Architecture des Ordinateurs et Systèmes d'Exploitation

Cours n°11

Les Processus :
Processus : appels systèmes
Communication inter processus

INSA ROUEN
Ph. Leray

Architecture des Systèmes d'Information
3ème année

Processus : appels systèmes 1/4

- **Appels (primitives) système permettant de manipuler des processus**
- **Comment les utiliser en C sous Unix ?**
  - **Le détail des fonctions systèmes est aussi donné par le *man* !**
  - ***man primitive_système* vous indiquera les librairies à inclure dans votre programme C *(#include <librairie.h>)***
- **Exemple :**

```
GETPID(2)        Linux Programmer's Manual        GETPID(2)

NAME
       getpid, getppid - get process identification

SYNOPSIS
       #include <unistd.h>
       pid_t getpid(void);
       pid_t getppid(void);

DESCRIPTION
       getpid  returns  the  process  ID  of the current process.
       (This is often used by routines that generate unique  temporary file names.)
       getppid  returns  the process ID of the parent of the current process.
```

Processus : appels systèmes 2/4

- ***int fork()***
  - **L'appel système *fork* duplique le processus.**
  - **Après l 'appel, l'exécution continue dans les 2 processus (fils et père)**
  - **Pour savoir dans quel processus (père/fils) on se retrouve, il faut regarder la valeur retournée par la fonction :**
    - » ***PID (fils)* pour le processus père**
    - » ***0* pour le processus fils**
    - » ***-1* en cas d'échec (manque de mémoire, trop de processus, …)**
- ***void exit(int status)***
  - **L'appel système *exit* termine le processus qui l'appelle**
  - **La valeur *status* est retournée au processus père pour indiquer une éventuelle erreur (0 sinon)**
- ***int sleep(int seconds)***
  - **L'appel système *sleep* bloque un processus pendant n secondes sauf si le processus reçoit un signal**
  - **(ne pas utiliser pour de la synchronisation de processus)**

Processus : appels systèmes 3/4

- ***int execlp(char *comm, char *arg, … , NULL)***
  - **L'appel système *execlp* recouvre un processus par un autre exécutable.**
  - **Après l 'appel, la fonction retourne :**
    - » ***-1* en cas d 'erreur**
    - » **rien sinon : le programme appelant n'existe plus, puisqu'il a été remplacé par la commande passée en paramètres !**
  - **Ex : execlp("ls", "ls", "-l", "/usr", NULL)**
    - » **il faut bien indiquer 2 fois *ls* (commande et argument[0])**

- ***int wait(0)***
  - **L'appel système *wait* bloque un processus jusqu'à la fin d'un de ses fils**
  - **Après l 'appel, la fonction retourne :**
    - » ***-1* en cas d 'erreur ou si le processus n 'a pas de fils**
    - » **le *PID* du fils qui s'est arrêté**

Processus : appels systèmes 4/4

- **Ex de prog. C**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main(void)  {
        int pid, i ;
        pid = fork() ;
        switch (pid) {
        case -1:
                printf(«  erreur : echec du fork()\n  »);
                exit(1);        break ;
        case 0:
                printf(« -------- Processus fils : pid =%d\n  », getpid() );
                sleep(100);
                printf(«  -------- fin du fils \n  »);
                exit(0);        break ;
        default:
                printf(«  Processus père : le fils à un pid=%d\n  », pid) ;
                wait(0);
                printf(«  Fin du père\n »)
        }
}
```

Création d 'un processus fils avec même segment de code que le père !

pid=0, ⇒ processus fils … c'est donc là qu'il faut mettre la partie de programme du fils

⇒ processus père …

c'est donc là qu'il faut mettre la partie de programme du père

Communication inter-processus

- **Signaux asynchrones**

- **Tubes (liés à la gestion de fichier)**

- **Files de messages**
- **Sémaphores**
- **Segments de mémoire partagée**

Signaux asynchrones

- **Signaux asynchrones**
  - **idée = avertir un processus d'un événement extérieur**
  - **généralement, un SE possède une liste fixée de signaux**
  - **lors de la réception d'un signal, le processus peut :**
    - » **l'ignorer**
    - » **mourir / stopper**
    - » **se réveiller**
    - » **exécuter une fonction donnée correspondant au signal**

- **Manipulation de signaux en C :**
  - ***int kill(int pid, int signum)***
    - » **envoyer le signal *signum* au process *pid***
  - ***void (*signal(int signum, void (*handler)(int)))(int)***
    - » **lier la réception du signal *signum* avec l'exécution de la fonction *handler***

Les signaux Linux

```
[pleray@servasi Processus]$ kill -l
```

```
 1) SIGHUP       : fin de session
 2) SIGINT
 3) SIGQUIT
 4) SIGILL
 5) SIGTRAP
 6) SIGIOT
 7) SIGBUS
 8) SIGFPE       : erreur de calcul
 9) SIGKILL      : le «signal tueur»
10) SIGUSR1      : utilisateur
11) SIGSEGV
12) SIGUSR2      : utilisateur
13) SIGPIPE
14) SIGALRM
15) SIGTERM      : terminaison propre
```

```
17) SIGCHLD      : fin du fils
18) SIGCONT      : reprise
19) SIGSTOP      : arrêt
20) SIGTSTP
21) SIGTTIN
22) SIGTTOU
23) SIGURG
24) SIGXCPU
25) SIGXFSZ
26) SIGVTALRM
27) SIGPROF
28) SIGWINCH
29) SIGIO
30) SIGPWR
```

Tubes

- Idée = échange de données entre 2 processus
- Tube (Unix) = fichier spécial avec :
  - 2 extrémités (lire/écrire)
  - lecture destructrice (l 'information lue est «enlevée» du tube)
  - gestion *fifo* : l 'information n°1 en écriture = n°1 en lecture
  - capacité finie
- Autres attributs d 'un tube :
  - nb de lecteurs = nb de descripteurs associés à l'entrée «lecture» du tube
  - nb d'écrivains = nb de descripteurs associés à l'entrée «écriture» du tube
- 2 types de tubes :
  - tubes ordinaires (adapté à la communication entre processus fils) ***
  - tubes nommés (communication entre processus sans parenté)

Tubes ordinaires

- Idée : tube = 2 fichiers sans nom utilisés pour communiquer

- Manipulation de tubes ordinaires en C :
  - *int pipe(int p[2])*
    - » création d'un tube
    - » retour : *p[0]*=fichier lecture *p[1]*=fichier écriture
  - *int read(p[0], buffer, n)*
    - » lecture d'au moins *n* caractères dans le tube
    - » opération bloquante si le tube est vide (permet de synchroniser)
  - *int write(p[1], buffer, n)*
    - » écriture des *n* caractères du *buffer* dans le tube

Files de messages

- **Idée = boîte à lettres « numérotée », avec une clé**

- **Créer ou trouver une file existante :**
  - ***int msgget(key_t cle, int option)***
    - » **retourne un numero de file (*msgid*) en fonction d'une clé**

- **Envoyer un message dans une file :**
  - ***int msgsnd(int msgid, const void *p_msg, int lg, int option)***
    - » **envoi bloquant ou non selon *option***

- **Extraire un message d'une file :**
  - ***int msgrcv(int msgid, const void *p_msg, int lg, long type, int option)***
  - **possibilité de définir des priorités avec *type***

Sémaphores

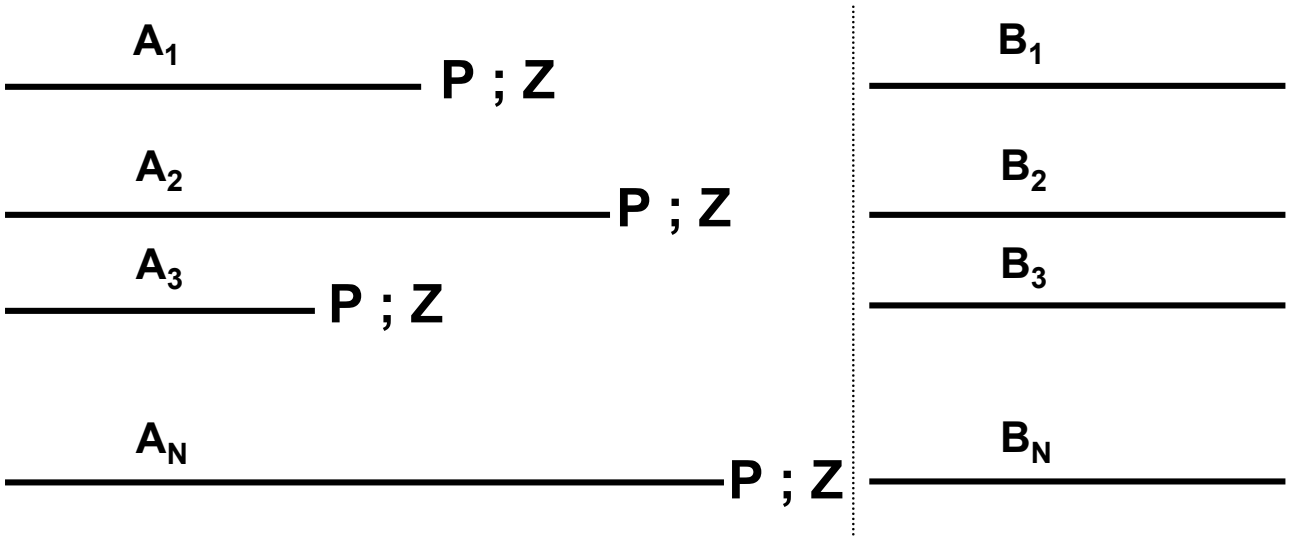
- **Mécanisme de synchronisation des processus**

- **Définition :**
  - **un sémaphore S est une variable entière (positive ou nulle)**
  - **2 opérations possibles :**
    - » **P(S) : si S=0 alors mettre le processus en attente, sinon S ← S-1**
    - » **V(S) : S ← S+1 ; réveiller un (ou plus) processus en attente**
  - **P et V non interruptibles**
  - **Mémorisation des opérations P non satisfaites (= processus en attente)**
  - **Fonction Z = attente qu'un sémaphore devienne nul [Unix]**

Sémaphores : exemple

- **Rendez-vous de processus**
  - **N processus P_i**
  - **le code de chaque P_i est partagé en 2 parties A_i ; B_i**
  - **on veut que tous les A_i soient exécutés avant de passer aux B_i**

  - **Utilisation d 'un sémaphore S initialisé à N**



Sémaphores : quelques primitives

- **Créer ou trouver un sémaphore existant :**
  - ***int semget(key_t cle, int nbsems, int option)***
    - » **retourne un n° de sémaphore (*semid*) en fonction d'une clé**
- **Initialiser la valeur d'un sémaphore :**
  - ***int semctl(int semid, int num, int cmd, arg)***
- **Appliquer une opération (P, V, Z) sur un sémaphore :**
  - ***int semop(int semid, int sops, nops)***

Segment de mémoire partagée

- **Au lieu de communiquer en échangeant de l 'information par un support externe, les processus vont partager un même espace physique**

- **Besoin d'une synchronisation des accès à cet espace pour éviter tout problème**

- **Segment de mémoire partagé :**
  - **existence indépendante des processus**
  - **rattaché à un processus lorsqu'il le demande**

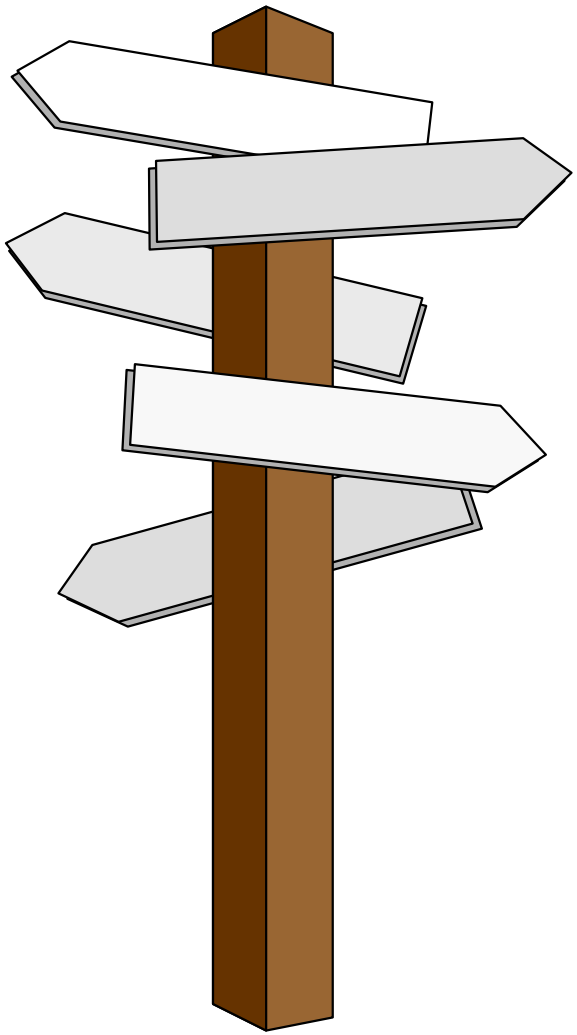
Segment de mémoire partagée : primitives

- **Créer ou trouver un segment partagé :**
  - ***int shmget(key_t cle, int taille, int option)***
    - » **retourne un n° de segment (*shmid*) en fonction d'une clé**

- **Attacher un segment partagé à un processus :**
  - ***void *shmat(int shmid, const void *adr, int option)***
    - » **attachement à l 'adresse *adr* du processus**

- **Détacher un segment partagé :**
  - ***void *shmdt(const void *adr)***

- **Opérations de contrôle :**
  - ***int shmctl(int shmid, int op, ...arg)***



Références

Processus : généralités

- Systèmes d'Exploitation - A. Tanenbaum (InterEditions)

Processus et programmation C sous UNIX/Linux :

- La programmation Unix - J.M. Rifflet (Ediscience)
- Programmer avec les outils GNU - M. Loukides & A. Oram (O'Reilly)
- Linux in a nutshell - E. Siever (O'Reilly)

A suivre :

- TP n°11 : Communication inter-Processus (C | Unix)
- Cours n°12 : Interblocage