
Les IPC

Inter process communication

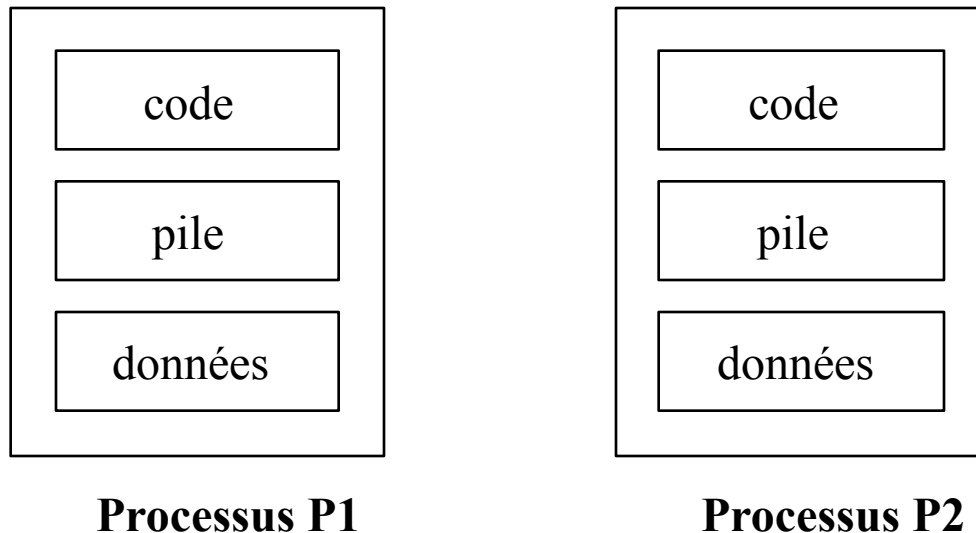
I –IPC : Outils et principes

II – Les IPC System V

III – Les IPC POSIX

Communication entre processus

Comment les processus peuvent communiquer, synchroniser ou partager des données?

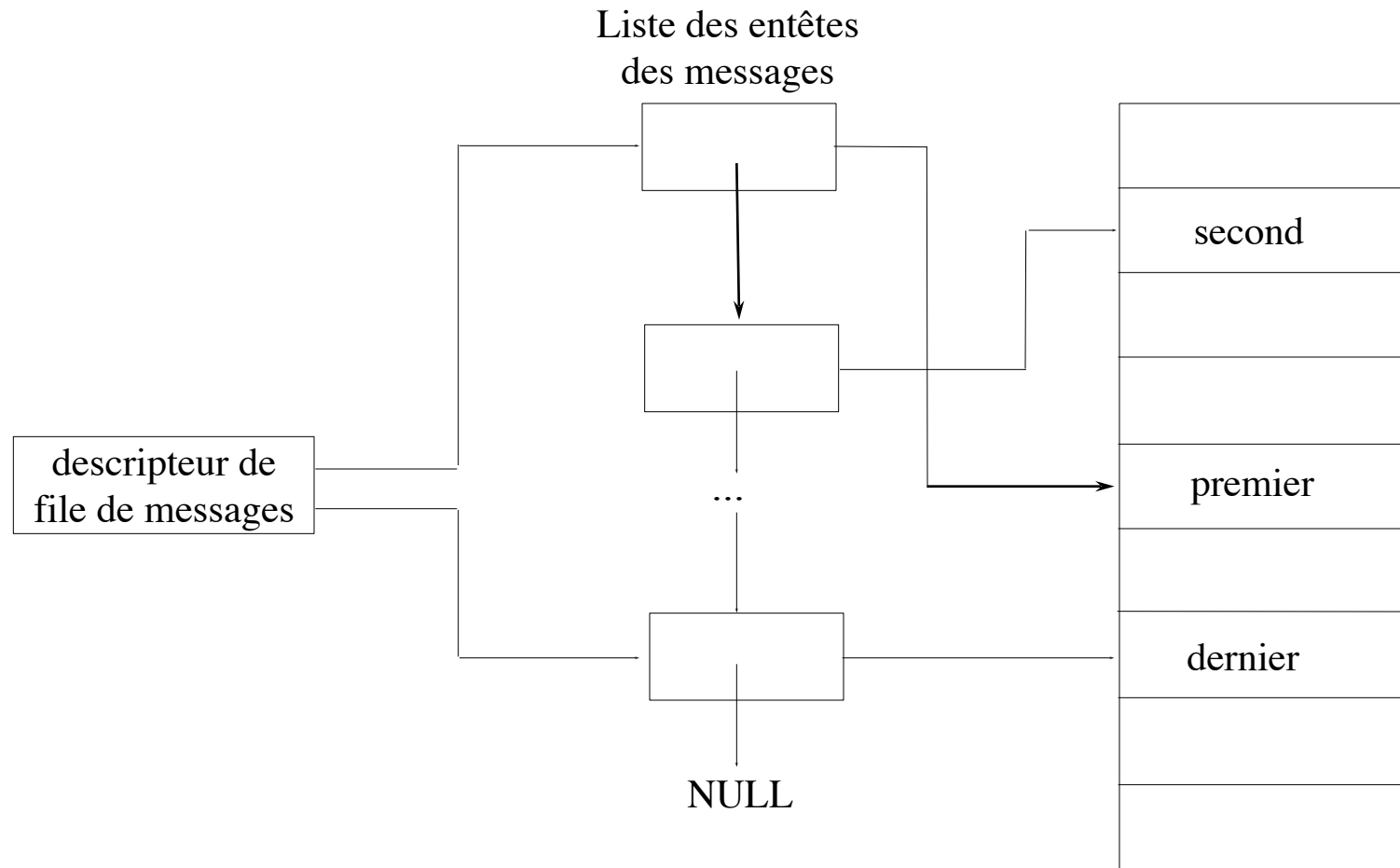


Processus ne partagent pas leur segments de données

I - IPC : Outils et principes

- **Inter-Process Communication (IPC)**
 - Modèle processus : moyen d'isoler les exécutions
 - Distinction des ressources, états, ...
 - Canaux de communication basiques : `wait`/`exit`, `kill`, ...
 - Pb : Nécessité de communication/synchronisation étroite entre processus
 - Canaux basiques pas toujours suffisants
 - Solutions par fichiers (*eg. tubes*) peu efficaces et pas forcément adaptées (*eg. priorités*)
 - Trois mécanismes de comm/synchro entre pcs locaux via la mémoire
 - Les files de messages
 - La mémoire partagée
 - Les sémaphores

Les files de messages



Les files de messages

■ Principe

- Liste chaînée de messages
 - Conservée en mémoire
 - Accessible par plusieurs processus
- Message
 - Structure complexe définie par l'utilisateur
 - Doit comporter un indicateur du type (*~ priorité*) de message
- Fonctionnement
 - accès FIFO (par défaut) + accès par type
 - limites : nb de msgs (MSGMAX), taille totale en nb de bytes (MSBMNB)
 - limite atteinte ⇒ écriture bloquante (par défaut)
 - file vide ⇒ lecture bloquante (par défaut)

Les files de messages

- **Avantages**

- Amélioration par rapport au concept de tube
 - organisée en message
 - contrôle sur l'organisation (priorités)
- Simplicité
 - Proche du fonctionnement naturel d'une application

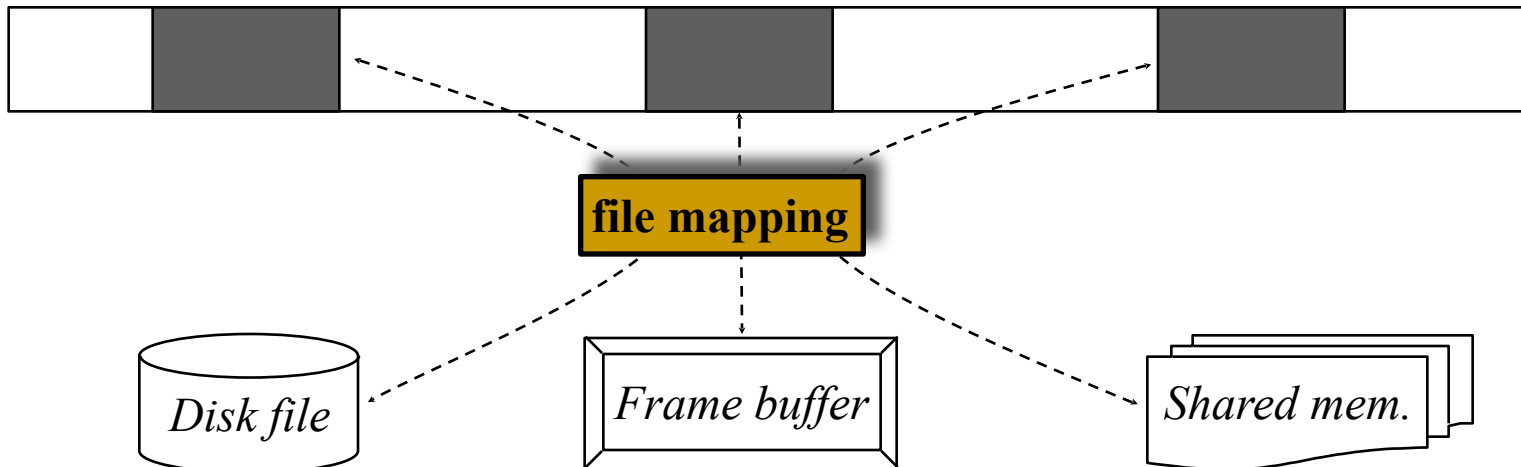
- **Désavantages**

- Reste basée sur du FIFO
 - Impossible d'organiser les accès différemment (*eg. pile*)
 - Pas d'accès concurrents à une même donnée
- Performances limitées
 - 2 copies **complètes** par msg : expéditeur → cache système → destinataire

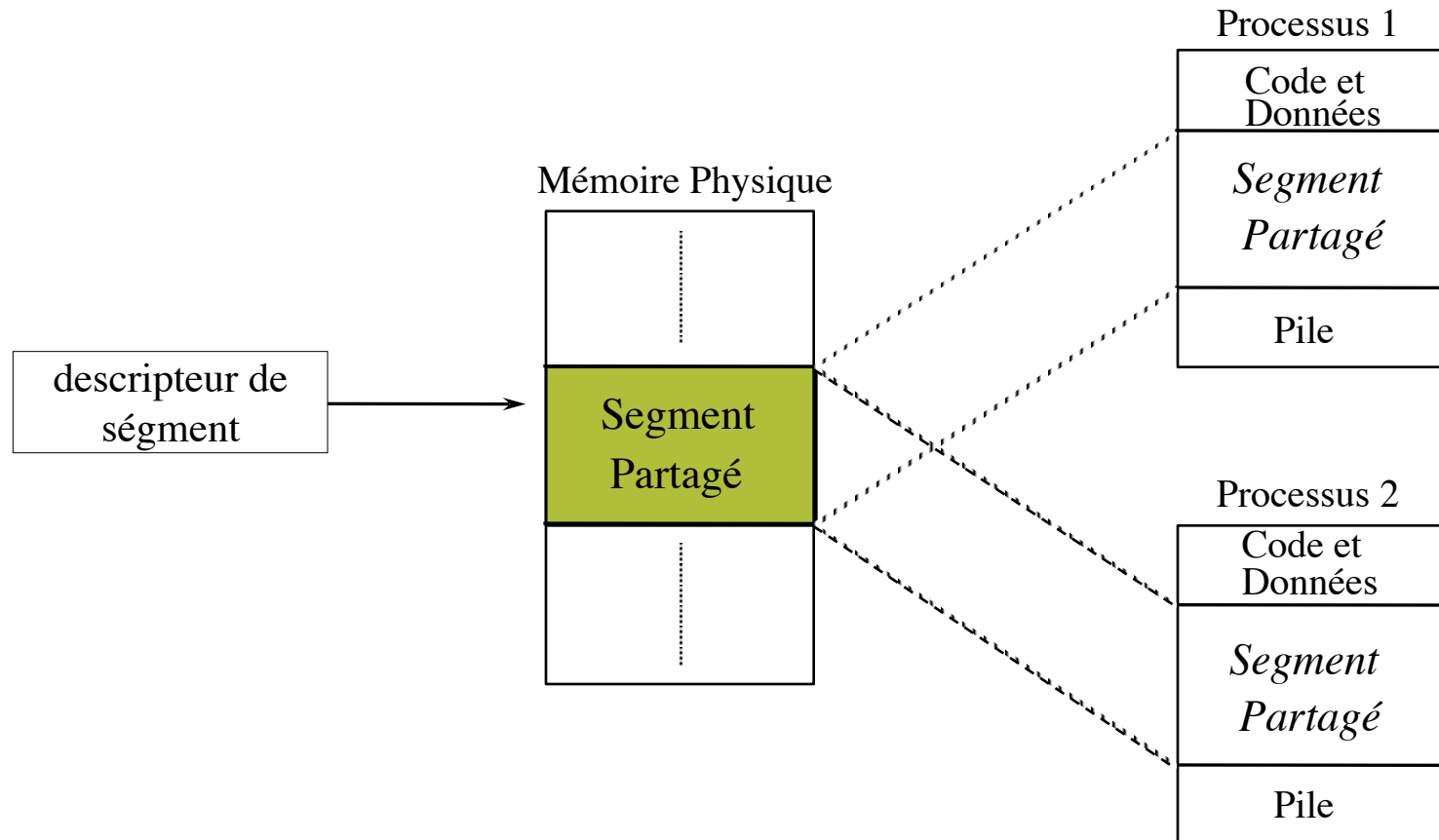
Segments de mémoire partagée

■ Principe

- Zone mémoire attachée à un processus mais accessible pour d'autres processus
- Liée à un autre service : file mapping
 - Etablissement d'une correspondance (attachement) entre :
 - ❑ Un fichier (ou un segment de mémoire)
 - ❑ Une partie de l'espace d'adressage d'un processus réservée à cet effet



Segments de mémoire partagée



Segments de mémoire partagée

- **Avantages**

- Accès totalement libre
 - Chaque processus détermine à quelle partie de la structure de données il accède
- Efficacité
 - Pas de recopie mémoire : tous les processus accèdent **directement** au même segment

- **Désavantages**

- Accès totalement libre
 - Pas de synchro implicite comme pour les tubes et les files de msgs
 - ⇒ Synchro doit être explicitée (sémaphores ou signaux)
- Pas de gestion de l'adressage
 - Validité d'un pointeur limitée à son esp. d'adressage
 - ⇒ Impossible de partager des pointeurs entre processus

Les sémaphores

■ Fonctionnement

- Demande d'accès (P - *proberen* ou "*puis-je ?*")
 - Décrémentation du compteur
 - Si compteur < 0 , alors blocage du pcs et insertion ds la file

- Fin d'accès (V - *verhogen* ou "*vas-y*")
 - Incrémentation du compteur
 - Si compteur ≤ 0 , alors déblocage d'un pcs de la file

Blocage, déblocage et insertion des processus dans la file sont des ops implicites

Les sémaphores

- **Principe (Dijkstra)**

- Mécanisme de synchronisation
 - accès concurrents à une ressource partagée (*eg. segment de mémoire*)
 - solution au problème de l'exclusion mutuelle

- **Structure sémaphore**

- un compteur : nb d'accès disponibles avant blocage
- une file d'attente : processus bloqués en attente d'un accès

Les sémaphores

- **Dysfonctionnements**

- Liés à leur utilisation franchement pas intuitive

- **Interblocage**

- 2 processus P et Q sont bloqués en attente
 - P attend que Q signale sa fin d'accès et Q attend que P signale sa fin d'accès

- **Famine**

Un processus est bloqué en attente d'une fin d'accès qui n'arrivera jamais

Effets des appels système

Appel à :

- **fork ()**

- Héritage de tous les objets IPC par le fils

- **exec () ou exit ()**

- 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

Les IPC

Inter process communication

I –IPC : Outils et principes

 **II – Les IPC System V**

III – Les IPC POSIX

II - IPC System V

- **Éléments communs à tous les mécanismes IPC System V**
- **Files de messages**
- **Segments de mémoire partagée**
- **Sémaphores**

IPC SysV : Caractéristiques

- **Ils sont extérieurs aux système de gestion de fichiers**
 - Pas désignés localement par de descripteurs
- **Gestion est faite par le système avec un table spécifique au type de l'objet**
- **Chaque objet dispose d'une identification interne**
 - Du point de vue externe, les objets sont identifiés par un clé.

IPC SysV : Caractéristiques communes

- **Une table par mécanisme**

- une entrée → une instance
- Une clé numérique par entrée

- **Un appel système **xxxget** par mécanisme**

xxx → **shm** (mémoire partagée), **msg** (files de messages) ou **sem** (sémaphores)

- crée une nouvelle entrée ou retrouve une déjà existante
- retourne un descripteur

Cas de création :

- `cle = IPC_PRIVATE`
- `IPC_CREAT` → `flags`
- `IPC_CREAT | IPC_EXCL` → `flags`

Les commandes `shell` associées

Deux commandes `shell`

- `ipcs`

- liste des ressources actives ainsi que leurs caractéristiques

```
$ipcs
IPC          ID      KEY      MODE      OWNER
  GROUP
Messages Queues:
q           0      0x00000000  --rw-----  root
  root
Shared Memory:
m           3      0x41442041  --rw-rw-rw   root
  root
Semaphores:
s           1      0x4144314d  --ra-ra-ra-  root
  root
```

- `ipcrm`

- suppression des ressources

```
$ipcrm -q 0 -m 3 -S 0x4144314d
```

Structure commune aux IPC SysV

- Une structure commune

```
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 sequence number */  
    key_t key; /* key */  
};
```

Définitions communes aux IPC SysV

■ Définitions communes

```
#define    IPC_CREAT    0001000    /* create entry if key
                                   doesn't exist */
#define    IPC_EXCL     0002000    /* fail if key exists */
#define    IPC_NOWAIT   0004000    /* error if request must wait */
#define    IPC_PRIVATE  (key_t) 0  /* private key */
#define    IPC_RMID     0          /* remove identifier */
#define    IPC_SET      1          /* set options */
#define    IPC_STAT     2          /* get options */
```

Définitions communes aux IPC SysV :

Composition d'une clé

- **Soumission d'une clé pour obtenir un descripteur d'IPC**
- **Composition d'une clé**
 - Fixée par l'utilisateur
 - Déterminée par le système

```
key_t ftok(path, code);  
char *path;  
char code;
```

Définitions communes aux IPC SysV :

Composition d'une clé

- **Exemple :**

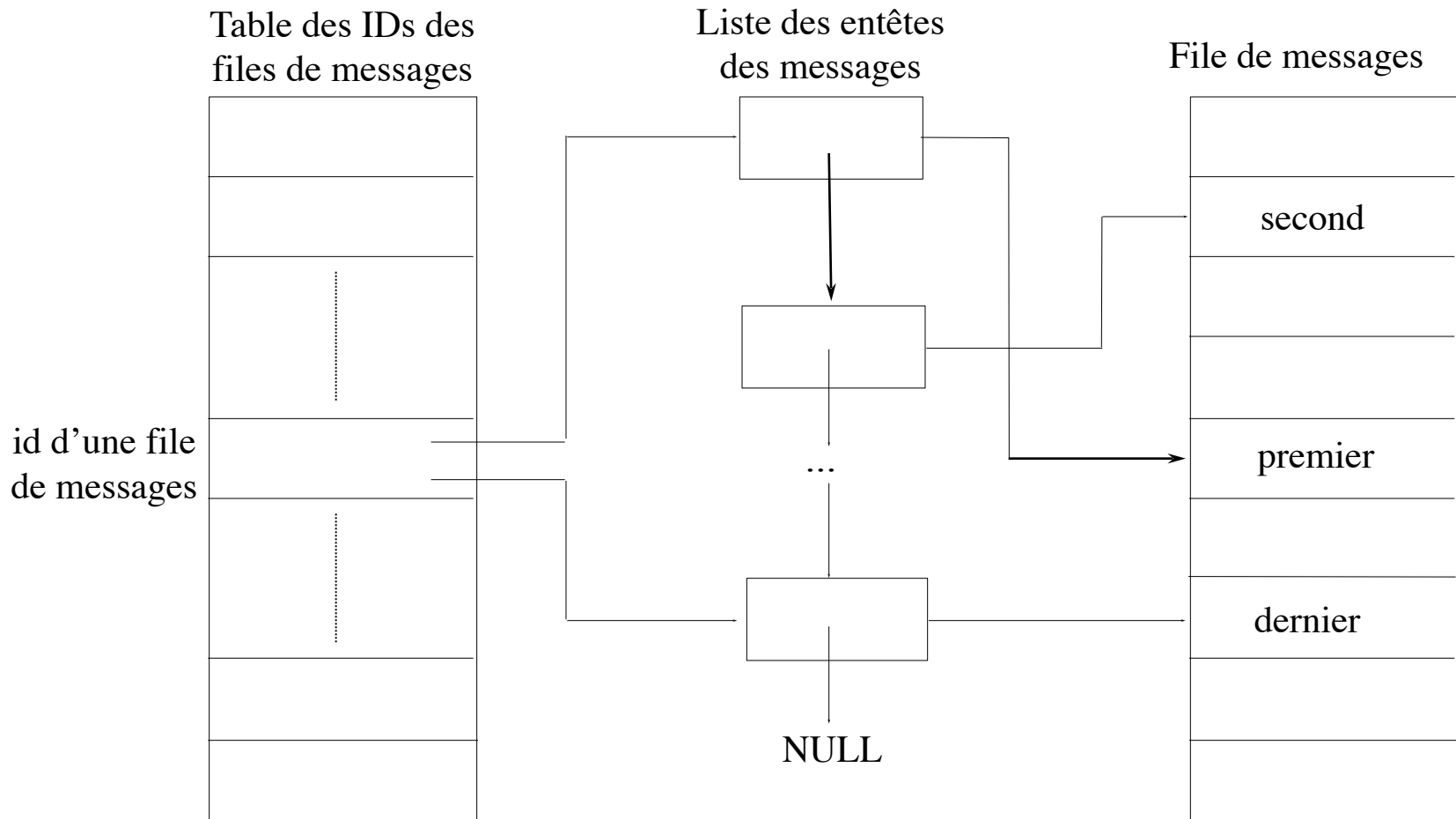
```
#include <sys/ipc.h>
... key_t key;
char *path = "/tmp";
key = ftok(path, 0);
```

- `path` doit exister tant que des clés y sont associées.
 - Si un fichier est déplacé entre deux appels à la fonction, la clé fournie sera différente.

IPC SysV : Caractéristiques communes

- **xxxget (key_t cle, int flag)**
 - Si clé = IPC_PRIVATE
 - Un nouveau objet est crée dans la table correspondante. Utilisé au sein du même processus ou entre père et fils.
 - Sinon
 - Si objet n'existe pas
 - ❑ Si (flag & IPC_CREAT)
 - Un nouveau objet est crée dans la table correspondante
 - ❑ Sinon
 - Erreur
 - Sinon ((flag & IPC_CREAT) && (flag_IPC_EXCL))
 - ❑ Erreur
 - Sinon
 - ❑ l'identification de l'objet est renvoyé

Les files de messages System V



Caractéristiques des files de messages

- **Paquets identifiables et indivisibles et non pas un flou de caractères**
- **Spécification de l'id. de file et non pas du processus lors d'une émission/réception**
- **Politique FIFO**
- **Connaissance + droits d'accès**
 - droits d'émission/réception
- **Un message :**
 - Type (entier dont l'interprétation est laissée à l'utilisateur) + Donnée (chaîne de caractères de longueur quelconque)
 - Un processus peut extraire un message en utilisant le type comme critère de sélection

Message

- **Message – composé par :**

- Type : nombre entier (long) strictement positif
- Donnée : Suite d'octets contigus en mémoire

```
struct msgbuf {                                /* définie dans sys/msg.h */
    long mtype; /* type du message */
    char mtext[1]; /* texte du message */
};
```

- **Possibilité de redéfinir en fonction de ses besoins**

- **Exemple:**

```
struct msg_buf {
    long type;
    struct msg {
        ....
    } corps;
};
```

```
struct msg_buf {
    long type;
    ....
};
```

Création d'une file de messages

- **Création d'une nouvelle file de messages**
ou recherche de l'identifiant d'une file déjà existante

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

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

```
int msgid = msgget(key_t cle, int flags);
```

- retourne un entier positif (id. de la file de msgs ds la table) en cas de succès; -1 sinon.
- Création si `cle = IPC_PRIVATE`
 - `IPC_CREAT` → `flags`
 - `IPC_CREAT | IPC_EXCL` → `flags`

Structure associée

Format d'une entrée dans la table de files de message

```
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 */  
};
```

Peut être accédée par l'intermédiaire de la fonction msgctl

Initialisations associées à une création

- Initialisations lors de la création d'une nouvelle entrée :

`msg_perm.cuid` et `msg_perm.uid` ← uid effectif du processus appelant
`msg_perm.cgid` et `msg_perm.gid` ← gid effectif du processus appelant
`msg_perm.mode` ← 9 bits de poids faible de l'entier `flags`
`msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, `msg_rtime` ← 0
`msg_qbytes` ← taille maximale permise par le système
`msg_ctime` ← heure courante

Emission d'un message

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

```
int msgsnd(int msgid, struct msgbuf *msg,  
           int taille, int flags);
```

- > Demande d'envoi dans la file `msgid` du message pointé par `msg`.
- > `taille` = longueur du texte du message
 - Octets occupés par le *type* ne sont pas comptabilisés
- > retourne 0 en cas de succès et -1 sinon.
- > Flags
 - `nul`
 - `IPC_NOWAIT`
 - Si la file est pleine l'appel à la primitive n'est pas bloquante

Propriétés d'une émission

- **Emission bloquante (défaut)**

- Si file pleine, le processus est suspendu jusqu'à :
 - extraction de messages de la file,
 - suppression du système de la file (retourne `-1` et `errno = EIDRM`),
 - réception d'un signal.
- Sinon,
 - insertion du message et de son type dans la file,
 - incrémentation du nombre de messages de la file,
 - mise à jour de l'identificateur du dernier écrivain,
 - mise à jour de la date de dernière écriture.

- **Emission non bloquante**

- Si file pleine et `IPC_NOWAIT` → `flags`,
 - le message n'est pas envoyé et
 - le processus reprend immédiatement la main.

Extraction d'un message d'une file

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

```
int msgrcv(int msgid, struct msgbuf *msg, int taille,  
           long type, int flags);
```

- Extraction quelconque ou sélective,
 - type = 0 → le 1er msg de la file, quel que soit son type,
 - type > 0 → le 1er msg du type désigné,
 - type < 0 → le 1er msg dont le type est > à la valeur absolue du type désigné
- bloquante par défaut,
- si `taille < taille du message`
 - Si `MSG_NOERROR → flags`
 - ❑ le système tronque le message sans générer d'erreur
 - ❑ le reste du texte est perdu.
 - Sinon, le système retourne une erreur (`errno = E2BIG`) et le msg reste dans la file.
- retourne le nb de caractères dont est composé le txt du msg en cas de succès -1 sinon.

Propriétés d'une extraction

- **Si aucun message ne répond aux conditions demandées**
 - $\text{IPC_NOWAIT} \not\subset \text{flags}$, alors le processus est suspendu jusqu'à :
 - arrivée d'un message satisfaisant les conditions demandées,
 - destruction de la file (retourne -1 et $\text{errno} = \text{EIDRM}$),
 - réception d'un signal.
 - $\text{IPC_NOWAIT} \subset \text{flags}$, alors :
 - le processus reprend immédiatement la main,
 - retourne -1 et $\text{errno} = \text{ENOMSG}$.
- **Sinon**
 - extraction effective du message de la file,
 - décrémentation du nombre de messages de la file,
 - mise à jour de l'identificateur du dernier lecteur,
 - mise à jour de la date de dernière lecture.

Contrôle de l'état d'une file

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

```
int msgctl(int msgid, int cmd, msgid_ds *buf);
```

- **consultation, modification des caractéristiques et suppression d'une file**

cmd	→	IPC_STAT	
		→	IPC_SET
			msg_perm.uid
			msg_perm.gid
			msg_perm.mode
			msg_qbytes
		→	IPC_RMID

Opérations permises uniquement si
uid effectif = super utilisateur

shm_perm.cuid
shm_perm.uid

Modification msg_qbytes → root

- **retourne 0 en cas de succès et -1 sinon**

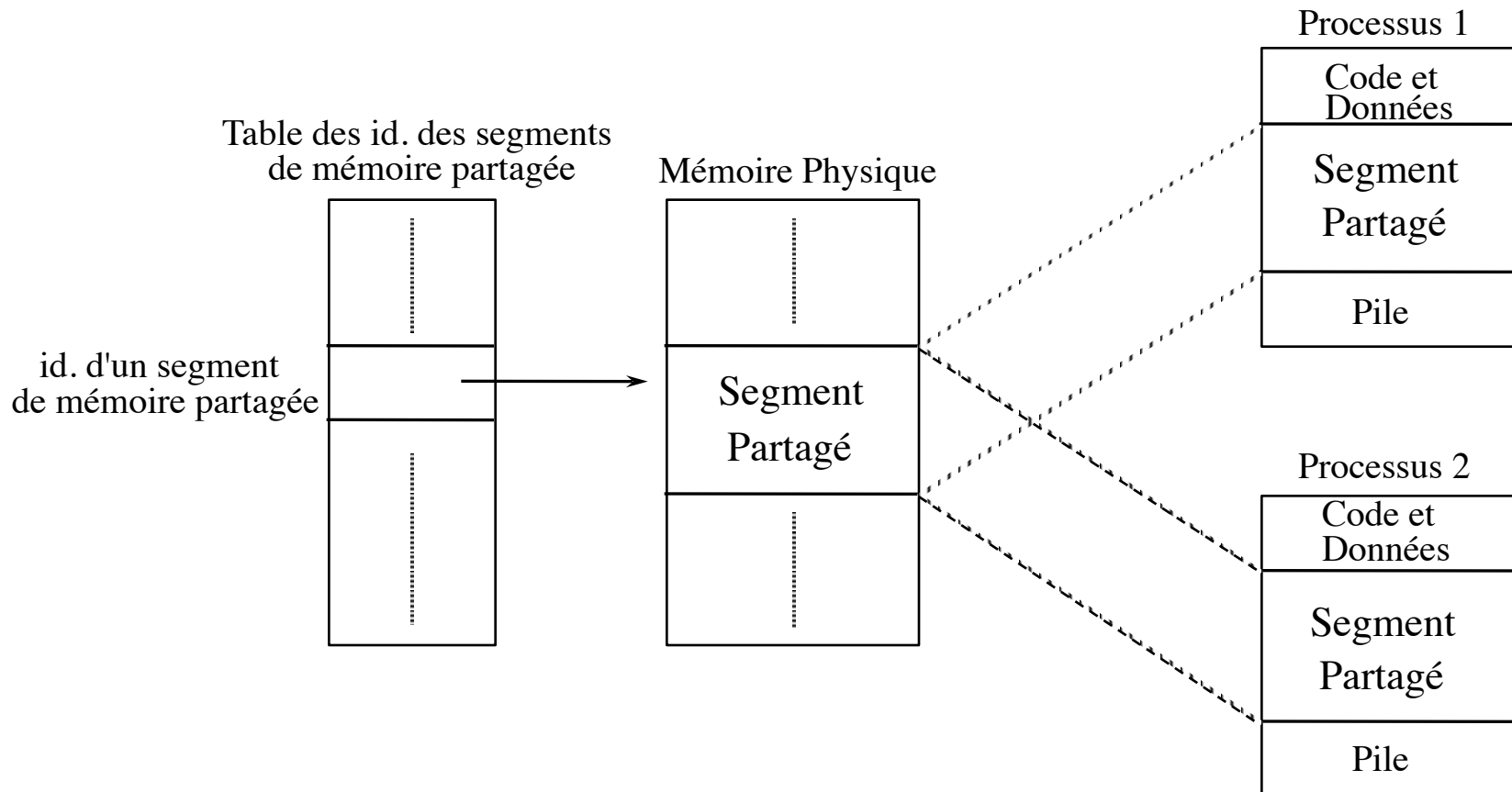
Example

```
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_SIZE 128
...
int msg_id; struct msqid_ds *buf; key_t cle;
struct message {long type;
                char texte[MSG_SIZE]; } msg;
char path[14]= "file_msg"; char code=' Q ';
...
cle = ftok(path, code);
msg_id = msgget (cle, 0666 | IPC_CREAT);
...
msg.type = 1;
for ( ; ; ) {
    printf ( "Entrer le texte a emettre \n");
    scanf("%s",msg.texte);
    msgsnd(msg_id , &msg , MSG_SIZE, 0);
    ...
}
...
```

Exemple (suite)

```
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSG_SIZE 128
...
int msg_id; struct msqid_ds *buf; key_t cle;
struct message {      long type;
                   char texte[MSG_SIZE]; } msg;
char path[14]= "file_msg"; char code=' Q ' ;
...
cle = ftok(path, code);
msg_id = msgget (cle, 0);
...
msg.type = 1;
for ( ; ; ) {
    msgrcv(msg_id , &msg , MSG_SIZE, 1L,0);
    printf
    ...
}
...
```

Segments de mémoire partagée



Segments de mémoire partagée

- **Processus partagent des pages physique par l'intermédiaire de leur espace d'adressage**
 - Pas de recopie d'information
 - Pages partagées deviennent de ressources critiques
 - Existence indépendantes des processus
 - Continue a exister jusqu'à une demande de suppression

Création d'un segment de mémoire partagée

- **Création d'un nouveau segment
ou recherche de l'identifiant d'un segment existant**

```
#include<sys/ipc.h>
#include<sys/shm.h>
int shmget(key_t cle, int taille, int flags);
```

- retourne -1 en cas d'échec
un entier positif (identificateur de segment dans la table) , sinon
- Création si `cle = IPC_PRIVATE`
 - `IPC_CREAT` → `flags`
 - `IPC_CREAT | IPC_EXCL` → `flags`

Structure associée

Format d'une entrée dans la table de segments de mémoire partagée

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* operation permission struct */
    uint shm_segsz;             /* size of segment in bytes */
    ushort shm_lpid;            /* pid of last shmop */
    ushort shm_cpid;            /* pid of creator */
    ushort shm_nattch;          /* number of current attaches */
    time_t shm_atime;           /* last shmat time */
    time_t shm_dtime;           /* last shmdt time */
    time_t shm_ctime;           /* last change time */
};
```

Peut être accédée par l'intermédiaire de la fonction shmctl

Initialisations associées à une création

- **Création d'une nouvelle entrée**

- Initialisations

- $\text{shm_perm.cuid et shm_perm.uid} \leftarrow \text{uid effectif du processus appelant}$
 - $\text{shm_perm.cgid et shm_perm.gid} \leftarrow \text{gid effectif du processus appelant}$
 - $\text{shm_perm.mode} \leftarrow \text{9 bits de poids faible de l'entier flags}$
 - $\text{shm_segsz} \leftarrow \text{taille}$
 - $\text{shm_lpid, shm_nattch, shm_atime et shm_dtime} \leftarrow 0$
 - $\text{shm_ctime} \leftarrow \text{heure courante}$

- Création effective au premier attachement

Attachement et détachement d'un segment de mémoire partagée

■ Attachement d'un segment

```
char *shmat(int shmid, void *adr, int flags);
```

- > rend l'adresse à laquelle le segment a été attaché
- > si 1er attachement, alors allocation effective de l'espace mémoire correspondant
- > si `adr = NULL` → le système choisit l'adresse d'attachement
- > possibilité d'attacher plus d'une fois un même segment par un processus
- > `SHM_RDONLY ∈ flags` → `SIGSEGV` en cas de tentative d'écriture
- > Accéder directement au travers de l'adresse renvoyé.

■ Détachement d'un segment

```
int shmdt(void *virtadr);
```

- > spécifier l'adresse et non pas l'identifiant
- > rend 0 en cas de succès et -1 sinon

Opérations de contrôle sur les segments de mémoire partagée

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

```
int shmctl(int shmid, int cmd, shm_id_ds *buf);
```

cmd	→	IPC_STAT	
		→	IPC_SET
			shm_perm.uid
			shm_perm.gid
			shm_perm.mode
	→	IPC_RMID	

Opérations permises uniquement si
uid effectif = super utilisateur

shm_perm.cuid
shm_perm.uid

Example

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

...
int shm_id; struct shmid_ds *buf; key_t cle;
char *p_int, *adr_att;
int taille = 1024;

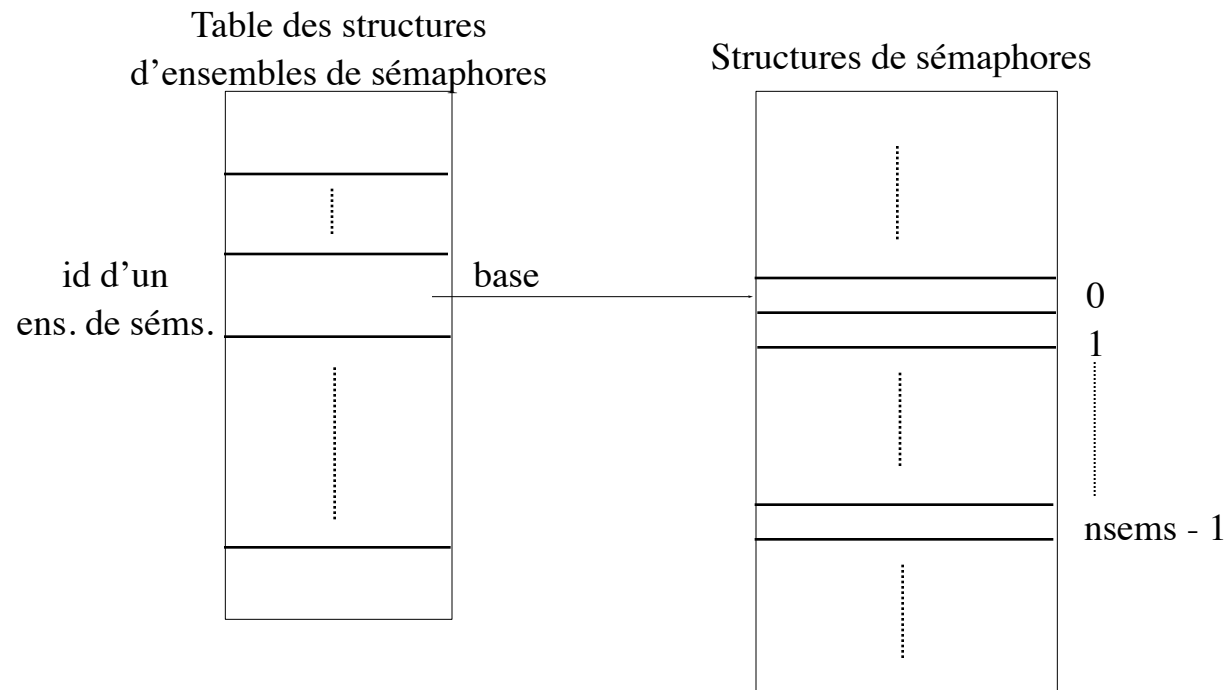
...
cle = ftok("mem_par", ' M ');
shm_id = shmget(cle, taille, 0666 | IPC_CREAT);
adr_att = shmat(shm_id, 0, 0600);

...
p_int = (int *)adr_att;
for (i=0; i<128; i++) *p_int++ = i;

...
shmdt(adr_att);
shmctl(shm_id, IPC_RMID, buf);

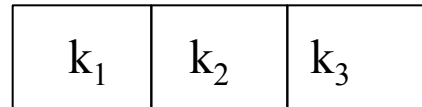
...
```

Les sémaphores



Ensemble de sémaphore

- **Gestion par ensembles de sémaphores, identifiés par des entiers**
 - Acquisition simultanée d'exemplaires multiples de plusieurs ressources différentes



Ensemble de sémaphores

op_1 op_2 op_3

Ensemble d'opérations

$op_i = P_n, V_n \text{ ou } Z$

Création d'un ensemble de sémaphores

- **Création d'un nouvel ens. de sémaphores**
ou recherche de l'identifiant d'un ens. de sémaphores

```
#include<sys/ipc.h>
#include<sys/sem.h>
    int sem_id = semget(key_t cle, int nsems, int flags);
```

- Retourne l'id. de l'ens. des sémaphores ds la table en cas de succès (> 0)
-1 sinon.
- Création si `cle == IPC_PRIVATE && IPC_CREAT ⊂ flags`
(possibilité : `IPC_EXCL → flags`)
 - Création d'un *ensemble* de sémaphores qui auront tous les mêmes droits

Structures associées

- **Ensemble de sémaphores**

```
struct semid_ds {  
    struct ipc_perm    sem_perm;    /* operation permission struct */  
    struct sem          *sem_base;  /* ptr to first semaphore in set */  
    ushort              sem_nsems;  /* # of semaphores in set */  
    time_t              sem_otime;  /* last semop time */  
    time_t              sem_ctime;  /* last change time */  
    [...]                
};
```

- **Sémaphore individuel**

```
struct sem {  
    ushort semval;    /* semaphore counter value */  
    short  sempid;    /* pid of last operation */  
    ushort semncnt;   /* # awaiting semval > cval */  
    ushort semzcnt;   /* # awaiting semval = 0 */  
};
```


Initialisations associées à une création

■ Initialisations lors de la création d'une nouvelle entrée

- `sem_perm.cuid` et `sem_perm.uid` ← uid effectif du processus appelant
- `sem_perm.cgid` et `sem_perm.gid` ← gid effectif du processus appelant
- `sem_perm.mode` ← 9 bits de poids faible de l'entier `flags`
- `sem_nsems` ← `nsems`
- `sem_otime` ← 0
- `sem_ctime` ← heure courante

Opérations sur les sémaphores

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

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

struct sembuf {
    ushort sem_num;           /* semaphore # */
    short   sem_op;           /* semaphore operation */
    short   sem_flag;         /* operation flags : SEM_UNDO,
IPC_NOWAIT */
};

> retourne 0 en cas de succès et -1 sinon
```

Opérations sur les sémaphores (suite)

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

- chaque op `sem_op` est exécutée sur le sémaphore correspondant à `semid` et `sem_num`
- Les `nsops` opérations sont réalisés atomiquement
 - traitées soit **toutes** à la fois, soit pas du tout
- Chaque opération peut être rendu individuellement non bloquante avec `IPC_NOWAIT` dans `sem_flags`.
- Opérations réalisées en séquence des `nsops` opérations
 - Aspect bloquant ou non dépend de celui de la première opération qui n'est pas réalisable.

Opérations sur les sémaphores (cont)

- **Si $\text{sem_op} > 0$: $V_{\text{sem_op}}$**
 - La valeur du sémaphore est augmenté de sem_op .
 - Processus en attente sont réveillés
- **Si $\text{sem_op} = 0$: Z**
 - Le processus est bloqué tant que le sémaphore n'est pas nul
- **Si $\text{sem_op} < 0$: $P_{|\text{sem_op}|}$**
 - Si l'opération n'est pas réalisable le processus est bloqué (si $\text{sem_flag} ! = \text{IPC_NOWAIT}$)
 - Si l'opération possible, la valeur du sémaphore est décrémenté de $|\text{sem_op}|$
 - Si valeur devient nul, tous les processus en attente de la nullité du sémaphore sont réveillés

SEM_UNDO et IPC_NOWAIT

- **Options de `sem_flag`**

- **SEM_UNDO**

- Valeur d'ajustement sera automatiquement ajoutée au sémaphore à la terminaison du processus.
- La valeur de n est ajoutée pour une opération P_n et $-n$ pour une opération V_n

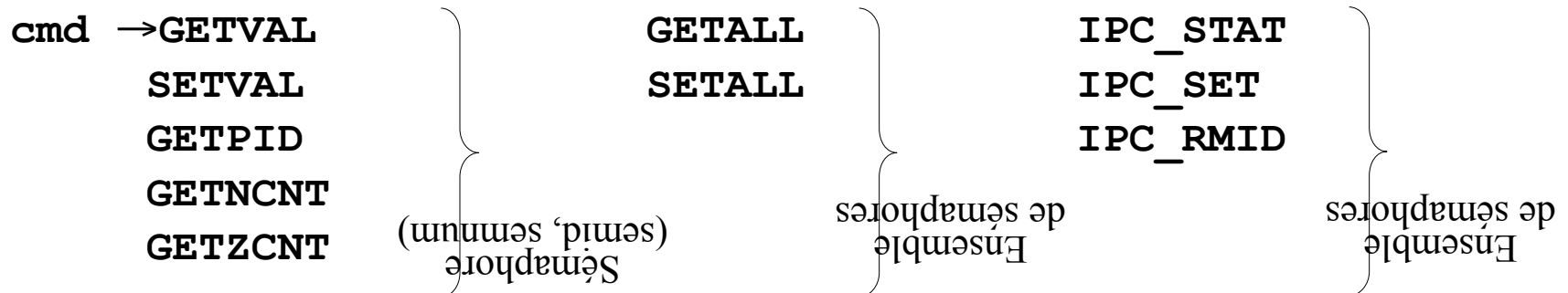
- **IPC_NOWAIT**

- Chaque opération peut être rendu individuellement non bloquante

Opérations de contrôle sur les sémaphores

```
#include<sys/ipc.h>
#include<sys/sem.h>
int semctl(int semid, int semnum, int cmd, union semun arg);

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



Example - semctl

```
union semun {
    int val;                // value for SETVAL
    struct semid_ds *buf;    // buffer for IPC_STAT, IPC_SET
    unsigned short int *array; // array for GETALL, SETALL
    struct seminfo *__buf;    // buffer for IPC_INFO
};

#define NS 3

int main(int argc, char* argv ){
    int      sem_id, sem_value, i;  key_t ipc_key;
    unsigned short int  sem_array[NS] = {3, 1, 2};
    union semun  arg;
    ipc_key = ftok(".", 'S');

    if ((sem_id = semget(ipc_key, NS, IPC_CREAT | 0660)) == -1) {
        perror("semget: IPC_CREAT | 0660"); return 1;
    }
}
```

Exemple – semctl (suite)

```
// set the semaphore value
arg.array = sem_array;
if (semctl(sem_id, 0, SETALL, arg) == -1) {
    perror("semctl: SETALL"); return 1;
}
// get the semaphore values
for (i = 0; i < NS; ++i) {
    if ((sem_value = semctl(sem_id, i, GETVAL, 0)) == -1) {
        perror("semctl: GETVAL"); return 1;
    }
    printf ("Semaphore %d : value %d \n », i ,sem_value );
}
//destruction des sémaphores
if (semctl(sem_id, 0, IPC_RMID, 0) == -1) {
    perror("semctl: IPC_RMID"); return 1;
}
}
```


Exemple P et V

```
void P (int sem)
{
    operation.sem_num = sem;
    operation.sem_op = -1;
    operation.sem_flg = SEM_UNDO;
    semop (sem_id, &operation, 1);
};
```

```
/* Primitive P () sur sémaphores */
/* sem = Identifiant du sémaphore */
/* Identification du sémaphore impliqué */
/* Définition de l'opération à réaliser */
/* Positionnement du bit SEM_UNDO */
/* Exécution de l'opération définie */
```

```
void V(int sem)
{
    operation.sem_num = sem;
    operation.sem_op = 1;
    operation.sem_flg = SEM_UNDO;
    semop (sem_id, &operation, 1);
};
```

```
/* Primitive V () sur sémaphores */
```

Exemple P et V_(suite)

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

#define SEM_EXCL_MUT          0
#define NB_SEM                1
char sem_path[14] = "ens_sem";      char sem_code = 'S';
int  FLAGS = 0666 | IPC_CREAT; key_t sem_cle; int  sem_id;
struct sembuf operation;

main ( ) {
    sem_cle = ftok(sem_path, sem_code);
    sem_id = semget (sem_cle , NB_SEM, FLAGS );
    semctl(sem_id, SEM_EXCL_MUT, SETVAL, 1);
    ...
        P(SEM_EXCL_MUT);
        /* Section Critique */
        V(SEM_EXCL_MUT);
    ...
    semctl(sem_id, SEM_EXCL_MUT, IPC_RMID, 0);
```

Exemple – ensemble sémaphore

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short int *array;
    struct seminfo *__buf;
} arg;

int  sem_id;
struct sembuf operations[3];
unsigned short sem_table [3] = {3,5,2};

main (int argc, char*argv);
    if (identSem=semget(IPC_PRIVATE,3,
        IPC_CREAT | IPC_EXCL | 0660))
        { perror("semctl: SETALL"); exit (3);
        }
arg.array = sem_table;
if (semctl(sem_id, 0, SETALL, arg == -1)
    { perror("semctl: SETALL"); exit (3);
    }
}
```

```
/* P(2) sur premier sémaphore */
operations[0].sem_num=0;
operations[0].sem_op=-2;
operations[0].sem_flg=0;

/* Z sur troisième sémaphore */
operations[2].sem_num=1;
operations[2].sem_op=0;
operations[2].sem_flg=0;

/* V(1) sur deuxième sémaphore */
operations[1].sem_num=2;
operations[1].sem_op=1;
operations[1].sem_flg=0;

semop(identSem,operations,3);
....
}
```

Exemple – opération Z

```
int main(int argc, char **argv)
{
    key_t clef;
    int semid;
    struct sembuf op;

    clef = ftok("\tmp", 0);
    semid = semget(clef, 1,
                  IPC_CREAT | IPC_EXCL | 0666);
    semctl(semid, 0, SETVAL, 1);

    op.sem_num = 0;
    op.sem_flg = 0;
```

```
    if (fork () != 0 )
    { op.sem_op = -1; //P
      semop(semid, &op, 1);

      wait (NULL);
      //Destruction du sémaphore
      semctl(semid, 0, IPC_RMID, 0);
    }
    else {
        // attendre le sémaphore devenir nul
        op.sem_op = 0;
        semop(semid, &op, 1);
        printf ("compteur semaphore=zero \n");
    }
}
```

```
return 0;
```

Résumé

	Files de msgs	Mémoire partagée	sémaphores
includes	sys/types.h sys/ipc.h sys/msg.h	sys/types.h sys/ipc.h sys/shm.h	sys/types.h sys/ipc.h sys/sem.h
Création/ouverture	msgget()	shmget()	semget()
contrôle	msgctl ()	shmctl ()	semctl ()
opérations	msgsnd () msgrcv ()	shmat () shmdt ()	semop ()
Structure associé	msqid_ds	shmid_ds	semqid_ds

Les IPC

Inter process communication

I –IPC : Outils et principes

II – Les IPC System V

➔ III – Les IPC POSIX

Files de messages POSIX

Fichier <mqueue.h>

- **Fonctions contenues dans la bibliothèque librt (real-time)**

\$ gcc -Wall -o monprog monprog.c -lrt

- **Accès**

- mq_open ⇒ créer / ouvrir une file en mémoire
- 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 ⇒ demander à être prévenu de l'arrivée d'un message

Attributs d'une file 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

Ouverture d'une file de messages

```
#include <mqueue.h>
```

```
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 (castable en int) positif en cas de succès, -1 sinon
- *flags* idem open
- *attrs* peut être rempli avant pour définir des valeurs (sauf *mq_flags*)

mq_flags consultable/modifiable avec :

```
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);
```

Fermeture d'une file de messages

- **`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) ;`**
 - 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 ont accès ont appelé **`mq_close`**

Ajout de message

- `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, $0 \leq \text{priority} \leq \text{MQ_PRIOMAX}$ (≥ 32 , déf. dans *limits.h*)
Si $\text{priority} > \text{MQ_PRIOMAX}$, l'appel échoue
 - File ordonnée par priorités, en FIFO pour les messages de même priorité
 - Appel bloquant si la file est **pleine** et `O_NONBLOCK` non spécifié

Retrait de message

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

- retourne le nb de bytes lus 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

Exemple file de message

```
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

int main (int argc, char* argv []) {
    mqd_t mqdes;
    char buf[ 100];
    unsigned int prio;

    if ( (mqdes = mq_open ("/file1",
O_RDWR | O_CREAT,
    0666, NULL)) == -1) {
        perror ("mq_open");
        exit (1);  }

    if (fork ( ) == 0) {
        if (mq_receive (mqdes, buf, MSG_SIZE,
            &prio) == -1) {
            perror ("mq_rec"); exit (1); }
        printf ("message:%s  prio:%d.\n",buf,prio);
    }
    else {
        if (mq_send (mqdes, "abcd",4, 0) == -1) {
            perror ("mq_send"); exit (1); }
        wait (NULL);
    }
    mq_close (mqdes);
    mq_unlink ("/file1");
    return (0);
}
```

Notification d'arrivée de message

```
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, désenregistrement de la demande

```
union sigval {
    int sival_int;   void *sival_ptr;
};

struct sigevent {
    int sigev_notify;           // SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD
    int sigev_signo;           //Signal number
    union sigval sigev_value; //Notif. data
    void (*)(union sigval) sigev_notify_function //Thread function
    void *sigev_notify_attributes; /* Thread function attributes */
};
```

Exemple mq_notify

```
#include <pthread.h>
#include <mqueue.h>
char buf[100]

static void /* Thread start function */
tfunc(union sigval sv) {
    char buf[100];
    mqd_t mqdes = *((mqd_t *) sv.sival_ptr);

    if ((nr=mq_receive(mqdes, buf, 100,
        NULL) )== -1) {
        perreur(" mq_receive" ); exit (1)}

    printf(« lu %ld « , (long) nr);
    exit(EXIT_SUCCESS);
}
```

```
int main(int argc, char *argv[])
{
    mqd_t mqdes;
    struct sigevent not;

    mqdes=mq_open(argv[1],O_RDONLY);
    if (mqdes == (mqd_t) -1) {
        perror("mq_open ");exit(1); }
    not.sigev_notify = SIGEV_THREAD;
    not.sigev_notify_function = tfunc;
    not.sigev_notify_attributes = NULL;
    not.sigev_value.sival_ptr = &mqdes;
    if (mq_notify(mqdes, &not) == -1) {
        perror("mq_notify"); exit(1); }
    pause();
}
```

Mémoire partagée POSIX

Fichier <sys/mman.h>

Fonctions contenues dans la bibliothèque librt (real-time)

```
$ gcc -Wall -o monprog monprog.c -lrt
```

- **Accès**

- shm_open ⇒ créer / ouvrir un segment en mémoire
- 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

Mémoire partagée POSIX

Savoir si un système implémente la mémoire partagée POSIX

Fichier <unistd.h>

mmap	_POSIX_MAPPED_FILES ou _POSIX_SHARED_MEMORY_OBJECTS
munmap	_POSIX_MAPPED_FILES ou _POSIX_SHARED_MEMORY_OBJECTS
shm_open	_POSIX_SHARED_MEMORY_OBJECTS
shm_unlink	_POSIX_SHARED_MEMORY_OBJECTS
ftruncate	_POSIX_MAPPED_FILES ou _POSIX_SHARED_MEMORY_OBJECTS
mprotect	_POSIX_MEMORY_PROTECTION
msync	_POSIX_MAPPED_FILES et _POSIX_SYNCHRONIZED_IO

Ouverture / Destruction d'un segment de mémoire partagée

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

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

- Crée un nouveau segment de taille 0
ou recherche le descr. d'un segment déjà existant
- Retourne un descripteur positif en cas de succès, -1 sinon
- *flags* idem open
- *mode* idem chmod

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

- Idem mq_unlink

Attachement / Détachement d'un segment de mémoire partagée

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

- Retourne NULL en cas d'échec,
l'@ d'un attachement de taille *len* à partir d'*offset* ds le segment de descr *fd* sinon
- *Addr* : adresse où attacher le segment en mémoire ; 0 ⇒ choix du système
- *prot*: protection associée (PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE)
- *flags*: mode de partage
 - MAP_SHARED : modifs visibles par tous les pcs ayant accès (partage)
 - MAP_PRIVATE : modifs visibles par le pcs appelant uniquement (shadow copy)
 - MAP_FIXED : force l'utilisation d'*addr*

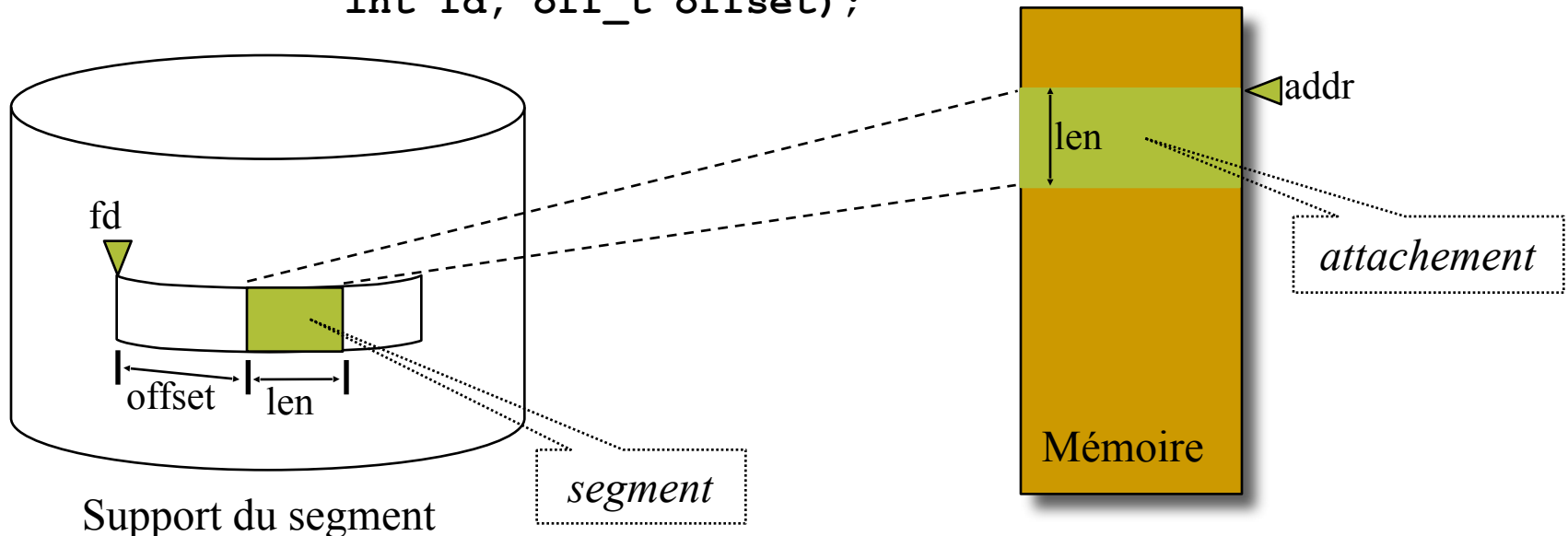
```
int munmap(caddr_t addr, size_t len);
```

- Détruit l'attachement de taille *len* à l'adresse *addr*
- Retourne -1 en cas d'échec, 0 sinon

Projection des fichiers

- Permet de projeter dans l'espace d'adressage du processus un segment de longueur *len* le segment $[offset, offset+len]$ du fichier associé à *fd*.

```
#include<sys/mman.h>
#include<sys/types.h>
void * mmap(void *addr, size_t len, int prot, int flags,
            int fd, off_t offset);
```



Exemple mmap

```
int main (int argc, char* argv []) {
    int fd1, fd2;  int *adr1;

    fd1 = open (argv[1], O_RDWR);
    fd2 = open (argv[1], O_RDWR);

    if ( (fd1 == -1) || (fd2 == -1) ) {
        printf ("open %s", argv[1]);
        return EXIT_FAILURE;
    }
```

```
    adr1=(char *) mmap (NULL,1024,PROT_READ|
    PROT_WRITE,MAP_SHARED, fd1,0);
```

```
    adr2=(char *) mmap (NULL,1024,PROT_READ|
    PROT_WRITE,MAP_SHARED, fd2,0);
```

```
    printf ("2- ème caractère adr1:%c \n", adr1[1] );
    adr[1]++;
    printf ("2- ème caractère adr1:%c \n", adr2[1] );
}
```

mmap_teste.c

```
> mmap_teste
2- ème caractère adr1 : a
2- ème caractère adr2 : b
```

Opérations sur un segment de mémoire partagée

```
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 du segment de descr. *fd*
 - nouvelle taille = length
 - si (ancienne taille > nouvelle taille), alors les données en excédent sont perdues
- Retourne -1 en cas d'échec, 0 sinon

Exemple1 – mémoire partagée

```
int *sp;
int main() {
    int fd;
    /* Creer le segment monshm, ouverture en R/W */
    if ((fd = shm_open("/monshm", O_RDWR | O_CREAT,
        0600)) == -1) {
        perror("shm_open");
        exit(1);}

    /* Allouer au segment une taille pour stocker un entier */
    if (ftruncate(fd, sizeof(int)) == -1) {
        perror("ftruncate");
        exit(1);}

    /* “mapper” le segment en R/W partagé */
    if ((sp = mmap(NULL, sizeof(int), PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, 0))
        == MAP_FAILED) {
        perror("mmap");
        exit(1);}

}
```

```
/* Accès au segment */
*sp = 10;

/* “detacher” le segment */
munmap(sp, sizeof(int));

/* detruire le segment */
shm_unlink("monshm");
return 0;
```

Exemple2 – mémoire partagée (sans mmap)

- Lire et écrire comme dans un fichier

```
char buf [20];
int main( int argc, char** argv )
{ int fd;
  if (argc > 3) {
    shm_unlink( "/monshm" );
    return EXIT_FAILURE;}
  fd = shm_open( "/monshm", O_RDWR
    | O_CREAT, 0666 );
  if( fd == -1 ) {
    fprintf( stderr, "Open failed:%s\n",
      strerror( errno ) );
    return EXIT_FAILURE;
  }
```

shm_ex2.c

```
if (fork() ==0) {
  write (fd,"ABCDEF",6);
  lseek (fd,SEEK_SET,0);
  read (fd,buf,3);
  printf ("fils: %s\n", buf); }
else {
  wait (NULL);
  read (fd, buf, 10);
  printf ("pere: %s\n", buf);}
close( fd );
shm_unlink("/bolts");
return EXIT_SUCCESS;
}
```

```
>shm_ex2 fic2
fils: ABC
Père: DEF
```


Sémaphores POSIX

- **Deux 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 : processus avec filiation, uniquement threads dans linux
 - Primitives de base : **sem_init**, **sem_destroy**, **sem_post**, **sem_wait**

- **Inclus dans la bibliothèque des pthreads**

- ```
$ gcc -Wall -o monprog monprog.c -lpthread
```

# Création de sémaphore nommé

---

```
#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 : Creation d'un sémaphore initialisé à 10**

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

# Création de sémaphore anonyme

---

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

- Crée et initialise le sémaphore *sem*.
  - *sem* : doit être alloué dans l'espace d'adressage du processus
- *pshared* != 0
  - partageable entre processus
- *pshared* == 0
  - partageable entre threads
- *val*: valeur initiale du sémaphore

Retourne -1 en cas d'erreur, 0 sinon

***Ex : création de sémaphore partagé, initialisé à 10***

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

# Opérations sur sémaphore

---

- **Opération P**

**int sem\_wait (sem\_t \*sem);**

- Attendre que le compteur soit supérieur à zéro et le décrémenter avant de revenir.

- **Opération V**

**int sem\_post (sem\_t \*sem);**

- Compteur incrémenté; un processus/thread en attente est libérée.

- **Opération P non bloquant**

**int sem\_trywait (sem\_t \*sem);**

- Fonctionnement égal à *sem\_wait* mais non bloquante.

- **Consultation compteur sémaphore**

**int sem\_getvalue (sem\_t \*sem, int \*valeur);**

- Renvoie la valeur du compteur du sémaphore *sem*. dans \*valeur.

# Fermeture / Destruction

---

- **Sémaphore nommé :**

- Fermer le sémaphore
- `int sem_close(sem_t *sem) ;`
- Détruire le sémaphore
- `int sem_unlink(const char *name) ;`

- **Sémaphore anonyme :**

- `int sem_destroy(sem_t *sem) ;`

# Exemple : sémaphores nommés

```
...
int main() {
 sem_t *smutex;

 /* creation d'un semaphore mutex initialisé à 1 */
 if ((smutex = sem_open("/monsem",
 O_CREAT | O_EXCL | O_RDWR, 0666, 1)) ==
 SEM_FAILED) {
 if (errno != EEXIST) {
 perror("sem_open"); exit(1);
 }

 /* Semaphore déjà créé, ouvrir sans O_CREAT */
 smutex = sem_open("/monsem", O_RDWR);
 }
}
```

```
/* P sur smutex */
sem_wait(smutex);

/* V sur smutex */
sem_post(smutex);

/* Fermer le semaphore */
sem_close(smutex);

/* Detruire le semaphore */
sem_unlink("/monsem");
return 0;
```