégé : elle effectue le V()
- la même tâche doit appeler un P() puis V()

`P(sem)` :

```
    Compteur = compteur - 1
    Si compteur < 0
        Mettre la tâche en file d'attente
    Sinon accorder la ressource
```

- on crée un sémaphore : son compteur est initialisé à N (pour N accès simultanés)
- Si plusieurs taches requièrent P(sem) (déjà à 0) : constitution de la file d'attente
- Une tâche qui acquiert la ressource "passe le P"

```
V(sem) :
    Compteur = compteur + 1
    Si compteur >= 0
        sortir une tâche de la file d'attente
```

Lorsqu'une tâche a effectué V(sem), le choix parmi les tâches en attente peut s'effectuer avec les mêmes critères que pour l'ordonnancement

Ressource fréquemment accédée : loi des priorités risquée, les tâches de faible priorité n'ont que peu de chances d'arriver à l'acquérir.

Pour signaler un évènement : sémaphore binaire (ne pas confondre avec le mutex, ici pas d'exclusion mutuelle)

- P() et V() découplés
- sémaphore binaire utilisé comme un mécanisme de signalisation :
  P() correspond à wait() et V() à signal()
- Une tâche fait le V() pour débloquer une autre tâche qui fait le P()

Tâche 1 : T_1	**Tâche 2 : T_2**
while(1){	while(1){
traitements;	P(s_1);
write(x);	read(x);
V(s_1);	traitements;
traitements;	write(y);
P(s_2);	V(s_2);
read(y);	traitements;
}	}

T1 fait des traitements, écris x et signale à T2 qu'il est prêt, fait quelques traitement, attend le résultat y puis le lit. T2 attends x puis le lit, fait des traitements, écris le résultat y et signale qu'il est prêt, fait d'autre traitements.

3.2 Manipuler un sémaphore en C

pour utiliser les sémaphores :

```
#include <pthread.h>   // pour les threads
#include <semaphore.h> // pour les semaphores
#include <fcntl.h>     // pour les flags O_CREAT, O_EXCL, ...
```

Sémaphores anonymes, non persistants (Ok sous linux, "deprecated" sous MacOS) :

```
int sem_init(sem_t * sem, int pshared, unsigned int value);
int sem_destroy(sem_t * sem);
```

- `sem_init` : par défaut le sémaphore est créé sur la pile du processus parent. Quand l'enfant est créé avec fork, il obtient une copie du sémaphore et non une référence sur le sémaphore du parent.
- nécessite d'avoir pshared>0 et de créer spécifiquement une zone mémoire partagée pour utiliser un sémaphore avec différents processus (fork).
- autre solution plus simple utiliser `sem_open`

créer un sémaphore nommé et persistant :

```
sem_t * sem_open(const char* name, int oflag, ...);
```

- sous linux, le sémaphore est créé dans `/dev/shm/` (répertoire ou sont stockés les sémaphores posix).

name : chaîne de caractères pour nommer le sémaphore
oflag : `O_CREAT` demande à créer le sémaphore que si il n'existe pas
`O_EXCL` retourne une erreur si le sémaphore existe déjà
arguments optionnels :

- les droits sur le sémaphore : 0777 ou `S_IRUSR | S_IWUSR`
  (S :Semaphore, I :Inode, R/W :Read/Write, USR :User)
- la valeur initiale du sémaphore

```
int sem_close (sem_t * sem);
```

Le processus indique qu'il n'a plus besoin de ce sémaphore, mais il n'est pas détruit, il subsiste sur la machine dans `/dev/shm/`.
Détruire un sémaphore :

```
int sem_unlink (const char * name);
```

Attention, il ne sera effectivement détruit que quand plus aucun processus ne l'utilisera.
=> la fonction retourne immédiatement, mais la destruction peut être reportée.

Pour bloquer/libérer le sémaphore :

```
int sem_wait (sem_t * sem); // primitive P()
int sem_post (sem_t * sem); // primitive V()
```

Autres fonctions pour demander le sémaphore :

```
int sem_trywait (sem_t * sem);
int sem_timedwait(sem_t * sem, const struct timespec * abs_timeout);
```

- trywait retourne une erreur au lieu de se bloquer si le sémaphore ne peut être acquis (errno == EAGAIN)
- timedwait prend une `struct timespec` qui indique une date limite au delà de laquelle, la fonction retourne une erreur (errno == ETIMEDOUT)

Pour connaitre la valeur du sémaphore :

```
int sem_getvalue(sem_t *__restrict __sem, int *__restrict __sval);
```

Attention : deprecated sous linux. On a jamais de garantie que la valeur obtenue n'est pas modifiée entre le moment ou elle a été consultée et le moment ou l'on prend connaissance de sa valeur. On peut utiliser cette valeur pour information, mais on ne doit pas trop s'y fier, ni compter dessus si on veut un programme robuste.