IPC system V

- Inter Process Communication : 3 mécanismes de communication entre processus d'un même système
  - les files de messages
  - les segments de mémoire partagée
  - les sémaphores
- objets externes au système de gestion de fichiers
  $\Rightarrow$ accessibles via des clés jouant le même rôle que les références pour les fichiers

généralités : le fichier /usr/include/sys/ipc.h(1)

- *structure ipc_perm* : permet contrôle de l'accès à un objet

```
struct ipc_perm  {
    ushort uid;         /* propriétaire */
    ushort gid;         /* groupe propriétaire */
    ushort cuid;        /* créateur */
    ushort cgid;        /* groupe créateur */
    ushort mode ;       /* droits d'accès */
    ushort seq;         /* nombre d'utilisation de l'entrée */
    key_t key;          /* clé */
}
```

- seuls les champs uid, gid, mode sont modifiables par un processus dont le propriétaire est uid ou cuid

généralités (2)

- identification des objets
  - un objet ↔une identification
  - à chaque mécanisme ↔ une table des objets du type correspondant (joue le même rôle que la table des i-noeuds pour les fichiers)
  - accès à un objet existant par l'intermédiaire de son identification interne ou d'une clé (permet par l'intermédiaire de primitives spécifiques de retrouver son identification)
  - type d'une clé : key_t (</sys/types.h>)

généralités (3)

- clé particulière : IPC_PRIVATE
- chaque mécanisme dispose de son propre ensemble de clés $\Rightarrow$ un segment de mémoire partagée et un sémaphore peuvent avoir la même clé

les primitives de la famille get (1)

- accès à un objet existant ou création de nouveaux objets
  - retour : identification de l'objet ou -1
  - au moins 2 paramètres : une clé et une option
  - clé = IPC_PRIVATE ⇒ création d'un nouvel objet
  - clé ≠ IPC_PRIVATE ⇒ recherche ou création :
    - ✓ 1er cas : IPC_CREAT non positionné
      - objet ∃ ⇒ retour de son identification
      - objet !∃ ⇒ erreur

les primitives de la famille get (2)

- ✓ 2ème cas : IPC_CREAT positionné et IPC_EXCL non positionné
  - objet ∃ ⇒ retour de son identification
  - objet !∃ ⇒ création et retour de l'identification de l'objet créé
- ✓ 3ème cas : IPC_CREAT positionné et IPC_EXCL positionné
  - objet ∃ ⇒ erreur
  - objet !∃ ⇒ création et retour de l'identification de l'objet créé
- ✓ autres cas : erreur

◆ Remarque : IPC_CREAT : bit particulier du mode d'un objet (mode : idem fichier mais seuls les bits r et w sont significatifs)

les primitives de la famille ctl ()

- au moins 3 paramètres
  - ✓ une identification d'objet (retour de get)
  - ✓ une option précisant la nature de l'opération
  - ✓ un argument pour l'opération réalisée
- opérations communes aux 3 mécanismes :
  - ✓ IPC_STAT : récupération des caractéristiques de l'objet
  - ✓ IPC_SET : modification de l'entrée associée dans la table avec le contenu de la zone mémoire dont l'adresse est donnée en paramètre
  - ✓ IPC_RMID : suppression de l'objet ($\Rightarrow$ le propriétaire du processus doit être le créateur, le propriétaire de l'objet ou le super-utilisateur)

les commandes shell

- la commande ipcs
  - consultation des tables
  - fournit :
    - T = (q, m ou s) = type de l'objet
    - ID : identification
    - KEY : clé
    - MODE : droits d'accès
    - OWNER
    - GROUP
- la commande ipcrm : permet de supprimer un objet en le désignant soit par sa clé, soit par son identification interne

les files de messages (1)

- communication indirecte entre processus
  - ✓ messages envoyés a une FdM et non à un processus
  - ✓ les messages sont typés
  - ✓ FDM gérées en FIFO
  - ✓ tout processus qui connaît une FDM peut l'utiliser
- fichier entête : /usr/include/sys/msg.h
  - structure générique d'un message

```
struct msgbuf {   mtyp_t m_type ; /* entier positif */
                  char mtext[1];
}
```

⇒ non utilisable telle quelle : l'utilisateur doit se définir sa propre structure (de nom ≠)

les files de messages (2)

- fonction msgget : int msgget (key_t clé, int option)

  ⇒ obtention d'une identification avec création evt

- fonction msgsnd : int msgsnd (int msgid, struct msgbuf *pmess, int lg, int opt)
  - ✓ msgid : identification de la FDM
  - ✓ pmess : pointeur sur le message à envoyer

    (remplacer le type struct msgbuf par le type de message défini)
  - ✓ lg : longueur du message (sans prendre en compte le type)
  - ✓ opt : option d'émission :

    IPC_NOWAIT ⇒ appel non bloquant (bloquant par défaut)

    ⇒ retour = -1 si file pleine (message non envoyé)
  - ✓ retour : -1 si échec

les files de messages (3)

- fonction msgrcv : int msgrcv(int msgid, struct msgbuf *pmess, int lgmax, long type, int option)
  - ✓ extraction de la FDM *msgid* d'un message de type *type* et rangement du message à l'adresse *pmess*
  - ✓ appel bloquant par défaut
  - ✓ appel non bloquant si le bit IPC_NOWAIT est positionné dans *option*
  - ✓ taille msg > lgmax ⇒ échec par défaut
  - ✓ taille msg > lgmax et bit MSG_NOERROR positionné dans option ⇒ message tronqué
  - ✓ retour : taille du message extrait (éventuellement taille du message tronqué)

les files de messages (4)

- remarque : en cas de troncature : aucune possibilité de connaître le nombre de caractères perdus

  - ✓ type 0 ⇒ 1er message (∀ son type)
  - ✓ type >0 ⇒ 1er message du type donné
  - ✓ type <0 ⇒ 1er message de type ≤ |type|
  - ✓ type du message extrait rangé dans le champs mtype de *pmess

- ☞ fonction msgctl : int msgctl(int msgid, int cmd, struct msqid_ds *pbuf)

  - ✓ cmd : IPC_RMID (pbuf = NULL)
  - ✓ IPC_STAT
  - ✓ IPC_SET

Files de messages : exemple (1)

- échange de messges par deux processus en utilisant UNE boite aux lettres :
  - le père envoie un message de type 1 au fils et attend un ack (message de type 2). Une fois l'ack reçu, il affiche son contenu, attend la mort du fils et détruit la boite aux lettres
  - Le fils attend le message de type 1 en provenance du père, affiche son contenu et renvoie un ack au père (message de type 2)

Files de messages : exemple (2)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define NB              10
#define CLE_MSG         (key_t)1000
#define TAILLE_MSG      20
```

Files de messages : exemple (2)

```
/* définition de la structure des messages */
typedef struct  {
    long type;
    char texte[TAILLE_MSG];
} Msg_requete;
main() {
    int msgid, n ;
    Msg_requete message;
    /* création de la FDM */
    if ((msgid = msgget(CLE_MSG, IPC_CREAT|IPC_EXCL|0666)) == -1) {
        perror ("msgget");
        exit(1);
    }
```

Files de messages : exemple (3)

```
/* création du processus fils */
if ((n=fork()) == -1) {
    perror ("fork");
    exit(2);
}
if (n==0){
    /* on est dans le fils */
    /* reception du message du père : message de type 1 */
    msgrcv(msgid, &message, TAILLE_MSG, 1, 0);
    /* affichage contenu message */
    printf("message reçu par le fils = %s\n", message.texte);
    /* preparation de l'ack : message de type 2 */
    message.type = 2;
    strcpy(message.texte, "msg reçu\n");
```

Files de messages : exemple (4)

```
    /* envoi de l'ack au père */
    msgsnd(msgid, &message, TAILLE_MSG, 0);
}
else {
    /* on est dans le père */
    /* preparation du message */
    message.type = 1 ;
    strcpy(message.texte, "hello\n");
    /* envoi du message au fils */
    msgsnd(msgid, &message, TAILLE_MSG, 0);
    /* attente de reception de l'ack : message de type 2 */
    msgrcv(msgid, &message, TAILLE_MSG, 2, 0);
    /* affichage ack reçu */
    printf("ack reçu par père : %s\n", message.texte);
```

Files de messages : exemple (5)

```
        /* attente de la mort du fils */
        wait(NULL);
        /* destruction de la fdm */
        msgctl(msgid, IPC_RMID, 0);
    }
}
```

Les segments de mémoire partagée (1)

- but : permettre à des processus quelconques de partager des zones de données physiques éventuellement structurées.
  - remarque : quand un processus fait un fork, le fils hérite d'une copie des données du père ($\Rightarrow$ une modification sur une donnée dans le processus fils est sans effet dans le processus père)
  - avantage mémoire partagée : aucune recopie d'information mais ...
  - ATTENTION : ressource critique !!!

Les segments de mémoire partagée (2)

- le fichier entête : /usr/include/sys/shm.h
- fonction shmget : int shmget (key_t clé, int taille, int option)
  - obtention de l'identification d'un segment après l'avoir éventuellement créé
  - la taille doit être compatible avec
    - ✓ SHMMIN et SHMMAX (cstes définies dans le fichier entête)
    - ✓ taille du segment s'il existe

Les segments de mémoire partagée (3)

- opérations sur segments
  - ✓ remarque : le code d'un programme exécutable contient des adresses virtuelles (pour permettre le chargement du programme à un endroit quelconque de la mémoire). Au cours de l'exécution, le module de gestion mémoire est chargé de traduire les adresses virtuelles en adresses physiques
  - ✓ $\Rightarrow$ un processus souhaitant utiliser un SMP doit lui attribuer une adresse virtuelle dans son espace d'adressage ; il pourra alors accéder à cet espace de la même manière qu'il accède à une zone de données allouée dynamiquement par malloc.

Les segments de mémoire partagée (4)

- ✓ ⇒ opération d'attachement du segment au processus
- ✓ opération inverse : détachement ; réalisée quand un processus n'utilise plus le segment

☞ fonction shmat : char * shmat(int shmid, char * adr, int option)

- ◆ retour : adresse à laquelle l'attachement a été réalisé effectivement (ie : @ par laquelle le 1er octet du segment sera disponible)
- ◆ pb : choix adresse
  - ✓ adr = NULL : le système décide l'@ d'attachement (recommandé pour assurer la portabilité) ⇒ paramètre option non significatif (0)

Les segments de mémoire partagée (5)

- ✓ adr ≠ NULL et bit SHM_RND positionné ⇒ adr est "arrondie" de façon à rendre cette @ correcte pour un segment
- ✓ adr ≠ NULL et bit SHM_RND non positionné ⇒ le système tente l'attachement à l'@ fournie

- ◆ autres options : positionnement du bit SHM_RDONLY ⇒ attachement du segment en lecture uniquement

☞ fonction shmdt : int shmdt (char *adr)

- ◆ détachement du segment préalablement attaché à l'@ adr par shmat (⇒ adr devient une @ illégale pour ce processus)

Les segments de mémoire partagée (6)

- fonction shmctl : int shmctl(int shmid, int op, struct shmid_ds *p_buf)
  - effectuer l'opération op sur le segment mémoire
    - ✓ IPC_STAT
    - ✓ IPC_SET
    - ✓ IPC_RMID : empêche un nouvel attachement du segment par un processus ∀ (suppression effective lors du dernier détachement)

mémoire partagée : exemple 1 (1)

- programme initialisant un tableau de NB nombres en mémoire partagée

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define NB 64
#define CLE (key_t)1000
main() {
    int shmid, * adr,  i;
    if  ((shmid=shmget  (CLE,  NB*sizeof(int),  IPC_CREAT|IPC_EXCL|0666))
          ==-1){
        perror("shmget"); exit(2);
    }
```

mémoire partagée : exemple 1 (2)

```
    /* attachement à une adresse non fixée */
    if ((adr = (int *) shmat(shmid, 0, 0)) == (int *)(-1)) {
        perror("shmat");
        exit(2);
    }

    /* init du générateur de nombre aléatoires */
    srand(getpid());

    /* init du tableau */
    for (i = 0; i < NB; i++)
    printf("%d ", adr[i] = rand() % 100);
    putchar('\n');
}
```

mémoire partagée : exemple 2 (1)

- programme relisant un tableau de NB nombres en mémoire partagée

```
#include ...

...

#define NB 64

#define CLE (key_t) 1000

main() {

   int shmid, *adr, i;

   if (( shmid = shmget (CLE, NB * sizeof(int), 0666)) == -1){

      perror ("shmget");    exit(2);

   }
```

mémoire partagée : exemple 2 (2)

```
    if (( adr = shmat(shmid, 0, SHM_RDONLY) == (int *)(-1)){
        perror("shmat"); exit(2);
    }
    for (i = 0; i < NB; i++)
    printf("%d ", adr[i]);
    putchar("\n");
}
```

Les sémaphores (1)

- exemples de problèmes non résolus par les sémaphores d'exclusion mutuelle et les primitives P et V classiques :
  - ex1 : un processus demande l'accès à m ressources
    - ✓ P(S1); P(S2); ... ; P(Sm); (atomicité non garantie)

      ⇒ un processus peut être bloqué sur P(Si) après s'être attribué S1, ..., Si-1

      ⇒ si le processus qui détient Si attend S1 pour libérer Si : deadlock

Les sémaphores (2)

- ex2 : un processus accède à m ressources identiques protégées par un sémaphore dont la valeur = nombre de ressources disponibles

⇒ même problème que précédemment : réalisé par P(S); P(S); ...; P(S);

✓ solution : définir les opérations Pn et Vn (n ≥ 0) telles que

- Pn(S) : si S < n alors
  attendre
  sinon
  S = S - n
  Fsi
- Vn(S) : S = S + n ; réveil d'un ou plusieurs processus en attente

Les sémaphores (3)

- sémaphores sous Unix : on manipule des ensembles de sémaphores (et non des sémaphores individuels) sur lesquels on peut réaliser un ensemble d'opérations de façon atomique
- le fichier entête : /usr/include/sys/sem.h
  - structure sembuf : décrit une opération élémentaire d'un ensemble

```
struct sembuf {
        ushort sem_num;    /* n° de sémaphore concerné */
        short sem_op;      /* opération sur sem */
        short sem_flg;     /* options de l'opération */
    }
```

Les sémaphores (4)

- structure sem : associée à chaque sémaphore individuel

```
struct sem {
        ushort semval;
        ushort sempid ; /* pid du dernier proc ayant appelé semop */
        ushort semcnt ; /* nombre de proc attendant semval > 0 */
        ushort semzcnt;/* nombre de proc attendant semval = 0 */
}
```

Les sémaphores (5)

- fonction semget : int semget(key_t cle, int nbsems, int options)
  - obtention de l'identification d'un ensemble de nbsems sémaphores avec création éventuelle
  - les différents sémaphores de l'ensemble ont pour n° : 0, 1, ..., nbsems - 1

Les sémaphores (6)

- fonction semop : int semop (int semid, struct sembuf *p_op, unsigned int nbop)
  - permet de réaliser automatiquement les nbop opérations situées à l'@ p_op sur l'ensemble de sémaphores semid
  - primitive bloquante par défaut : si une des opérations ne peut être réalisée, processus mis en attente ($\Rightarrow$ aucune opération réalisée)

Les sémaphores (7)

- une opération = un entier court
  - ✓ valeur > 0 ⇒ opération de type V
    - incrémentation du sémaphore concerné avec la valeur ⇒ réveil de tous les processus attendant que cette valeur augmente
  - ✓ valeur = 0 ⇒ permet de tester que la valeur du sémaphore est nulle. le processus sera bloqué si la valeur n'est pas nulle
  - ✓ valeur < 0 ⇒ opération P

```
si semval > |valeur| alors
    semval = semval - |valeur|
sinon
    attendre
 fsi
```

Les sémaphores (8)

- options des opérations
  - ✓ bit IPC_NOWAIT : permet de rendre une opération non bloquante : ie : lors de semop si une opération ne peut être réalisée, aucune opération n'est réalisée mais le processus n'est pas bloqué (retour de semop = -1)
  - ✓ bit SEM_UNDO : permet de supprimer les blocages d'un processus par la terminaison incorrecte d'un autre : si le processus fait un exit, le noyau inverse l'effet de chaque opération de sémaphore que le processus a réalisé
    - ex : un processus verrouille une ressource. Il reçoit un kill ⇒ les autres processus ne peuvent plus utiliser la ressource. Si SEM_UNDO est positionné, le noyau déverrouille la ressource avant de réaliser l'exit.

Les sémaphores (9)

- fonction semctl : int semctl(int semid, int semnum, int op, union semun args)

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