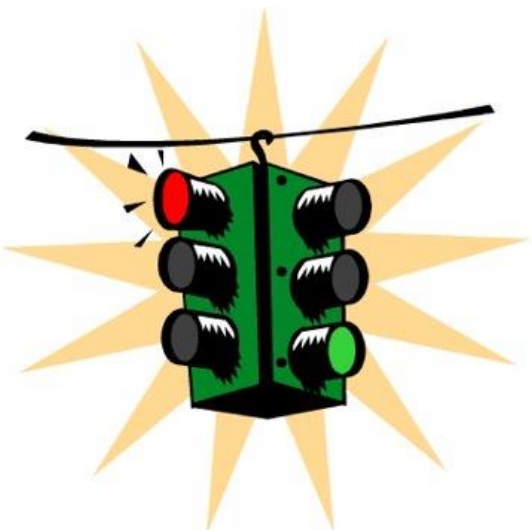
Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes d'exploitation 2

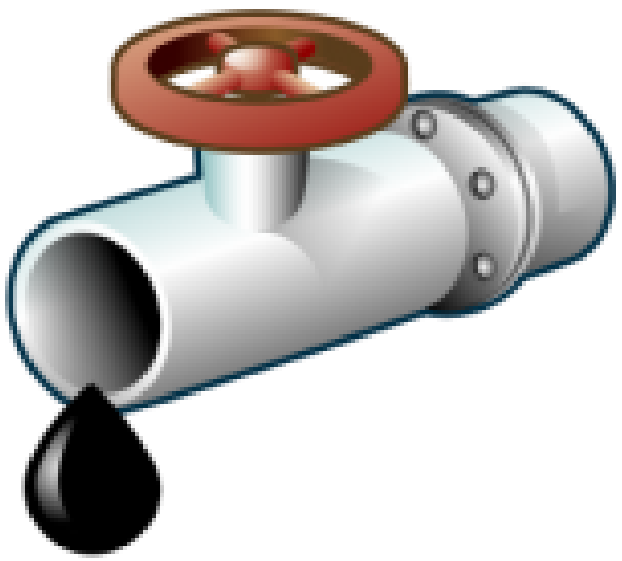
IPC: Inter-Process Communication

IPC: Inter-Process Communication

- Tubes nommés (FIFO)
- Mémoire partagée
- Sémaphores



Tube nommés



Tubes anonymes vs. tubes nommés

Tubes:

- FIFO: L'ordre des caractères en entrée est conservé en sortie
- L'opération de lecture est destructive !
- Tube unidirectionnel : pour pouvoir communiquer dans les 2 sens avec 2 processus, il faut 2 tubes

Tubes anonymes (anonymous pipe):

- non visibles dans l'arborescence
- entre processus de même "parenté" (`fork()`)

Tubes nommés (named pipe)

- visibles dans l'arborescence
- entre processus pas forcément liés par filiation

Tubes nommés

Tubes nommés:

- Mécanisme de communication entre plusieurs processus qui connaissent le **nom du tube**.
- Peuvent être lus ou écrits comme des **fichiers ordinaires**:
    - en utilisant les primitives systèmes (bas niveau) :
      `read, write,…`
    - en utilisant les fonctions de la librairie (haut niveau):
      `fprintf,…`

L'écriture est atomique si le nombre de caractères à écrire est inférieur à `PIPE_BUF`, qui est la taille du tube sur le système.
`PIPE_BUF` peut être définit dans `<limits.h>`.
`PIPE_BUF` vaut au minimum 512 (dans le standard POSIX)

Tubes nommés: bloquant / non-bloquant

- En **mode bloquant** (mode par défaut):
  - la lecture depuis un tube vide est bloquante (attente de données)
  - l'écriture dans un tube plein est bloquante (attente d'une possibilité d'écriture)
  - Il y a automatiquement synchronisation des processus qui ouvrent en mode bloquant un tube nommé.
- En **mode non bloquant** (O_NONBLOCK), seule l'ouverture en lecture réussit dans tous les cas.
  L'ouverture en écriture en mode non bloquant d'un tube nommé ne fonctionne que si un autre processus a déjà ouvert en mode non bloquant le tube en lecture.

Tube nommé: création/destruction

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

int unlink(const char *pathname);

/* returns 0 on success or
    -1 on error (sets errno) */
```

`mode` : permission en lecture et écriture.

La fonction `unlink()` ne fait que détruire la référence, le tube n'est réellement détruit que lorsque son compteur de références est nul.

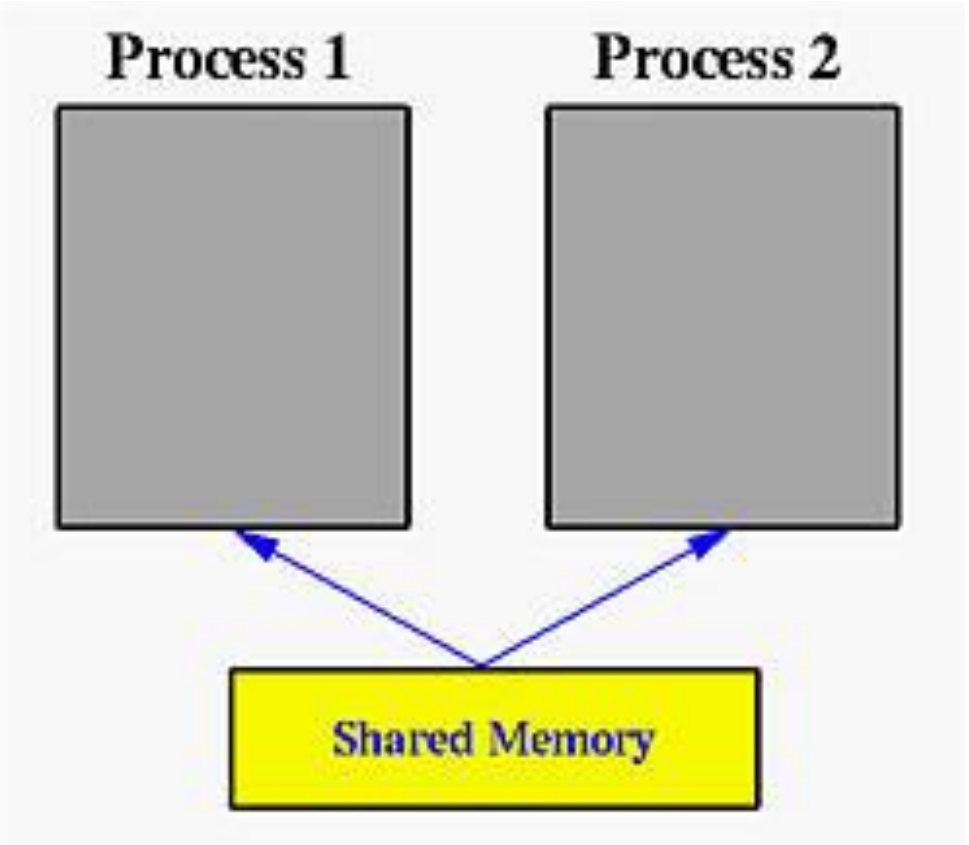
Tube nommé: ouverture

```
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *filename, int flags);

/* Exemples:  */
/* en lecture seule */
open("fifo", O_RDONLY);

/* en écriture seule */
open("fifo", O_WRONLY);

/* lecture non bloquante: réussit à tous les coups */
open("fifo", O_RDONLY|O_NONBLOCK);

/* écriture non-bloquante: réussit ou échoue */
open("fifo", O_WRONLY|O_NONBLOCK);
```

Tube nommé: exemple

```
  #define fifo_name "myFifo"
...
  // open pipe
  if(mkfifo(fifo_name, 0666)==-1) {
        perror("Impossible de créer le pipe avec mkfifo...");
        exit(EXIT_FAILURE);
  }

  pid_t pid = fork();
  else if(pid != 0)   {  // parent process reads pipe
        int fdRead;
        fdRead=open(fifo_name, O_RDONLY);
        char c;
        read(fdRead, (void *)&c, 1);
        printf("%c\n", c);
        close(fdRead);
        unlink(fifo_name);
  }
  else    { // child processe writes on pipe
        char c='a';
        int fdWrite;
        fdWrite=open(fifo_name, O_WRONLY);
        write(fdWrite, (void *)&c, 1);
        close(fdWrite);
        unlink(fifo_name);
  }
```

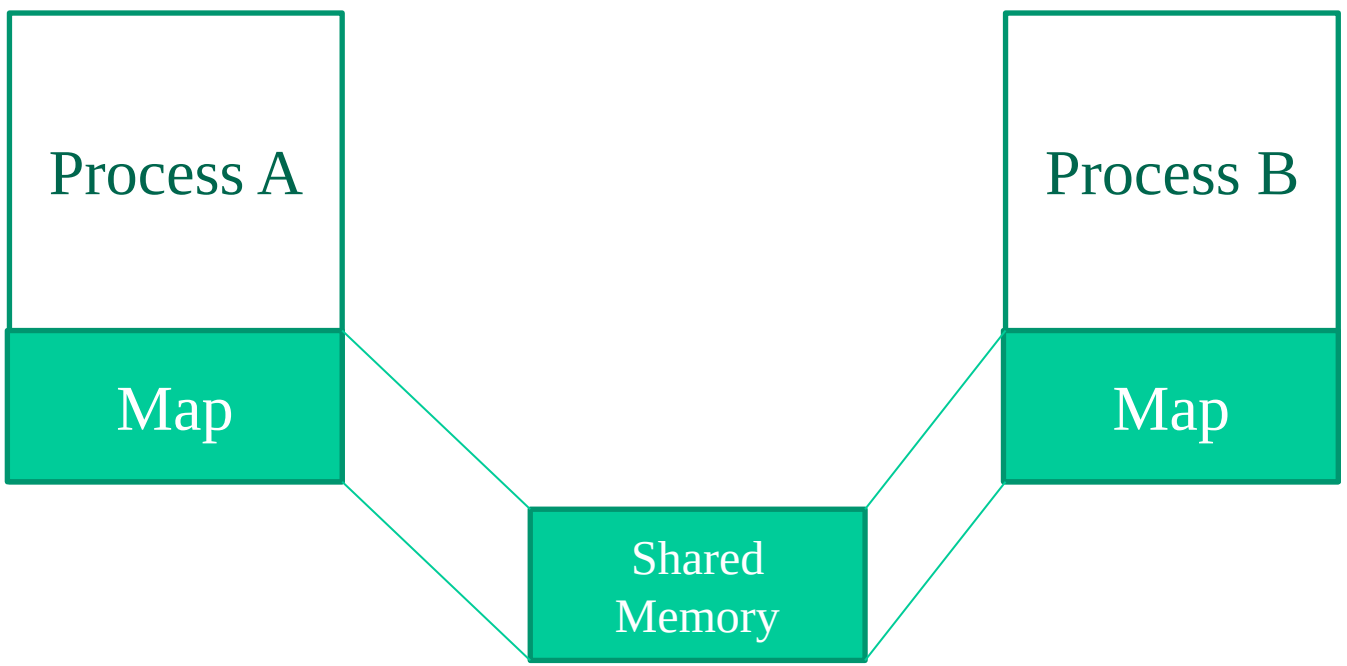
Mémoire partagée



Mémoire partagée (shared memory)

Il est possible de définir une zone (segment) de **mémoire partagée** entre plusieurs processus

⇨ Un segment peut faire partie de l'espace d'adressage de plusieurs processus.



Mémoire partagée (shared memory)

Pour partager un segment de mémoire entre 2 processus **P1** et **P2** , il faut procéder aux étapes suivantes:

- **P1** (ou **P2)** doit créer un segment de mémoire partagé **S**
- **P1** doit attacher **S** dans son espace d'adressage
- **P2** doit attacher **S** dans son espace d'adressage

⇨ le segment **S** est partagé: même zone en mémoire physique

Mémoire partagée (POSIX)

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

/* créer ou ouvrir un segment */
int shm_open(const char *name, int oflag, mode_t mode);
/* returns file descriptor or -1 on error (sets errno) */

/* détruire un segment */
int shm_unlink(const char *name);
/* returns 0 on success or -1 on error (sets errno) */
```

- Les `oflag` sont décrits au slide suivant.
- Il faut compiler avec l'option `-lrt` pour avoir l'édition des liens avec la bibliothèque temps-réel `librt`.
- Les segments de mémoire partagée apparaissent dans l'arborescence des fichiers, par exemple sous `/dev/shm/`

Mémoire partagée: oflags

Le flag `oflag` est définit avec les flags utilisés pour l'ouverture des fichiers:

- `O_RDONLY`: ouvrir l'objet en lecture seule.
- `O_RDWR`: ouvrir l'objet en lecture et écriture.
- `O_CREAT`: crée l'objet de mémoire partagée s'il n'existe pas.
- `O_EXCL`: si `O_CREAT` était précisé et si un objet de mémoire partagée avec le même nom existait déjà, renvoie une erreur.

Mais on ne peut pas utiliser `O_WRONLY` (pas défini pour la mémoire partagée)

Mémoire partagée: `ftruncate`

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

int ftruncate(int fd, off_t length);
/* returns 0 on success or -1 on error (sets
errno) */
```

Tronque un fichier à la taille donnée

⇨ ne s'utilise pas que pour la mémoire partagée!

Mapping de la mémoire partagée

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

/* attacher un segment dans l'espace du processus */
void *mmap( void *start, size_t length,
            int prot,   int flags,
            int fd,     off_t offset);

/* returns pointer to segment or MAP_FAILED on error
(sets errno) */

/* détacher un segment de l'espace du processus */
int munmap(void *start, size_t length);
/* returns 0 on success or -1 on error (sets errno) */
```

`prot` définit la protection lors du mapping:

- `PROT_NONE` : mémoire inaccessible
- `PROT_READ`: peut être lu
- `PROT_WRITE`: peut être écrit
- `PROT_EXEC`: peut être exécuté

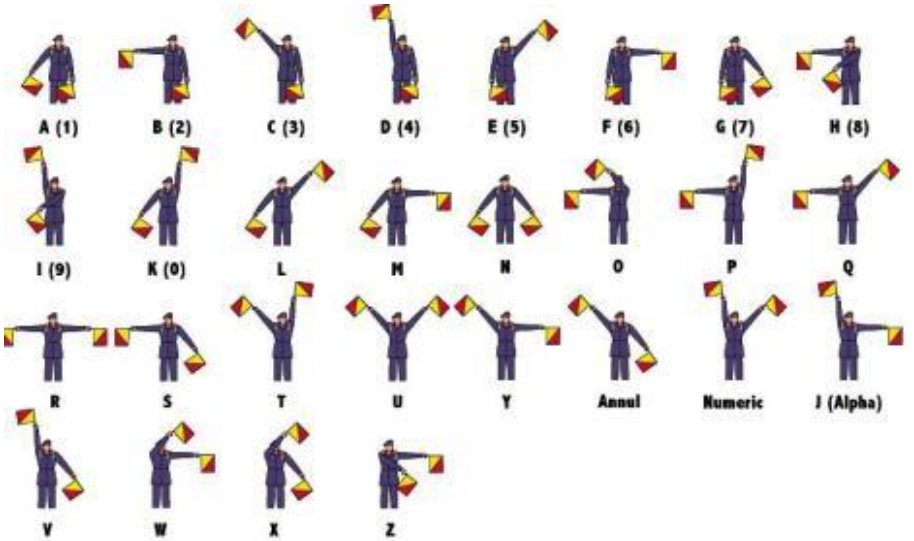
```c
/* Créer le segment MEMORY_SEGMENT, accès R/W */
int mfd = shm_open(MEMORY_SEGMENT, O_CREAT|O_RDWR, 0777);
if (mfd == -1) {
      perror("opening/creating shared memory segment");
      exit(1);}

/* Allouer au segment une taille prédéfinie */
if (ftruncate(mfd, SH_SIZE) == -1) {
      perror("setting memory size");
      exit(1);}

/* Mapper le segment partagé en R/W */
void *maddr = mmap(NULL, SH_SIZE, PROT_READ | PROT_WRITE,
                   MAP_SHARED, mfd, 0);
if (maddr == MAP_FAILED) {
      perror("setting up the mapping");
      exit(1);}
char *c = maddr;
char *msg="My message";
memcpy (c, msg, strlen(msg)+1);

/* détacher le segment */
if (munmap(maddr, SH_SIZE) == -1) perror("unmapping segment");

/* détruire le segment */
if (shm_unlink(MEMORY_SEGMENT) == -1) perror("unlinking segment");
```

Exemple d'accès à la mémoire partagée

```
typedef struct mainMemoryStruct
{
        int myValue1;
        int myValue2;
        char myFlag;
} MyStruct;
```

```
#define MY_SHMEM "/sharedMem1"
…
MyStruct * myStructure;
int shmem_fd = shm_open(MY_SHMEM, O_CREAT|O_RDWR, 0777);
int myStructureSize = sizeof(MyStruct);
if (shmem_fd == -1) {
        perror("opening/creating shared memory");
        return;
}
myStructure = (MyStruct*)mmap(NULL, myStructureSize ,
                                    PROT_READ | PROT_WRITE,
                                    MAP_SHARED, shmem_fd, 0);
if (myStructure == MAP_FAILED) {
        perror("setting up the mapping");
}
```

Sémaphores


A (1)
B (2)
C (3)
D (4)
E (5)
F (6)
G (7)
H (8)
I (9)
K (0)
L
M
N
O
P
Q
R
S
T
U
Y
Annul
Numeric
J (Alpha)
V
W
X
Z

Sémaphores

Plusieurs implémentations des sémaphores sous UNIX :

- Les ***MUTEX POSIX***: sémaphores binaires (LOCKED ou UNLOCKED). Ils permettent de synchroniser des threads appartenant à un même processus.
- Les ***sémaphores POSIX***: sémaphore à compteur. Ils permettent de synchroniser des processus différents.
- Les ***sémaphores UNIX*** : tableaux de sémaphores. Les opérations peuvent alors s'exécuter de manière atomique sur un ou plusieurs sémaphores => d'allocations simultanées de plusieurs ressources différentes.

Sémaphore

Un sémaphore est une variable entière partagée, positive, et accessible au travers de deux opérations atomiques:

- ***P(S)*** : ("*Prendre!*" ou abaisser le sémaphore)
  - si *S==0* alors mettre le processus en attente
  - sinon *S=S-1*
- ***V(S)*** : ("*Vas-y* !" ou libérer le sémaphore)
  - *S=S+1*
  - réveiller un processus en attente

Sémaphore POSIX

- Un **sémaphore POSIX** est un sémaphore à compteur pouvant être partagé par plusieurs threads de plusieurs processus
- Le compteur associé à un sémaphore peut prendre des valeurs plus grande que 1
- La prise d'un sémaphore dont le compteur est négatif ou nul bloque l'appelant.

Sémaphore anonymes POSIX

Un **sémaphore anonyme** n'a pas de nom. Il est placé dans une région de la mémoire qui est partagée entre:

- plusieurs threads ⇨ variable globale
- plusieurs processus ⇨ mémoire partagée

Opérations:

- `sem_init()`: initialiser le sémaphore
- `sem_post()`: libérer le sémaphore
- `sem_wait()`: abaisser le sémaphore
- `sem_destroy()`: destruction du sémaphore

Sémaphore nommés POSIX

Un sémaphore nommé est identifié par un nom (commençant par un '/'). Plusieurs processus peuvent utiliser un même sémaphore nommé en passant le même nom à `sem_open()`.

Opérations:

- `sem_open`(): créer / ouvrir
- `sem_post()`: libérer le sémaphore (incrémente)
- `sem_wait()`: abaisser le sémaphore (décrémente)
- `sem_close()`: fermer le sémaphore
- `sem_unlink()`: supprimer le sémaphore

Ouverture d'un sémaphore nommé POSIX

```
#include <semaphore.h>

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

sem_open() crée un nouveau sémaphore POSIX ou en ouvre un existant.

sem_open() retourne un pointeur sur le sémaphore ou **SEM_FAILED** en cas d'erreur.

Si **O_CREAT** est spécifié dans `oflag`, le sémaphore est créé s'il n'existe pas déjà.

Si **O_CREAT** est spécifié dans `oflag`, 2 autres arguments doivent être fournis:

- mode spécifie les permissions à placer sur le nouveau sémaphore
- value spécifie la valeur initiale du nouveau sémaphore.

Si **O_CREAT** est spécifié et que le sémaphore `name` existe déjà, `mode` et `value` sont ignorés.

Destruction d'un sémaphore nommé POSIX

```
#include <semaphore.h>

int sem_unlink(const char *name);

int sem_close(sem_t *sem);
```

- L'appel à **`sem_close()`** indique la fin de l'utilisation du sémaphore par le processus appelant.
- La fonction **`sem_unlink()`** détruit le sémaphore.
- La destruction d'un sémaphore n'est effective que lorsque tous les processus qui l'utilisent ont détruit le sémaphore par un appel à **`sem_unlink()`** (ou à `exit()` ou `exec()`).

Opérations sur les sémaphores POSIX

```
#include <semaphore.h>

/* Opération P() */
int sem_wait(sem_t *sem)

/* Opération P() en mode non bloquant */
int sem_trywait(sem_t *sem)

/* Opération V() */
int sem_post(sem_t *sem)

/* Récupère la valeur courante du sémaphore */
int sem_getvalue(sem_t *sem, int *valeur) :
```

```
…
#include <semaphore.h>

#define SEMFILE "/testsem"

int main(void){
    sem_t *sem;
    pid_t pid;
    int val, i;

    if ((sem = sem_open(SEMFILE, O_CREAT, 0666, 2)) == SEM_FAILED ) { exit(1); }
    if ((pid = fork()) == 0) {
        while (1) {
            printf(" child waiting...\n");
            if ( sem_wait(sem) == -1 ) { exit(2); }
            if( sem_getvalue(sem, &val) == -1 ) { exit(2); }
            printf(" child got semaphore; value = %d\n", val);
        }
    }
    printf("parent ready for post loop\n");
    for (i = 0; i < 3; i++) {
        printf("parent is posting\n");
        if( sem_post(sem) == -1) { exit(2); }
        sleep(1);
    }
    sem_close(sem);
    sem_unlink(SEMFILE);
    kill(pid, SIGKILL);
    waitpid(pid, NULL, 0);
    return EXIT_SUCCESS;
}
```

```
$ ./semtest
parent ready for post loop
parent is posting
  child waiting...
  child got semaphore; value = 2
  child waiting...
  child got semaphore; value = 1
  child waiting...
  child got semaphore; value = 0
  child waiting...
parent is posting
  child got semaphore; value = 0
  child waiting...
parent is posting
  child got semaphore; value = 0
  child waiting...
```