Processus

Les processus : Rappels (?)

Un processus :

- a un numéro (le PID = Process IDentifier)
- a un père.
- a un propriétaire (réel et effectif)
- a un état : en exécution, en sommeil, stoppé ou zombie
- a un niveau de priorité
- un répertoire courant
- (certains processus) peuvent être “légers” (“threads”) (on y reviendra dans quelques cours)
- une table de descripteurs de fichiers
- une table de pages
- renvoie un code retour (un entier entre 0 et 255)

Voir les processus

- Pour voir tous les processus tournant, avec leur lien de parenté : `ps faux`
- Pour voir en temps réel (utilisation CPU, mémoire...) : `top`
- Toutes les informations dans le système procfs (`/proc/`)

État d'un processus

Un processus peut être :

- actif (i.e. en exécution)
- prêt (en attente d'exécution)
- suspendu (par exemple avec ctrl+z)
- en sommeil : il attend un évènement
  - sleep, pause...
  - attente sur une I/O
- zombi

Ordonnancement

Deux processus ne peuvent pas s'exécuter en même temps sur un même coeur.

L'OS découpe le temps en petits bouts, et fait tourner les processus les uns après les autres.

L'OS choisi l'ordre, en essayant de respecter la priorité des processus (voir `nice`)

Passage d'un processus à un autre : commutation de contexte (context switch).

- assez lent...
- transparent pour nous

Gestion de processus : syscalls

- `getpid()` renvoie le PID du processus courant
- `getppid()` renvoie le PID du père

- `getuid()` renvoie l'UID de l'utilisateur processus courant
- `geteuid()` renvoie l'UID de l'utilisateur effectif du processus courant
- Ces UIDs peuvent être différents si le binaire est en "setuid"

- `setuid()` et `seteuid()` permettent de changer les utilisateurs, si on a les droits !

- Obtenir/changer le répertoire courant : `getcwd()` / `chdir()`
- Obtenir le temps CPU consommé (en mode utilisateur et système) : `times()`
- Modifier le masque de création de fichiers : `umask()`

- Pour voir/changer les "limites" d'un processus : `ulimit`, `getrlimit`, `setrlimit`

Priorité d'un processus

- Chaque processus a une priorité :
  - généralement, un nombre entre -20 et 19, et par défaut : 0.
  - plus le nombre est élevé, moins le processus aura du temps de calcul.

- Il est possible de lancer un processus avec une priorité plus basse avec `nice`
- Il est possible de diminuer la priorité d'un de ses processus en exécution :
  - Commande : `renice`
  - Appel système : `nice`

- Seul `root` a le droit d'augmenter une priorité.

Comment lancer un processus ?

Il faut différencier le fait de :

- créer un nouveau processus : le processus appelant continue de vivre, et un nouveau processus naît
- exécuter un binaire spécifié : il n'y a pas de nouveau processus, l'ancien processus est "écrasé" par le nouveau

Créer un nouveau processus (création) se fait avec `fork`

Exécuter un binaire (recouvrement) se fait avec `exec...`

Créer un nouveau processus qui est l'exécution d'un binaire se fait avec la combinaison de `fork` et `exec...`.

`int system(const char *command)`

(Pas un appel système. Donné à titre informatif.)

Un moyen simple de lancer un processus depuis un programme est d'utiliser `system` (dans <stdlib.h>).

`system` lance un shell (`/bin/sh`) qui exécutera `command`. Une fois la commande terminée, la fonction retournera le code retour de la commande.

Exemple : `system("ls");`

Les exec*

```c
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
    char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
    char *const envp[]);
```

- Elles ne créent pas un nouveau processus : elles remplacent (recouvrent) le processus courant par l'exécution du fichier en argument.
- En particulier, le PID, L'UID, les fichiers ouverts sont conservés (sauf si option O_CLOEXEC)
- Si l'exécution se fait normalement, elles ne retournent jamais !

`pid_t fork(void)`

`fork()` (dans <unistd.h>) est la commande pour lancer un nouveau processus.

Elle duplique le processus courant, pour créer un processus fils.

- Dans le père, elle renvoie le PID du fils (et rien ne change)
- Dans le fils (le nouveau processus) :
  - elle retourne 0
  - le fils aura un nouveau PID
  - son père sera le processus père

`pid_t fork(void)`

- `fork` retourne donc deux fois, une fois dans le père, une fois dans le fils
- Tout se passe comme si toute la mémoire du processus appelant fork est copiée.
- (En pratique, le système copie seulement si c'est nécessaire.)
- Les deux processus sont concurrents. On ne peut pas dire lequel des deux retournera en premier.

Exemple : fork

```c
#include <unistd.h>

int main()
{
  if(fork()==0) {
    /* si on est ici, on est le fils */
    /*...*/
    return 0; /* fin du fils */
  }
  /* si on est ici, on est le pere */
  /*...*/
  return 0; /* fin du pere */
}
```

fork et descripteurs de fichiers

- Lors d'un fork, la table des descripteurs du processus est copiée.
- Les descripteurs des deux processus (père et fils) référencient les mêmes fichiers ouverts par le système (comme après un `dup`)
- En particulier, si un des deux processus modifie le curseur d'un descripteur (read/write/lseek...), cela affectera le curseur du même descripteur de l'autre processus

fork et descripteurs de fichiers

```
int main()
{
   int fd=open("sortie.txt",O_CREAT | O_RDWR
       ,0644);
   if(fork()==0) {
      write(fd,"A",1);
      close(fd);
      return 0;
   }
   write(fd,"B",1);
   close(fd);
   return 0;
}
```

sortie.txt : AB ou BA

Exemple : fork + exec

```c
#include <unistd.h>

int main()
{
  if(fork()==0) {
    /* si on est ici, on est le fils */
    execlp("xeyes","xeyes",NULL);
    return 1; /* si on se trouve ici, c'est qu'
        execlp a echoue */
  }
  /* si on est ici, on est le pere */
  /*...*/
  return 0; /* fin du pere */
}
```

`pid_t wait(int *ptr)`

`wait` : attend jusqu'à ce qu'un processus fils termine.

Plus précisément :

- Si un processus fils termine avant l'appel de `wait` de son père, il devient zombi.

- Si un processus n'a pas de fils : `wait` renvoie -1.
- Si un processus a un fils zombi : `wait` renvoie le PID du fils zombi, et efface ce processus de la liste des processus.
  - Si `ptr` n'est pas NULL, `wait` copie le "statut" dans l'entier pointé par `ptr` (voir man).
- Si un processus a des fils, mais pas pas de fils zombi : `wait` attend jusqu'à ce qu'un fils devienne zombi, puis idem.

Attente d'un processus particulier :

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Exemple : fork + exec + wait

```c
#include <unistd.h>

int main()
{
  int status;
  if(fork()==0) {
    /* si on est ici, on est le fils */
    execlp("xeyes","xeyes",NULL);
    return 1; /* si on se trouve ici, c'est qu'
        execlp a echoue */
  }
  /* si on est ici, on est le pere */
  wait(&status);
  return 0; /* fin du pere */
}
```

`pid_t setsid(void)`

Un processus peut se détacher de son père en appelant `setsid()`.

Plus précisément, cette fonction sert à créer une nouvelle session.
Le processus appelant devient leader de cette session.

Exemple : nohup & (un premier essai...)

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
  argc -= 1;
  argv += 1;

  if(fork()) {
    /* si on est ici, on est le pere */
    return 0;
  }
  /* si on est ici, on est le fils */
  setsid();
  execvp(*argv, argv);
  return 1; /* si on se trouve ici, c'est qu'
     execvp a echoue */
}
```

Communication inter processus : avant goût

IPC = Inter Processus Communucation

Comment faire communiquer des processus ?

- via les entrées sorties : pas dynamique
- fichier standards : archaïque, lent, problèmes de synchronisation
- fichiers tubes
- Signaux : information très limitée (mais ça sert à plein de niveau)
- partage de mémoire
- par un canal réseau ...

[C : Pointeurs sur fonctions]

- Une fonction dispose également d'une adresse mémoire
- C'est son point d'entrée , i.e. l'adresse mémoire où commence la liste des instructions en langage machine
- Il est possible de manipuler les adresses des fonctions en C, et d'avoir des pointeurs sur des fonctions
- Il est obligatoire de savoir manipuler les pointeurs sur fonctions pour gérer les signaux et les threads...

Syntaxe en C

Le type d'un pointeur sur fonction doit contenir les types des paramètres de la fonction, et le type de retour.

- les paramètres n'ont pas besoin d'avoir de nom :
- le compilateur doit juste savoir quel type empiler sur la pile

Pour déclarer un pointeur sur une fonction :

```
type_retour (*nompointeur) (liste_arguments...);
```

Syntaxe en C

Exemple :

```
int (*fct) (int);
```

déclare fct comme étant un pointeur sur une fonction prenant en argument un entier, et revoyant un entier

Appeler une fonction pointée se fait de la même manière que pour une fonction normale.

Exemple

```
int carre(int x) {return x*x;}

int cube(int x) {return x*x*x;}

void iter(int (*fct)(int)) {
  int i;
  for(i=1;i<=10;i++)
    printf("%d : %d\n",i,fct(i));
}

void main() {
  int (*x)(int);
  x=carre;
  iter(x);
  iter(cube);
}
```

avec typedef

On peut simplifier les choses avec typedef :

- `typedef int (*typefctintint) (int);`

Définit typefctintint comme étant le type pointeur sur une fonction int $\rightarrow$ int ;

- `typefctintint fct=carre;`

Exemple 1 : atexit

`atexit` enregistre une fonction qui sera appelée à la fin (normale) du processus (après un `exit`, ou au retour du `main`)

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Exemple 2 : qsort

`qsort` est une fonction de la libc effectuant un *quick sort*.

```
#include <stdlib.h>
void qsort(
  void *base,
  size_t nmemb,
  size_t size,
  int (*compar)(const void *, const void *)
  );
```

On doit passer en argument l'adresse de la fonction de comparaison (`compar`) que `qsort` doit utiliser.

Les signaux

Les signaux : introduction

- Les signaux permettent de notifier des évènements à un processus.
- Il s'agit d'un moyen de communication limité :
  - ponctuel
  - unique information : le numéro du signal, un entier entre 1 et (généralement) 64.
- Mais très important sous Unix.

Les signaux : introduction

Beaucoup de mécanismes sous Unix utilisent des signaux. Par exemple :

- `ctrl + c` (arrêt d'une tache)
- `ctrl + z` (mise en pause d'une tache)
- Tuer un processus par `kill`
- Erreur de segmentation
- Division par 0...

Signaux standard

- SIGTERM (15) : Signal de fin (signal par défaut de `kill`)
- SIGINT (2) : Terminaison depuis le clavier (ctrl + c)
- SIGKILL (9) : Tuer un processus (on ne peut pas le contourner)
- SIGSTOP (19) : Arrêt (pause) du processus (ctrl + z)
- SIGCONT (18) : Continuer si en pause
- SIGALRM (14) : Temporisation `alarm` écoulée.
- SIGUSR1 (10) : Signal utilisateur 1.
- SIGUSR2 (12) : Signal utilisateur 2.
- SIGCHLD (17) : Fils arrêté ou terminé

Les erreurs :

- SIGFPE (8) : Erreur mathématique virgule flottante.
- SIGPIPE (13) : Écriture dans un tube sans lecteur.
- SIGSEGV (11) : Référence mémoire invalide.
- SIGILL (4) : Instruction illégale.
- ...

Signaux générés

Un signal est généré par un évènement :

- Envoi d'un signal par un autre processus
- Action sur le terminal (ctrl+c, ctrl+z...)
- Erreur (arithmétique, de segmentation ...)
- Minuterie
- Arrêt ou terminaison d'un fils...

Lorsque le signal est délivré à un processus, une action se produit :

- action par défaut
- signal ignoré
- effectuer une action choisie : handler

Un signal généré, mais pas (encore) délivré, est pendant

Si le même signal est généré plusieurs fois, on est pas sûr qu'il sera délivré le même nombre de fois

Envoyer un signal

Depuis le shell : `kill -sig pid`, où :

- *sig* est le signal : le nom (KILL, STOP,CONT...) ou numérique
- *pid* est le PID du processus à qui on lance le signal

Appels systèmes :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
int raise(int sig);
unsigned int alarm(unsigned int s)
```

`kill` envoie le signal `sig` au processus `pid`.
`raise` envoie le signal `sig` au processus courant.
`alarm` envoie le signal SIGALRM s secondes plus tard. (Si $s = 0$, annule l'alarme)

Réception : comportement par défaut

À la réception d'un signal, un comportement par défaut est définit. Celui-ci peut être :

- Terminer le processus
  - KILL, TERM, ALARM, INT, FPE, PIPE, USR1, USR2...
- Terminer le processus avec fichier `core`
  - ILL, SEGV
- Signal ignoré
  - CHLD
- Suspension du processus
  - STOP
- Continuation du processus
  - CONT

Il est possible d'ignorer ce comportement par défaut, ou d'en définir un autre, pour tous les signaux, sauf SIGSTOP et SIGKILL

Ensemble de signaux

`sigset_t` est un type pour un ensemble de signaux. Une variable de ce type peut être manipulé par les fonctions suivantes.

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum
    );
```

Masquer des signaux

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
    sigset_t *oldset);
```

- how :
  - SIG_SETMASK : nouveau masque = set
  - SIG_BLOCK : nouveau masque = ancien masque $\cup$ set
  - SIG_UNBLOCK : nouveau masque = ancien masque $\setminus$ set
- Masquer un signal ne veut pas dire l'ignorer.
- Si un signal masqué est généré, il reste pendant, sauf si le comportement par défaut est de l'ignorer.

Lister les signaux pendants

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Copie dans set la liste des signaux pendants.

C'est particulièrement utile si des signaux sont masqués (et non ignorés).

Changer le comportement : `signal`

On peut demander à exécuter une fonction (handler) en cas de réception d'un signal.

L'ancienne interface Unix (non POSIX) est la suivante. Donnée à titre indicatif (car plus simple à comprendre). Ne pas utiliser.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t
    handler);
```

`handler` est soit :

- SIG_IGN : ignore le signal
- SIG_DFL : action par défaut
- l'adresse d'une fonction prenant un entier
  - à la réception d'un signal, la fonction sera appelée, avec comme argument le numéro du signal.

Changer le comportement : signal

```
void handler(int i)
{
    printf("signal recu : %d\n",i);
}

int main()
{
    signal(SIGUSR1,handler);
    signal(SIGUSR2,handler);
    sleep(10000);
    return 0;
}
```

sigaction

L'interface à utiliser pour manipuler les handlers est `sigaction`

```
struct sigaction {
 void      (*sa_handler)(int);
 void      (*sa_sigaction)(int, siginfo_t *,
    void *);
 sigset_t    sa_mask;
 int         sa_flags;
};

int sigaction(int signum,
    const struct sigaction *act,
    struct sigaction *oldact);
```

sigaction

- sa_handler le handler (comme `signal`)
- sa_flags : options (void man)
- sa_mask : liste des signaux à masquer en plus, le temps que le handler s'exécute

- Si `act` n'est pas NULL : nouveau handler à installer
- Si `oldact` n'est pas NULL : l'ancien handler est copié dans la structure pointée

- On peut utiliser `sa_sigaction` à la place de `sa_handler` pour avoir un comportement plus fin (voir man).

Attente de signaux

```
#include <unistd.h>
int pause(void);

#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

- pause met le processus en pause, jusqu'à ce qu'un signal (n'importe lequel) arrive.
- Problème : un signal non masqué peut arriver avant l'appel à pause(), et être "perdu"...
- `sigsuspend` change temporairement le masque des signaux masqués, et attend jusqu'à ce qu'un signal arrive.

Signaux et appels systèmes

- Certains appels systèmes peuvent être interrompus par un signal.
  - Dans ce cas, l'appel système échoue, et le code retour (errno) sera EINTR
- Lors d'un fork, les signaux pendants ne sont pas hérités (le masque et les handlers, si)
- Lors d'un exec, les handlers ne sont pas hérités (le masque et les signaux pendants, si)

Suite :

- IPC
- Threads
- Réseau...