UNIVERSITÉ PARIS SUD
IUT D'ORSAY

DUT Informatique
M3101 Système S3
2015 / 2016

Travaux Dirigés n° 1 : Processus et signaux

Objectifs : savoir créer les processus Unix avec l'appel système `fork()` et les synchroniser avec le processus-père par `exit()` et `wait()`. Maîtriser la communication entre processus par signaux Unix.

1 Création et identification de processus

Dans les systèmes Unix (dont Linux et OS X), la création d'un nouveau processus se fait par la primitive `fork()` :

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

L'appel système `fork()` crée un nouveau processus, dit « fils », qui exécute le même code et a le même contenu de la mémoire que le processus appelant, dit « père ». Elle renvoie une valeur entière n qui indique :

— si $n > 0$, on est dans le processus père, n vaut l'identifiant de processus (PID) du fils,
— si $n = 0$, on est dans le processus fils,
— si $n = -1$, `fork()` a échoué, aucun nouveau processus n'a été créé.

Le processus créé est une copie presque exacte de son père. À part le code de programme et la mémoire, le processus fils « hérite » de son parent le répertoire courant, les fichiers ouverts, les variables d'environnement, les gestionnaires de signaux (*signal handlers*, voir ci-dessous), etc.

Dans la page `man 2 fork` vous trouverez la liste des propriétés que `fork()` ne passe pas du père au fils. La plus notable parmi elles est, bien sûr, l'identifiant unique de processus ou le PID. Le processus père récupère le PID du fils nouveau-né par la valeur de retour de `fork()`. Pour déterminer son propre PID ou le PID de son père, un processus peut se servir des primitives suivantes :

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

2 Terminaison de processus

Un processus peut terminer son exécution en appelant la primitive `exit()` :

```
#include <stdlib.h>
void exit(int status);
```

Le processus passe en argument à `exit()` son *code de retour* : un nombre entier entre 0 et 255. Dans le langage C, exécuter `return` n depuis la fonction `main()` est équivalent à l'appel `exit(`n`)`. Un processus peut être également terminé par un signal, envoyé par un autre processus ou par le système d'exploitation lui-même (on appelle ça la terminaison anormale).

Un processus père peut obtenir des informations sur la terminaison d'un fils par l'appel `wait()` :

```
#include <sys/wait.h>
pid_t wait(int * pointer_to_status);
```

À l'appel de `wait()`, le processus père s'endort dans l'attente de la mort d'un de ses fils. À la mort d'un fils, `wait()` renvoie le PID du fils mort : on ne sait pas par avance qui va mourir mais on sait après coup qui est mort. Dans le cas ou un ou plusieurs fils sont morts avant l'appel de `wait()`, l'appel retourne immédiatement. Dans le cas où il n'y a plus (ou pas) de fils à attendre, `wait()` renvoie -1. Si l'appelant passe en argument de `wait()` un pointeur non-NULL sur un entier, par exemple `&status`, le noyau placera à cette adresse la cause de la mort du fils avec le code de retour (en cas de sortie par `exit()`) et le numéro du signal qui a causé la mort (en cas de terminaison anormale).

Les macros `WIFEXITED(status)` et `WEXITSTATUS(status)` permettent, respectivement, de vérifier si le fils a terminé par `exit()` et d'extraire le code de retour. Les macros `WIFSIGNALED(status)` et `WTERMSIG(status)` permettent, respectivement, de vérifier si le fils a été tué par un signal et d'extraire son numéro. Voir la page `man 2 waitpid` pour une version plus générique de `wait()`.

Tout processus mort reste dans la table de processus du noyau en état « zombie », avant que son père (s'il est vivant) ne récupère les informations sur sa terminaison par `wait()` ou `waitpid()`. Si le processus père termine avant ses fils, les processus orphelins sont « adoptés » par le processus 1, `init`, qui fait les appels nécessaires de `wait()` automatiquement.

3 Signaux

Un signal est un mécanisme permettant d'informer un processus que quelque chose s'est produit dans le système. Les signaux Unix ont des origines diverses, ils peuvent être retransmis par le noyau (division par zéro, overflow, instruction interdite etc.), envoyés depuis le clavier par l'utilisateur (touches `<Ctrl-Z>`, `<Ctrl-C>`...), émis par la commande `kill` depuis le shell ou par la primitive `kill()` dans un programme C/C++.

La primitive `kill()` prend en argument le PID du processus destinataire du signal et le numéro du signal à envoyer. Sa valeur de retour est 0 en cas de succès ou -1 en cas d'échec. Son prototype est le suivant :

```
#include <signal.h>
int kill(pid_t pid_du_destinataire, int nom_ou_numero_du_signal);
```

Dans la table suivante vous trouverez quelques signaux souvent utilisés. Pour chaque signal, on indique son nom (que vous pouvez passer en argument de `kill()`), son numéro sur l'architecture x86, ainsi que le comportement par défaut à la réception de ce signal.

Nom	N°	Par défaut	Commentaire
`SIGHUP`	1	terminaison	déconnexion du terminal gérant
`SIGINT`	2	terminaison	interruption par l'utilisateur (touche `<Ctrl-C>`)
`SIGKILL`	9	terminaison	terminaison immédiate, ne peut pas être capturé ou ignoré
`SIGSEGV`	11	terminaison	accès à l'adresse mémoire invalide (*segmentation fault*)
`SIGPIPE`	13	terminaison	écriture dans un tube (*pipe*) sans lecteur
`SIGALRM`	14	terminaison	temporisation mise par `alarm()` écoulée
`SIGTERM`	15	terminaison	terminaison immédiate
`SIGUSR1`	10	terminaison	signal utilisateur
`SIGUSR2`	12	terminaison	signal utilisateur
`SIGCHLD`	17	ignoré	processus fils terminé ou arrêté
`SIGCONT`	18	continuation	continuer l'exécution si arrêté
`SIGSTOP`	19	arrêt	arrêt immédiat, ne peut pas être capturé ou ignoré
`SIGTSTP`	20	arrêt	arrêt par l'utilisateur (touche `<Ctrl-Z>`)

Un processus peut choisir d'accepter le comportement par défaut à la réception d'un signal, d'ignorer ou de redéfinir le traitement du signal. Cela se fait en utilisant la primitive `signal()` :

```
#include <signal.h>
int signal(int nom_ou_numero_du_signal, sighandler_t gestionnaire);
```

Le paramètre `gestionnaire` peut être
— la macro `SIG_IGN` pour ignorer le signal,
— la macro `SIG_DFL` pour restaurer le comportement par défaut,
— le nom de la fonction C à appeler en cas de réception du signal. Cette fonction doit prendre un seul argument de type **`int`** : le numéro du signal qui l'aura appelée.

Remarque : sur certaines versions d'Unix (dont AIX, mais pas Linux), après exécution de la fonction spécifique définie par l'utilisateur, le comportement par défaut est rétabli. Si on veut conserver le comportement spécifique il faut rappeler `signal` dans la fonction spécifique. Voir la page `man 2 sigaction` pour une version plus générique et plus portable de `signal`.

Si l'utilisateur installe une fonction C comme gestionnaire d'un signal (3e cas d'utilisation de `signal()`), alors cette fonction sera immédiatement exécutée à la réception de ce signal. Une fois le gestionnaire terminé, l'exécution du programme reprend de l'endroit où elle a été interrompue. Si pendant l'exécution de la fonction gestionnaire pour un signal, ce même signal arrive pour la 2e (3e, 4e...) fois, la fonction gestionnaire *n'est pas interrompue.* Tout signal est considéré comme *bloqué* pendant l'exécution de son gestionnaire, et le noyau va attendre que le gestionnaire retourne pour le rappeler juste après. Notez bien que le système retient juste le fait qu'un signal bloqué est reçu, mais pas le nombre de fois qu'il est reçu. Ainsi, même si le signal arrive cent fois pendant qu'il est bloqué, le noyau va rappeler le gestionnaire une seule fois. Voir la page `man 7 signal` pour une description détaillée de ce mécanisme.

Pour avertir le processus père de la mort d'un fils, le système lui envoie le signal `SIGCHLD`. Ainsi, une façon d'éviter l'apparition de zombies est d'installer un gestionnaire de `SIGCHLD` qui appelle `wait()` dès que le signal est reçu. Une autre façon consiste à appeler `signal(SIGCHLD,SIG_IGN);` dans le processus père : cela empêche complètement la création de zombies. Dans ce cas, un appel à `wait()` met le processus en attente jusqu'à la terminaison de tous ses fils et ensuite renvoie -1.

4 Exercices

Exercice 1 : Qui suis-je ? Considérons le programme suivant (les lignes **`#include`** sont omises).

```
1  int main () {
2    pid_t child = fork();
3    if (child == -1) { perror("fork() error"); exit(1); }
4  //printf("My PID is %d.\n", getpid());  // à ajouter pour question (b)
5    if (child == 0) {
6      printf("Child process: my PID is %d.\n", getpid());
7      exit(0);                          // à enlever pour question (b)
8    }
9    printf("Now my PID is %d.\n", getpid());
10   exit(0);
11 }
```

Question (a) : Donnez les affichages obtenus sur la sortie standard à l'exécution.

Question (b) : Même question en ajoutant la ligne 4 et en enlevant la ligne 7.

Exercice 2 : Un long dimanche de calcul. Il était une fois Bob, un jeune programmeur audacieux qui voulut calculer l'âge du capitaine. Pour cela il lui fallut faire la somme de deux nombres entiers : le nombre de beaux jours à Londres à l'an de grâce 1605 (entre 0 et 127) et le nombre d'étudiants en 2e année de DUT Info qui viendront à l'IUT le 16/05 (entre 0 et 127). Comme chacun des deux nombres demandait environ 6 heures de calcul, Bob décida de paralléliser le travail et écrivit le programme suivant (les lignes **#include** et le traitement d'erreurs sont omises).

```
int sunny_days_in_London(int year)       { /* 6 heures de calcul */ }
int second_year_presence(int day) { /* encore 6 heures de calcul */ }

int main () {
  int captains_age = 0;
  pid_t cpid = fork();
  if (cpid != 0) captains_age += sunny_days_in_London(1605);
  else           captains_age += second_year_presence(1605);
  printf("Le capitaine a %d ans et mon PID est %d\n", captains_age, (int) getpid());
  return 0;
}
```

Expliquez l'erreur de Bob et donnez l'algorithme d'une solution au problème en utilisant les primitives `exit()` et `wait()`. Détaillez les actions du père et du fils.

Combien de temps durera l'exécution de votre programme ?

Exercice 3 : `fork()` d'artifice. Lorsqu'on exécute le programme ci-dessous, combien y a-t-il de processus créés ?

```
int main(int argc, char ** argv) {
  pid_t id;
  int i, N = 0;
  if (argc > 1) N = atoi(argv[1]);
  for (i = 0; i < N; i++) {
    id = fork();
    if (id == -1) { perror("fork() error"); exit(1); }
    printf("I am %d, son of %d.\n", getpid(), getppid());
  }
  printf("%d out.\n", getpid());
  return 0;
}
```

Donner la formule générale permettant de trouver le nombre de processus en fonction de `N`.

Exercice 4 : Pause café. La primitive **`int`** `pause(`**`void`**`)` met le processus en état d'attente jusqu'à la réception d'un signal qui termine le processus ou provoque l'exécution d'une fonction gestionnaire. Soit le programme

```
int main() {
  printf("PID %d waiting for a signal.\n", getpid());
  pause();
  printf("PID %d out.\n", getpid());
  return 0;
}
```

Le processus correspondant a pour numéro 12345.

Que se passe-t-il si on tape la commande `kill -USR1 12345` dans un autre terminal ?

Comment faire pour que le message « `PID 12345 out` » s'affiche (donnez le code à ajouter).

Exercice 5 : world ! Hello, Considérons le programme à trou suivant :

```
int main () {
  pid_t id = fork();
  if (id == 0) {
    printf("Hello, ");
    exit(0);
  }
                    // <-- TROU
  printf("world!");
  return 0;
}
```

On souhaite compléter le programme afin d'assurer que les mots sont toujours affichés dans le bon ordre, indépendamment des décisions de l'ordonnanceur.

La primitive `int sleep(int n)` met le processus en état d'attente pendant `n` secondes. Est-ce que l'on peut compléter le programme par un appel de `sleep(1)` ? `sleep(100)` ? Justifiez la réponse.

Est-ce que l'on peut compléter le programme par un appel de `pause()` ? Justifiez la réponse.

Est-ce que l'on peut se servir de `pause()` en installant un gestionnaire tout simple pour `SIGCHLD`, comme on l'a fait dans l'exercice 4 pour `SIGUSR1` ?

Est-ce que l'on peut compléter le programme par un appel de `wait()` ? Justifiez la réponse.

Exercice 6 : Et mon courroux, coucou ! Bob n'est pas satisfait par nos réponses à l'exercice précédent. Et si on se servait du gestionnaire pour empêcher de lancer `pause()` dans le cas où le signal arrive tôt ? Introduisons pour cela un drapeau global :

```
int flag = 0;
void handler (int sig) { flag = 1; }
int main () {
  signal(SIGCHLD, handler);
  if (fork() == 0) { printf("Hello"); exit(0); }
  while (!flag) pause();
  printf(" world");
  return 0;
}
```

Pourquoi est-t-il important d'installer le gestionnaire avant d'appeler `fork()` ?

Pourquoi est-t-il important de tester le drapeau dans une boucle ?

Est-ce que le programme de Bob assure le bon ordre d'affichage ?

Exercice 7 : Un, deux, beaucoup. Soit le programme suivant

```
int count = 0;
void handler(int sig) { count++; }
int main() {
  int i;
  signal(SIGUSR1, handler);
  if (fork() == 0) {
    for (i = 0; i < 256; i++)
      kill(getppid(), SIGUSR1);
    exit(0);
  }
  wait(NULL);
  printf("Final: %d\n", count);
  return 0;
}
```

Qu'est-ce que ce programme affiche ? Justifiez la réponse.

Exercice 8 : Jusqu'à l'infini en huit secondes. La primitive `alarm()`

```
#include <unistd.h>
unsigned int alarm(unsigned int nb_sec);
```

programme un « réveil » qui enverra un signal `SIGALRM` au processus appelant dans `nb_sec` secondes. Tout appel de `alarm()` annule et remplace l'éventuelle alarme précédente. L'appel `alarm(0)` annule toute alarme précédente et ne programme pas de nouvelle alarme.

Écrivez un programme qui incrémente en boucle un compteur de type **`unsigned int`** et affiche la valeur courante du compteur toutes les secondes. Pour obtenir la valeur maximale du type **`unsigned int`** vous pouvez utiliser la constante `UINT_MAX` déclarée dans `limits.h`, ou bien écrire `((unsigned int) (-1))`.