

Les IPC System V

Cyril Rabat

`cyril.rabat@univ-reims.fr`

Licence 3 Informatique - Info0601 - Systèmes d'exploitation - concepts avancés

2021-2022



Cours n°9

Présentation des mécanismes IPC System V
Files de messages, mémoire partagée, sémaphores

Version 10 février 2022

Table des matières

- 1 Les IPC *System V*
- 2 Les files de messages
- 3 Les segments de mémoire partagée
- 4 Les tableaux de sémaphores

Présentation des IPC System V

- IPC pour *Inter Process Communication*
- Trois types :
 - Les tableaux de sémaphores
 - Les files de messages
 - Les segments de mémoire partagée
- Gérés au niveau du système par trois tables indépendantes
- Indépendants du VFS
 - ↪ N'utilisent pas de descripteur de fichier

Convention de nommage (en C)

- Chaque outil :
 - Correspond à un préfixe
 - Possède un jeu de fonctions propre :
 - ↪ Chacune commençant par un même préfixe
- Tableaux de sémaphores :
 - ↪ Préfixe : `sem`
 - ↪ Fonctions : `semget`, `semop`, `semctl`
- Files de messages :
 - ↪ Préfixe : `msg`
 - ↪ Fonctions : `msgget`, `msgsnd`, `msgrcv`, `msgctl`
- Segments de mémoire partagée :
 - ↪ Préfixe : `shm` (pour *Shared Memory*)
 - ↪ Fonctions : `shmget`, `shmat`, `shmdt`, `shmctl`

Accès aux IPC

- Chaque outil :
 - ↪ Identification par une **clé externe unique** (type `key_t`)
 - ↪ Manipulation par un **descripteur interne**
- Comparaison avec la manipulation d'un fichier :
 - ↪ clé externe \Rightarrow nom du fichier
 - ↪ descripteur interne \Rightarrow descripteur de fichier
- Connaissance du descripteur interne :
 - ↪ Via un appel système en fournissant la clé
 - ↪ Par héritage
- Comment deux processus peuvent accéder au même outil ?
 - ❶ Clé stockée en dur dans l'application
 - ↪ Avec un `#DEFINE` ou via le `makefile`
 - ❷ Échange de la clé (via un moyen de communication quelconque)
 - ❸ Calcul à partir d'un nom de fichier et d'une valeur :
 - ↪ Fonction `ftok`

Il est possible d'utiliser la même clé externe pour les trois outils.

Fonction `ftok`

- **En-tête de la fonction (S3) :**

- `key_t ftok(char *pathname, int proj_id)`
- *Inclusions* : `sys/types.h` et `sys/ipc.h`

- **Paramètre(s) :**

- `pathname` : nom de fichier
- `proj_id` : valeur quelconque (attention, 8 bits de poids faible utilisés)

- **Valeurs retournées et erreurs générées :**

- Une clé **qu'on espère unique** ou -1 en cas d'erreur
- Erreurs possibles (les mêmes que `stat`) :
 - `ENOENT` : nom de fichier invalide
 - `EACCESS` : accès interdit

Généralement, nous n'utiliserons pas cette solution en INFO0601.

Shell : liste des IPC

- Liste des IPC (accessibles) : commande `ipcs`
- Attention : clé en hexadécimal !

Illustration

```
> ipcs
```

```
----- Segment de mémoire partagée -----
```

clé	shmid	propriétaire	perms	octets	nattch	états
0x00000dc0	2981893	cyril	600	1000	0	
...						

```
----- Tableaux de sémaphores -----
```

clé	semid	propriétaire	perms	nsems
0x000001c2	786433	cyril	600	5
...				

```
----- Files de messages -----
```

clé	msqid	propriétaire	perms	octets utilisés	messages
0x000007e0	262147	cyril	600	0	0
...					

Shell : création et suppression d'IPC

- Suppression : commande `ipcrm`
 - ↪ File de messages : `-Q clé` ou `-q identificateur`
 - ↪ Segment de mémoire partagée : `-M clé` ou `-m identificateur`
 - ↪ Tableau de sémaphores : `-S clé` ou `-s identificateur`
- Création : commande `ipcmk`
 - ↪ File de messages : `-Q`
 - ↪ Segment de mémoire partagée : `-M taille`
 - ↪ Tableau de sémaphores : `-S taille`
 - ↪ Mode : `-p mode`

ipcmk est une commande Linux. Pas de normalisation !

Présentation des files de messages

- Fonctionnement type boîte à lettres
- Envoi/réception de messages
- Deux types d'acteurs :
 - ↔ Le producteur : envoi de messages
 - ↔ Le consommateur : lecture des messages
- Un processus peut avoir les deux rôles
- **Pas de synchronisation nécessaire :**
 - ↔ Gestion propre au système

Quelques constantes liées aux files de messages

Affichage via la commande `ipcs`

```
> ipcs -l
```

```
----- Limites de messages -----
```

```
nombre maximal de files système = 2002
```

```
taille maximale des messages (octets) = 8192
```

```
taille maximale par défaut des files (octets) = 16384
```

Ces constantes dépendent de la configuration du noyau.

Création et accès à une file de messages

- Utilisation de la fonction `msgget` :
 - Création d'une file
 - Accès à une file existante
- Possibilité de créer une file "locale" :
 - Utilisation de la clé `IPC_PRIVATE` (constante) :
 - ↪ La clé est dans ce cas 0
 - Accès ensuite par les processus fils :
 - ↪ Héritage du descripteur
 - Mais non protégée de l'extérieur !
 - ↪ ... à condition d'obtenir l'identificateur interne

Fonction msgget (1/2)

En-tête de la fonction (S2)

- `int msgget(key_t cle, int options)`
- *Inclusions* : `sys/types.h`, `sys/ipc.h` et `sys/msg.h`

Paramètre(s)

- `cle` : la clé ou `IPC_PRIVATE`
- `options` :
 - `IPC_CREAT` : crée la file
 - `IPC_EXCL` : génère une erreur si la file existe déjà
 - Les modes d'accès (voir les fichiers)

Contrairement à la fonction `open`,
les droits d'accès sont spécifiés via le paramètre `options`

Fonction msgget (2/2)

Valeurs retournées et erreurs générées

- Retourne l'identificateur interne ou -1 en cas d'échec
- Quelques erreurs possibles :
 - EEXIST : la file existe déjà (IPC_CREAT + IPC_EXCL)
 - EACCES : accès interdit
 - ENOENT : pas de file associée à la clé (et pas de IPC_CREAT)
 - ENOSPC : nombre maximum de files atteint

Exemple de création d'une file de messages

```
#define CLE 2021

int main() {
    int msqid;

    if((msqid = msgget(CLE,
                      S_IRUSR | S_IWUSR |
                      IPC_CREAT | IPC_EXCL)) == -1) {
        if(errno == EEXIST)
            fprintf(stderr, "File_(cle=%d)_existante\n", CLE);
        else
            perror("Erreur_lors_de_la_creation_");
        exit(EXIT_FAILURE);
    }

    /* Utilisation de la file */

    return EXIT_SUCCESS;
}
```

Visualisation depuis le shell

Vérification de la création

```
> ipcs -q
```

```
----- Files de messages -----
```

clé	msqid	propriétaire	perms	octets utilisés	messages
0x000007e5	262147	cyril	600	0	0

Affichage des informations

```
> ipcs -q -i 262147
```

```
File de messages msqid=262147
```

```
uid=1000 gid=1000 cuid=1000 cgid=1000 mode=0600
```

```
cbytes=0 qbytes=16384 qnum=0 lspid=0 lrpipid=0
```

```
send_time=Non initialisé
```

```
rcv_time=Non initialisé
```

```
change_time=Mon Jan 25 07:32:04 2021
```

Exemple de création d'une file privée

```
int main() {
    int msqid;

    if((msqid = msgget(IPC_PRIVATE,
                      S_IRUSR | S_IWUSR |
                      IPC_CREAT | IPC_EXCL)) == -1) {
        perror("Erreur_lors_de_la_creation_de_la_file_");
        exit(EXIT_FAILURE);
    }

    /* Utilisation de la file */

    return EXIT_SUCCESS;
}
```


Visualisation depuis le shell

Vérification de la création

```
> ipcs -q
```

```
----- Files de messages -----
```

clé	msqid	propriétaire	perms	octets utilisés	messages
0x00000000	294916	cyril	600	0	0

Affichage des informations

```
> ipcs -q -i 294916
```

```
File de messages msqid=294916
```

```
uid=1000  gid=1000      cuid=1000  cgid=1000  mode=0600
```

```
cbytes=0  qbytes=16384  qnum=0    lspid=0    lrpid=0
```

```
send_time=Non initialisé
```

```
rcv_time=Non initialisé
```

```
change_time=Mon Jan 25 07:45:21 2021
```

Les messages

- Utilisation de structures pour l'envoi/la réception
- Premier champ obligatoire = type du message :
 - ↪ De type `long` mais **strictement** positif
- Les autres champs : personnalisés
- Correspondent en pratique à une zone mémoire (un tampon) :
 - ↪ Toutes les données doivent être contiguës en mémoire
 - ↪ Pas de pointeurs (*cf* cours n°4)

```
struct message_t {  
    long type;          /* Obligatoire */  
    /* Données utilisateur */  
}
```

Envoi de messages

- Utilisation de la fonction `msgsnd` :
 - ↪ Nécessite d'avoir l'identificateur interne de la file
- Passage de l'adresse mémoire du message + la taille :
 - ↪ Attention : **la taille ne prend pas en compte le champ `type` !**
 - ↪ Une copie mémoire est réalisée
- La primitive est bloquante :
 - ↪ Blocage si la file est pleine
 - ↪ Possible de la rendre non bloquante avec `IPC_NOWAIT`

Fonction msgsnd (1/2)

En-tête de la fonction (S2)

- `int msgsnd(int msqid, struct msgbuf * msg, size_t taille, int options)`
- *Inclusions* : `sys/types.h`, `sys/ipc.h` et `sys/msg.h`

Paramètre(s)

- `msqid` : l'identificateur interne de la file
- `msg` et `taille` : l'adresse mémoire du message et sa taille
- `options` :
 - `IPC_NOWAIT` : rend l'écriture non bloquante
 - `MSG_NOERROR` : pas d'erreur si message trop grand (tronqué)

Fonction msgsnd (2/2)

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
 - EAGAIN : message trop grand
 - EACCES : accès interdit
 - EFAULT : pointeur hors de l'espace d'adressage
 - EINTR : interruption par un signal
 - EINVAL : identificateur non valide

Exemple d'envoi d'un message

```
typedef struct {  
    long type;           /* Type du message */  
    char msg[256];       /* Texte */  
} message_t;  
  
...  
message_t msg = { 1, "Bonjour" };  
  
if(msgsnd(msqid,  
          &msg,  
          sizeof(message_t) - sizeof(long),  
          0) == -1) {  
  
    ...  
}
```

Réception de messages

- Utilisation de la fonction `msgrcv` :
 - ↪ Nécessite d'avoir l'identificateur interne de la file
- Récupère le prochain message ...
- ... ou le prochain message correspondant à un type spécifié
- Lecture bloquante :
 - ↪ Possible de la rendre non bloquante
 - ↪ Utilisation de l'option `IPC_NOWAIT`
- La lecture détruit le message de la file

Fonction `msgrcv` (1/2)

En-tête de la fonction (S2)

- `ssize_t msgrcv(int msqid, struct msgbuf * msg, size_t taille, long type, int options)`
- *Inclusions* : `sys/types.h`, `sys/ipc.h` et `sys/msg.h`

Paramètre(s)

- `msqid` : l'identificateur de la file
- `msg` et `taille` : l'adresse mémoire du message et la taille
- `type` : le type du message
 - 0 : premier message qui vient
 - >0 : premier message du type spécifié
 - <0 : premier message dont le type inférieur ou égal à `|type|`
- `options` :
 - `IPC_NOWAIT` : rend la lecture non bloquante (erreur `ENOMSG`)
 - `MSG_EXCEPT` : lit les messages qui ne sont pas du type spécifié
 - `MSG_NOERROR` : tronque le message si sa taille est trop grande

Fonction `msgrcv` (2/2)

Valeurs retournées et erreurs générées

- Retourne la taille du message lu (sans le type) ou -1 en cas d'échec
- Quelques erreurs possibles :
 - `E2BIG` : message trop grand (et pas `MSG_NOERROR`)
 - `EACCES` : accès interdit
 - `EFAULT` : pointeur hors de l'espace d'adressage
 - `EINTR` : interruption par un signal
 - `ENOMSG` : pas de message et `IPC_NOWAIT` spécifié
 - `EINVAL` : identifiant ou taille invalide

Exemple de réception d'un message

```
typedef struct {  
    long type;  
    char msg[256];  
} message_t;  
  
...  
message_t msg;  
  
if(msgrcv(msqid,  
    &msg,  
    sizeof(message_t) - sizeof(long),  
    0,  
    0) == -1) {  
  
    ...  
}
```

Structure msqid_ds

- msqid_ds : structure associée à la file de messages
- Mise à jour en fonction des envois/réceptions

```
struct msqid_ds {  
    struct ipc_perm msg_perm; /* permissions */  
    time_t msg_stime;         /* heure dernier appel à msgsnd */  
    time_t msg_rtime;         /* heure dernier appel à msgrcv */  
    time_t msg_ctime;         /* heure dernière modification */  
    msgqnum_t msg_qnum;       /* nombre de messages dans la file */  
    msglen_t msg_qbytes;      /* taille max. de la file */  
    pid_t msg_lspid;          /* pid du dernier processus qui a  
                               appelé msgsnd() */  
    pid_t msg_lrpid;          /* pid du dernier processus qui a  
                               appelé msgrcv() */  
};
```

Récupération des informations ou suppression de la file

- Fonction `msgctl` permet :
 - ↪ De récupérer les informations sur une file de messages
 - ↪ De modifier les informations sur une file de messages
 - ↪ De supprimer une file de messages
- Dépend d'un paramètre appelé `commande` :
 - `IPC_RMID` : supprime la file
 - `IPC_STAT` : récupère les informations
 - `IPC_SET` : modifie les informations
- Un troisième paramètre (sauf pour `IPC_RMID`) est utilisé :
 - ↪ Pour récupérer ou modifier les infos
- Ce qui peut être modifié :
 - ↪ Les permissions
 - ↪ La taille (seul `root` autorisé pour `taille > MSGMNB`)

Fonction `msgctl` (1/2)

En-tête de la fonction (S2)

- `int msgctl(int msqid, int commande, struct msqid_ds *buf)`
- *Inclusions* : `sys/types.h`, `sys/ipc.h` et `sys/msg.h`

Paramètre(s)

- `msqid` : l'identificateur de la file
- `commande` :
 - `IPC_RMID` : supprime la file
 - `IPC_STAT` : récupère les informations (dans le paramètre `buf`)
 - `IPC_SET` : modifie les informations
- `buf` : données à modifier ou récupérer

Fonction `msgctl` (2/2)

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
 - `EACCES` : accès interdit
 - `EFAULT` : `buf` invalide (avec `IPC_STAT` et `IPC_SET`)
 - `EINVAL` : des paramètres possèdent une valeur incorrecte

Exemple de suppression d'une file de messages

```
#define CLE 2021

/* Récupération de la file */
if((msqid = msgget(CLE, 0)) == -1) {
    perror("Erreur_lors_de_la_récupération_de_la_file_");
    exit(EXIT_FAILURE);
}

/* Suppression de la file */
if(msgctl(msqid, IPC_RMID, 0) == -1) {
    perror("Erreur_lors_de_la_suppression_de_la_file_");
    exit(EXIT_FAILURE);
}
```

Présentation des segments de mémoire partagée

- Chaque processus possède son propre espace d'adressage :
↪ Impossible pour un autre processus d'y accéder
- Idée de cet outil IPC :
↪ Créer un segment de mémoire, partagé entre plusieurs processus
- Extension de la mémoire du processus :
↪ Chaque processus doit l'attacher à son espace d'adressage
- Fonctionne de la même manière que les files de messages :
↪ Segment identifié par une clé
↪ Les processus qui la connaissent peuvent s'y attacher

Comme pour les *threads*, la mémoire est partagée
donc attention aux accès concurrents!

Quelques constantes liées aux segments mémoire

Affichage via la commande `ipcs`

```
> ipcs -l
```

```
----- Limites de la mémoire partagée -----  
nombre maximal de segments = 4096  
taille maximale de segments (kilooctet) = 32768  
total de mémoire partagée maximal (kilooctet) = 8388608  
taille minimale de segments (octet) = 1
```

Ces constantes dépendent de la configuration du noyau.

Création et accès à un segment de mémoire

- Utilisation de la fonction `shmget` :
 - ↪ Création d'un segment mémoire
 - ↪ Accès à un segment existant
- Même principe que pour `msgget` :
 - ↪ On précise en plus la taille (en octets)
- Taille du segment :
 - Arrondie (en mémoire) au multiple supérieur de `PAGE_SIZE`
 - Comprise dans un intervalle fixé par le système

Le segment est créé / récupéré, mais pas utilisable par le processus !

Fonction shmget (1/2)

En-tête de la fonction (S2)

- `int shmget(key_t cle, int taille, int options)`
- *Inclusions* : `sys/ipc.h` et `sys/shm.h`

Paramètre(s)

- `cle` : la clé ou `IPC_PRIVATE`
- `taille` : la taille du segment
- `options` :
 - `IPC_CREAT` : alloue le segment mémoire
 - `IPC_EXCL` : génère une erreur si le segment existe déjà
 - Les modes d'accès (voir les fichiers)

Fonction shmget (2/2)

Valeurs retournées et erreurs générées

- Retourne l'identificateur interne ou -1 en cas d'échec
- Quelques erreurs possibles :
 - `EEXIST` : segment existe déjà (`IPC_CREAT + IPC_EXCL`)
 - `EINVAL` : segment trop petit ou trop grand
 - `ENOENT` : pas de segment associé à la clé (et pas de `IPC_CREAT`)
 - `ENOSPC` : nombre maximum de segments atteint ou mémoire maximum atteinte

Exemple de création

```
#define CLE 3520

int main() {
    int shmid;

    if((shmid = shmget(CLE, 1000,
                      S_IRUSR | S_IWUSR |
                      IPC_CREAT | IPC_EXCL)) == -1) {
        if(errno == EEXIST)
            fprintf(stderr, "Segment_(cle=%d)_existant\n", CLE);
        else
            perror("Erreur_lors_de_la_création_du_segment_");
        exit(EXIT_FAILURE);
    }

    /* Suite du code */

    return EXIT_SUCCESS;
}
```

Visualisation depuis le shell

Vérification de la création

```
> ipcs -m
```

```
----- Segment de mémoire partagée -----
```

clé	shmid	propriétaire	perms	octets	nattch	états
0x00000dc0	2981893	cyril	600	1000	0	

Affichage des informations

```
> ipcs -m -i 2981893
```

```
Mémoire partagée segment shmid=2981893
uid=1000 gid=1000 cuid=1000 cgid=1000
mode=0600 access_perms=0600
octets=1000 lpid=0 cpid=9559 nattch=0
att_time=Non initialisé
det_time=Non initialisé
change_time=Sat Jan 30 13:49:07 2021
```

Attachement/détachement du segment mémoire

- Pour utiliser un segment mémoire :
 - Le processus doit attacher le segment à son espace d'adressage
↔ En lecture ou en lecture/écriture
 - Utilisation de la fonction `shmat`
 - Possible de l'attacher plusieurs fois
- Le processus doit ensuite se détacher :
 - Utilisation de la fonction `shmdt`
 - Détache une instance uniquement
 - Ne détruit pas le segment

Un segment mémoire ne peut pas être détruit
si un processus y est encore attaché.

Fonction shmat (1/2)

En-tête de la fonction (S2)

- `void *shmat(int shmid, const void *shmaddr, int options)`
- *Inclusions* : `sys/types.h` et `sys/shm.h`

Paramètre(s)

- `shmid` : l'identificateur du segment mémoire
- `shmaddr` : mettre à NULL
↪ Possible de spécifier une adresse mais alignée sur une page
- `options` :
 - `SHM_RDONLY` : segment en lecture seule
 - `SHM_RND` : aligne le segment sur une page

Fonction `shmat` (2/2)

Valeurs retournées et erreurs générées

- Retourne l'adresse d'attachement ou `(void*)-1` en cas d'échec
- Quelques erreurs possibles :
 - `EACCES` : pas les permissions
 - `EINVAL` : `shmid` incorrect, `shmaddr` non aligné
 - `ENOMEM` : pas assez de mémoire

Fonction `shmdt`

En-tête de la fonction (S2)

- `int shmdt(const void *shmaddr)`
- *Inclusions* : `sys/types.h` et `sys/shm.h`

Paramètre(s)

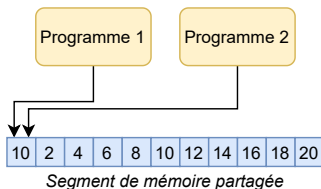
- `shmaddr` : adresse d'attachement (retournée par `shmat`)

Valeurs retournées et erreurs générées

- Retourne 0 en cas de réussite ou -1 en cas d'échec
- Erreur possible :
 - `EINVAL` : `shmaddr` invalide ou pas de segment

Présentation de l'exemple

- Deux programmes différents attachés à un segment de mémoire partagée
- Premier programme :
 - Crée un segment de mémoire partagée
 - Stocke la taille du tableau et les entiers
- Deuxième programme :
 - S'attache au segment de mémoire partagée
 - Récupère le tableau d'entiers



La taille du segment est exactement de 11 entiers

Exemple complet (1/4) : `memoire1.c` (début)

```
#define CLE 1056

int main() {
    int shmid;
    int *adresse;

    if((shmid = shmget(CLE,
                      sizeof(int) * 11,
                      S_IRUSR | S_IWUSR | IPC_CREAT)) == -1) {
        perror("Erreur_lors_la_récupération_du_segment_");
        exit(EXIT_FAILURE);
    }

    if((adresse = shmat(shmid, NULL, 0)) == (void*)-1) {
        perror("Erreur_lors_de_l'attachement_");
        exit(EXIT_FAILURE);
    }

    ...
}
```

Exemple complet (2/4) : `memoire1.c` (fin)

```
...
int i;
adresse[0] = 10;
for(i = 1; i <= 10; i++)
    adresse[i] = i * 2;

if(shmdt(adresse) == -1) {
    perror("Erreur_lors_du_détachement_");
    exit(EXIT_FAILURE);
}

return EXIT_SUCCESS;
}
```

Exemple complet (3/4) : `memoire2.c` (début)

```
#define CLE 1056

int main() {
    int shmid;
    int *adresse;

    if((shmid = shmget(CLE, 0, 0)) == -1) {
        perror("Erreur_lors_de_la_récupération_du_segment_");
        exit(EXIT_FAILURE);
    }

    if((adresse = shmat(shmid, NULL, 0)) == (void*)-1) {
        perror("Erreur_lors_de_l'attachement_");
        exit(EXIT_FAILURE);
    }

    ...
}
```

Exemple complet (4/4) : `memoire2.c` (fin)

```
...
int i;
printf("[");
for(i = 1; i <= adresse[0]; i++) {
    printf("%d", adresse[i]);
    if(i < adresse[0]) printf(",_");
}
printf("]\n");

if(shmdt(adresse) == -1) {
    perror("Erreur_lors_du_détachement_");
    exit(EXIT_FAILURE);
}

return EXIT_SUCCESS;
}
```

exit, exec et fork

- Après un `exit` :
 - ↪ Tous les segments sont détachés
- Après un `fork` :
 - ↪ Héritage des segments de mémoire partagée
- Après un `exec` :
 - ↪ Tous les segments sont détachés

Récupération des informations ou suppression du segment

- Fonction `shmctl` permet :
 - De récupérer les informations sur un segment mémoire
 - De modifier les informations sur un segment mémoire
 - De supprimer un segment mémoire
- Dépend d'un paramètre appelé commande :
 - `IPC_RMID` : marque un segment pour la destruction
 - `IPC_STAT` : récupère les infos
 - `IPC_SET` : modifie les infos
- Un troisième paramètre (sauf pour `IPC_RMID`) est utilisé :
 - Pour récupérer ou modifier les informations
 - Seules les permissions peuvent être modifiées
- Destruction d'un segment si plus aucun processus attaché

Structure `shmid_ds`

- `shmid_ds` : structure associée aux segments de mémoire
- Mise à jour en fonction des attachements/détachements

```
struct shmid_ds {  
    struct ipc_perm msg_perm; /* permissions */  
    size_t shm_segsz;         /* taille du segment */  
    time_t shm_atime;         /* heure dernier attachement */  
    time_t shm_dtime;         /* heure dernier détachement */  
    time_t shm_ctime;         /* heure dernière modification */  
    pid_t shm_cpid;           /* PID du créateur */  
    pid_t shm_lpid;           /* PID du dernier processus qui a  
                               attaché/détaché */  
    shmatt_t shm_nattch;      /* nombre d'attachements actuels */  
};
```

Fonction shmctl (1/2)

En-tête de la fonction (S2)

- `int shmctl(int shmid, int commande, struct shmid_ds *buf)`
- *Inclusions* : `sys/ipc.h` et `sys/shm.h`

Paramètre(s)

- `shmid` : l'identificateur du segment mémoire
- `commande` :
 - `IPC_RMID` : supprime le segment
 - `IPC_STAT` : récupère les infos (dans le paramètre `buf`)
 - `IPC_SET` : modifie les infos
- `buf` : données à modifier ou récupérer

Fonction `shmctl` (2/2)

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
 - `EACCES` : accès interdit
 - `EFAULT` : `buf` invalide (avec `IPC_STAT` et `IPC_SET`)
 - `EINVAL` : paramètres possèdent une valeur incorrecte

Exemple de suppression d'un segment de mémoire partagée

```
#define CLE 1056

// Récupération du segment de mémoire partagée
if((shmid = shmget(cle, 0, 0)) == -1) {
    perror("Erreur_lors_de_la_récupération_du_segment_");
    exit(EXIT_FAILURE);
}

// Suppression du segment de mémoire partagée
if(shmctl(shmid, IPC_RMID, 0) == -1) {
    perror("Erreur_lors_de_la_suppression_du_segment_");
    exit(EXIT_FAILURE);
}
```

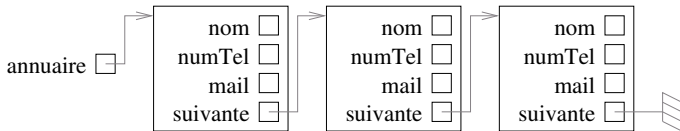
Un peu plus loin avec les segments

- Un segment de mémoire partagée peut contenir plusieurs types de données :
 - ↪ Évite la création (coûteuse) de plusieurs segments
 - ↪ Complique la récupération/modification des données
- Éviter d'utiliser directement des pointeurs :
 - ↪ Solution peu lisible
 - ↪ Pas efficace (opérations nécessaires pour l'accès aux données)
- Une solution :
 - ↪ Créer une structure qui permet de mapper les données
 - ↪ Attention avec les pointeurs !

**Il ne faut absolument pas recopier les données
en mémoire pour chaque processus !**

Exemple d'application

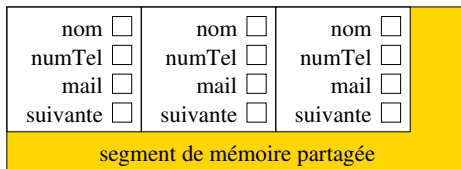
- Un annuaire contient un ensemble d'entrées :
↳ Nom, numéro de téléphone et adresse de courriel
- Implémentation sous la forme d'une liste chaînée :
↳ Permet d'ajouter/insérer, supprimer, etc.



État mémoire de la liste

Comment partager cette liste chaînée entre plusieurs processus via un segment de mémoire partagée ?

Première solution : explications

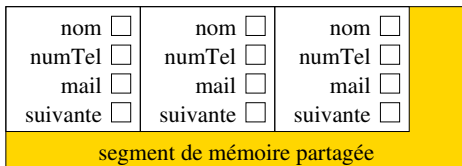


État mémoire de la liste

- Copie des éléments de la liste dans le segment mémoire
- Pas de modification des structures :
 ↪ Pas de `malloc` à faire, pointage sur le segment



Première solution : problèmes

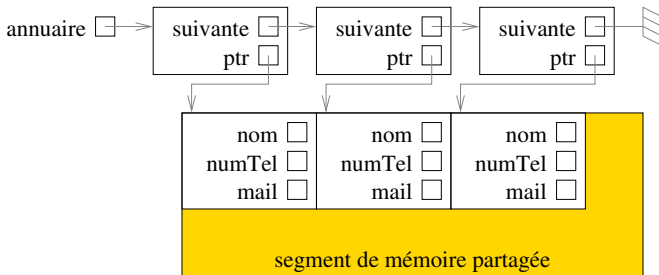


État mémoire de la liste

- Problème avec les pointeurs “suivante”
- Adresses mémoires différentes d'un processus à un autre
↪ Même pour le même segment mémoire
- Solution : suivante n'est plus un pointeur mais une position dans le segment



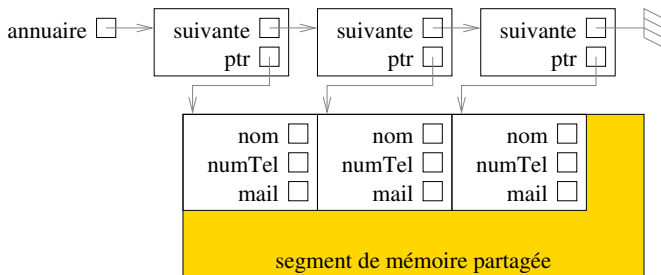
Deuxième solution : explications



État mémoire de la liste

- Création d'une liste en dehors du segment
- Chaque élément de la liste pointe vers le segment
 - ⇨ Le segment ne contient que les données utiles

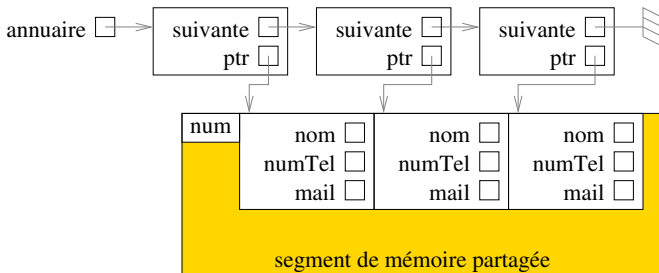
Deuxième solution : problèmes



État mémoire de la liste

- Comment récupérer la liste ?
⇨ Nombre d'éléments inconnus
- Que faire pour "synchroniser" les différentes listes :
⇨ Données partagées, mais pas les éléments de la liste

Deuxième solution "améliorée" : explications

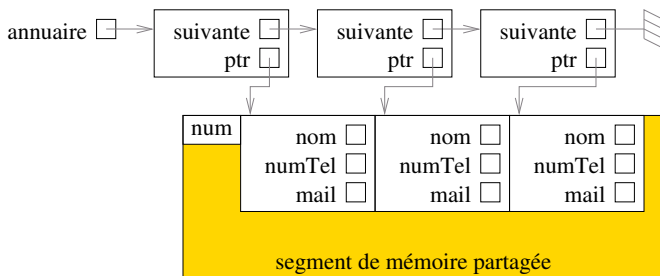


État mémoire de la liste

- Ajout du nombre d'éléments
- Permet "facilement" de créer une liste pour chaque processus



Deuxième solution "améliorée" : problèmes



État mémoire de la liste

- Comment faire pour "synchroniser" les différentes listes :
↪ Données partagées, mais pas les éléments de la liste

Mapping des données d'un segment (1/3)

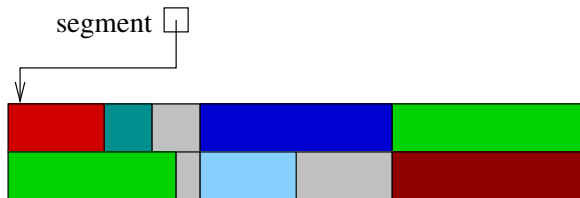
- Données à partager représentées par une structure
- Idée :
 - Utilisation d'un pointeur de structure
 - Pointage direct vers le segment

```
/* Exemple de structure */  
typedef struct {  
    float a;  
    short b;  
    long c;  
    char d[15];  
    int e;  
    double f;  
} segment_t;
```



Mapping des données d'un segment (2/3)

```
...  
int shmid = shmget(CLE, sizeof(segment_t),  
                  S_IRUSR | S_IWUSR | IPC_CREAT);  
segment_t *segment = shmat(shmid, NULL, 0);  
segment->a = 1;  
segment->b = 2;  
segment->c = 3;  
strcpy(segment->d, "Toto");  
...
```

Mapping des données d'un segment (3/3)



segment de mémoire partagée

Légende					
	char		short		float
	int		long		double

Représentation du segment

Attention à l'alignement des structures ! N'utiliser que la structure !

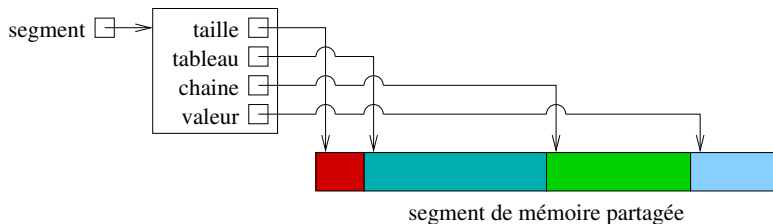
Champs dynamiques (1/5)

- Que se passe-t-il si la structure possède des champs dynamiques ?
- Impossible d'utiliser la technique précédente :
↔ Problème des pointeurs !
- Solution :
 - Instancier une structure
 - Initialiser les champs de la structure vers le segment

```
/* Exemple de structure */  
typedef struct {  
    int *tableau;  
    char *chaine;  
    double valeur;  
} segment_t;
```

Champs dynamiques (2/5)

```
/* Structure utilisée pour le mapping */  
typedef struct {  
    size_t *taille; /* Taille du tableau */  
    int *tableau;  
    char *chaine;  
    double *valeur;  
} segment_t;
```



Représentation du segment

Champs dynamiques (3/5)

```
/* Création du segment, initialisation + mapping */
segment_t *segment_creer(void **adresse, size_t taille,
                        int *tableau, char *chaine,
                        double valeur) {

    int shmid, i;
    size_t tailleSegment;
    segment_t *segment;

    tailleSegment = sizeof(size_t) + taille * sizeof(int) +
                    (strlen(chaine) + 1) * sizeof(char) +
                    sizeof(double);
    shmid = shmget(CLE_SEGMENT, tailleSegment,
                  IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
    *adresse = shmat(shmid, NULL, 0);
    segment = (segment_t*)malloc(sizeof(segment_t));
    ...
}
```

Champs dynamiques (4/5)

```
...
segment->taille = (size_t)*adresse;
*segment->taille = taille;

segment->tableau = (int*)&segment->taille[1];
for(i = 0; i < taille; i++)
    segment->tableau[i] = tableau[i];

segment->chaine = (char*)&segment->tableau[*segment->taille];
strcpy(segment->chaine, chaine);

segment->valeur = (double*)&segment->chaine[strlen(segment->chaine)
    + 1];
*segment->valeur = valeur;

return segment;
}
```

Champs dynamiques (5/5)

```
/* Mapping côté client */
segment_t *segment_mapping(void *adresse) {
    segment_t *segment;

    segment = (segment_t*)malloc(sizeof(segment_t));

    segment->taille = (size_t*)adresse;
    segment->tableau = (int*)&segment->taille[1];
    segment->chaine = (char*)&segment->tableau[*segment->taille];
    segment->valeur = (double*)&segment->chaine[strlen(segment->chaine)
        + 1];

    return segment;
}
```

Présentation des tableaux de sémaphores

- Implémentation des sémaphores :
↔ Opérations P et V
- Manipulation de tableaux de sémaphores
- En un seul appel :
 - Plusieurs opérations
 - Un ou plusieurs sémaphores concernés
 - Toutes les opérations réalisées de manière atomique
- Ajout d'une troisième opération : ATT
↔ Attente que la valeur d'un sémaphore soit à 0

Rappel

P pour *Proberen* et V pour *Verhogen*

Quelques constantes liées aux sémaphores

Affichage via la commande `ipcs`

```
> ipcs -l
```

```
----- Limites des sémaphores -----  
nombre maximal de tableaux = 128  
nombre maximal de sémaphores par tableau = 250  
nombre maximal de sémaphores système = 32000  
nombre maximal d'opérations par appel semop = 32  
valeur maximal de sémaphore = 32767
```

Ces constantes dépendent de la configuration du noyau.

Création et accès

- Utilisation de la fonction `semget` :
 - ↪ Création d'un tableau de sémaphores
 - ↪ Récupération d'un tableau existant
- Même principe que `msgget` :
 - ↪ On précise en plus le nombre de sémaphores

```
/* Structure représentant un sémaphore */
struct semaphore {
    atomic_t count;           /* Valeur actuelle */
    int sleepers;            /* Nombre processus endormis */
    wait_queue_head_t wait;  /* File d'attente des processus
                               actuellement endormis */
}
```


Fonction semget (1/2)

En-tête de la fonction (S2)

- `int semget(key_t cle, int nbSems, int options)`
- *Inclusions* : `sys/types.h`, `sys/ipc.h` et `sys/sem.h`

Paramètre(s)

- `cle` : la clé ou `IPC_PRIVATE`
- `nbSems` : nombre de sémaphores dans le tableau
- `options` :
 - `IPC_CREAT` : crée le tableau de sémaphores
 - `IPC_EXCL` : génère une erreur si le tableau existe déjà
 - Les modes d'accès (voir les fichiers)

Fonction semget (2/2)

Valeurs retournées et erreurs générées

- Retourne l'identificateur interne ou -1 en cas d'échec
- Quelques erreurs possibles :
 - EEXIST : tableau existe déjà (`IPC_CREAT + IPC_EXCL`)
 - EINVAL :
 - ↪ `nbSems > SEMMSL` ou `nbSems < 0`
 - ↪ Le tableau existe et `nbSems >` à la taille actuelle
 - ENOSPC :
 - ↪ nombre maximum de tableaux atteint (`=SEMMNI`)
 - ↪ nombre maximum de sémaphores atteint (`=SEMMNS`)

Exemple de création

```
#define CLE 450

int main() {
    int semid;

    if((semid = semget(CLE, 5,
                      S_IRUSR | S_IWUSR |
                      IPC_CREAT | IPC_EXCL)) == -1) {
        if(errno == EEXIST)
            fprintf(stderr, "Tableau_(cle=%d)_existant\n", CLE);
        else
            perror("Erreur_lors_de_la_creation_");
        exit(EXIT_FAILURE);
    }

    /* Suite du code */

    return EXIT_SUCCESS;
}
```

Visualisation depuis le shell (1/2)

Vérification de la création

```
> ipcs -s
```

```
----- Tableaux de sémaphores -----
```

clé	semid	propriétaire	perms	nsems
0x000001c2	786433	cyril	600	5

Visualisation depuis le shell (2/2)

Affichage des informations

```
> ipcs -s -i 786433
```

```
Tableaux de sémaphores semid=786433
```

```
uid=1000          gid=1000          cuid=1000          cgid=1000
```

```
mode=0600, access_perms=0600
```

```
nsems = 5
```

```
otime = Non initialisé
```

```
ctime = Sat Jan 30 14:27:13 2021
```

semnum	valeur	ncount	zcount	PID
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

Opérations sur les sémaphores (1/2)

- Opérations spécifiées via la structure `sembuf`
- On indique le numéro du sémaphore et l'opération :
 - Valeur positive : opération V
 - Valeur négative : opération P
 - Valeur nulle : opération ATT
- Plusieurs options :
 - `IPC_NOWAIT` : opération non bloquante (mais erreur si pas possible)
 - `SEM_UNDO` : mémorise l'opération pour un retour arrière

```
/* Structure représentant une opération */
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
}
```

Opérations sur les sémaphores (2/2)

- Structure(s) `sembuf` passée à la méthode `semop` :
 - ↪ Possible de passer un tableau d'opérations
 - ↪ Bloquant tant qu'une seule n'est pas possible
- Opérations réalisées de manière atomique :
 - ↪ Dans l'ordre du tableau

`semop` réalise des opérations mais ne permet pas d'initialiser les sémaphores !

Fonction semop (1/2)

En-tête de la fonction (S2)

- `int semop(int semid, struct sembuf *sops, unsigned nsops)`
- *Inclusions* : `sys/types.h`, `sys/ipc.h` et `sys/sem.h`

Paramètre(s)

- `semid` : identifiant du tableau de sémaphores
- `sops` : tableau d'opérations (ou une seule)
- `nsops` : nombre d'opérations dans le tableau

Fonction semop (2/2)

Valeurs retournées et erreurs générées

- Retourne 0 en cas de réussite ou -1 en cas d'échec
- Quelques erreurs possibles :
 - EAGAIN : une opération n'a pu être réalisée avec option `IPC_NOWAIT`
 - EINVAL : tableau inexistant, `smid` ou `nsops` incorrects
 - EINTR : signal reçu pendant l'attente

Exemple d'opérations

```
struct sembuf op;

/* Réalisation de V(S) */
op.sem_num = 0;
op.sem_op = 1;
op.sem_flg = 0;
if(semop(semid, &op, 1) == -1) {
    perror("Erreur_lors_de_l'opération_sur_le_sémaphore_");
    exit(EXIT_FAILURE);
}

/* Réalisation de P(S) */
op.sem_num = 0;
op.sem_op = -1;
op.sem_flg = 0;
if(semop(semid, &op, 1) == -1) {
    perror("Erreur_lors_de_l'opération_sur_le_sémaphore_");
    exit(EXIT_FAILURE);
}
```

Exemple d'opérations multiples

```
struct sembuf op[2];

/* Réalisation de V(S0) et P(S1) */
op[0].sem_num = 0;
op[0].sem_op = 1;
op[0].sem_flg = 0;

op[1].sem_num = 1;
op[1].sem_op = -1;
op[1].sem_flg = 0;

if(semop(semid, op, 2) == -1) {
    perror("Erreur_lors_des_opérations_sur_le_sémaphore_");
    exit(EXIT_FAILURE);
}
```

semctl : la fonction couteau suisse

- Fonction `semctl` permet :
 - De modifier/récupérer les informations d'un tableau de sémaphores
 - De supprimer un tableau de sémaphores
 - De récupérer/modifier les valeurs d'un ou de tous les sémaphores
- Dépend du paramètre `commande` :
 - `IPC_RMID` : suppression immédiate du tableau
 - `IPC_STAT` : récupère les infos
 - `IPC_SET` : modifie les infos (les permissions)
 - `GETVAL` et `SETVAL` : récupère et modifie la valeur d'un sémaphore
 - `GETALL` et `SETALL` : idem, mais de tous les sémaphores

Fonction `semctl` : format général

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande, union semun args)`
- *Inclusions* : `sys/types.h`, `sys/ipc.h` et `sys/sem.h`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : numéro du sémaphore ou paramètre ignoré
- `commande` : la commande
- `args` : les arguments correspondant à la commande

Fonction semctl : union senum

```
union senum {  
    int val;  
    struct semid_ds *buf;  
    unsigned short *array;  
};
```

L'union n'est pas définie en général :
on utilise l'un des trois champs directement.

Fonction `semctl` : suppression du tableau (1/2)

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande)`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : ignoré
- `commande` : `IPC_RMID`

Fonction `semctl` : suppression du tableau (2/2)

Valeurs retournées et erreurs générées

- Retourne 0 ou -1 en cas d'échec
- Quelques erreurs possibles :
 - `EACCES` : accès interdit
 - `EIDRM` : le tableau a déjà été supprimé
 - `EINVAL` : des paramètres possèdent une valeur incorrecte

Exemple de création/suppression

```
int semid;

/* Création d'un tableau de 10 sémaphores */
if((semid = semget(CLE,
                  10,
                  S_IRUSR | S_IWUSR |
                  IPC_CREAT | IPC_EXCL)) == -1) {
    perror("Erreur_lors_de_la_création_");
    exit(EXIT_FAILURE);
}

...

/* Suppression du tableau */
if(semctl(semid, 0, IPC_RMID) == -1) {
    perror("Erreur_lors_de_la_suppression_du_tableau_");
    exit(EXIT_FAILURE);
}
```

Fonction `semctl` : récupération/modification infos (1/2)

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande, struct semid_ds *args)`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : ignoré
- `commande` :
 - `IPC_STAT` ou `IPC_SET` : récupère ou modifie les informations
- `args` : données à modifier ou récupérer

Fonction `semctl` : récupération/modification infos (2/2)

```
struct semid_ds {  
    struct ipc_perm sem_perm; /* permissions */  
    time_t sem_otime;         /* heure dernière opération semop */  
    time_t sem_ctime;         /* heure dernière modification */  
    unsigned short sem_nsems; /* nombre de sémaphores dans le  
                               tableau */  
};
```

Suivant les systèmes, `sem_nsems` peut être un `unsigned long`.

Exemple de récupération d'informations

```
int semid;
struct semid_ds sem_buf;

if((semid = semget(CLE, 0, 0)) == -1) {
    perror("Erreur_lors_de_la_recupération_du_tableau_");
    exit(EXIT_FAILURE);
}

if(semctl(semid, 0, IPC_STAT, &sem_buf) == -1) {
    perror("Erreur_lors_de_la_recupération_d'infos_");
    exit(EXIT_FAILURE);
}

if(sem_buf.sem_otime == 0)
    printf("Date_der._op._:encore_aucune_opération\n");
else
    printf("Date_der._op._:%s", ctime(&sem_buf.sem_otime));
printf("Date_der._modif._:%s", ctime(&sem_buf.sem_ctime));
printf("Nb._sémaphores_: %ld\n", sem_buf.sem_nsems);
```

Fonction `semctl` : modification d'une valeur

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande, int valeur)`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : numéro du sémaphore dans le tableau
- `commande` : SETVAL
↪ Peut débloquent des processus !
- `valeur` : nouvelle valeur

Valeurs retournées

- 0 en cas de réussite ou -1 en cas d'échec

Exemple de modification d'une valeur

```
int semid, num = 0, valeur;

/* Récupération du tableau de sémaphores */
if((semid = semget(CLE, 0, 0)) == -1) {
    perror("Erreur_lors_de_la_récupération_du_tableau_de_sémaphores_")
    ;
    exit(EXIT_FAILURE);
}

/* Récupération de la valeur du sémaphore */
if((semctl(semid, num, SETVAL, valeur) == -1) {
    perror("Erreur_lors_de_la_modification_de_la_valeur_du_sémaphore_")
    );
    exit(EXIT_FAILURE);
}
```

Fonction `semctl` : récupération d'une valeur

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande)`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : numéro du sémaphore dans le tableau
- `commande` : GETVAL

Valeurs retournées

- Valeur du sémaphore ou -1 en cas d'échec

Exemple de récupération d'une valeur

```
int semid, num = 0, valeur;

/* Récupération du tableau de sémaphores */
if((semid = semget(CLE, 0, 0)) == -1) {
    perror("Erreur_lors_de_la_récupération_du_tableau_de_sémaphores_")
    ;
    exit(EXIT_FAILURE);
}

/* Récupération de la valeur du sémaphore */
if((valeur = semctl(semid, num, GETVAL)) == -1) {
    perror("Erreur_lors_de_la_récupération_de_la_valeur_du_sémaphore_")
    );
    exit(EXIT_FAILURE);
}
```


Fonction `semctl` : modification de toutes les valeurs

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande, unsigned short *tableau)`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : ignoré
- `commande` : SETALL
- `tableau` : tableau avec les nouvelles valeurs

La modification peut entraîner le déblocage de processus !

Exemple de modification des valeurs

```
int semid;
unsigned short *tableau = { 0, 0, 0, 0, 0 };

/* Recuperation du tableau de semaphores */
if((semid = semget(CLE, 0, 0)) == -1) {
    perror("Erreur_lors_de_la_recuperation_du_tableau_de_semaphores_")
    ;
    exit(EXIT_FAILURE);
}

/* Modification des valeurs */
if(semctl(semid, 0, SETALL, tableau) == -1) {
    perror("Erreur_lors_de_la_modification_des_valeurs_");
    exit(EXIT_FAILURE);
}
```

Fonction `semctl` : récupération de toutes les valeurs

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande, unsigned short *tableau)`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : ignoré
- `commande` : `GETALL`
- `tableau` : tableau contenant les valeurs récupérées

Informations sur les sémaphores

- Chaque sémaphore d'un tableau est associé aux valeurs :
 - `unsigned short semval` : valeur du sémaphore
 - `unsigned short semzcnt` : nombre de processus bloqués sur ATT
 - `unsigned short semncnt` : nombre de processus bloqués sur P
 - `pid_ sempid` : dernier processus agissant
- La valeur est récupérée par `GETVAL`, modifiée par `SETVAL`
- Les autres champs :
 - ↪ Mis à jour automatiquement par le système
 - ↪ Récupération avec des commandes spécifiques

Fonction `semctl` : récupération informations générales

En-tête de la fonction (S2)

- `int semctl(int semid, int semnum, int commande)`

Paramètre(s)

- `semid` : identificateur du tableau de sémaphores
- `semnum` : numéro du sémaphore
- `commande` :
 - `GETNCNT` : récupère `semzcnt` (nombre de proc. bloqués sur ATT)
 - `GETZCNT` : récupère `semncnt` (nombre de proc. bloqués sur P)
 - `GETPID` : récupère `sempid` (PID du processus agissant)

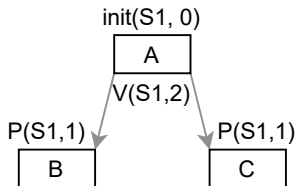
Valeurs retournées

- Nombre de processus, le PID ou -1 en cas d'échec

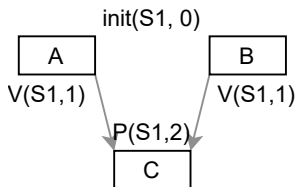
Retour sur les opérations

- Opérations plus générales que les sémaphores de *Dijkstra*
- Les **opérations généralisées** :
 - $P(S, X)$: test de la valeur du sémaphore S
 - Si la valeur est $\leq X$, le processus est bloqué
 - Sinon, on soustrait X à la valeur du sémaphore
 - $V(S, X)$: ajout de X à la valeur du sémaphore
 - $ATT(S)$: processus bloqué tant que la valeur de $S \neq 0$
- Initialisation avec `init(S, valeur)`
- Liste d'opérations réalisées de manière atomique :
 \hookrightarrow Exemple : $[P(S_1, 2), V(S_2, 1), P(S_3, 2)]$

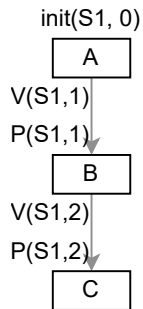
Exemples d'utilisation



Exemple 1



Exemple 2



Exemple 3

Autre exemple (1/2)

- Supposons une zone de mémoire partagée
- Opération de lecture : plusieurs processus autorisés
- Opération d'écriture : un seul processus autorisé et pas de lecteur
- Comment résoudre avec des sémaphores ?



Autre exemple (2/2)

- Supposons une zone de mémoire partagée
- Opération de lecture : plusieurs processus autorisés
- Opération d'écriture : un seul processus autorisé et pas de lecteur

Solution : trois sémaphores

- Un sémaphore S_1 utilisé pour compter le nombre de lecteurs actuels :
 $\hookrightarrow \text{init}(S_1, 0)$
- Un sémaphore S_2 pour compter le nombre d'écrivains en attente :
 $\hookrightarrow \text{init}(S_2, 0)$
- Un sémaphore S_3 pour bloquer l'accès en écriture :
 $\hookrightarrow \text{init}(S_3, 1)$