Process

Questions principales

Un processus effectue un fork(). Quel est le rôle de la table des interruptions, de la table des processus, de l'ordonnanceur, de l'OS dans ce cas ?

- Table des handles : La table des handles est copié du père au fils.
- Table des interruptions : Le fils hérite d'une copie des fonctions associées aux signaux du père. Cependant les signaux en attente sont évidemment supprimé. Il faut préciser que lors d'un appel à une fonction `exec`, les fonctions associées aux signaux du process sont remises par défaut sauf pour les signaux ignorés, ces derniers sont préservés.
- Table des processus : Dans la table des processus, l'entrée du père est copiée avec tout son contexte pour créer le processus fils.
- Ordonnanceur : Le fork produit une interruption, l'ordonnanceur va placer le processeur fils dans sa file puis choisir un nouveau processus à faire tourner.
- OS : L'OS est chargé de traiter l'interruption, de sauvegarder le contexte du processus, de traiter l'interruption puis de donner la main à l'ordonnanceur.

wait(), wait4() : Quelle est l'utiité ? Arguments ? Valeur de retour ?

```
pid_t wait(int * status);
pid_t waitpid(pid_t pid, int * status, int options);
pid_t wait3(int * status, int options, struct rusage * rusage);
pid_t wait4(pid_t pid, int * status, int options, struct rusage * rusage);
```

Toutes ces fonctions attendent que un ou plusieurs process changent d'état. C'est à dire :

- Un process enfant se termine
- Un process enfant est stoppé par un signal
- Un process enfant reprend son cours suite à un signal.

Dans le cas où le fils se termine, un wait permet au système de libérer les ressources associées au process fils. Si le wait n'est pas effectué, le process fils sera zombie (voir question zombie).

wait3 et wait4 sont comme waitpid mais retournent une structure `rusage` qui donne des information sur l'utilisation des ressources dans le process fils.

wait3() et wait4() sont marqués comme deprecated dans le manuel, il faut utiliser waitpid() à la place.

- wait attend qu'un de ses fils se termine.
- waitpid attend qu'un de ses fils spécifié par pid se termine.
  - si pid = -1 : attend n'importe quel fils.
  - si pid = 0 : attend n'importe quel fils ayant le même group ID que le père.
  - si pid > 0 : attend le fils dont son pid est égal au pid spécifié.
  - options :
    - WNOHANG : retourne immédiatement si aucun fils n'est fini.
  - status : si non NULL : stocke la valeur de retour du process, cette valeur de retour peut être consulté grâce à des macro (voir man 2 wait)
- wait3 attend pour tous les fils.
- wait4 attend pour un ou plusieurs fils spécifié.
- Si rusage n'est pas NULL, rusage contiendra des informations sur le fils.
- waitpid fait tout.
- Valeur de retour :
  - wait : PID du fils terminé, -1 si erreur.
  - waitpid : PID du fils dont l'état a changé. SI WNOHANG est spécifié et que plusieurs fils sont spécifié mais n'ont pas encore changé d'état, alors 0 est retourné. -1 si erreur.
  - wait3 et wait4 retourne comme waitpid.

Ces codes suivant sont égaux :

```
wait3(&status, options, rusage);
waitpid(-1, &status, options);
wait(&status);
```

Ainsi que ces deux codes ci :

```
wait4(pid, &status, options, rusage);
waitpid(pid, &status, options);
```

Wait(&status) appelle waitpid de cette manière :

```
waitpid(-1, &status, NULL);
```

Quelques champs utiles de rusage (man getrusage pour la liste complète) :

```
struct rusage
{
    struct timeval ru_utime; // user CPU time used
    struct timeval ru_stime; // system CPU time used
    long ru_nsignals; // signals received
    long ru majflt; // page fault
    long ru_minflt; // page reclaims;
    etc.
}
```

Exemple : attend tous les fils

```
while(wait(&status) > 0);
while(waitpid(-1, &status, 0) > 0);
```

Définir un zombie, son utilité, les problèms que cela génère, comment les créer et comment les détruire Un processus zombie est un processus terminé qui n'a pas été liberé par son parent via l'appel wait.

Lors de l'appel du System Call exit d'un fils, le père doit le "release" pour enlever son entrée dans la table des processus. Il est préférable d'attendre les processus fils le plus rapidement possible pour chaque fork.

pour en créer il suffit de faire un process vide :

```
if(pid_f=fork()==0);if(pid_f=fork()==0);if(pid_f=fork()==0);if(pid_f=fork()==0);
// JaumainStyle quadruplette de process zombie.
```

Pour les détruire, il est possible de les attendre à un moment choisi dans le code mais il se peut que un processus reste zombie durant ce laps de temps. Il est donc préférable de lancer une fonction de nettoyage au trigger SIGCHLD qui est lancé à la mort d'un fils.

```
void infanticide(int sg)
{
    if(sg = SIGCHLD) while(waitpid(-1, NULL, WNOHANG) > 0);
}

signal(SIGCHLD, infanticide);

if(fork() == 0);
```

execve(), execl, execv, execlp : Quelle est l'utilité ? Arguments ? Valeur de retour ? Quel est le rôle de la table des interruptions, de la table des processus, de l'ordonnanceur, de l'OS dans ce cas ?

```
execl(CHEMIN_COMMANDE, NOM_COMMANDE, argument, argument, ..., NULL);
```

```
execv(CHEMIN_COMMANDE, NOM_COMMANDE, arguments[argument, argument, ..., NULL]);
```

```
execlp(NOM_FICHIER_COMMANDE, argument, argument, ..., NULL);
execvp(NOM_FICHIER_COMMANDE, arguments[argument, argument, ..., NULL]);
```

Il est possible aussi d'utiliser execle, execve, execlpe, execvpe qui prennent un tableau de variables d'environnement en plus (accompagné de sa taille).

Cette appel système appel un logiciel externe en remplacement l'intégralité de son code, de ses données par celui appellé. Les arguments (varargs ou tableau) nécessite le nom de la commande en premier argument qui simulera le char * argv[] d'un processus principal. La table des handle est préservée, la table des interruptions est préservée. Le PID est préservé.

La valeur de retour est est `-1` et **peut être percu uniquement** lors d'une erreur puisque dans le cas contraire le code suivant exe* n'existe plus.

Questions secondaires

(fork) Quelle est la conséquence pour une variable x définie avant le fork() ? Que vaut &x ? Il s'agit d'une copie donc la variable x n'est pas dans le même segment de data que le père. Ce n'est pas la même adresse.

(fork) Quelle est la conséquence pour une lecture dans un ficheir qui a été ouvert avant le fork() ? Le fichier est lu correctement puisque la table des handle est copié, par contre le pere et le fils partage l'entrée du fichier dans la table des fichiers et ouvert et donc l'indice de position dans le fichier est ausis partagée.

(fork) fork() dans une boucle : Que se passe-t-il si la boucle est un for(i = 0; i < 3; i++) 1 process => 2 process => 4 process => 8 process mais de manière asynchrone, dans tous les cas, il y a 8 zombies à la fin de "toutes les boucles".