# Programmation Système

## Communication inter-processus

Emmanuel Bouthenot (emmanuel.bouthenot@u-bordeaux.fr)

Licence professionnelle ADSILLH - Université de Bordeaux

2018-2019

# Communication inter-processus - IPC

Ce que l'on nomme les IPC (**I**nter **P**rocessus **C**ommunication) recouvre 3 mécanismes de comunication entre processus :

1. Les files de messages (**message queues**)
2. La mémoire partagée (**share memory**)
3. Les sémaphores (**semaphores**)

# API des IPC

Sous UNIX, il existe deux APIs :

1. **System V IPC** (1983)
   - Plus compliquée à utiliser
   - Non fondée sur le principe de descripteurs de fichiers
2. **IPC Posix** (2000)
   - Également normée dans SuSv4
   - API moderne
   - Sous Linux, nécessite un noyau $>= 2.6.10$

# Files de messages - Message queues

- Ce mécanisme est utilisé pour envoyer ou recevoir des données entres processus.
- Les données ne sont pas envoyées à destination d'un processus en particulier.
- Mécanisme proche des tubes à la différence que les données ne sont pas lues comme un flux d'octets. On écrit ou on lit un message complet.

# Ouvrir une file de messages

```
mqd_t mq_open(const char *name, int flags);
mqd_t mq_open(const char *name, int flags, mode_t mode,
                                struct mq_attr *attr);

// Renvoie un descripteur de file de messages en cas de succès,
// Retourne -1 en cas d'échec avec errno positionnée
```

- **name** : nom de la file (doit commencer par un / et ne pas en comporter d'autres)
- **flags**, **mode**: arguments identiques à ceux de **open**
- **attr** : attributs de la file (nombre maximal de messages, taille maximal d'un message)

# Envoi d'un message dans une file

```
int mq_send(mqd_t mqdes, const char *buffer,
                size_t len, unsigned int prio);

int mq_timedsend(mqd_t mqdes, const char *buffer,
                    size_t len, unsigned int prio,
                    const struct timespec *abs_timeout);

// Renvoie 0 en cas de succès,
// Retourne -1 en cas d'échec avec errno positionnée
```

- Les messages seront dépilés dans l'ordre décroissant de priorité (priorité 0 en dernier)
- Si la file est pleine, l'appel à **mq_send()** reste bloquant jusqu'à ce qu'il y'ai de la place libre
- La version temporisée (**mq_timedsend()** échoue si le message n'a pas pu être mis en file avant un instant donné

# Lecture d'un message dans une file

```
ssize_t mq_receive(mqd_t mqdes, char *buffer,
                   size_t len, unsigned int *prio);

ssize_t mq_timedreceive(mqd_t mqdes, char *buffer,
                        size_t len, unsigned int *prio,
                        const struct timespec *abs_timeout);

// Renvoie le nombre d'octets du message reçu en cas de succès,
// Retourne -1 en cas d'échec avec errno positionnée
```

- Le **buffer** doit est assez grand pour contenir le plus grand message que la file puisse transporter
- Cette dimension est accessible / modifiable avec **mq_getattr** et **mq_setattr**.

# Fermeture / Suppression d'une file de messages

```c
int mq_close(mqd_t mqdes);

// Renvoie 0 en cas de succès,
// Retourne -1 en cas d'échec avec errno positionnée

int mq_unlink(const char *name);

// Renvoie 0 en cas de succès,
// Retourne -1 en cas d'échec avec errno positionnée
```

- Même après fermeture, une file reste persistante, à la disposition des autres processus
- Après suppression d'une file, les processus qui ont déjà ouvert cette file peuvent continuer à l'utiliser, son contenu sera supprimé lorsque tous les processus qui utilisent cette file l'auront fermé

# Envoi de messages dans une file - exemple (1)

```c
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char **argv)
{
    mqd_t mq;
    int  priority;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s queue priority message\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (sscanf(argv[2], "%d", &priority) != 1) {
        fprintf(stderr, "Invalid priority: %s\n", argv[2]);
        exit(EXIT_FAILURE);
    }
    if ((mq = mq_open(argv[1], O_WRONLY | O_CREAT, 0644, NULL)) == (mqd_t) -1)
        perror("Opening message queue failed");
        exit(EXIT_FAILURE);
    }
    if (mq_send(mq, argv[3], strlen(argv[3]), priority) == -1) {
```

# Envoi de messages dans une file - exemple (2)

```
        perror("Unable to send message to queue");
        exit(EXIT_FAILURE);
    }
    if (mq_close(mq) == -1) {
        perror("Unable to close queue");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

# Lecture de messages dans une file - exemple (1)

```c
#include <fcntl.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char * argv[])
{
    int     n;
    mqd_t   mq;
    struct mq_attr attr;
    char * buffer = NULL;
    unsigned int priority;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s queue\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((mq = mq_open(argv[1], O_RDONLY)) == (mqd_t) -1) {
        perror("Opening message queue failed");
        exit(EXIT_FAILURE);
    }
    if (mq_getattr(mq, &attr) != 0) {
        perror("Unable to get message queue attributes");
```

# Lecture de messages dans une file - exemple (2)

```
        exit(EXIT_FAILURE);
    }
    if ((buffer = (char *) malloc(attr.mq_msgsize)) == NULL) {
        perror("Unable to allocate memory");
        exit(EXIT_FAILURE);
    }
    if ((n = mq_receive(mq, buffer, attr.mq_msgsize, &priority)) < 0) {
        perror("Unable to receive message from queue");
        exit(EXIT_FAILURE);
    }
    if (mq_close(mq) == -1) {
        perror("Unable to close queue");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "[%d] %s\n", priority, buffer);
    free(buffer);
    return EXIT_SUCCESS;
}
```

# Files de messages - Démo

```
> gcc -Wall -lrt mq-send.c -o mq-send
> ./mq-send /plop 10 "Msg 10"
> ./mq-send /plop 30 "Msg 30"
> ./mq-send /plop 20 "Msg 20"
> ./mq-recv /plop
[30] Msg 30
> ./mq-recv /plop
[20] Msg 20
> ./mq-recv /plop
[10] Msg 10
```

# Mémoire partagée - Shared memory

- Le débit d'une file de messages est assez limité car 2 appels systèmes (**mq_send()** et **mq_receive()** sont nécessaires pour communiquer.
- La mémoire partagée est un mécanisme beaucoup plus rapide qui permert de partager une zone mémoire entre deux ou plusieurs processus.
- Le partage de mémoire entre plusieurs processus consiste à ouvrir un segment de mémoire avec **shm_open()** et à la projeter dans l'espace mémoire du processus avec **mmap()**

# mmap / munmap

**mmap()** crée une nouvelle projection dans l'espace d'adressage virtuel du processus appelant. L'adresse de démarrage de la nouvelle projection est indiquée dans addr. Le paramètre length indique la longueur de la projection.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
                int fd, off_t offset);

//  mmap() renvoie un pointeur sur la zone de mémoire
// En cas d'échec il retourne la valeur MAP_FAILED et
// errno contient le code d'erreur.

int munmap(void *addr, size_t length);

// munmap() renvoie 0 s'il réussit. En cas d'échec, -1 est
//renvoyé et  errno contient le code d'erreur.
```

# mmap / munmap (suite)

- **prot** : PROT_EXEC, PROT_READ, PROT_WRITE ou PROT_NONE (pas d'accès)
- **flags** : MAP_SHARED (partage entre plusieurs processus), MAP_PRIVATE (les modifications de la projection ne sont pas visibles depuis les autres processus projetant la zone mémoire), etc.
- **fd** : Descripteur de fichier
- **offset** : décalalage dans le fichier pointé par **fd**

# Ouverture / fermeture d'un segment de mémoire partagé

```c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>

int shm_open(const char *name, int flags, mode_t mode);

// retourne un descripteur de fichier non négatif ou
// -1 en cas d'échec

int shm_unlink(const char *name);

// renvoie 0 s'il réussit ou -1 en cas d'erreur
```

- **name** : nom de la file (doit commencer par un / et ne pas en comporter d'autres)
- **flags**, **mode**: arguments identiques à ceux de **open**

# Accès à un segment de mémoire partagé - exemple (1)

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>

int main (int argc, char **argv)
{
    int fd;
    long int *counter;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s segment\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = shm_open(argv[1], O_RDWR | O_CREAT, 0600)) == -1) {
        perror("Opening shared memory segment failed");
        exit(EXIT_FAILURE);
    }
    if (ftruncate(fd, sizeof(long int)) != 0) {
        perror("Unable to truncate shared memory segment");
        exit(EXIT_FAILURE);
    }
```

# Accès à un segment de mémoire partagé - exemple (2)

```
    counter = mmap(NULL, sizeof(long int), PROT_READ | PROT_WRITE, MAP_SHARED,
    if (counter == MAP_FAILED) {
        perror("Unable to map shared memory segment");
        exit(EXIT_FAILURE);
    }
    for(long int i=0; i< 1000000000; i++) {
        (*counter)++;
    }
    fprintf(stdout, "counter=%ld\n", (*counter));
    return EXIT_SUCCESS;
}
```

# Accès à un segment de mémoire partagé - Démo

```
> gcc -Wall -lrt shm-inc.c -o shm-inc
> ./shm-inc /foobar
counter=1000000000
> ./shm-inc /foobar
counter=2000000000
> ls -l /dev/shm
-rw------- 1 user group 8 Nov 15 00:00 foobar
> cat /dev/shm/foobar
5w
> hexdump /dev/shm/foobar
0000000 9400 7735 0000 0000
0000008
> echo "7*16^7 + 7*16^6 + 3*16^5 + 5*16^4 +
              9*16^3 + 4*16^2 + 0*16^1 + 0*16^0" | bc
2000000000
> rm -f /dev/shm/foobar
> ./shm-inc /foobar & ./shm-inc /foobar
counter=1004416125
```

# Sémaphores

- Les sémaphores permettent de synchroniser des processus et d'organiser l'accès concurrent à des ressources partagées
- Un sémaphore est un entier dont la valeur ne doit jamais descendre en dessous de zéro. Deux opérations peuvent être effectuées sur un sémaphore :
  - Incrémenter la valeur du sémaphore : **V()** (du néerlandais Verhogen, Dijkstra)
  - Décrémenter la valeur du sémaphore : **P()** (Proberen)
- Si la valeur d'un sémaphore est zéro, une opération **P()** bloquera jusqu'à ce que la valeur devienne supérieure à zéro
- Les sémaphores POSIX ont deux formes possibles : les sémaphores anonymes et les sémaphores nommés.

# Sémaphores anonymes

- Il est placé dans une région de la mémoire qui est partagée entre plusieurs threads ou processus
- Un sémaphore partagé entre threads est placé dans une zone mémoire partagée entre les threads d'un processus à travers par exemple une variable globale
- Un sémaphore partagé entre processus doit être placé dans une région mémoire partagée à travers par exemple un segment de mémoire partagée POSIX

# Création / destruction d'un sémaphore anonyme

```c
#include <semaphore.h>

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

// Renvoie 0 en cas de succès
// Retourne -1 en cas d'échec avec errno positionnée

int sem_destroy(sem_t *sem);

// Renvoie 0 en cas de succès
// Retourne -1 en cas d'échec avec errno positionnée
```

- **value** : la valeur initiale du sémaphore

# Création / fermeture d'un sémaphore nommé

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

sem_t *sem_open(const char *name, int flags);
sem_t *sem_open(const char *name, int flags,
                mode_t mode, unsigned int value);

// Renvoie l'adresse du nouveau sémaphore en cas de succès
// Retourne SEM_FAILED en cas d'échec avec errno positionnée

int sem_close(sem_t *sem);

// Renvoie 0 en cas de succès
// Retourne -1 en cas d'échec avec errno positionnée
```

- **name** : nom du sémahpore
- **flags**, **mode**: arguments identiques à ceux de **open**
- **value** : la valeur initiale du sémaphore

# Opérations sur les sémaphores

```c
#include <semaphore.h>

#include <semaphore.h>

int sem_wait(sem_t *sem);

// Renvoie 0 en cas de succès
// Retourne -1 en cas d'échec avec errno positionnée

int sem_post(sem_t *sem);

// Renvoie 0 en cas de succès
// Retourne -1 en cas d'échec avec errno positionnée
```

Incrémenter la valeur du sémaphore se fait avec un **sem_post()** tandis décrémenter se fait avec **sem_wait()**.

# Sémaphore versus Mutex : quelle différence ?

**Mutex**:
Is a key to a toilet. One person can have the key - occupy the toilet - at the time. When finished, the person gives (frees) the key to the next person in the queue.

Mutexes are typically used to serialise access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

**Semaphore**:
Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.

A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).

**The Toilet Example, 2005, © Niclas Winquist**

# Utilisation d'un sémaphore - exemple (1)

```c
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int i;
    sem_t *sem;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s sem_name\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    sem = sem_open(argv[1], O_RDWR | O_CREAT, 0666, 1);
    if (sem == SEM_FAILED) {
        perror("Unable to open semaphore");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "[%d] Sempahore %s created\n", getpid(), argv[1]);
    for (i = 0; i < 3;  i ++) {
        fprintf(stdout, "[%d] waiting...\n", getpid());
```

# Utilisation d'un sémaphore - exemple (2)

```
        sem_wait(sem);
        fprintf(stdout, "\t[%d] semaphore locked\n", getpid());
        sleep(4);
        fprintf(stdout, "\t[%d] semaphore released\n", getpid());
        sem_post(sem);
        sleep(2);
    }
    return EXIT_SUCCESS;
}
```

# Utilisation d'un sémaphore - Démo

```
> gcc -pthread -lrt -Wall sem-lock.c -o sem-lock
> ./sem-lock foobar
    & ./sem-lock foobar
    & ./sem-lock foobar
> ls -l /dev/shm
-rw-r--r-- 1 user group 32 Nov 15 00:00 sem.foobar
```

```
[16597] Sempahore foobar created
[16597] waiting...
        [16597] semaphore locked
[16599] Sempahore foobar created
[16599] waiting...
[16600] Sempahore foobar created
[16600] waiting...
        [16597] semaphore released
        [16599] semaphore locked
[16597] waiting...
        [16599] semaphore released
        [16600] semaphore locked
[16599] waiting...
        [16600] semaphore released
        [16597] semaphore locked
[16600] waiting...
        [16597] semaphore released
        [16599] semaphore locked
[16597] waiting...
        [16599] semaphore released
        [16600] semaphore locked
[16599] waiting...
        [16600] semaphore released
        [16597] semaphore locked
[16600] waiting...
        [16597] semaphore released
        [16599] semaphore locked
        [16599] semaphore released
        [16600] semaphore locked
        [16600] semaphore released
```

# Bibliographie


[**APUE**] Advanced Programming in the UNIX Environment.
W. Richard Stevens and Stephen A. Rago.
*Addison-Wesley Professional*, 2005.

[**TLPI**] The Linux Programming Interface.
Michael Kerisk.
*No Starch Press*, 2010.