INSA
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
CENTRE VAL DE LOIRE

Module système II
Les Inter Process Communication (IPC)

Support de cours 2

3Ème année INSA CVL

Département STI

2017-2018

M. Szpieg

Concept des IPC système V

- Mécanisme de communication entre processus
- Ils permettent de partager des données, de synchroniser des processus entre eux
- Complètent les signaux et tubes qui comportent un certain nombre de limitations

Les mécanismes au nombre de trois

- Files de messages (Message Queues) système de boîte à lettres permettant à un processus d ’y déposer un ou plusieurs messages et à d’autres de venir les lire.
- Mémoire partagée
- Sémaphores

système V

La gestion des clefs system V

- Pour qu'un processus accède à un IPC il faut qu'il obtienne une clef (numéro unique qui identifie l 'IPC)
- Cette clef peut être générée manuellement ou par l'utilisation de la fonction ftok() « File To key »
- La fonction « ftok » permet de générer la clef

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

key_t ftok(char *pathname, char proj)
```

Les droits d’accès

- IPC possède des droits d’accès pour assurer la confidentialité
- Chaque IPC possède une structure ipc_perm contenant les informations de permission

```
struct ipc_perm {
ushort uid; /* owner’s user id */
ushort gid; /* owner’s group id */
ushort cuid;/* creator’s user id */
ushort cgid; /* creator’s group id */
ushort mode; /* access modes */
ushort seq; /* slot usage seq. number */Identification de la ressource
key_t key; /* key */ };
```

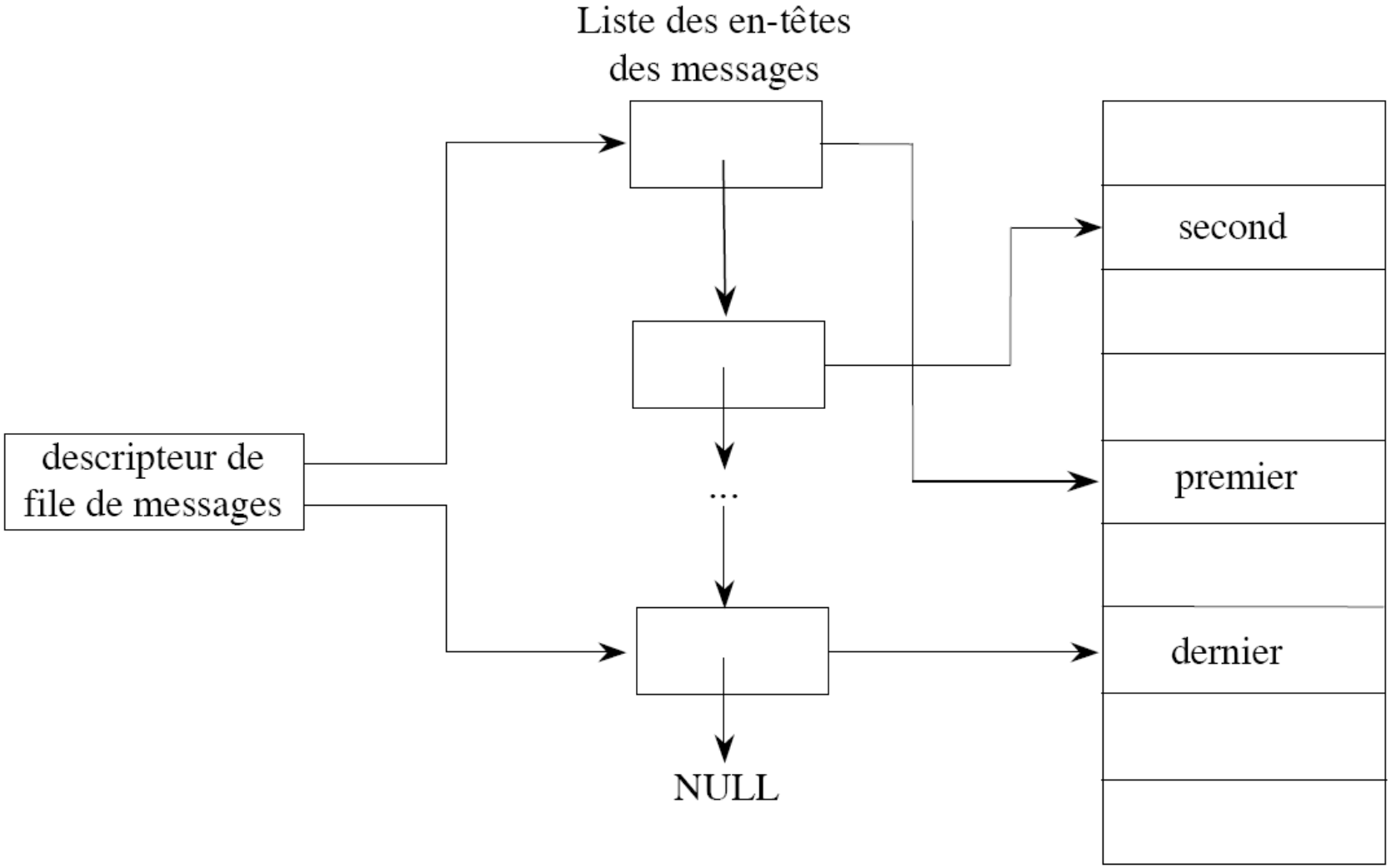
Les catégories d’appels système

- Création : msgget,semget,shmget
  Attention la ressource créée est gérée par le noyau, ainsi si le processus se termine, la ressource peut toujours exister. (utilisation des commandes ipcs et ipcrm à partir de l’interpréteur de commande pour les gérer).

- Contrôle : msgctl, semctl, shmctl , semop, shmop
- Communication : msgsnd, msgrcv

système V

Les files de messages



Les files de messages

- Principe : liste chaînée de messages en mémoire accessibles par plusieurs processus.
- Le message doit comporter à son début, une étiquette (priorité sous la forme d'un entier long), cette étiquette est appelée « type » puis sa structure est définie par le programmeur.
- L'accès à ces messages s'effectue de manière FIFO mais la notion de priorité (étiquette) est traitée.
- Le nombre de messages simultanés est limité à MSGMAX. MSBMNB donne le nombre maximun d'octets dans la file /usr/include/linux/msg.h

Les files de messages

- basées sur du FIFO
- Pas d'accès concurrents à une même donnée
- 2 recopies complètes par msg : expéditeur -> cache système ->destinataire

msgget() 2 rôles :

- Création d ’une nouvelle file de messages
- Recherche d’une file de messages existante (créée par un autre processus) grâce à sa clef

Arguments de msgget()

- int msgget (key_t clef, int option)
- 2 cas :
  - si la clef n'est pas utilisée par un autre processus, et si IPC_CREAT est dans l'argument option, alors la file est créée
  - si la clef est utilisée par un autre processus et si les droits d 'accès sont permis, alors on peut lire et écrire.
  - Remarque si l'option contient IPC_CREAT et IPC_EXCL alors la file est créée si la clef n'est pas en utilisation sinon la fonction échoue.
- La fonction renvoie un identificateur de file de messages.

Contrôle des files de messages

- int msgctl (int msqid, int cmd ,struct msquid_ds *buf)

  -msqid identificateur de file
  -cmd opérations à effectuer :

  - MSG_STAT, copie la table associée à la file messages à l'adresse *buf
  - IPC_SET, possible d'accéder à trois champs de la structure msg_perm.uid, msg_perm.gid et msg_perm.mode etc.
  - IPC_RMID, destruction de la file de messages

Structure msqid_ds

```
Struct msqid_ds {
struct ipc_perm msg_perm;/* operation permission struct */
struct msg *msg_first; /* ptr to first message on q */
struct msg *msg_last; /* ptr to last message on q */
ushort msg_cbytes; /* current # of bytes on q */
ushort msg_qnum; /* # of messages on q */
ushort msg_qbytes; /* max # of bytes on q */
ushort msg_lspid; /* pid of last msgsnd */
ushort msg_lrpid; /* pid of last msgrcv */
time_t msg_stime; /* last msgsnd time */
time_t msg_rtime; /* last msgrcv time */
time_t msg_ctime; /* last change time */
};
```

Émission de messages

- int msgsnd(int msqid, struct msgbuf *msgp, int msgtaille, int msgopt);
  - msgbuf structure à définir, le premier champ est le type de message **obligatoirement** sous la forme d'un entier long.
  - msgtaille taille des données à envoyer (!! sans l'entier long)
  - si msgopt=IPC_NOWAIT alors l'appel n'est pas bloquant.

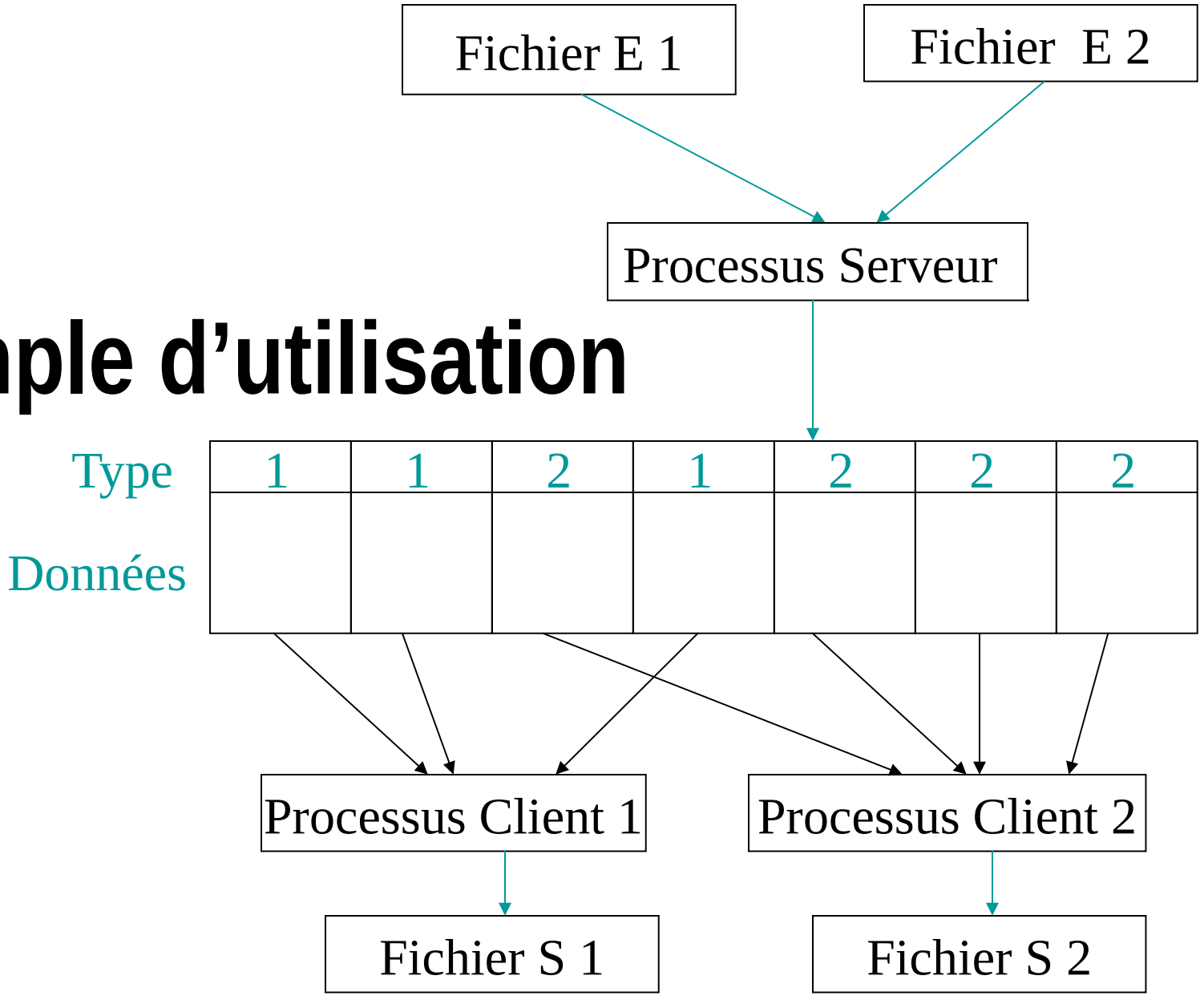
- Remarque : pour que l'envoi soit possible, il faut posséder les droits d'écrire dans la file de messages

Réception de messages

- int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);
  - message récupéré à l'adresse *msgp
  - taille de la zone mémoire de récupération msgsz
  - msgtyp type de message à récupérer, c'est le premier champ de la structure msgbuf qui indique ce type
- - deux options dans msgflg
    ❶MSG_NOERROR si la taille du message > msgsz alors le message est tronqué
    ❷ IPC_NOWAIT, si la file est vide, l'erreur ENOMSG est retournée sinon l'appel est suspendu jusqu'à l'arrivée d'une donnée

système V

Exemple d'utilisation



Sémaphores

- Gestion par ensemble de sémaphores
- Structure d'un sémaphore
  – un compteur : nombre d'accès disponibles avant blocage
  – une file d’attente : processus bloqués en attente d'un accès
- - opération P « P - proberen traduction puis-je ? » (test compteur si >0 alors décrémentation et utilisation de la ressource sinon attend)
  - Opération V « V - verhogen traduction vas-y» (incrémente le compteur et libère la ressource)

système V

Sémaphores

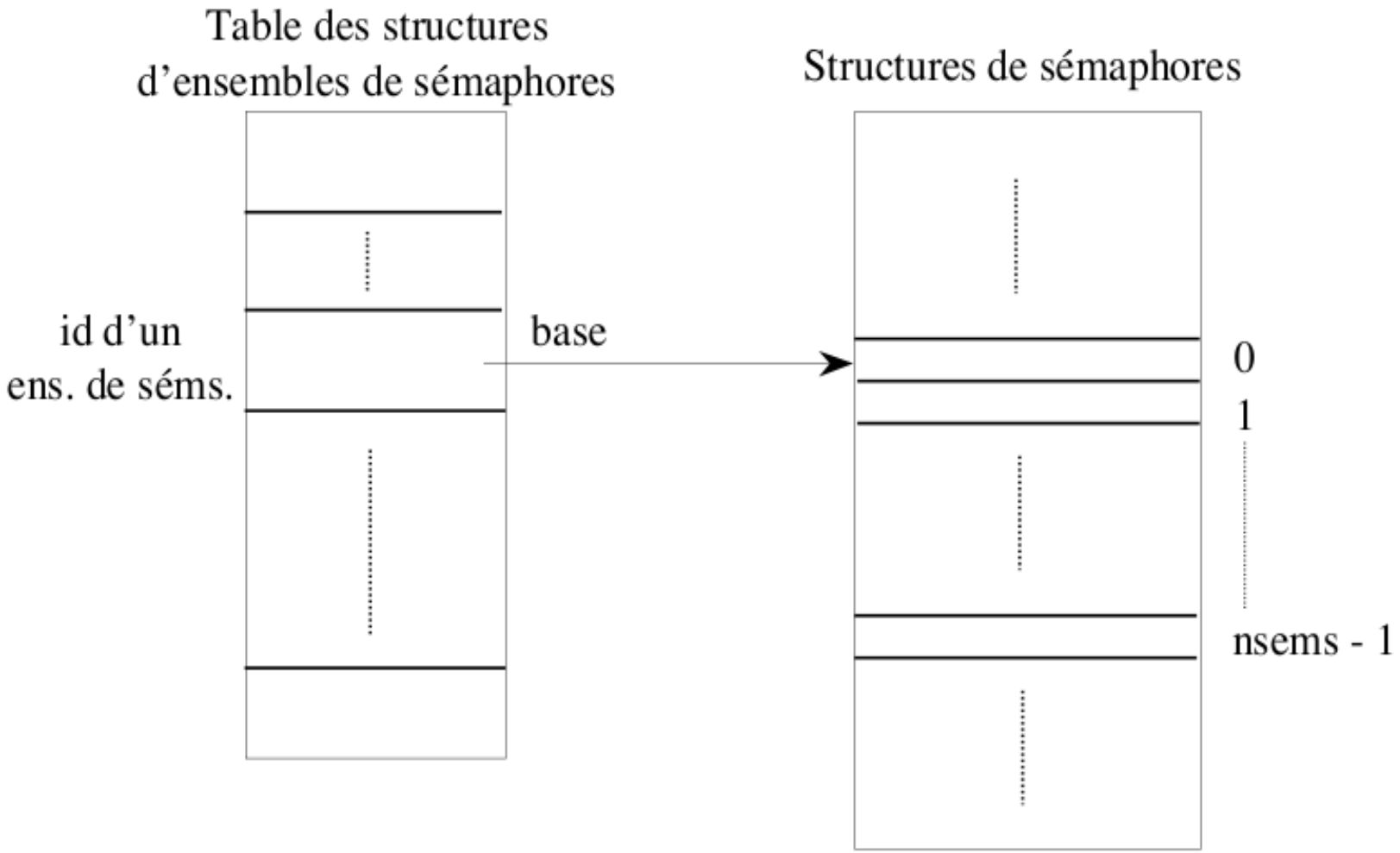


Table des structures
d'ensembles de sémaphores
Structures de sémaphores
id d'un
ens. de séms.
base
0
1
nsems - 1

Struct liée au sémaphore

Chaque sémaphore dans un ensemble de sémaphores se voit associer les valeurs suivantes :

```
unsigned short  semval;   /* valeur du sémaphore   */
unsigned short  semzcnt;  /* # Attente pour zéro   */
unsigned short  semncnt;  /* # Attente d'incrément */
pid_t           sempid;   /* dernier processus agissant */
```

Appel système relatif aux sémaphores

- int semget (key_t key, int nsems, int semflg);
  idem à msgget mais pour un tableau de sémaphores
  - nsems : nombre de sémaphores

Opérations sémaphores

int semop (int semid, struct sembuf *sops, unsigned nsops);

- nsops nombre d'opérations à effectuer
- sembuf structure contenant un ensemble de sémaphores

Chaque élément dans le tableau pointé par sops indique une opération à effectuer sur un sémaphore

```
struct sembuf{
unsigned short sem_num;  /* Num. du sémaphore*/
short   sem_op;   /* Opération sur le sémaphore */
short  sem_flg;  /* Options pour l'opération
   [IPC_NOWAIT,SEM_UNDO]*/ };
```

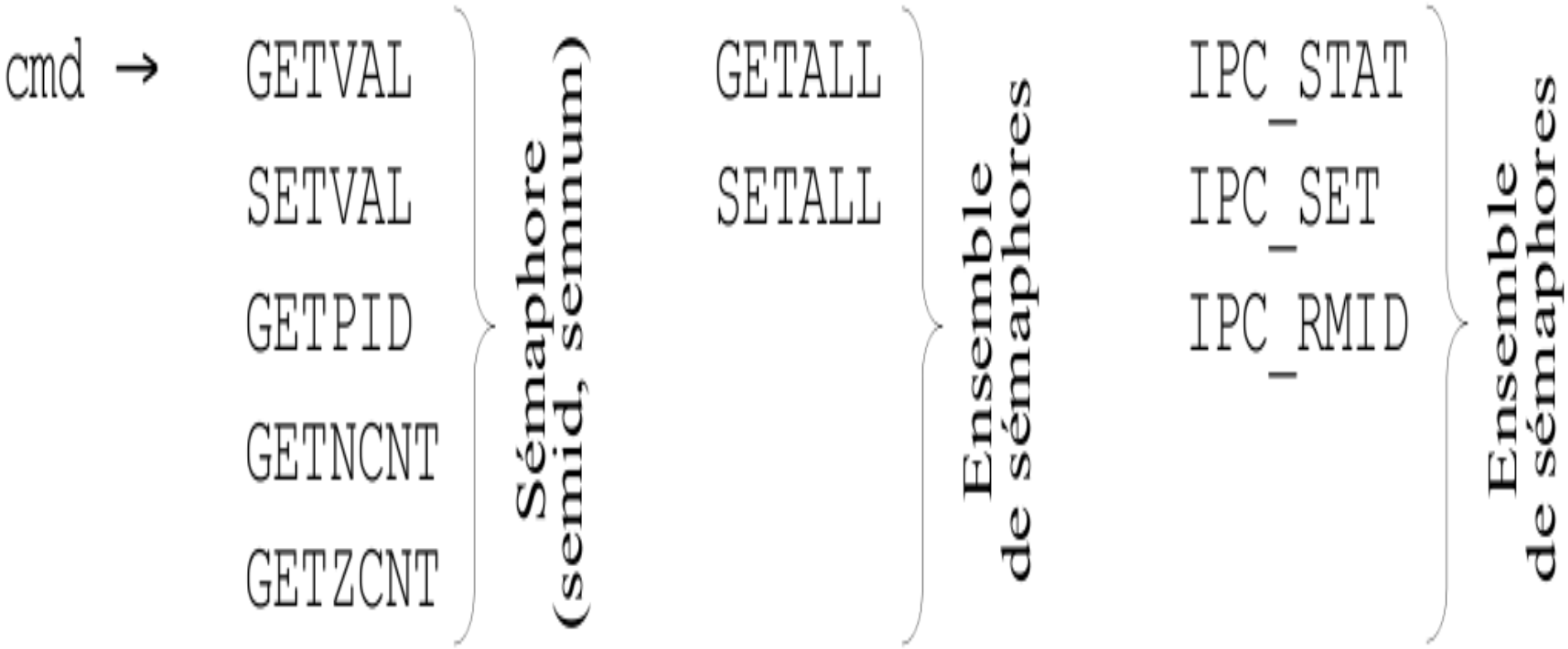
Contrôle des sémaphores

- int semctl (int semid, int semnun, int cmd, union semun arg);
  - suivant cmd, la valeur « val » de l'union peut représenter soit le nombre soit le numéro du sémaphore

```
union semun {
int val;
struct semid_ds *buf;
u_short *array;
};
```

Contrôle des sémaphores

- Valeurs de « cmd »



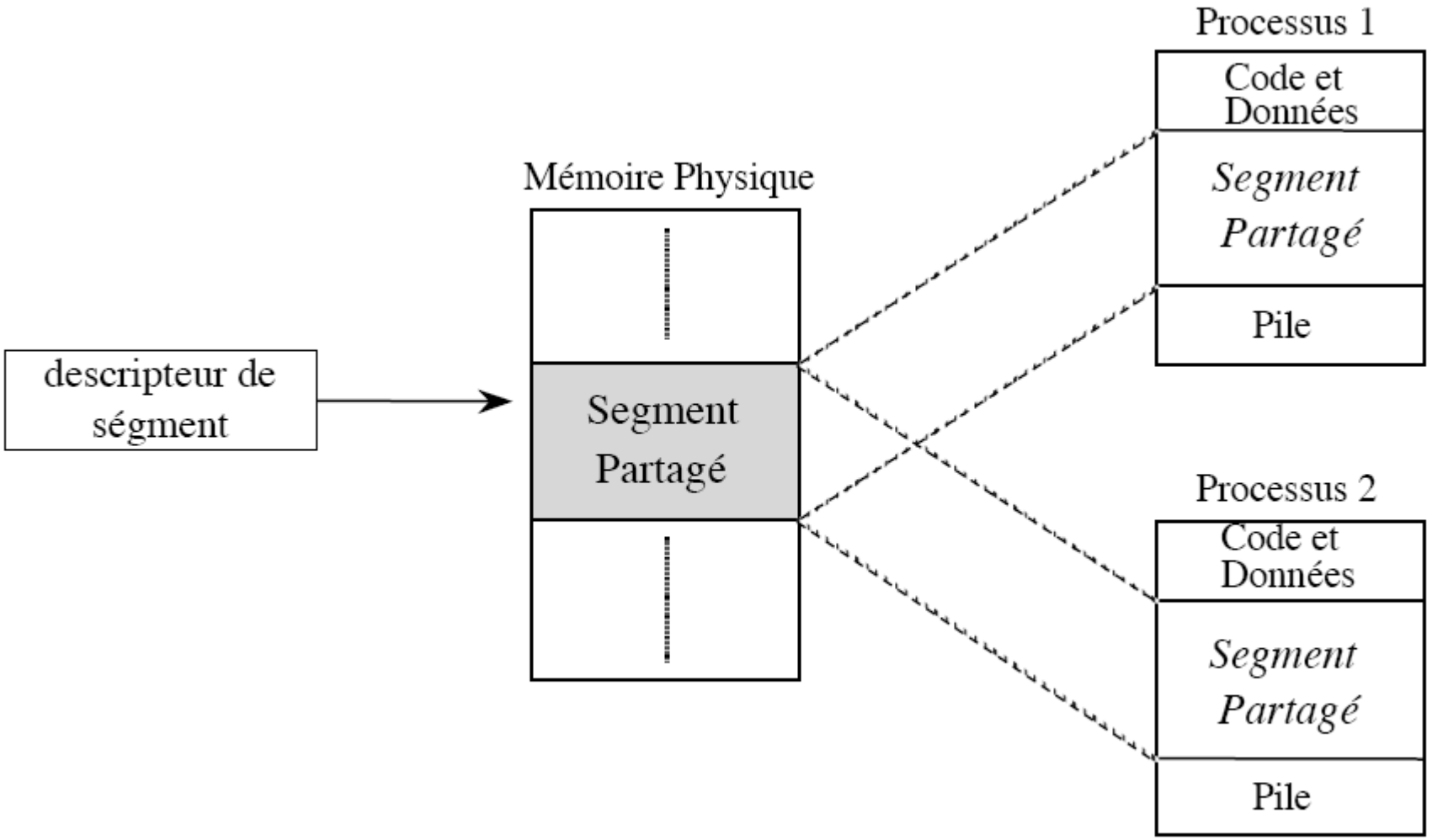
système V

Mémoires partagées (Shared Memory)

- Permet aux processus de partager une partie de l’espace adressable.

Utilisation de sémaphores pour gérer les accès concurrents.

Mémoires partagées (Shared Memory)



Mémoires partagées (Shared Memory)

- Accès libre à l’espace de données
- Pas de recopie des données
- Gestion des accès concurrents à prévoir
- Les pointeurs ne sont valables que dans l’espace d’adressage du processus

Création et recherche d'une mémoire partagée

- int shmget (key_t clef, int taille, int option);
  -taille est la taille allouée, elle doit être un multiple de 4ko

Attachement d ’une zone mémoire à un processus

- void *shmat (int shmid ,const void *shmaddr, int option);
  - si shmaddr=NULL, Linux cherche une adresse
  sinon essai du système à l’adresse mémoire virtuelle du processus indiqué.

Détachement d’un processus d’une zone mémoire

- int shmdt (const void *shmaddr)

Contrôle des zones de mémoire partagée

- int shmctl (int shmid, int cmd,struct shmid_ds *buf);

  - IPC_STAT : donne les informations sur la mémoire partagée.

  - IPC_SET : valide les changements sur les champs uid,gid,mode.

  - IPC_RMID : permet de marquer un segment mémoire pour sa destruction

Les IPCs et l'héritage

- Appel système fork()
  – Héritage de tous les objets IPC par le fils
- Appel système exec()
  – Tous les accès à des objets IPC sont perdus
  ATTENTION : les objets ne sont pas détruits
  – Dans le cas de la mémoire partagée, le segment est détaché

IPCs System V et POSIX

IPC System V ≠ IPC POSIX !

- POSIX est un standard portable
- POSIX demande un système "thread-safe"
- IPC POSIX sont globalement + simples d'utilisation, et + fonctionnels
- API System V requiert un appel système par fonction
- Les sémaphores POSIX se manipulent à l'unité
- Les IPC POSIX apparaissent dans l’espace de noms du système de fichiers « /dev/shm »

Les IPC POSIX ne sont pas dans la Glibc mais dans les bibliothèques « rt » ou « pthread »

Les files de messages

#include <mqueue.h>

Accès

mq_open ⇒ créer / ouvrir une file en RAMDISK

mq_close ⇒ fermer l'accès à une file

mq_unlink ⇒ détruire une file

mq_getattr ⇒ obtenir les attributs de la file (taille, mode d'accès, ...)

mq_setattr ⇒ modifier le mode d'accès (O_NONBLOCK)

Opérations sur une file

mq_send ⇒ déposer un message

mq_receive ⇒ retirer un message

mq_notify ⇒ signal d'arrivée d'un message

Les files de messages

```
struct mq_attr {
[...]
long mq_maxmsg; //max nb of msgs in queue
long mq_msgsize;//max size of a single msg
long mq_flags; //behaviour of the queue
long mq_curmsgs; //nb of msgs currently in queue
[...]
};
```

mq_flags = O_NONBLOCK ou 0

files de messages «open»

mqd_t mq_open(char* name, int flags, struct mq_attr* attrs);

Crée une nouvelle file ou recherche le descr. d'une file déjà existante

Retourne un descripteur positif si succès, -1 sinon

flags idem open

attrs peut être rempli avant pour définir des valeurs (sauf mq_flags)

mq_flags : O_RDONLY ; O_WRONLY ; O_RDWR.

mq_getattr(mqd_t mq descr, struct mq_attr* attrs);

mq_setattr(mqd_t mq descr, struct mq_attr* new_attrs

struct mq_attr* old_attrs);

files de messages «open»

mq_getattr() renvoie une structure mq_attr dans le tampon pointé par attr. Cette structure est définie comme suit :

```
struct mq_attr {
long mq_flags;       /* Drapeaux : 0 or O_NONBLOCK */
long mq_maxmsg;  /* Max. # de messages dans la file */
long mq_msgsize;     /* Max de la taille du message (octets) */
mq_curmsgs;     /* # de messages actuellement dans la file */
        };
```

files de messages «close»

int mq_close(mqd_t mqdescr);

– retourne 0 en cas de succès, -1 sinon.

– Automatiquement appelé lors de la terminaison du pcs

– Pas d'effet sur l'existence ou le contenu de la file

int mq_unlink(char* mqname);

– retourne 0 en cas de succès, -1 sinon.

– Détruit la file associée à mqname ainsi que son contenu

Après l'appel, plus aucun processus ne peut ouvrir la file

Destruction effective une fois que tous les processus qui y ont accès ont appelé mq_close

files de messages «send»

int mq_send(mqd_t mqdescr, const char* msg_data, size_t msg_length, unsigned int priority);

retourne 0 en cas de succès, -1 sinon.

msg_data le contenu du message

msg_length la taille du message

priority sa priorité, 0 ≤ priority ≤ MQ_PRIOMAX (≥ 32, déf. dans limits.h)

Si priority > MQ_PRIOMAX, l'appel échoue

– File ordonnée par priorités, en FIFO pour les msgs de même prio

– Appel bloquant si la file est pleine et O_NONBLOCK non spécifié

files de messages «recv»

int mq_receive(mqd_t mqdescr, const char* msg_data, size_t msg_length, unsigned int *priority);

– retourne le nb de octets lu en cas de succès, -1 sinon.

– msg_data le contenu du message

– msg_length la taille du message

Si msg_length > mq_attr.mqmsgsize, l'appel échoue

– priority sa priorité

– Appel bloquant si la file est vide et O_NONBLOCK non spécifié

files de messages «notif»

int mq_notify(mqd_t mqdescr, const struct sigevent* notification);

retourne 0 en cas de succès, -1 sinon

Appel non bloquant

Un seul processus par file peut demander à être notifié

Notification si aucun processus n'est bloqué en attente de msg

Après notification, dés-enregistrement de la demande

⇒ pour être notifié à chaque arrivée de msg, il faut refaire un appel après chaque notification

files de messages «notif»

struct sigevent { [...]
int sigev_notify //Notification type
(SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD)
int sigev_signo //Signal number
union sigval sigev_value
//Notif. data
void (*)(union sigval) sigev_notify_function
//Notif. function
[...] };

Mémoire partagée

#include<sys/mman.h>

shm_open ⇒ créer / ouvrir un segment en RAMDISK

close ⇒ fermer un segment

mmap ⇒ attacher un segment dans l'espace du processus

munmap ⇒ détacher un segment de l'espace du processus

shm_unlink ⇒ détruire un segment

Opérations sur un segment

mprotect ⇒ changer le mode de protection d'un segment

ftruncate ⇒ allouer une taille à un segment

msync ⇒ mettre à jour le segment mémoire

Mémoire partagée «open»

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

– Crée un nouveau segment de taille 0 ou recherche le descripteur d'un segment déjà existant

– Retourne un descripteur positif en cas de succès, -1 sinon

– flags idem open

– mode idem chmod

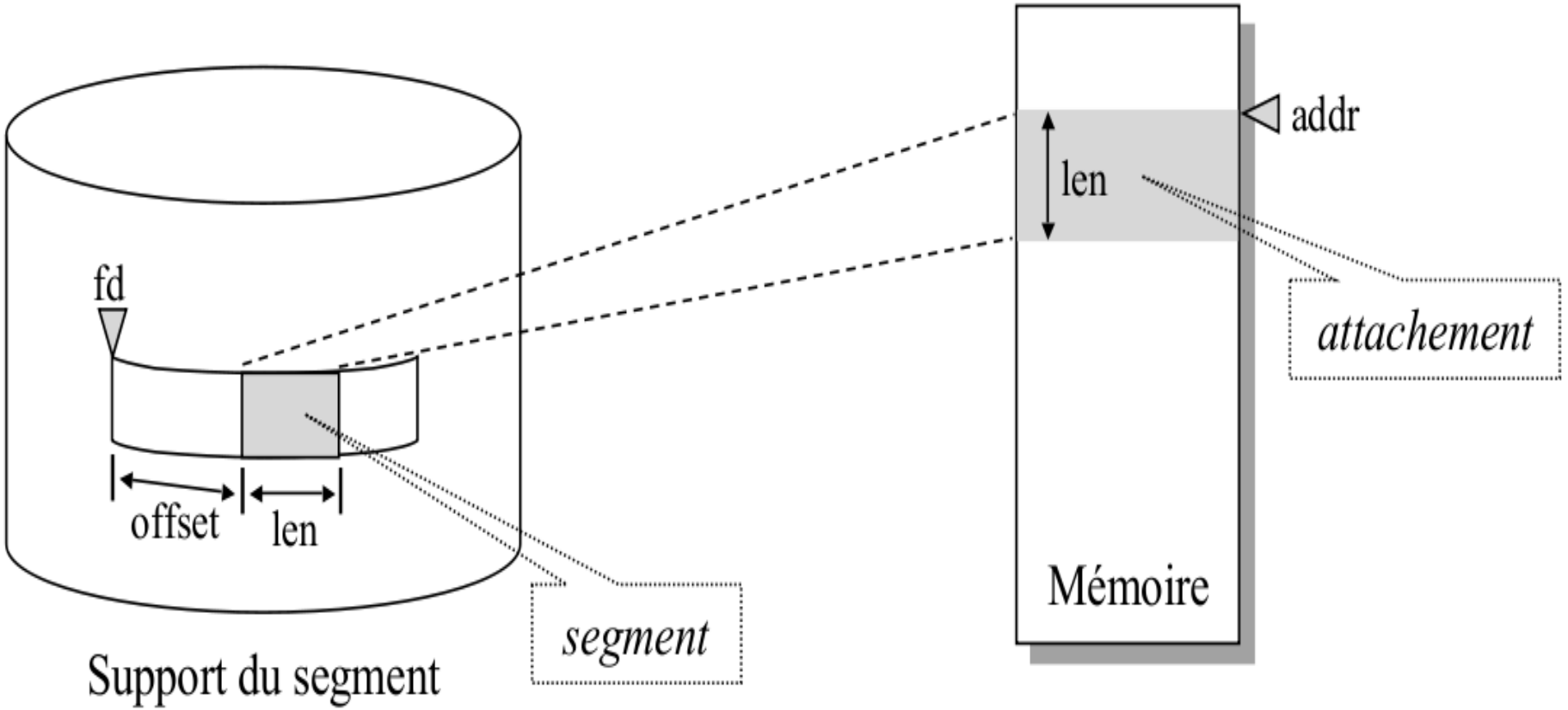
Destruction

int shm_unlink(const char *name);

Idem mq_unlink

Mémoire partagée «attach»

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



Mémoire partagée «attach»

void * mmap(void *addr, size_t len, int prot, int flags,

int fd, off_t offset);

– Retourne NULL en cas d'échec, ou l'adresse du segment partagé

– addr

adresse où attacher le segment en mémoire ; 0 ⇒ choix du système

– prot

protection associée (PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE)

Mémoire partagée «attach»

– flags

mode de partage

MAP_SHARED : modifs visibles par tous les processus ayant accès (partage)

MAP_PRIVATE : modifs visibles par le processus appelant uniquement (shadow copy)

MAP_FIXED : force l'utilisation d'addr

int munmap(caddr_t addr, size_t len);

– Détruit l'attachement de taille len à l'adresse addr

– Retourne -1 en cas d'échec, 0 sinon

Opérations mémoire

void * mprotect(caddr_t addr, size_t len, int prot);

– Modifie la protection associée au segment

PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE – Retourne -1 en cas d'échec, 0 sinon

int ftruncate(int fd, off_t length);

– Définit la taille « length » du segment de descripteur fd

si (ancienne taille > nouvelle taille), alors les données en excédent sont perdues

int msync(void *addr, size_t len, int flags);

– Met à jour le segment associé à l'attachement d'adresse addr et de taille len

– flags MS_SYNC, MS_ASYNC

Sémaphores

2 types de sémaphores :

- Sémaphores nommés
  - Portée : tous les processus de la machine
  - Primitives de base : sem_open, sem_close, sem_unlink, sem_post, sem_wait
- Sémaphores anonymes (memory-based)
  - Portée : threads du même processus ou threads de processus différents
  - Primitives de base : sem_init, sem_destroy, sem_post, sem_wait

Inclus dans la bibliothèque des pthreads

Sémaphores nommés

#include <semaphore.h>

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

– Crée ou ouvre le sémaphore de nom name

– oflag, mode idem open

– value valeur initiale du compteur

Retourne un pointeur sur le sémaphore, NULL en cas d'erreur

Ex : Création d'un sémaphore initialisé à 10

```
sem_t *s;
s = sem_open(« monsem », O_CREAT | O_RDWR, 0600, 10);
```

Sémaphores anonymes

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

Crée et initialise le sémaphore sem

– Si pshared vaut 0, il est partagé entre les threads d'un processus et devrait être situé à une adresse visible par tous les threads (ex : variable globale)

–Si pshared n'est pas nul, le sémaphore est partagé entre processus et devrait être situé dans une région de mémoire partagée

– value : valeur initiale du sémaphore

Ex : création de sémaphore partagé, initialisé à 10

```
sem_t s;
sem_init(&s, 1, 10);
```

Sémaphores close destroy

Sémaphore nommé :

1) Fermer le sémaphore

int sem_close(sem_t *sem);

2) Détruire le sémaphore

int sem_unlink(const char *name);

Sémaphore anonyme :

int sem_destroy(sem_t *sem);