Volatiana RALAMBONDRAINY – Samy FOUILLEUX – Maxime CHAMBREUIL
ASI3 – Année 2001 / 2002

Architecture des Ordinateurs & Systèmes d'Exploitation
TP 10

Exercice 1 : La gestion des processus sous UNIX

Processus fils / pères PID PPID

session 1

La commande tty donne le terminal dans lequel on travaille : dev/pts/13. Puis on exécute plusieurs commandes à la fois en les mettant dans des pipes cat|tail|pr|wc

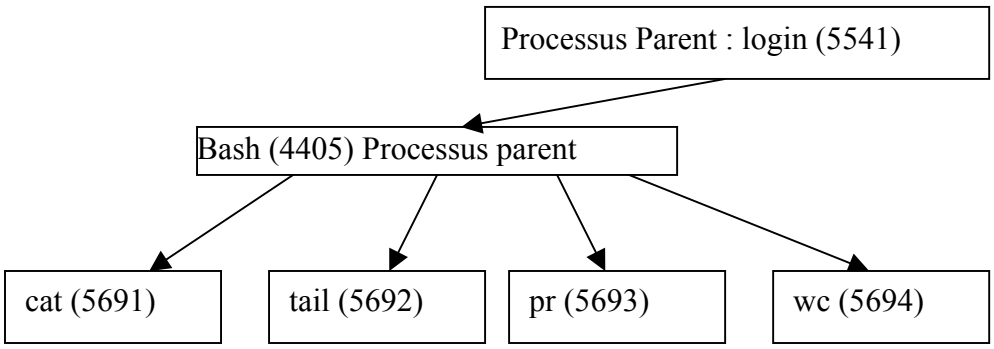
session 2

on tape ps -l -t dev/pts/13 et on obtient

```
[vola :-)vralambo]>>ps -l -t /dev/pts/13
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
100 S     0  5541  5540  0  60   0   -   585 wait4  pts/13   00:00:00 login
100 S   560  5543  5541  0  60   0   -   572 wait4  pts/13   00:00:00 bash
000 S   560  5691  5543  0  60   0   -   396 read_c pts/13   00:00:00 cat
000 S   560  5692  5543  0  60   0   -   405 pipe_r pts/13   00:00:00 tail
000 S   560  5693  5543  0  60   0   -   439 pipe_r pts/13   00:00:00 pr
000 S   560  5694  5543  0  60   0   -   398 pipe_r pts/13   00:00:00 wc
```

Le processus qui correspond au bash a comme identifiant de processus 5543 et identifiant de proccesus père 5541 car il s'agit d'un fils du processus login. De même toutes les processus « pipés » ont tous un PID différent car à chaque instruction correspond un processus par contre, en tant que fils du bash ils ont tous le même PPID (5543).

Schéma d'enchaînement de création de processus



session 1

L'option 'f' qu'on rajoute à la commande ps permet d'avoir l'arborescence (f='forest') des processus dans la dernière colonne, arborescence qui correspond bien à notre schéma ci-dessus.

```
[vola :-)vralambo]>>ps f -l -t /dev/pts/13
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY        TIME CMD
100 S     0  5541  5540  0  60   0    -   585 wait4  pts/13     0:00 login -- vr
100 S   560  5543  5541  0  60   0    -   572 wait4  pts/13     0:00 -bash
000 S   560  5691  5543  0  60   0    -   396 read_c pts/13     0:00  \_ cat
000 S   560  5692  5543  0  60   0    -   405 pipe_r pts/13     0:00  \_ tail
000 S   560  5693  5543  0  60   0    -   439 pipe_r pts/13     0:00  \_ pr
000 S   560  5694  5543  0  60   0    -   398 pipe_r pts/13     0:00  \_ wc
```

Processus daemons

session 2

La commande top nous donne les renseignement suivants :

```
 PID USER   PRI  NI  SIZE  RSS SHARE STAT  CPU MEM   TIME COMMAND
5567 vralambo  12   0  1108 1108   816 R     2,5  0,2   0:01 top
5487 arogozan   8   0 16108  15M  8896 S     2,1  3,1   0:11 netscape-commun
6667 root       2   0     0    0     0 SW    0,1  0,0  14:28 nfsd
6669 root       1   0     0    0     0 SW    0,1  0,0  14:27 nfsd
6680 root       1   0     0    0     0 SW    0,1  0,0  14:21 nfsd
```

Elle permet de voir à un instant t donné tous les états des processus

PID	Numéro du processus
USER	utilisateur
PRI	Priorité courante du processus
NI	Priorité initiale du processus
SIZE	Taille du processus
RSS	Mémoire totale physique utilisée par le processus en kilo Bytes
SHARE	Mémoire partagée
STAT	Status du processus
%CPU	Le temps utilisé par la tâche depuis le dernier raffraichissement de l'ecran
%MEM	Le partage de la tâche au niveau de la mémoire physique
TIME	Temps total utilisé par la tâche depuis le début

COMMAND	Nom de la commande

Pour afficher seulement la liste des processus en cours, on utilise l'option u lors de l'exécution de top et on rentre l'identifiant de l'utilisateur.

Etat d'un processus

session 2 : La commande top q permet de voir en temps réel le déroulement des processus. On voit bien le champ PRI qui change en permanence, la priorité du processus change en permanence selon l'ordonnancement exécuté par le SE.

session 1 : Pour suspendre la commande en avant-plan : ctrl +z. On vérifie avec jobs. Pour la remettre à l'avant-plan on fait fg (foreground) et bg à l'arrière plan (background).Lorsqu'on applique la commande jobs, le système nous retourne tous les processus qui tournent avec un numéro relatif. fg %1 permet de mettre en avant plan le processus qui a le numéro relatif 1 et fg 1 le processus qui a un '1' dans son nom.On peut très bien faire fg nomduprocessus comme fg netscape par exemple.

Priorités

session 1 : on relance la commande en arrière plan avec (commande)&

session 2 :

Pour trouver l'aide on-line, on lance la commande top q et ensuite pendant l'execution de ce programme on tape ? on obtient l'aide. Pour restreindre à l'affichage de nos processus seuls, il suffit de taper u lors de l'exécution du top et de rentrer le login vralambo
La variable pri correspond à la priorité du processus lancée, elle varie à cause de l'ordonnanceur qui affecte les priorités.
Pour baisser la priorité du bash, on utilise la commande renice dont la nomenclature est : renice 9 PIDduprocessusbash

0: old priority 0, new priority 9

Top ou ps permet de vérifier la valeur de la priorité
.
Quand on essaie de baisser la priorité du bash à 4, le système nous répond permission non accordée ! Normal car il est possible de restreindre l'accès, mais pas de s'octroyer des libertés supplémentaires seul l'utilisateur possèdant les accès (root, par ex.) peut le faire.

Pour la commande more/etc/passwd & la priorité est de 9, la même que celle de son père (bash). En baissant la priorité du bash, on constate que la priorité du processus fils more n'a pas du tout été affectée. Lorsqu'on modifie la priorité du père, on n'influe pas sur la priorité des fils déjà crées.

Exercice 2 : Scripts bash

1)Script de fin de TP :

On veut un programme qui tous les vendredis à une certaine heure nous affichera « fin du tp ». Il y a un fichier demon qui s'appelle crontab qui execute de manière cyclique certaines taches. Donc il faut faire un programme qui affiche « fin du tp » et dans le fichier crontab il faut préciser que ce programme s'exécutera chaque vendredi à 17.40. Les * signifient que cela marche quelque soit le mois ou l'année.

```
#!bin/bash
# finTP

      echo 'fin du TP'
fi
```

et dans le fichier etc/crontab rajouter la ligne
40 17 * * 6//home/etud/asi01/vralambo/SE_2001/TP10/ `finTP`

2) On désire un script qui affiche l'ensemble des processus d'un utilisateur donné en paramètre et leurs états respectifs. On utilise ps avec le format long (-l) pour avoir les caractéristiques désirées des processus. Pour restreindre l'affichage aux processus de l'utilisateur il faut utiliser l'option –u et comme on met le nom d'utilisateur qui est en paramètre $1. Le problème : récupérer les champs intéressants. Le cut ne marche pas très bien car on se rend compte que le nombre d'espaces n'est pas le meme à chaque ligne, donc il est difficile de préciser les différents champs avec le délimiteur espace. Une astuce est de definir le format de ps qui nous convient avec l'option –o
Pour cela on utilise la commande `ps -u $1 -o state -o cmd` >resultat qu'on met dans un fichier resultat.
A partir de ce moment on peut utiliser cut parce que le 1er champ se réduit qu 'à une seule lettre et donc on pourra faire un test sur cette première
if (var=D)then echo « sommeil continu »
if (var =R) then echo « en cours »
if(var =S) then echo « sommeil »
if(var =T) then echo « stoppé »
if(var =Z) then echo « zombie »
Cependant, pour réussir à faire une boucle qui descend à chaque fois pour le processus suivant, je n'ai pas trouvé d'autre moyen que de mettre le résultat de la commande dans un fichier résultat et de faire une boucle for i= 1 jusqu'au nombre de ligne de résultats (obtenu par wc –l). il ne faut pas oublier d'effacer le fichier resultat à la fin du script, c'est plus propre. Comme je n'ai pas trouvé de boucle for, on se débrouille avec while pour un tant qu'on n'a pas atteint la fin du fichier, et après on désire la i-eme ligne donc on va prendre les i dernières lignes depuis la fin avec 'tail' et prendre la première de ces dernières lignes avec 'head'.

```
#!bin/bash
# etats processus

if (test -e $2);
      then echo "Utilisateur:$1"


      ps  -u $1 -o state -o cmd > resultat
      nbprocessus=$(wc -l<resultat)
      i=1
      while [ $i -le $nbprocessus ]
```

```
      do
            com=$(tail -$i resultat|head 1|cut -d" " -f1)
            etat=$(tail -$i resultat|head 1|cut -d" " -f2)

            let i=i+1
            echo -n " $com : "
            case $et in
                  D) echo "endormi (non-interruptible)";;
                  S) echo "endormi" ;;
                  R) echo "en cours" ;;
                  T) echo "stoppe" ;;
                  Z) echo "zombi" ;;
                  *) echo "inconnu" ;;
            esac
      done
rm resultat
else echo " veuillez rentrer 1 seul parametre"
fi
```

3)Question piège , à chaque fois qu'on lance un processus on réinitialise le bash, donc impossible d'afficher les processus en cours.

4) De la meme manière que dans le 1 on utilise crontab avec le script placeoccupee. La fonction pour trouver la place utilisée est du avec l'option h pour qu'elle soit plus lisible. Et éditer après le fichier etc/crontab pour que l'opération soit périodique.

!bin/bash
#placeoccupee
du –h>placeoccupee|mail user

Architecture des Ordinateurs & Systèmes d'Exploitation
TP 11

Objectifs :

- Manipulation de processus : utilisation des primitives systèmes C sous Unix
- Communication de processus : utilisation des primitives systèmes C sous Unix

Exercice 1 : Manipulation de processus

On écrit et on compile le programme essai.c.

```
void main()
{
  int i;
for(i=0;i<5;i++)
    { printf("\n je suis le programe qui boucle\n");
    }
}
```

1 –
Pour exécuter le programme en avant-plan, il suffit de taper *./essai*, ce qui lance le programme qui doit être dans le répertoire courant.
Lorsqu'on fait CTRL C, le processus associé à ce programme est alors arrêté.

2 –
On ré exécute le programme en avant-plan. Lorsqu'on fait CTRL Z, le processus associé à ce programme est alors suspendu. En tapant jobs, on peut voir la liste des processus qui sont soit en arrière-plan soit suspendus.

3 –
Pour exécuter le programme en arrière-plan , il faut taper *./essai &*. Mais l'affichage continue à s'exécuter en avant-plan. Pour ne plus être dérangé par l'affichage, il faut envoyé la sortie du programme dans un fichier. Mais si on l'envoie dans un fichier normal, celui ci va grossir jusqu'à remplir tout le disque dur. Pour éviter cela, on va envoyer la sortie du programme dans un fichier ou toutes les données sont supprimées.
On tape la commande *./essai > /dev/null &*. Le programme va alors s'exécuter en arrière-plan sans que l'on soit dérangé par les affichages. Le processus associé à ce programme est alors en arrière-plan.

Exercice 2 : Création de processus en C sous Unix

A –
Programme concur.c

```
void main()
{       int pid,i;
        pid=fork();
        if(pid!=0)
        {for(i=0;i<10;i++)    {printf("\n Je suis le processus pere %d",i);sleep(1);}        }
```

```
        if(pid==0)
        {for(i=0;i<10;i++)    {printf("\n Je suis le processus fils %d", i); sleep(1);}        }
}
```

1 –
On exécute le programme en avant-plan. On obtient :

```
Je suis le processus fils 0
Je suis le processus pere 0
Je suis le processus fils 1
Je suis le processus pere 1
Je suis le processus fils 2
Je suis le processus pere 2
```

On voit que le processus père et le processus fils exécute une fois la boucle l'un après l'autre. On peut signaler que ce n'est pas obligatoirement le fils qui commence. Cela peut être le père.

2 –
On rajoute une boucle d'attente et un *sleep* au programme essai.c.

```
void main()
{
  int i;
  for(i=0;i<10;i++)
    { printf("\n je suis le programme qui boucle\n");sleep(1);
    }
}
```

On exécute les programmes concur.c et essai.c (commande : *./essai & ./concur &*). Les deux programmes sont donc lancé en arrière-plan. On obtient :

```
Je suis le processus fils 0
Je suis le processus pere 1

je suis le programme qui boucle
Je suis le processus fils 1

je suis le programme qui boucle
Je suis le processus pere 2
```

On voit bien apparaître l'entrelacement des trois processus. On peut aussi remarquer que les deux processus qui ont été lancés dans le même programme ne se suivent pas car le troisième processus est apparu entre les deux autres.

3 –
On modifie le programme concur.c pour que le processus fils exécute le programme essai.c.

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
main()
{       int pid,i,a;
        pid=fork();
        if(pid!=0)
        {for(i=0;i<3;i++)
                {printf("\n Je suis le processus pere %d",i); sleep(1);}
```

```
        }
        if(pid==0)
        {for(i=0;i<3;i++)
                {printf("\n Je suis le processus fils de mon pere %d\n", i);
                a=execlp("./essai","./essai",NULL);
                printf("le programme marche pas\n",a) ;}
        }
}
```

On obtient alors :

```
Je suis le processus fils de mon pere 0
je suis le programe qui boucle
je suis le programe qui boucle
je suis le programe qui boucle
je suis le programe qui boucle
je suis le programe qui boucle
Je suis le processus pere 0
Je suis le processus pere 1
Je suis le processus pere 2
```

On s'aperçoit d'abord que c'est le programme fils qui commence. Celui-ci entre dans la boucle du programme qui le concerne, et affiche qu'il est le fils. Ensuite la commande execlp fait que le processus fils exécute le code de essai .c. Il est donc normal qu'il affiche les 5 lignes du programme essai, puis qu'il ne fasse rien ensuite. En effet la commande execlp fait que le processus fils oublie son code de départ. Une fois qu'il a finit de faire essai.c, il n'affiche pas la ligne « le programme ne marche pas », mais il meurt. Le père continue a exécuté le code original et s'arrête lorsqu'il a fait 3 fois la boucle.
En résumé, la commande execlp remplace virtuellement le code du processus par le code du programme qu'il doit exécuter. Le processus ne peut alors plus revenir sur son code original.

B –

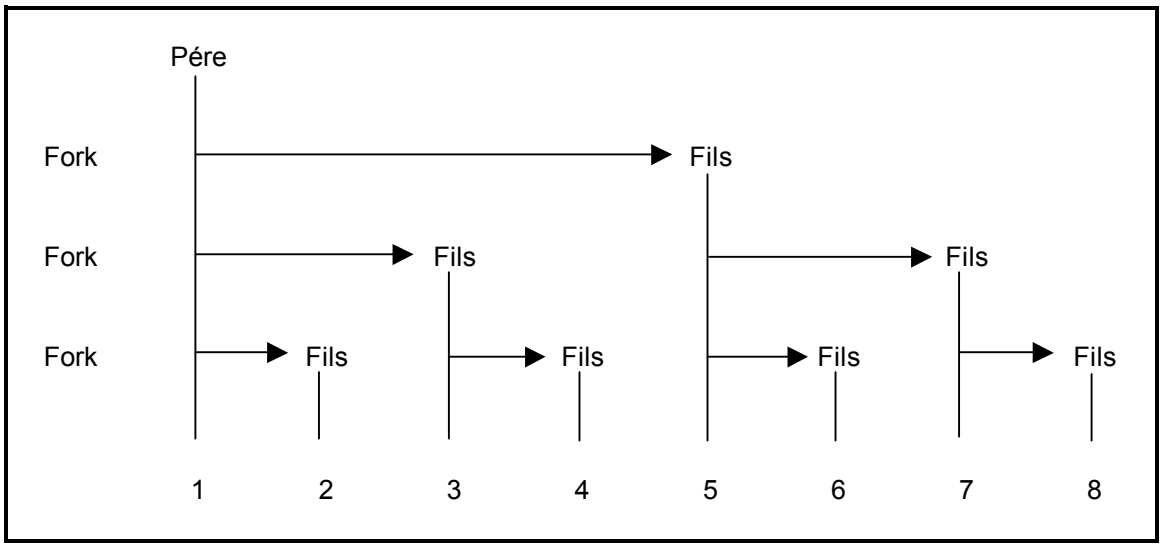
Programme bidon.c

```
void main()
{
int a=0;
int pid,pid2,pid3;
pid=fork();
if(pid==0)
{printf("coucou fork 1");}
pid2=fork();
if(pid2==0)
{printf("coucou fork 2");}
pid3=fork();
if(pid3==0)
{printf("coucou fork 3");}
}
```

Le programme nous donne :

```
coucou fork 1
coucou fork 2
coucou fork 2
coucou fork 3
coucou fork 3
```

coucou fork 3
coucou fork 3
On voit que nous avons 7 affichages, ce qui veut dire que 7 processus fils ont été créés. En comptant le processus père, cela nous fait 8 processus lancés par le programme bidon.c. On peut résumer ce qui s'est passé par le graphique suivant.



On voit que 8 processus sont créés.

Exercice 3 : Envoi de signaux

Programme demo-signal.c

```
#include<signal.h>
void handler(int sig)
{printf("Signal SIGUSR1 recu\n");
  signal(sig,handler);}
void main()
{signal(SIGUSR1,handler);
 for(;;){printf("a\n");sleep(1);}}
```

1 –
On exécute le programme en arrière-plan. Pour que le message s'affiche, il faut envoyer au processus le signal SIGUSR1, qui est un signal définit par l'utilisateur. Pour cela, on tape dans une autre console la commande *kill –SIGUSR1 6512*, le numéro donné étant le numéro de PID du processus. On peut aussi taper la commande *kill –10 6512*, qui fait la même chose, a part qu'on ne définit pas le signal par son nom (*SIGUSR1*) mais par son numéro (*10*).

2 –
On veut maintenant réaliser un programme qui remplace la commande précédente. Pour cela il faut utiliser dans le programme la fonction *execlp* qui nous permet de lancer la commande souhaitée.
Programme envoi-signal.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
void main()
```

```
{char pid[10];
 printf("Entrez le numero de PID\n");
 scanf("%s",&pid);
 execlp("kill","kill","-10",pid,NULL);}
```

Il nous suffit alors de rentrer le numéro de PID du processus pour que la commande soit exécutée.

3 –

On veut modifier le programme demo-signal.c pour que celui-ci réagisse à deux signaux différents. Le premier fait incrémenter un compteur, le deuxième fait afficher la valeur de ce compteur.

```
#include<signal.h>
int compteur=0;
void handler(int sig)
{
printf("Signal SIGUSR1 recu\n");
compteur++;
signal(sig,handler);
}

void handler2(int sig)
{
printf("Signal SIGUSR2 recu : nb iteration : %d\n",compteur);
signal(sig,handler2);
}

void main()
{
signal(SIGUSR1,handler);
signal(SIGUSR2,handler2);
for(;;){printf("a\n");sleep(1);}
}
```

Le processus nous donne affiche *Signal SIGUSR1 recu* lorsque l'on envoie un signal SIGUSR1, et il affiche bien le nombre de fois que l'on a envoyé un signal SIGUSR1 lorsqu'on lui envoie un signal SIGUSR2.

Exercice 4 : Sémaphores

1) Utilisation de base d'un sémaphore en C

- Le programme se découpe comme suit : on a d'abord la déclaration des librairies utiles pour l'utilisation des sémaphores. Ensuite, on trouve 4 procédures : P, V, Z et init_sem. Dans chacune de ses procédures, on a une déclaration d'une variable structurée op de type sembuf, puis une initialisation de ses champs avant de faire une commande. Enfin, on a le programme principal, reconnaissable par main. Ce programme déclare un entier semid, qui est initialisé comme étant la clé d'un IPC (fichier de communication inter processus) à l'aide de la fonction semget. Enfin, le programme ne s'arrête pas : La boucle for n'a pas de condition d'arrêt.

Après avoir exécuter ce programme en tâche de fond (./demo-semaphore &), la commande ipcs nous donne :

```
[TP11] maxounet >> ipcs
------ Segments de mémoire partagée --------
touche   shmid   propriétaireperms   octets   nattch   statut
0x00280267 1       root     644      1048576   1
------ Tables de sémaphores --------
touche   semid   propriétaireperms   nsems   statut
0x00280269 0       root     666      14
0x000005eb 2565    mchambre  666      1
------ Files d'attente de messages --------
touche   msqid   propriétaireperms   octets utilisésmessages
```

On a les trois types d'IPC : mémoire partagée, sémaphore ou message. Pour chacun, on a sa clé (ou touche), son identifiant (~id), son propriétaire, ses permissions d'accès, le nombre de sémaphore de l'utilisateur, l'espace mémoire qu'il occupe et son statut. Après avoir tué le processus associé à demo-semaphore, le sémaphore est toujours là. Pour le supprimer, il faut spécifier à la commande ipcrm que ce sera un IPC de type sémaphore et l'identifiant de cet IPC (ici : ipcrm sem 2565).

- La chaîne « Operation P » s'est affichée 5 fois donc la fonction s'est exécutée 4 fois au moins. En remplaçant dans la boucle, l'opération P par deux P et une V, on devrait obtenir 3 affichages de la chaîne, puisqu'on a eu un problème à la 5ème itération de la fonction P. Or on a toujours 4 affichages. La fonction P suspend l'affichage si semid = 0, or semid est initialisé à 4 grâce à la fonction init_sem et surtout à la variable val0,qui ne prend que 4 comme valeur. En arrivant sur la boucle, le processus se déroule comme suit :

On affiche la chaîne
On fait 2 fois la fonction P : semid = 2
On fait la fonction V : semid = 3
On affiche
On fait 2 fois la fonction P : semid = 1
On exécute la fonction V : semid = 2
On affiche
On fait 2 fois la fonction P : semid = 0 (en sortie de la fonction)
On fait la fonction V : semid = 1
On affiche
On fait la fonction P une fois : semid = 0
On refait la fonction P mais comme l'argument est nul, le processus est suspendu.

On a bien 4 affichages de la chaîne.

- On a les invites d'exécution de la commande V, qui décrémente semid à chaque fois qu'on le souhaite.

- On a les affichages d'exécution (Operation P) de demo-semaphore, puis les invites d'exécution de la fonction v par demo-semaphore2. Par contre, je n'ai jamais vu l'affichage de demo-semaphore3 donc semid ne s'annule pas.

2) Programmation du « rendez-vous de processus » vu dans le cours

Pour que les 2 processus s'arrête en même temps (lors de la saisie du caractere par l'utilisateur, on initialise le sémaphore avant que le fils ne soit créé. Puis on fait un P après les boucles pour endormir le processus fils. Dans le processus père, on attend le caractère. Une fois qu'il est saisi on réveille le processus fils avec un V. On a donc l'affichage de la mort de la famille en même temps.