TP 1 – Les processus

A. Gestion des processus

• Fonctions utiles : (N'hésitez pas à utiliser le MAN – Manuel du Programmeur Unix)

#include <unistd.h> ***unsigned int sleep(unsigned int seconds)***

Le processus appelant s'endort pour environ seconds secondes. La valeur renvoyée par sleep est la différence entre le nombre demandé et le nombre de secondes effectives de sommeil. Cette valeur peut être > 0 car sleep est suspendue par n'importe quel signal arrivant au processus. La plupart du temps, on se contente d'utiliser sleep comme une procédure : sleep(2) ;

#include <unistd.h> **pid_t fork() ;**

Le processus appelant crée un processus identique à lui-même ; le processus appelant est le père et le processus créé est le fils. Attention, tout appel à la fonction fork() provoque cette création ! fork() rend la valeur 0 dans le processus fils, et le numéro d'identification (PID) du fils créé dans le processus père. On récupère donc cette valeur par une instruction du type : pid_t ident = fork(). En cas d'erreur, fork renvoie -1.

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

Renvoient au processus appelant son numéro d'identification (getpid) ou le numéro d'identification de son père (getppid). Renvoient -1 en cas d'erreur.

void exit(int status)

Le processus appelant met fin à son exécution. Un appel exit(*status*) est équivalent à un return status dans la fonction main d'un programme C.L'entier status est un code de terminaison et est rendu au processus père si celui-ci attend la fin de son fils (voir wait).

#include <sys/types.h> #include <sys/wait.h>

pid_t wait(int *stat loc)

Cette fonction permet à un processus d'attendre la mort d'un de ses fils. Si le processus n'a pas eu de fils, ou si tous les fils sont morts au moment de l'appel de wait, la fonction rend -1. Si un fils est déjà mort, la fonction rend le numéro d'identification de ce fils. Sinon le processus est bloqué jusqu'au décès d'un fils (le premier qui meurt) et rend son numéro (cette attente n'est donc pas sélective).

Cette fonction s'utilise de 2 façons :

- wait(NULL) donne le fonctionnement décrit ci-dessus ;
- wait(&status) la valeur renvoyé par le fils par exit est stockée dans l'entier status.

include <sys/types.h> #include <sys/wait.h>

int waitid (idtype_t idtype, id_t id, siginfo_t *infop, int options);

fournit des moyens plus fins de contrôler quels changements d'états attendre. Les arguments idtype et id sélectionnent le(s) fils à attendre, comme suit :

- *idtype == P_PID Attendre la fin du processus numéro id.*
- *idtype == P_PGID Attendre la fin de n'importe quel processus fils appartenant à un groupe de processus d'ID id.*
- *idtype == P_ALL Attendre n'importe quel fils ; l'argument id est ignoré.*

  Les changements d'état à attendre sont indiqués par un OU binaire des attributs suivants dans le paramètre options :

- *WEXITED Attendre les fils qui se sont terminés.*
- *WSTOPPED Attendre les enfants qui ont été arrêtés par un signal.*
- *WNOHANG Comme pour waitpid().*
- *WNOWAIT Laisser le fils dans un état prêt ; un appel ultérieur à wait() pourra de nouveau fournir des informations sur l'état du fils. Si l'appel réussit, waitid() remplit les champs suivants de la structure siginfo_t pointée par infop :*
  - *si_pid L'identifiant de processus du fils.*
  - *si_uid L'UID réel du fils. Ce champ n'est pas rempli par la plupart des autres implémentations.*
  - *si_signo Toujours SIGCHLD*
  - *si_status Soit le code de retour du fils donné à _exit(2) ou exit(3), soit le signal ayant provoqué la terminaison, l'arrêt, ou la relance du fils. Le champ si_code permet de comment interpréter ce champ.*
- WCONTINUED Attendre les enfants précédemment arrêtés qui ont été relancés par le signal SIGCONT. Les attributs suivants peuvent également être utilisés dans **options**.

Voir également

- **wait3 (status, options, rusage)**
- **wait4 (pid,status,options, rusage)**

B. Recouvrement

Un processus peut lancer un programme exécutable en utilisant un des appels système **exec.** Le code du programme remplace le code du processus en cours. Il existe de nombreuses versions de fonctions exec (execl, execv, execle, execve*, execlp, execvp). Les différentes versions portent sur la manière de transmettre les arguments et l'environnement, et sur la méthode pour accéder au programme à lancer.

```
#include <unistd.h>
int execlp(char *path, char *arg0, char *arg1, ..., char *argn, NULL)
```

path est une chaîne de caractères donnant le nom du fichier qui contient le programme à exécuter (cherche dans les chemins du PATH, arg0 contient par convention le nom du fichier (sans répertoire) et les autres arg sont les paramètres du programme à exécuter. La liste des paramètres doit se terminer par un pointeur nul (NULL).

Exemple : execlp("ls", "ls", ".c", NULL) ;*

C. Informations statistiques sur les processus

a) struct rusage

include <sys/resource.h>

int getrusage(int who, struct rusage * stat)

who indique quelles sont les statistiques dont j'ai besoin. Cet argument peut prendre la forme de :

- RUSAGE_SELF pour obtenir les informations concernant le processus appelant
- RUSAGE_CHILDREN pour obtenir les informations concernant le processus fils terminé

rusage est une structure qui est remplie lors de l'appel. Dans notre cas seul deux champs nous intéressent :

- ru_utime de type timeval qui permet de donner le temps passé (en seconde) par le processus en mode utilisateur.
- ru_stime de type timeval qui permet de donner le temps passé (en seconde) par le processus en mode noyau.

b) struct siginfo_t

```
typedef struct {
    int     si_signo;         /* Signal number */
    int     si_code;          /* Signal code */
    int     si_trapno;        /* Trap number for hardware-generated signal
                                 (unused on most architectures) */
    union sigval si_value;    /* Accompanying data from sigqueue() */
    pid_t   si_pid;           /* Process ID of sending process */
    uid_t   si_uid;           /* Real user ID of sender */
    int     si_errno;         /* Error number (generally unused) */
    void   *si_addr;          /* Address that generated signal
                                 (hardware-generated signals only) */
    int     si_overrun;       /* Overrun count (Linux 2.6, POSIX timers) */
    int     si_timerid;       /* (Kernel-internal) Timer ID
                                 (Linux 2.6, POSIX timers) */
    long    si_band;          /* Band event (SIGPOLL/SIGIO) */
    int     si_fd;            /* File descriptor (SIGPOLL/SIGIO) */
    int     si_status;        /* Exit status or signal (SIGCHLD) */
    clock_t si_utime;         /* User CPU time (SIGCHLD) */
    clock_t si_stime;         /* System CPU time (SIGCHLD) */
} siginfo_t;
```

Exercices

1. Création

Lancement de commandes en premier/arrière plan

a) Examinez **prg1.c** tapez echo $$ pour avoir le pid du shell puis exécutez plusieurs fois **prg1**.

Que déduisez-vous de l'ordre d'apparition des messages ?

b) Afin de provoquer la terminaison du processus fils après celle de son père, tapez maintenant *prg1 -1*. Que pensez-vous du ppid du processus fils ?

c) Tapez : *prg1 -2 prg1 2*

Expliquez la raison pour laquelle le shell vous rend la main tout de suite dans un cas et pas dans l'autre ?

d) Modifiez **prg1.c** pour que le processus père écrive sur la sortie standard juste après la terminaison du processus fils quelle que soit la valeur du paramètre temps.

2. Terminaison du processus fils et affichage de sa valeur de retour

a) Programmer l'interaction Père Fils en considérant que le père récupère la valeur de stratus du Fils. Les traces d'exécution de ce programme pourraient être :

je suis le père: mon pid est 4243 je suis le fils: mon pid est 4244 pid de mon père, 4243 pid de mon fils, 4244 je suis le père; le pid de mon fils mort est 4244 je suis le père; le code retour de mon fils est 0

Appeler ce programme prg2.c

b) Modifier le programme de sorte à utiliser wait3() et wait4(). Appeler ce programme prg2bis.c
c) Modifier le programme de sorte à utiliser à récupérer les informations concernant l'activité du processus en mode noyau et en mode user. Appeler ce programme prg2ter.c

3. Recouvrement

a) Examinez **prg3_avant.c** et **prg3_apres.c** puis exécutez **prg3_avant.** Pourquoi le message "ne passe pas ici" n'apparaît-il pas ?

b) Afin de comprendre l'association fork + exec utilisée lors du lancement de commandes Unix par le shell, créez deux exécutables père et fils qui affichent respectivement "je suis le pere", "je suis le fils" tels que le processus père n'affiche son message qu'après celui du processus fils.

c) Écrire un programme qui crée 2 processus l'un faisant la commande ls -l, l'autre ps -l. Le père devra attendre la fin de ses deux fils et afficher quel est celui qui se termine en premier.

4. Informations du processus

A partir du programme **prg4.c** modifier les paramètres afin de récupérer les valeurs correspondantes à la structure de siginfo_t.