Programmation temps-réel : Interblocage et modèles de synchronisation

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

2016-2017

Table des matières

1 L'inter-blocage : le problème du « dîner des philosophes »

Autre version moins connue "the Dining Programmers" : des programmeurs, autour d'une table ronde, mangent des sushi et n'ont qu'une baguette entre chaque assiette.

1.1 Problème énoncé par *Edsger Dijkstra* (wikipédia)

La situation est la suivante :
- cinq philosophes (initialement mais il peut y en avoir beaucoup plus) se trouvent autour d'une table ;
- chacun des philosophes a devant lui un plat de spaghetti ;
- à gauche de chaque plat de spaghetti se trouve une fourchette.

Un philosophe n'a que trois états possibles :
- penser pendant un temps indéterminé ;
- être affamé (pendant un temps déterminé et fini, sinon il y a famine) ;
- manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation :
- quand un philosophe a faim, il va se mettre dans l'état « affamé » et attendre que les fourchettes soient libres ;
- pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à droite (c'est-à-dire les deux fourchettes qui entourent sa propre assiette) ;
- si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

1.2 Interblocage

Si les philosophes peuvent librement prendre une fourchette, un inter-blocage peut survenir. L'inter-blocage, étreinte fatale ou "deadlock" intervient si aucun philosophe ne mange et que tous saisissent la fourchette située à leur droite puis attendent indéfiniment pour obtenir la fourchette à leur gauche.

1.3 Conditions de survenue d'un inter-blocage

1. **Exclusion mutuelle** : Chaque ressource est soit attribuée à un seul, soit disponible
2. **Détention et attente** : Les entités qui détiennent des ressources peuvent en demander de nouvelles (sans relâcher celles qu'elles détiennent)
3. **Pas de réquisition** : Les ressources obtenues par un processus ne peuvent lui être retirées de force
4. **Attente circulaire** : Il y a un cycle orienté d'au moins 2 entités, chacune en attente d'une ressource détenue par une autre entité (du cycle).

1.4 Traitements possible

1. Le prévenir : imposer une contrainte pour qu'une de 4 circonstances n'apparaisse pas.
2. Le contrôler : Ajouter des contrôles pour verrouiller une ressource que s'il n'y a pas de risque d'inter-blocage.
3. Le guérir : terminer une des tâche pour débloquer les autres.

Nier une des circonstances :
1. **Exclusion mutuelle** : difficile de s'en passer
2. **Détention et attente** :
   - Acquérir toutes les ressources en même temps : section atomique, la phase d'acquisition est elle-même une section critique (section critique → limite le parallélisme, voir `semop` sous unix).
   - Relâcher une ressource acquise si l'autre ne peut pas l'être
   (trylock → risque de livelock : personne ne verrouille rien).
3. **Pas de réquisition** : Les ressources obtenues par un processus sont retirées de force, il se termine ou revient en arrière (difficile, tâches pas forcément annulable, rollback → nécessite de mémoriser l'état précédent pour un retour arrière).

4. **Attente circulaire** : On impose un ordre sur les ressources. Toutes les entités doivent demander toutes les ressources toujours dans le même ordre afin de ne jamais « fermer » le cycle.

1.5 Quelques solutions au problème des philosophes

Nier la circonstance 2 : utiliser un mutex pour protéger l'acquisition des fourchettes. Inconvénient : on ralenti tout le monde...
Nier la circonstance 2 : utiliser timedlock ou trylock. Tous les philosophes risquent de s'attendre et personne ne mange.
Nier la circonstance 4 : introduire un gaucher. Un des philosophes prendra sa fourchette gauche avant la droite. Dans le cas d'un nombre pair de philosophes, on impose à un sur deux d'être gaucher : N/2 philosophes mangent en même temps.
Nier la circonstance 4 : ajouter un sémaphore qui permet à seulement N-1 philosophes de manger en même temps. Un des philosophe laisse les fourchettes à ses voisins. Ce sémaphore permet de s'assurer qu'ils mangeront chacun leur tour.
Nier la circonstance 4 : solution de Chandy/Misra (1984) Introduire une notion d'ordonnancement et de requête. Les philosophes demandent poliment les fourchettes. Si deux philosophes veulent acquérir la même fourchette, on la donne à celui qui a le plus petit identifiant. Si un des philosophes possède la fourchette, il la cède à son voisin si il a fini de manger. Si il n'a pas encore mangé, il attend la deuxième fourchette et mange un peu avant de la lui passer.

2 Rendez-vous

Si envoyeur non bloquant, on peut avoir besoin de gérer des acquittements par retour de signalisation

```
Tâche 1 : T1                Tâche 2 : T2
signal(T2, evt);            wait(evt);
wait(ack);                  traiter le signal
                            signal(T1, ack);
```

l'appelant (T1) demande un rendez vous à une tâche T2 qui fournit différents services. L'appelant requiert un rendez-vous sur ce service, avec des paramètres en entrée et en sortie Il reste bloqué la tâche (T2) accepte le rendez-vous, traite les données et envoie un signal (acknowledge)
nombreuses variations :

- différents types de signalisation
- retour de l'acquittement dès l'activation de la tâche
- ou seulement lorsqu'elle devient courante
- ...

Exemple : machine MPPA du labo : T1 est un cluster qui demande une transmission de données vers la machine hôte, T2 est le processus responsable des échanges de données.

2.1 Variables conditions

Permet de bloquer un thread tant qu'une condition n'est pas remplie. Le thread est mit en sommeil ce qui lui évite de consommer inutilement les ressources du processeur.
Initialisation de la variable de condition :

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; // initialisation statique, pas de vérification d'erreur
                                                // idem : pthread_cond_init(&cond, NULL)

int pthread_cond_init(pthread_cond_t* cond, pthread_condattr_t* attr);
```

`pthread_condattr_t` : structure contenant des attributs comme la portée de la variable de condition (partagée par plusieurs processus, ou privée i.e. partagée uniquement par les threads).

Destruction :

```
int pthread_cond_destroy(pthread_cond_t* cond);
```

Détruit la variable conditionnelle mais ne libère pas l'espace mémoire.

Pour attendre la condition :

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);
```

Attente sous la protection d'un mutex. Si la condition n'est pas remplie, le mutex est libéré avant de rentrer en attente. Le mutex est re-verrouillé à la réception du signal.

```
pthread_mutex_lock(mutex);
    while(condition_is_false)
        pthread_cond_wait(cond_var, mutex);
pthread_mutex_unlock(mutex);
```

Attente limitée dans le temps par une date d'échéance :

```
int pthread_cond_timedwait(pthread_cond_t* cond, pthread_mutex_t* mutex,
                           const struct timespec* abstime);
```

Pour envoyer un signal et réveiller les threads en attente (la condition est remplie) :

```
int pthread_cond_signal(pthread_cond_t* cond);
```

Doit être utilisé sous la protection du même mutex que le thread qui attend, sinon le signal peut être envoyé entre le moment ou la condition est testée et le moment ou le thread se bloque en attente.

```
pthread_mutex_lock(mutex);
    pthread_cond_signal(cond_var);
pthread_mutex_unlock(mutex);
```

Pour réveiller plusieurs threads à la fois :

```
int pthread_cond_broadcast(pthread_cond_t* cond);
```

Si on envoie un signal unique, certains threads ne sont pas réveillés et restent bloqués indéfiniment. Avec broadcast, tous les threads se réveillent à chaque signal envoyé.

condition_exemple.c

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

pthread_cond_t cond;
pthread_mutex_t mutex; // pour protéger cond

void* th(void* p) {
    int i;

    sleep(4); // => les signaux non receptionnés sont perdus...

    for (i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);
        // ici on verrifie si des données sont disponibles
        // avant de se mettre en attente
        printf("thread %d : wait signal\n", (int) pthread_self());
        pthread_cond_wait(&cond, &mutex);
        printf("thread %d : signal reçu !\n", (int) pthread_self());
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    int i;
    srand(time(NULL)); // initialisation de rand (attention : non thread safe!)
```

```
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_t tids[3];

    pthread_create(&tids[0], NULL, th, NULL);
    pthread_create(&tids[1], NULL, th, NULL);
    pthread_create(&tids[2], NULL, th, NULL);

    for (i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);
        printf("main thread : 1 signal sent\n");
        pthread_cond_signal(&cond);   // debloque un seul
        pthread_mutex_unlock(&mutex);

        sleep(3); // travaille

        pthread_mutex_lock(&mutex);
        pthread_cond_broadcast(&cond); // debloque tous
        printf("main thread : signal sent for all\n");
        pthread_mutex_unlock(&mutex);

        sleep(3); // travaille
    }

    pthread_join(tids[0], NULL);
    pthread_join(tids[1], NULL);
    pthread_join(tids[2], NULL);

    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

2.2 Barrières

Permet de créer un point de synchronisation. Un nombre permet d'indiquer le nombre de tâches à attendre devant barrière. Quand toutes les tâches sont regroupées à la barrière elle s'ouvre et la suite des instructions est exécutée.

Initialisation/destruction de la barrière :

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                         const pthread_barrierattr_t *restrict attr, unsigned count);
```

Pour placer une barrière :

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

La barrière se débloque quand le nombre d'appel de cette fonction atteint la valeur `count`.

barriere_exemple.c

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

pthread_barrier_t debut_reponse;
```

```
pthread_barrier_t fin_reponse;

void* th(void* p) {
    int i;
    for (i = 0; i < 5; i++) {
        // attend la question...
        pthread_barrier_wait(&debut_reponse);
        printf("thread %d : HOURA !\n", (int) pthread_self());
        sleep(rand() % 3);
        pthread_barrier_wait(&fin_reponse);
    }
    return NULL;
}

int main() {
    int i;
    srand(time(NULL)); // initialisation de rand (attention : non thread safe!)

    pthread_barrier_init(&debut_reponse, NULL, 4); // 3 threads + main thread
    pthread_barrier_init(&fin_reponse, NULL, 4); // 3 threads + main thread
    pthread_t tids[3];

    pthread_create(&tids[0], NULL, th, NULL);
    pthread_create(&tids[1], NULL, th, NULL);
    pthread_create(&tids[2], NULL, th, NULL);

    printf("main thread : ok, je commence !\n");

    for (i = 0; i < 5; i++) {
        printf("main thread : hip hip hip ?\n");
        sleep(1);
        pthread_barrier_wait(&debut_reponse);
        // attend la réponse...
        pthread_barrier_wait(&fin_reponse);
    }

    pthread_join(tids[0], NULL);
    pthread_join(tids[1], NULL);
    pthread_join(tids[2], NULL);

    pthread_barrier_destroy(&debut_reponse);
    pthread_barrier_destroy(&fin_reponse);

    return 0;
}
```

3 Modèle Producteur/consommateur

zones mémoire tampon entre un producteur et un consommateur

- liste de N tampons mémoire
- un acteur retire un tampon de la liste, le vide ou le remplit et le remet dans la liste
- le producteur et le consommateur ne doivent pas accéder au même tampon en même temps
- le producteur doit attendre si tout est plein
- le consommateur doit attendre si tout est vide

```
mutex m = 1;         // protège l'accès au tampon
semaphore plein = 0; // présence d'un tampon contenant un message ?
semaphore vide = n;  // limite à n le nombre de tampons
```

```
buf_type tampon[n];
producteur() {
    buf_type *tp, *new;
    while(1){
        new = produire();      // création un nouvel élément
        P(vide);               // y a t-il un tampon libre ?
        P(m);
        tp = obtenir(tampon);    // obtention du tampon
        V(m);
        copier(new, tp);
        P(m);
        placer(tp, tampon);     // met tampon dans liste des tampons
        V(m);
        V(plein);              // signale la présence d'un tampon plein
    }
}
consommateur() {
    buf type *tp;
    while(1){
        P(plein);                // y a t-il un message à consommer ?
        P(m);
        tp = obtenir(tampon);    // obtention du tampon plein
        V(m);
        consommer(tp);
        P(m);
        placer(tp, tampon);      // met tampon dans liste des tampons
        V(m);
        V(vide);                 // signale le libération du tampon consommé
    }
}
```

4 Modèle Lecteurs et les écrivains

partager une ressource entre un ensemble de tâches pouvant avoir des rôles distincts :

- lecteurs et écrivains.
- les lecteurs peuvent consulter la ressource ensemble
- un écrivain demande un accès exclusif à la ressource pour la modifier

4.1 1ere solution : les écrivains attendent quand un lecteur consulte.

```
ressource_type *ressource;
int nb_lect = 0;
mutex nb_lecteur = 1
mutex verrou_ecrivain = 1;

lecteur() {
    P(nb_lecteur) // un seul lecteur modifie le nombre à la fois
    nb_lect = nb_lect + 1;
    if(nb_lect == 1) // seul le premier verrouille l'accès aux écrivains
        P(verrou_ecrivain);
    V(nb_lecteur)
    //pas d'attente pour les autres lecteurs
    lecture(ressource);  // dure un temps aléatoire
    P(nb_lecteur)
    nb_lect = nb_lect - 1;
    if(nb_lect == 0)
        V(verrou ecrivain);
    V(nb_lecteur);
}
```

```
ecrivain() {
    P(verrou_ecrivain); // compétition avec le premier lecteur
    ecriture(ressource);
    V(verrou_ecrivain); // libère le premier lecteur
}
```

4.2 2ere solution : s'assurer que les lecteurs pourront consulter en priorité

```
ressource_type *ressource;
int nb_lect = 0;
mutex nb_lecteur = 1
mutex verrou_ecrivain = 1;
mutex ecrit_attente = 1;

lecteur() {
    P(nb_lecteur) // un seul lecteur modifie le nombre à la fois
    nb_lect = nb_lect + 1;
    if(nb_lect == 1) // seul le premier verrouille l'accès aux écrivains
        P(verrou_ecrivain);
    V(nb_lecteur)
    //pas d'attente pour les autres lecteurs
    lecture(ressource);  // dure un temps aléatoire
    P(nb_lecteur)
    nb_lect = nb_lect - 1;
    if(nb_lect == 0)
        V(verrou ecrivain);
    V(nb_lecteur);
}

ecrivain() {
    P(ecrit_attente);   // les autres écrivains restent bloqués ici
    P(verrou_ecrivain); // compétition avec le premier lecteur
    ecriture(ressource);
    V(verrou_ecrivain); // libère le premier lecteur
    V(ecrit_attente); // un lecteur est passé avant de laisser le passage au prochain écrivain
}
```

La fin d'un écrivain libère le mutex verrou_ecrivain libérant ainsi un éventuel lecteur avant de libérer le sémaphore ecrit_attente

4.3 3eme solution : s'assurer que les écrivains pourront écrire en priorité

```
ressource type *ressource;
int nb_lect = 0;
int nb_ecriv = 0;
mutex nb_lecteur = 1
mutex nb_ecrivain = 1
mutex verrou_lecteur = 1;
mutex verrou_ecrivain = 1;
mutex lecteur_attente = 1;

ecrivain() {
    P(nb_ecrivain);
    nb_ecriv = nb_ecriv + 1;
    if(nb_ecriv == 1)
        P(verrou_lecteur); // stoppe les prochains lecteurs
    V(nb_ecrivain);
    P(verrou_ecrivain); // peut créer une file d'attente d'écrivains
    ecriture(ressource);
```

```
    V(verrou_ecrivain);
    P(nb_ecrivain);
    nb_ecriv = nb_ecriv - 1;
    if(nb_ecriv == 0)
        V(verrou lecteur);
    V(nb_ecrivain); // c'est le dernier écrivain qui débloque le ou les lecteurs
}

lecteur() {
    // filtre les lecteurs un par un
    // compétition entre un lecteur et des écrivains
    P(lecteur_attente); // empêche les autres lecteurs de passer
    P(verrou_lecteur);  // le verrou est obtenu si tous le dernier écrivain le relâche
    P(nb_lecteur);
    nb_lect = nb_lect + 1;
    if(nb_lect == 1)
        P(verrou_ecrivain); // compétition avec les écrivains (le premier gagne)
    V(nb_lecteur);
    V(verrou_lecteur);
   // un écrivain peut obtenir le verrou pendant que les autres lecteurs attendent (lecteur_attente)
    V(lecteur_attente);
    lecture(ressource);
    P(nb_lecteur);
    nb_lect = nb_lect - 1;
    if(nb_lect == 0)
        V(verrou_ecrivain);
    //le dernier libère les écrivains
    V(nb_lecteur);
}
```

4.4 4eme solution : chances égales

```
ressource_type *ressource;
int nb_lect = 0;
mutex verrou_lecteur = 1;
mutex verrou_ecrivain = 1;
mutex fifo = 1;

ecrivain() {
    P(fifo); // file d'attente de lecteurs et écrivains
    P(verrou_ecrivain);
    V(fifo)
    ecriture(ressource);
    V(verrou_ecrivain);
}

lecteur() {
    P(fifo); // file d'attente de lecteurs et écrivains
    P(verrou_lecteur);
    nb_lect = nb_lect + 1;
    if(nb_lect == 1)
        P(verrou_ecrivain); // compétition avec les écrivains (le premier gagne)
    V(verrou_lecteur);
    V(fifo);
    lecture(ressource);
    P(verrou_lecteur);
    nb_lect = nb_lect - 1;
    if(nb_lect == 0)
        V(verrou_ecrivain);
    V(verrou_lecteur);
}
```

4.5 Variante : 2 groupes de lecteurs

Autre cas similaire équivalent à 2 groupes de "lecteurs" : deux groupes de voitures qui doivent emprunter un chemin étroit en sens opposé. Plusieurs véhicules peuvent passer dans un sens, mais ils bloquent la route pour l'autre groupe.

4.6 Verrou rwlock

Utiles pour une ressource associée à :

- N accès simultanés en lecture
- 0 ou 1 accès en écriture sans accès en lecture

Pour initialiser / détruire le verrou :

```
int pthread_rwlock_init(pthread_rwlock_t * rwl, const pthread_rwlockattr_t * attr);
int pthread_rwlock_destroy(pthread_rwlock_t* rwl);
```

Pour demander le verrou en lecture :

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwl);
int pthread_rwlock_tryrdlock(pthread_rwlock_t* rwl);
```

- rdlock : bloque le thread en attente du verrou
- tryrdlock : retourne une erreur si le verrou ne peut être acquis immédiatement

Idem pour acquérir le verrou en écriture :

```
int pthread_rwlock_wrlock(pthread_rwlock_t* rwl);
int pthread_rwlock_trywrlock(pthread_rwlock_t* rwl);
```

Pour relacher le verrou :

```
int pthread_rwlock_unlock(pthread_rwlock_t* rwl);
```

Exemple :

thread_verrou_rwlock.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>

pthread_rwlock_t lock;
char data = 'A';

void* lecteur(void *arg) {
    int id = *((int*) arg);
    int i;

    for (i = 0; i < 5; i++) {
        printf("Lecteur %d veut consulter la ressource\n", id);
        pthread_rwlock_rdlock(&lock);
        printf("Lecteur %d consulte la ressource : %c\n", id, data);
        sleep(rand() % 2); // on simule un traitement long
        printf("Lecteur %d a fini de consulter la ressource : %c\n", id, data);
        pthread_rwlock_unlock(&lock);
        sleep(rand() % 3); // le lecteur fait une pause
    }
    printf("Lecteur %d : s'en va\n", id);
    return NULL;
```

```
}

void* ecrivain(void *arg) {
    int id = *((int*) arg);
    int i;

    for (i = 0; i < 5; i++) {
        printf("Ecrivain %d veut modifier la ressource\n", id);
        pthread_rwlock_wrlock(&lock);
        printf("Ecrivain %d modifie la ressource : %c\n", id, data);
        sleep(rand() % 2); // on simule un traitement long
        data++;
        printf("Ecrivain %d : maintenant la ressource est %c\n", id, data);
        pthread_rwlock_unlock(&lock);
        sleep(rand() % 3); // l'écrivain fait une pause
    }
    printf("Ecrivain %d : s'en va\n", id);
    return NULL;
}

int main() {
    int i, nb[8];
    srand(time(NULL)); // initialisation de rand
    pthread_t tid[8];

    pthread_rwlock_init(&lock, NULL);
    for (i = 0; i < 5; i++) {
        nb[i] = i;
        pthread_create(&tid[i], NULL, lecteur, (void*) &nb[i]);
    }
    for (i = 0; i < 3; i++) {
        nb[i+5] = i;
        pthread_create(&tid[i+5], NULL, ecrivain, (void*) &nb[i+5]);
    }

    for (i = 0; i < 8; i++) {
        pthread_join(tid[i], NULL);
    }
    puts("Consultation et modifications terminées");

    pthread_rwlock_destroy(&lock);
    return 0;
}
```

5 Boîtes aux lettres

Mécanisme d'échange de messages asynchrones
Le message est de taille limitée : type primitif ou pointeur
boîte aux lettres :

- composé d'une file de messages et d'une file de tâches
- une des files est vide à tout moment
- fonctions send() et receive() utilisables par les tâches

receive(mailbox, out_message)	send(mailbox, in_message)

if(file_message == VIDE){
  mettre la tâche en file;
  suspendre la tâche;
}
sortir out_message de la file;
	insérer in_message dans la file
	if(file des tâches != VIDE){
	  sortir la 1ère tâche de la file;
	  l'activer;
	}

- receive() bloquant : tant qu'il n'y a pas de message disponible
- send() non bloquant

5.1 Files de messages

Pour créer une file de messages. La file est nommée et persistante. Elle peut être partagée entre 2 processus.
on retrouve les fonction `open/close/unlink` comme pour un sémaphore nommé.
Les attributs permettent de préciser le nombre maximum de messages stockés dans la file ainsi que leur taille.
`mq_receive/mq_send` permetent respectivement de recevoir ou envoyer un message dans la file.
`mq_notify` permet de mettre en place un système de notification : un événement est déclenché à l'arrivée d'un message.

```
// créer une queue
mqd_t mq_open(const char* name, int oflag, ...);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);

//oflag :
// O_RDONLY : Open the queue to receive messages only.
// O_WRONLY : Open the queue to send messages only.
// O_RDWR : Open the queue to both send and receive messages.

// O_NONBLOCK : Open the queue in nonblocking mode. fail with the error EAGAIN.
// O_CREAT : Create the message queue if it does not exist.
// O_EXCL | O_CREAT : fail with the error EEXIST if the queue already exists.

// mode : permissions
// attr : les attributs

// obtenir les attributs
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

Les attributs sont directement réglés en accédant aux membres de la structure :

```
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

// changer les attributs
int mq_setattr(mqd_t mqdes, struct mq_attr *newattr, struct mq_attr *oldattr);

// fermer la queue
int mq_close (mqd_t mqdes);

// détruire le fichier représentant la queue sur le système
int mq_unlink (const char *name);

// enregistre un évènement qui sera déclanché à la réception d'un message
// à renouveler pour chaque message
int mq_notify (mqd_t mqdes, const struct sigevent* notif);

// recevoir un message
ssize_t mq_receive (mqd_t mqdes, char* msgPtr, size_t msgLen, unsigned int* msgPrio);
```

```
// envoyer un message
int mq_send (mqd_t mqdes, const char* msgPtr, size_t msgLen, unsigned int msgPrio);
```

exemple :

mqueue_exemple.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>
#include <fcntl.h>           /* For O_* constants */
#include <mqueue.h>

// compiler avec  $ gcc -Wall mqueue_exemple.c -lrt


#define QUEUE_NAME  "/test_queue"
#define MAX_SIZE    1024
#define MSG_STOP    "exit"

enum {READER, WRITER} r_w;

void usage(char* n) {
    fprintf(stderr, "usage : %s [R|W]\n", n);
    exit(1);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        usage(argv[0]);
    } else {
        if (!strcmp(argv[1], "R"))
            r_w = READER;
        else if (!strcmp(argv[1], "W"))
            r_w = WRITER;
        else
            usage(argv[0]);
    }

    mqd_t mq;
    struct mq_attr attr;
    char buffer[MAX_SIZE+1];

    /* initialize the queue attributes */
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    /* open/create the mail queue */
    if (r_w == READER)
        mq = mq_open(QUEUE_NAME, O_CREAT | O_RDONLY, 0644, &attr);
    else
        mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
    assert((mqd_t)-1 != mq);

    if (r_w == READER) {
        int must_stop = 0;
```

```
        do {
            ssize_t bytes_read;

            /* receive the message */
            bytes_read = mq_receive(mq, buffer, MAX_SIZE, NULL);
            assert(bytes_read >= 0);

            buffer[bytes_read] = '\0';
            if (!strncmp(buffer, MSG_STOP, strlen(MSG_STOP))) {
                must_stop = 1;
            } else {
                printf("Received: %s", buffer);
                sleep(3);
            }
        } while (!must_stop);
    } else {
        printf("Send to server (enter \"exit\" to stop it):\n");

        do {
            printf("> ");
            fflush(stdout);

            memset(buffer, 0, MAX_SIZE);
            fgets(buffer, MAX_SIZE, stdin);

            /* send the message */
            assert(0 <= mq_send(mq, buffer, MAX_SIZE, 0));

        } while (strncmp(buffer, MSG_STOP, strlen(MSG_STOP)));
    }

    /* cleanup */
    assert(mq_close(mq) != (mqd_t)-1);
    if (r_w == READER) assert(mq_unlink(QUEUE_NAME) != (mqd_t)-1);

    return 0;
}
```

mqueue_notify_exemple.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>
#include <fcntl.h>           /* For O_* constants */
#include <mqueue.h>

// compiler avec  $ gcc -Wall mqueue_exemple.c -lrt


#define QUEUE_NAME  "/test_queue"
#define MAX_SIZE    1024
#define MSG_STOP    "exit"

enum {READER, WRITER} r_w;

void usage(char* n) {
    fprintf(stderr, "usage : %s [R|W]\n", n);
```

```
    exit(1);
}

typedef struct {
    union sigval sv;
    struct sigevent * se;
} sv_se;


static void                     /* Thread start function */
tfunc(union sigval arg)
{
    sv_se * args = arg.sival_ptr;
    struct mq_attr attr;
    ssize_t bytes_read;
    char *buffer;
    mqd_t mq = *((mqd_t *) args->sv.sival_ptr);

    /* Determine max. msg size; allocate buffer to receive msg */
    assert(mq_getattr(mq, &attr) != -1);
    buffer = malloc(attr.mq_msgsize);
    assert(buffer != NULL);

    /* receive the message */
    bytes_read = mq_receive(mq, buffer, attr.mq_msgsize, NULL);
    assert(bytes_read >= 0);
    buffer[bytes_read] = '\0';
    if (!strncmp(buffer, MSG_STOP, strlen(MSG_STOP))) {
        free(buffer);
        exit(EXIT_SUCCESS); /* Terminate the process */
    } else {
        printf("Received: %s", buffer);
        fflush(stdout);
    }
    free(buffer);
    assert(mq_notify(mq, args->se) != -1); // set new notification
    puts("end");
}



int main(int argc, char* argv[]) {
    if (argc < 2) {
        usage(argv[0]);
    } else {
        if (!strcmp(argv[1], "R"))
            r_w = READER;
        else if (!strcmp(argv[1], "W"))
            r_w = WRITER;
        else
            usage(argv[0]);
    }

    mqd_t mq;
    struct sigevent se;
    struct mq_attr attr;
    char buffer[MAX_SIZE+1];

    /* initialize the queue attributes */
```

```
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = MAX_SIZE;
    attr.mq_curmsgs = 0;

    /* open/create the mail queue */
    if (r_w == READER) {
        mq = mq_open(QUEUE_NAME, O_CREAT | O_RDONLY, 0644, &attr);
        assert((mqd_t)-1 != mq);

        /* create the notifyer */
        //se.sigev_notify = SIGEV_SIGNAL;
        //se.sigev_signo = SIGUSR1;
        se.sigev_notify = SIGEV_THREAD;
        se.sigev_notify_function = tfunc;
        se.sigev_notify_attributes = NULL;
        sv_se args;
        args.sv.sival_ptr = &mq;
        args.se = &se;
        se.sigev_value.sival_ptr = (void *) &args;   /* Arg. to thread func. */
        assert(mq_notify(mq, &se) != -1);
    } else {
        mq = mq_open(QUEUE_NAME, O_CREAT | O_WRONLY, 0644, &attr);
        assert((mqd_t)-1 != mq);
    }

    if (r_w == READER) {
        while (1) {
            pause();
        }
    } else {
        printf("Send to server (enter \"exit\" to stop it):\n");

        do {
            printf("> ");
            fflush(stdout);

            memset(buffer, 0, MAX_SIZE);
            fgets(buffer, MAX_SIZE, stdin);

            /* send the message */
            assert(0 <= mq_send(mq, buffer, MAX_SIZE, 0));

        } while (strncmp(buffer, MSG_STOP, strlen(MSG_STOP)));
    }

    /* cleanup */
    assert(mq_close(mq) != (mqd_t)-1);
    if (r_w == READER) assert(mq_unlink(QUEUE_NAME) != (mqd_t)-1);

    return 0;
}
```

6 Autre outils

6.1 Verrou tournant : spinlock

- Attente active : scrutation
- Utile dans un système temps réel pour avoir un temps de réaction très court

- Inconvénient : on monopolise le processeur pour attendre
- Au niveau du système le spinlock est associé avec un masquage des interruptions :
  - ➤ pour exécuter une section critique de manière atomique
  - ➤ pour éviter une interruption de processus plus prioritaires qui font une attente active sur le même verrou : deadlock
- Système mono-processeur : on doit s'assurer que les interruptions ne sont pas masquées pendant qu'on fait l'attente active (mais uniquement quand on tient le verrou) sinon aucun autre processus ne peut relâcher le verrou...

Utilisé dans les cas suivants :
- changement de contexte impossible (par exemple en l'absence d'OS)
- changement de contexte plus long que le temps d'attente moyen (contrainte temps réel souple) ou maximum (contrainte temps réel dur). en général c'est le cas que dans les systèmes SMP.
- Le spinlock n'a d'intérêt que pour les systèmes multiprocesseur pour exécuter des tâches pseudo-atomiques.

Manipulation du verrou tournant :

```
int pthread_spin_init (pthread_spinlock_t *spinner, int pshared);
int pthread_spin_destroy (pthread_spinlock_t *spinner);
int pthread_spin_lock (pthread_spinlock_t *spinner);
int pthread_spin_trylock (pthread_spinlock_t *spinner);
int pthread_spin_unlock (pthread_spinlock_t *spinner);
```

7 Mutex et Semaphores : limites pour le Temps réel

7.1 Protection contre les accès concurrents : mutex

- Attention aux blocages !
- Une tâche de faible priorité peut bloquer une tâche de haute priorité
- Ou au contraire une tâche de faible priorité peut subir une famine
- Solutions :
  - ➤ Gestion des attentes par FIFO et augmenter la priorité des sections critiques
  - ➤ Fournir des primitives de type "test de disponibilité"
  - ➤ Associer un temps d'attente maximum aux primitives bloquantes
- risque de bottleneck : un "embouteillage" ou "goulot d'étranglement" se forme au niveau de l'accès au mutex.

Algorithme sans mutex : la perte de données due à des accès concurrents peut être négligeable par rapport au temps de verrouillage d'un mutex et du ralentissement provoqué par l'attente du mutex.
=> la possibilité d'un algorithme sans mutex est à étudier.

Compromis entre efficacité et risque en cas de données "compartimentées" (exemple : un arbre avec plusieurs noeuds)
Verrouiller tout l'arbre pour modifier un nœud empêche les autres threads de travailler sur le même arbre même si ils veulent modifier un autre nœud : perte de temps. Verrouiller chaque nœud séparément : gourmand, programme difficile à maintenir et risque d'inter-blocage si un thread doit verrouiller plusieurs nœuds à la fois. Il faut trouver un compromis entre les deux.

7.2 Synchronisation : sémaphores

- risque d'une tâche de faible priorité qui en bloque une de plus haute priorité
- risque d'inter-blocage

Solution
- débloquer les tâches en attente en fonction de leur priorité
- faire un test de disponibilité : une tâche qui n'obtient pas une ressource continue son travail et réessaie plus tard
- associer un temps maximum aux primitives bloquantes
  - ➤ prévenir les inter-blocages
  - ➤ éviter qu'une tâche monopolise une ressource

8 Processus lourds

Priorité des processus : tester dans un shell

```
$ a.out          # priorité "normale"
$ a.out &        # tâche de fond : basse priorité
$ nice a.out     # augmente de 10 la priorité (-n X : augmente de X)
```

Différents appels système pour démarrer un processus dans un programme :

- system()
- fork()
- vfork()
- Famille des exec
- Famille des spawn

leur utilisation dépends de la portabilité et des fonctionnalités recherchées.

8.1 system

`int system(const char* cmd)`

exécution de `cmd` avec `/bin/sh -c cmd`

- pendant cmd :
  - ➤ SIGCHLD bloqué : utilisé pour réveiller un processus dont un enfant vient de mourir
  - ➤ SIGINT ignoré : interruption (Ctrl+C)
  - ➤ SIGQUIT ignoré : arrêt demandé avec core dump, copie de la mémoire vive et des registres (Ctrl-\)
- retourne 127 si échec de /bin/sh
- retourne -1 si erreur de cmd
- sinon retourne la valeur de retour de cmd

system_exemple.c

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

int main(int argc, char * argv[]){

  printf("Je vais lancer system(\"echo bonjour\")\n");
  int status = system("echo bonjour");
  printf("La commande a retourné %d\n", status);

  printf("Je vais lancer system(\"ls -z\")\n");
  status = system("ls -z");
  printf("La commande a retourné %d\n", status);

  return 0;
}
```

8.2 fork

`pid_t fork(void)`

Création d'une copie du processus courant :

- avec nouveau pid
- avec nouveaux descripteurs de fichier
- avec nouveaux flux de répertoire

- avec nouveaux timers d'exécution
- sans conservation des verrous
- sans conservation des signaux
- sans conservation des alarmes
- retourne 0 dans processus-fils
- retourne pid-fils dans processus-père
- retourne -1 en cas d'échec

fork_exemple.c

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

int main(int argc, char * argv[]){
    int pid, status = 0;

    int var = 33; // cette variable sera copiée

    pid = fork();

    if (pid < 0) {
        perror("fork"); // echec lors de la creation
        return -1 ;
    }
    if (pid == 0) { // enfant
        printf("Je suis l'enfant, pid=%d\n", getpid());
        printf("Mon parent est %d\n", getppid());
        var += 4;
        printf("var chez l'enfant = %d\n", var);
        sleep(3);
        _exit(0); // "it is wrong to call exit
                  // since buffered data would then be flushed twice"
    }
    else { // parent
        printf("Je suis le parent, pid=%d\n", getpid());
        printf("J'attends mon enfant\n");
        var += 1;
        printf("var chez le parent = %d\n", var);
        wait(&status); // la fin du parent ne doit pas faire
                       // de l'enfant un zombie
        printf("Mon enfant a retourné, status=%d\n", status);
    }

    return 0;
}
```

parent qui n'attend pas son enfant avec wait (processus zombie) :

zombie.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main () {
  pid_t child_pid;
  /* Crée un processus fils. */
  child_pid = fork ();

  if (child_pid > 0) {
```

```
    /* Nous sommes dans le processus parent. Attente d'une minute. */
    sleep (60);
  }
  else {
    /* Nous sommes dans le processus fils. Sortie immédiate. */
    exit (0);
  }

  return 0;
}
```

```
$ gcc -Wall zombie.c -o make-zombie # compilation
$ ./make-zombie                    # execution, création d'un processus zombie
$ ps -e -o pid,ppid,stat,cmd       # liste des identifiants des processus,
                                   # des processus pères,
                                   # de leur statut et
                                   # de la ligne de commande du processus.
```

En plus du processus père `make-zombie`, un autre processus `make-zombie` est affiché. Il s'agit du processus fils ; notez que l'identifiant de son père est celui du processus `make-zombie` principal. Le processus fils est marqué comme `<defunct>` et son code de statut est `Z` pour zombie.

Que se passe-t-il lorsque le programme principal de `make-zombie` se termine sans appeler `wait` ? Le processus zombie est-il toujours présent ? Non

Relancer `ps` et notez que les deux processus `make-zombie` ont disparu. Lorsqu'un programme se termine, un processus spécial hérite de ses fils, le programme `init`, qui s'exécute toujours avec un identifiant de processus valant 1 (il s'agit du premier processus lancé lorsque Linux démarre). Le processus `init` libère automatiquement les ressources de tout processus zombie dont il hérite.

8.3 vfork

```
pid_t vfork(void)
```

Suspend le père jusqu'à :
- suspension du fils par exécution d'un autre processus
- terminaison du fils

Père et fils partagent le même espace adressage

vfork_exemple.c

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

int main(int argc, char * argv[]){
    int pid, status = 0;

    int var = 33; // cette variable sera partagée

    pid = vfork();

    if (pid < 0) {
        perror("fork"); // echec lors de la creation
        return -1 ;
    }
    if (pid == 0) { // enfant
        printf("Je suis l'enfant, pid=%d\n", getpid());
        printf("Mon parent est %d\n", getppid());
```

```
        var += 4;
        printf("var chez l'enfant = %d\n", var);
        sleep(3);
        _exit(0); // "call _exit rather than exit if you can't execve,
                  // since exit will flush and close standard I/O channels"
    }
    else { // parent
        printf("Je suis le parent, pid=%d\n", getpid());
        printf("J'attends mon enfant\n");
        var += 1;
        printf("var chez le parent = %d\n", var);
        wait(&status); // la fin du parent ne doit pas faire
                       // de l'enfant un zombie
        printf("Mon enfant a retourné, status=%d\n", status);
    }

    return 0;
}
```

➥ utilisé quasiment exclusivement pour lancer un exec en suivant.
- Non efficient pour les autres usages, mieux vaut utiliser des threads.
- la différence est qu'il y a un nouveau pid

8.4 exec et spawn

Equivalent de fork + exec sous UNIX
spawn : disponible sous QNX par exemple

QNX : système d'exploitation compatible POSIX, conçu principalement pour le marché des systèmes embarqués micro-noyau => ensemble de "serveurs", services que l'on peut désactiver (plus léger).

différences entre ces deux familles :
- exec
  - ➤ insertion dans la séquence d'instructions courante
  - ➤ sans changement de pid
- spawn
  - ➤ création d'une nouvelle séquence d'instructions
  - ➤ nouveau processus, nouveau pid
- POSIX :
  - ➤ spawn()
  - ➤ execl(), execle(), execlp(), execlpe(), execp(), execv(), execve(), execvp()
- non POSIX :
  - ➤ spawnl(), spawnle(), spawnlp(), spawnlpe(), spwanp(), spwanv(), spwanve(), spwanvp(), spwanvpe()
  - ➤ execvpe()
- signification des suffixes :
  - ➤ l, spécifie une liste de paramètres
  - ➤ e, spécifie un environnement
  - ➤ p, spécifie un chemin
  - ➤ v, spécifie un vecteur de paramètres

exemples :

execl_exemple.c

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

```c
int main(int argc, char * argv[]){
    int pid, status = 0;

    pid = vfork();

    if (pid < 0) {
        perror("fork"); // echec lors de la creation
        return -1 ;
    }
    if (pid == 0) { // enfant
        printf("Je suis l'enfant, pid=%d\n", getpid());
        printf("Mon parent est %d\n", getppid());
        execl("/bin/echo", "echo", "bonjour", "!", NULL);
        _exit(0);
    }
    else { // parent
        printf("Je suis le parent, pid=%d\n", getpid());
        printf("J'attends mon enfant\n");
        wait(&status);
        printf("Mon enfant a retourné, status=%d\n", status);
    }

    return 0;
}
```

execv_exemple.c

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

int main(int argc, char * argv[]){
    int pid, status = 0;

    pid = vfork();

    if (pid < 0) {
        perror("fork"); // echec lors de la creation
        return -1 ;
    }
    if (pid == 0) { // enfant
        printf("Je suis l'enfant, pid=%d\n", getpid());
        printf("Mon parent est %d\n", getppid());
        char* com[] = {"echo", "bonjour", "!", NULL};
        execv("/bin/echo", com);
        _exit(0);
    }
    else { // parent
        printf("Je suis le parent, pid=%d\n", getpid());
        printf("J'attends mon enfant\n");
        wait(&status);
        printf("Mon enfant a retourné, status=%d\n", status);
    }

    return 0;
}
```

intérêt du chemin :

```
$ ls -al /bin/gzip
```

```
-rwxr-xr-x  1 root root 47624 Jul 30 18:13 /bin/gzip
$ ls -al /bin/gunzip
lrwxrwxrwx  1 root root 4 Aug 18 17:11 /bin/gunzip -> gzip
$ ls -al /bin/zcat
lrwxrwxrwx  1 root root 4 Aug 18 17:11 /bin/zcat -> gzip
```

8.5 Communication entre/avec les processus : Les signaux

```
$ man signal // explications sur les signaux
$ kill -l    // liste des signaux
```

- un signal provoque une interruption
  ex : Ctrl + C : SIGINT
  Ctrl + Z : SIGTSTP
  kill xxx : SIGTERM => ne peut pas être ignoré
  kill -9 xxx : SIGKILL => ne peut pas être ignoré
- on peut associer un "signal handler", un gestionnaire pour traiter le signal

sign_handler_exemple.c

```c
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

// norme ISO/ANSI du langage C de 1989 :
//  "Seul le type sig_atomic_t garantit l'atomicité de l'accès à l'objet"
sig_atomic_t sigusr1_count = 0;

void handler (int signal_number) {
    printf("reception du signal %d\n", signal_number);
    ++sigusr1_count;
}

// quelque chose de long
int fib (int i) {
    if (i == 0)
        return 0;
    else if (i == 1)
        return 1;
    else
        return fib(i-1) + fib(i-2);
}

int main () {
    // struct sigaction
    // {
    //   void     (*sa_handler)(int); /* Adresse du gestionnaire */
    //   sigset_t sa_mask;            /* Masque des signaux bloques pendant
    //                                 * l'exécution du gestionnaire */
    //   int      sa_flags;           /* Ignore pour l'instant */
    // }

    struct sigaction sa;
    sa.sa_handler = &handler; // gestionnaire de signal à utiliser
    sigemptyset(&sa.sa_mask); // on ne bloque pas de signaux spécifiques
    sa.sa_flags = 0;          // théoriquement ignoré
    sigaction (SIGINT, &sa, NULL);  // mise en place du gestionnaire
```

```
    /* Faire quelque chose de long ici.  */
    int i = 0;
    for (i = 0; i < 45; i++)
        printf("fib %d = %d\n", i, fib(i));

    printf ("SIGINT a été reçu %d fois\n", sigusr1_count);
    return 0;
}
```

Un processus peut envoyer un signal à un autre :

signal_sender.c

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

int main(int argc, char * argv[]){
  int pid = 0;

  if (argc > 1)
   pid = atoi(argv[1]);
  else {
   fprintf(stderr, "usage : %s [pid]\n", argv[0]);
   exit(1);
  }

  printf("J'envoie un signal SIGUSR1 au processus %d\n", pid);
  kill(pid, SIGUSR1);

  return 0;
}
```

signal_handler.c

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

void handler (int signal_number) {
  printf("reception du signal %d\n", signal_number);
}

int main(int argc, char * argv[]){
  int pid = getpid();
  printf("Je suis le processus %d\n", pid);
  printf("J'attends le signal SIGUSR1\n");

  struct sigaction sa;
  sa.sa_handler = &handler; // gestionnaire de signal à utiliser
  sigemptyset(&sa.sa_mask); // on ne bloque pas de signaux spécifiques
  sa.sa_flags = 0;          // théoriquement ignoré
  sigaction (SIGUSR1, &sa, NULL);  // mise en place du gestionnaire

  // boucle infinie
  while(1) {
   printf("#"); fflush(stdout);
   sleep(1);
```

```
  }

  return 0;
}
```

Lancer `signal_handler.c` puis lancer `signal_sender.c` avec le pid de l'autre processus en argument.

8.6 Communication entre/avec les processus : Les pipes

Échange de données entre les processus :
`int pipe(int pipefd[2]);`

pipe_exemple.c

```
#include <stdio.h>
#include <memory.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
 int pfd[2]; // to store pipe infos
 if (pipe(pfd) == -1) {
     printf("pipe failed\n");
     return 1;
 }

 int pid = fork();

  if (pid < 0) {
     printf("fork failed\n");
     return 2;
  }
  else if (pid == 0) { // child
     close(pfd[1]);    // close the unused write side
     dup2(pfd[0], 0);  // connect the read side with stdin
     close(pfd[0]);    // close the read side

     execlp("sort", "sort", (char *) 0); // execute sort command
     printf("sort failed");  // if execlp returns, it's an error
     return 3;
  }
  else { // parent
     close(pfd[0]);     // close the unused read side
     dup2(pfd[1], 1);   // connect the write side with stdout
     close(pfd[1]);     // close the write side

     execlp("ls", "ls", (char *) 0); // execute ls command
     printf("ls failed"); // if execlp returns, it's an error
     return 4;
  }
  return 0;
}
```

signaux/pipes : mécanismes lourds (temps de traitement des interruptions, mise en place d'un pipe, ...), pour communiquer entre différentes sous-tâches, les processus légers sont plus efficaces.