Chapitre 3

Les processus : création et destruction

Dans le premier chapitre, nous avons vu ce qui se passe au démarrage quand s'installe sur l'ordinateur un super-programme : le *noyau* (*kernel*) du système d'exploitation. Un des rôles du kernel est de coordonner l'action des autres programmes qui tournent sur l'ordinateur, en les isolant les uns des autres tout en leur permettant de communiquer entre eux d'une façon contrôlée.

Une fois le chapitre assimilé, vous aurez une idée plus précise de ce que sont un thread et un processus et vous serez en mesure d'écrire des programmes C qui en fabriquent autant que de besoin.

Attention, c'est *le* chapitre crucial du cours.

3.1 Le processus : définitions

Il y a plusieurs façons de définir ce qu'est un processus dans le contexte d'un système d'exploitation. Les deux définitions les plus importantes sont du point de vue de l'utilisateur et du point du vue du système d'exploitation.

3.1.1 Processus = un programme qui tourne

Du point de vue de l'utilisateur, un processus est un programme qui tourne.

Un programme, c'est un ensemble d'instructions et de données pour effectuer une tâche ; la tâche peut être simple, comme dans les commandes `sleep` ou `echo`, ou plus complexe comme dans les commandes `emacs` ou `gcc`.

Si on prend un programme simple, comme `sleep`, il est évident qu'on peut lancer ce programme plusieurs fois. On peut même le lancer plusieurs fois en même temps comme dans

```
$ sleep 10 & sleep 15 & sleep 20 &
```

Ici le programme `sleep` a été lancé trois fois, avec trois arguments différents : il est donc en train de s'exécuter trois fois dans le système. Ce sont ces exécutions qu'on appelle des processus.

La définition n'est pas complètement satisfaisante parce qu'un programme complexe, comme `gcc` va en fait lancer l'exécution de plusieurs processus qui

sont chargés chacun d'une partie du travail. C'est néanmoins souvent une bonne façon de voir un processus.

3.1.2 Processus = entité à laquelle le système attribue les ressources

Le système d'exploitation est chargé, entre autres, d'arbitrer entre les différents programmes qui tournent sur la machine pour leur attribuer les ressources disponibles. Les ressources peuvent être de la mémoire centrale, de l'espace disque, du temps d'exécution sur le processeur, etc.

Du point de vue du système, l'unité qui utilise des ressources est le *processus*. Le système gardera en mémoire les ressources utilisées par chaque processus et utilisera cette information pour décider de l'attribution des ressources futures.

Cette définition n'est pas complètement satisfaisante parce qu'il existe une entité plus petite que le processus, qu'on appelle un *thread* ou *processus léger* (*lightweight process*) à laquelle le système attribue également des ressources.

3.1.3 Processus, tâche ou job

Le processus est un concept qu'on rencontre dans (presque) tous les systèmes d'exploitation mais il porte des noms variés. Dans certains systèmes, ce qu'on appelle un processus sous Unix s'appelle un *job* (attention, sous Unix les jobs sont autre chose, comme on l'a vu dans le chapitre sur le shell) ; dans d'autres cela s'appelle une *tâche* (*task* en anglais). Dans les sources du noyau Linux, la structure principale utilisée pour représenter un processus porte le nom `task_struct`.

3.1.4 Qu'est-ce qui caractérise un processus ?

Cherchons à énumérer ce que nous pouvons deviner de ce qui caractérise un processus. Une autre façon de poser cette question est : *étant donné un processus, de quelles informations ai-je besoin à un moment précis pour être en mesure de le recréer à l'identique s'il venait à disparaître ?*

Parce qu'un processus est un programme en cours d'exécution, il faut bien sur savoir de quel programme il s'agit ; il faut donc connaître les instructions qui le composent ; soulignons que les instructions sont fixées une fois pour toutes (lors de la compilation) et qu'elles ne seront jamais modifiées : plusieurs processus qui exécutent le même programme ont les mêmes instructions.

Il faut aussi connaître ses données, qui vont probablement changer de valeurs au cours de l'exécution. A mesure que le programme s'exécute, de nouvelles données peuvent même apparaître, parce qu'on a utilisé l'opérateur `new` en C++ ou la fonction `malloc` en C, ou parce qu'on a appelé des fonctions qui ont utilisé la pile pour sauver des valeurs. Les données d'un processus lui sont propres, plusieurs processus qui exécutent le même programme ne peuvent pas partager leurs données. (En fait, pour économiser la mémoire, on s'efforce de distinguer les données qui ne seront jamais modifiées pour pouvoir les partager entre processus ; c'est pour cette raison que gcc interdit d'écrire à l'intérieur des chaînes de caractères si on n'a pas compilé avec l'option `-fwritable-string`.)

Parce que le programme est en cours d'exécution, il est aussi nécessaire de connaître l'adresse de l'instruction courante, généralement stockée dans un re-

gistre qu'on appelle `PC` comme *Program Counter* (parfois appelé *pointeur ordinal* en français).

Il y a d'autres choses également, comme le répertoire de travail (le *working directory* affiché par la commande `pwd`). On a vu dans le chapitre sur le shell que chaque processus possède un *environnement*. Il a aussi des fichiers ouverts, au moins pour lire, afficher les messages ordinaires et afficher les erreurs : ces descripteurs de fichiers peuvent varier d'un processus à l'autre, comme on l'a vu avec les redirections d'entrée sortie. En pratique, les connexions réseau que le processus a pu ouvrir se comportent, du point de vue du système, à peu près comme des descripteurs de fichiers.

Il y a beaucoup d'autres caractéristiques qu'il est nécessaire de connaître pour tout savoir sur un processus ; nous en verrons un certain nombre dans la suite du cours ; la chose importante à retenir pour le moment : un processus est une chose assez complexe.

Le PID

Pour nommer les processus, le système (Unix ou Linux) leur colle un numéro d'identité nommé PID comme *Process ID*. Le premier processus au démarrage du système reçoit le PID numéro 1, le second le PID 2 et ainsi de suite. On verra plus loin dans ce chapitre ce qui se passe quand on a atteint le PID maximum.

3.1.5 Les commandes `ps` et `top`

La commande `ps` (comme *process status*) permet d'examiner certaines caractéristiques des processus. Un exemple de sortie :

```
 PID TTY          TIME CMD
4937 pts/1    00:00:00 bash
4976 pts/1    00:00:09 emacs
5123 pts/1    00:00:00 ps
```

La commande `ps` décrit chaque processus par une ligne. La colonne `PID` contient son numéro, la colonne `TTY` la fenêtre depuis laquelle il a été lancé, la colonne `TIME` le temps pendant lequel il a calculé (en minutes, secondes et centièmes de secondes) et la colonne `CMD` la commande qu'il est train d'exécuter.

Pour avoir des informations plus détaillées, on peut utiliser `ps -l`. Nous reviendrons, au long du cours, sur le rôle de la plupart des colonnes.

Par défaut, `ps` n'informe que sur les processus lancés depuis la même fenêtre. Pour avoir des informations sur *tous* les processus, il faut utiliser `ps ax`.

La commande `top` fonctionne presque comme la commande `ps` mais elle met à jour les informations sur les processus à intervalles réguliers (toutes les trois secondes par défaut) et elle place en tête de liste les processus actifs. Quand un ordinateur est lent, une des première chose à faire est de regarder avec `top` s'il y a des processus qui font de nombreux calculs.

Exercice 3.1 — Compiler le programme suivant, qui calcule beaucoup (des calculs sans intérêt), le lancer et regarder l'état des processus avec `top` pendant qu'il tourne. Le lancer plusieurs fois en même temps. Combien de fois faut-il lancer le programme en même temps pour commencer à en ressentir les effets ?

```
# include <stdio.h>
```

```
int
main(){
      int x;

      for(;;)
              x += 1;
      return 0;
}
```

3.2 La mort d'un processus

Commençons par examiner la fin des processus, qui est plus facile que la création.

Pour se terminer, un processus demande au noyau du système de le supprimer avec un appel système spécifique, qui s'appelle `exit`. (En réalité, la fonction `exit` fait le ménage avant de demander au noyau de terminer le processus. L'appel système lui-même s'appelle `_exit`.)

Il y a un paramètre à `exit`, qui est le compte-rendu de l'exécution du programme. Par convention, un processus pour qui tout s'est bien passé renvoie la valeur 0 ; n'importe quelle autre valeur signale un problème. C'est facile de s'en souvenir parce que c'est le contraire de ce qui se passe en C.

Un programme C qui ne fait que sortir :

```
int main(){ exit(0); }
```

Compilez et testez le programme, constatez qu'il n'imprime pas la valeur de sortie. On peut récupérer cette valeur dans la variable shell `$?`.

Ce programme existe déjà, sous le nom de `true`. On l'utilise pour la programmation shell quand on veut faire une boucle infinie. Il y a un programme symétrique nommé `false`, qui ne fait rien mais s'arrête avec un compte-rendu différent de 0.

```
$ true
$ echo $?
0
$ false
$ echo $?
1
$ gcc correct.c
$ echo $?
0
$ touch incorrect.c
$ gcc incorrect.c
(.text+0x18): undefined reference to ‘main’
collect2: ld returned 1 exit status
$ echo $?
1
```

Exercice 3.2 — Le programme suivant se contente de sortir avec le compte rendu d'exécution qu'on lui a passé sur la ligne de commande :

```
/* ca-sortir.c
 Un programme qui s'arrete avec la valeur fournie en argument
*/
# include <stdio.h>
# include <stdlib.h>

int
main(int ac, char * av[]){
        if (ac != 2){
                fprintf(stderr, "usage: %s N\n", av[0]);
                return 1;
        }
        return atoi(av[1]);
}
```

Le compiler sous le nom `sortir`. L'appeler avec des valeurs diverses et consulter la valeur de la variable `$?`. En conclure le type de données dans lequel la valeur de sortie est stockée. Exemples d'appel :

```
$ sortir 0 ; echo $?
0
$ sortir 23 ; echo $?
23
$ sortir 1234 ; echo $?
210
```

Regarder notamment les valeurs quand on l'appelle avec `sortir 1024` et `sortir 1025`.

Exercice 3.3 — (facile)
Que vaut la valeur de sortie d'un processus interrompu avec CTRL C (par exemple interrompre la commande `sleep 60` et consulter la valeur de `$?`). Et si on l'interrompt avec Contrôle-\ ?

Exercice 3.4 — (plus difficile)
Que vaut la valeur de sortie d'un processus interrompu parce qu'il tente de lire le contenu de l'adresse 0 ? Attention, ici il faut que vous fassiez un programme C parce que les programmes ordinaires se gardent bien de lire cette adresse.

Exercice 3.5 — (encore plus difficile) Que vaut la valeur de sortie pour un programme qui essaye d'effectuer une division par 0 ? Attention, si on fait un programme élémentaire comme `int main(){ 23 / 0; }` gcc va détecter la division par 0 (et donner un warning) et la retirer du programme.

3.3 La création d'un nouveau processus 1 : `fork`

On pourrait imaginer d'avoir un appel système qui servirait à créer un processus. Un problème c'est que cet appel système devrait avoir autant d'arguments qu'un processus a de caractéristiques (et elles sont nombreuses) : il faudrait spécifier le programme à lancer, le répertoire de travail, les fichiers à ouvrir, la fenêtre à utiliser etc. Il parait que la fonction de création de processus sous Windows a quinze paramètres : c'est beaucoup trop pour qu'on l'utilise facilement.

Sous Unix, la création de processus se fait en deux temps, avec deux appels système plutôt simples : fork et exec. Commençons par examiner `fork`.

L'appel système `fork` est tellement simple qu'il en est étrange : quand un processus appelle `fork`, le système en fait une copie intégrale, avec le même code, les mêmes données, le même point d'exécution, les mêmes fichiers ouverts etc. Seules deux choses changent dans la copie : le numéro du processus (le PID) est différent et dans la copie, `fork` renvoie 0 alors que dans l'original il renvoie le PID de la copie.

On parle souvent de processus *parent* pour l'original et de processus *enfant* pour la copie.

L'exemple le plus simple de fork :

```
/* cb-fork-simple.c
 Un fork simple
 */
# include <stdio.h>
# include <unistd.h>

int
main(){
  if (fork())
    printf("Vrai\n");
  else
    printf("Faux\n");
  return 0; /* ou return EXIT_SUCCESS; */
}
```

Quand le processus appelle `fork`, le système le copie dans un processus enfant. Dans le parent, `fork` renvoie le PID du processus enfant et il imprime `Vrai`. Dans l'enfant, `fork` renvoie 0 et il imprime `Faux`.

Dans un vrai programme, il faut prendre en compte le fait que l'appel système `fork` peut échouer. Dans ce cas, il renvoie une valeur inférieure à 0. Pour cette raison, la manière ordinaire d'utiliser `fork` est la suivante :

```
int
main(){
  int t;

  ...
  t = fork();
  if (t == -1){              // erreur
     traitement d'erreur
  } else if (fork() == 0){ // enfant
     traitements spécifiques de l'enfant
  } else {                 // parent
     traitement spécifiques du parent
  }
  return 0;
}
```

Exercice 3.6 — Qu'imprime le programme suivant et pourquoi ?

```
/* cc-fork-etoile.c
 Fork et printf : les noeuds d'un arbre binaire
*/
# include <stdio.h>
# include <unistd.h>

int
main(){
  int i;

  setbuf(stdout, NULL);
  for(i = 0; i < 10; i++){
    printf("%d\n", i);
    fork();
  }
  return 0;
}
```

3.3.1 La bombe fork

Le programme suivant est simple mais dangereux :

```
int
main(){
    for(;;)
        fork();
}
```

Quand on le lance, il crée un processus puis le parent et l'enfant créent au tour de boucle suivant un autre processus ; au tour suivant on a huit processus, puis seize, puis trente deux, etc. Très rapidement, l'ordinateur ne va plus faire autre chose qu'essayer de créer des processus et toutes les autres opérations seront ralenties de façon significative. On peut essayer de supprimer les processus mais ils sont créés avec fork plus vite qu'on ne les détruit.

En pratique, la solution la plus aisée consiste à redémarrer l'ordinateur. On appelle ce programme la *bombe fork*.

On peut essayer d'éviter le problème avec l'appel système `ulimit` qui permet de placer des limites aux ressources qu'un utilisateur peut obtenir. On peut notamment limiter le nombre de processus créés. En général, la limite n'est pas définie par défaut.

3.4 Attendre la fin d'un processus enfant : `wait`

Il existe un appel système `wait` qui permet à un processus parent d'attendre la fin d'un processus enfant et récupérer alors la valeur avec laquelle celui-ci s'est terminé (l'argument qu'il a fourni à `exit`).

On passe à `wait` l'adresse d'un entier où le compte-rendu sera placé et il renvoie le PID du processus qui s'est terminé, ce qui est utile pour les parents qui ont plusieurs enfants.

Avec `wait`, on peut écrire un programme vraisemblable, même s'il ne fait rien d'utile, avec la création d'un processus.

```
/* cd-fork-wait.c
 lancer un processus, attendre qu'il se termine
 */
# include <stdio.h>
# include <unistd.h>
# include <stdlib.h>

int
main(){
  int t, pid, etat;

  printf("Je suis le processus de depart\n");
  t = fork();
  if (t == -1){
    perror("fork");
    return 1;
  } else if (t == 0){
    printf("enfant: j'attend 5 secondes\n");
    sleep(5);
    printf("enfant: je termine\n");
    return 0;
  } else {
    for(;;){
      pid = wait(&etat);
      if (pid == t)
        break;
      printf("parent: wait a rendu la valeur %d, j'attend encore\n", pid);
      if (pid == -1)
        perror("wait");
    }
    printf("parent: l'enfant s'est termine avec la valeur %d\n", etat);
    printf("parent: je termine\n");
    return 0;
  }
}
```

Notez le traitement en cas d'erreur : on appelle la fonction `perror` qui imprime le message standard qui détaille la raison pour laquelle `fork` n'a pas fonctionné. Sur un système bien configuré, ce message sera dans la langue choisie par l'utilisateur.

(Pour faire échouer le fork et voir le message d'erreur, on peut utiliser la commande `ulimit` qui permet de limiter les ressources, notamment le nombre de processus qu'un utilisateur peut lancer. Pour ce faire, il faut commencer par compter le nombre de processus qu'on a déjà lancés, par exemple avec `ps -u nom-de-login | wc`. Si cette commande a compté 19 lignes, c'est qu'on a 16 processus actifs (puisqu'il faut retirer la première ligne de la sortie de `ps` qui nomme les colonnes et les deux processus `ps` et `wc`). On va donc limiter le nombre de processus à 18 avec `ulimit -u 18`. Maintenant, quand le shell lance

la commande `a.out`, cela fonctionne mais quand notre programme tente de créer un nouveau processus, cela échoue puisqu'on a atteint le nombre maximum de processus.)

Exercice 3.7 — (facile) Que se passe-t-il quand un processus fait un `wait` alors qu'il n'a pas de processus enfant ?

Exercice 3.8 — (moyen) wait attend-il la fin (1) d'un processus enfant direct ou bien (2) d'un processus enfant ou d'un descendant d'un processus enfant ? Pour trouver la réponse, le plus simple est de faire une expérience. Écrire un programme P1 qui crée un processus P2 et attend la fin d'un enfant avec wait. L'enfant P2 crée un petit enfant P3 et attend 20 secondes. Le petit enfant attend 10 secondes. Quand on lancera le programme, s'il se termine au bout de 10 secondes, c'est que wait attend la fin d'un enfant *ou* d'un petit enfant. S'il se termine au bout de 20 secondes, c'est qu'il attend la fin de l'enfant. On peut connaître le temps qui s'écoule entre le lancement d'une commande et sa fin avec la commande `time`.

3.4.1 Le PID maximum

On a vu dans le chapitre sur le shell que le système attribue un PID à chaque processus en incrémentant un compteur. Une question à se poser est ce qui se passe quand ce compteur atteint la valeur maximum.

Plutôt que de trouver (difficilement) cette réponse dans la documentation ou en lisant les sources du noyau, on peut écrire un programme expérimental, qui va créer des processus jusqu'à ce que le PID du nouveau processus soit plus petit que celui du précédent : cela signifiera probablement que le précédent avait atteint le PID maximum.

Le programme suivant est une manière simple (et fausse) de faire :

```
/* ce-pidmax1.c
 Trouver le PID maximum (programme faux)
 */
# include <stdio.h>
# include <unistd.h>

int
main(){
  int oldpid = 0;
  int newpid;

  for(;;){
    newpid = fork();
    if (newpid == 0){
      // enfant : ne rien faire
      return 0;
    }
    // parent
    if (newpid < oldpid){
      printf("Le PID max semble etre %d\n", oldpid);
      return 0;
    }
```

```
    oldpid = newpid;
  }
}
```

Quand on l'exécute, il donne des résultats qui ne semblent pas cohérents :

```
$ gcc -g -Wall ce-pidmax1.c
$ a.out
Le PID max semble être 8880
$ a.out
Le PID max semble être 12636
$ a.out
Le PID max semble être 16392
```

Une rapide inspection du code met à jour qu'on a oublié que l'appel système `fork` peut échouer. Ajoutons un test dans le parent pour ce cas juste après l'appel à fork :

```
if (newpid < 0){
  perror("fork");
   return 1;
}
```

Maintenant, quand on exécute le programme, on est prévenu si la création du nouveau processus échoue. Effectivement :

```
$ a.out
fork: Resource temporarily unavailable
```

On peut considérer ceci comme une piqûre de rappel : *il faut toujours tester les valeurs renvoyées par les appels système pour détecter et traiter les erreurs.*

Ce qui provoque sans doute l'erreur : le processus parent crée trop de processus enfants avant que ceux-ci aient le temps de se terminer. Pour résoudre le problème, il suffira alors que le parent attende la fin de l'enfant, comme dans le programme (correct) suivant :

```
/* ce-pidmax3.c
 Trouver le pid maximum (programme corrige, avec wait)
 */
# include <stdio.h>
# include <unistd.h>
# include <assert.h>

int
main(){
  int oldpid = 0;
  int newpid;
  int t, etat;

  for(;;){
    newpid = fork();
    if (newpid == 0){
      // enfant : ne rien faire
```

```
      return 0;
    }
    // parent
    if (newpid < 0){
      perror("fork");
      return 1;
    }
    t = wait(&etat);
    assert(t >= 0);
    if (newpid < oldpid){
      printf("Le PID max semble etre %d\n", oldpid);
      return 0;
    }
    oldpid = newpid;
  }
}
```

Ici, j'utilise `assert` qui permet de vérifier simplement que la condition est remplie. Si ce n'est pas le cas, le programme sera interrompu violemment avec un message d'erreur. Cette fonction est pratique pour vérifier que les cas « *qui ne peuvent pas se produire* » ne se produisent effectivement pas.

Maintenant, quand on lance ce programme, on obtient toujours la même réponse :

```
$ a.out
Le PID max semble être 32767
$ a.out
Le PID max semble être 32767
```

Cette valeur de 32767 semble assez particulière pour faire confiance à ce résultat : il s'agit de $2^{15} - 1$, la plus grande valeur positive qu'on peut stocker dans un entier sur deux octets.

Le même programme tournant sur d'autres systèmes Unix (FreeBSD et Solaris) me donne des résultats différents.

3.4.2 Le temps nécessaire pour créer un processus

De quel ordre de grandeur est le temps nécessaire pour créer un processus ? On peut utiliser la programme précédent comme base pour le savoir : pour avoir un temps mesurable, créons par exemple 100 000 processus ; divisons le temps que cela prend par 100 000 et nous obtiendrons une bonne approximation du temps pour créer un processus.

Le programme modifié est le suivant :

```
/* cf-forktime.c
 Combien de temps pour un fork ?
 */
# include <stdio.h>
# include <unistd.h>
# include <assert.h>

enum {
```

```
  Nfois = 100 * 1000,
};

int
main(){
  int i, t, etat, newpid;

  for(i = 0; i < Nfois; i++){
    newpid = fork();
    if (newpid == 0){
      // enfant : ne rien faire
      return 0;
    }
    // parent
    assert(newpid > 0);
    t = wait(&etat);
    assert(t == newpid);
  }
  return 0;
}
```

J'ai ici utilisé `assert` pour vérifier que fork ne produit pas d'erreurs. J'ai aussi défini la constante `Nfois` en utilisant un `enum`, parce que je cela trouve plus lisible qu'un `#define`.

Pour mesurer le temps nécessaire, pas besoin de sortir son chronomètre : il existe une commande `time` qui lance une commande, attend qu'elle se termine puis affiche le temps qu'elle a pris :

```
$ gcc -g -Wall cf-forktime.c
$ time a.out

real 0m16.352s
user 0m1.408s
sys 0m14.789s
$ time a.out

real 0m14.012s
user 0m1.216s
sys 0m12.737s
$ time a.out

real 0m16.056s
user 0m1.216s
sys 0m14.749s
```

La commande `time` affiche trois temps : le temps `real` est le temps qu'aurait mesuré un chronomètre entre le lancement de la commande et sa fin ; il peut varier largement pour une même commande en fonction de la charge de l'ordinateur. Le temps `user` est celui pendant lequel le processus a effectué des calculs. Le temps `sys` est celui pendant lequel le noyau du système a traité les appels système du processus. Notons que les valeurs sont susceptibles de varier d'un

appel à l'autre, en fonction de ce qui se passe sur l'ordinateur (ce ne sont que des statistiques) ; en revanche, l'ordre de grandeur reste constant. Il faut donc, sur mon ordinateur, environ 0,15 millisecondes pour créer et terminer un processus. Une autre façon de voir, c'est qu'on peut y créer environ 6500 processus par seconde.

Exercice 3.9 — (facile) Sur votre ordinateur, quels sont les résultats ?

3.5 La création d'un nouveau processus 2 : `exec`

L'appel système `fork` permet de créer un nouveau processus mais celui-ci n'est qu'une copie presque à l'identique de l'ancien. Il est donc complété par un second appel système qui change le programme exécuté par un processus en laissant toutes ses autres caractéristiques inchangées : il s'agit de l'appel système `exec`.

On donne à l'appel système `exec` deux arguments (au moins) qui sont d'une part le fichier qui contient le programme et d'autre part les arguments tels qu'ils apparaîtront dans `argc` et `argv`, les deux paramètres de la fonction `main`. Le système va modifier le processus pour qu'il commence à exécuter le programme contenu dans le fichier depuis le début mais (presque) tous les autres paramètres du processus comme son PID, son répertoire courant, les fichiers qu'il a ouvert, etc. restent identiques.

Il existe plusieurs fonctions C qui permettent d'utiliser l'appel système `exec` ; les deux plus importantes sont `execl` et `execv`. Voici un programme simple qui utilise `execl` :

```
/* cg-exec-simple.c
 Un execl tout simple
 */
# include <stdio.h>
# include <unistd.h>

int
main(){
  int t;

  t = execl("/bin/echo", "commande", "arg1", "arg2", NULL);
  if (t < 0){
    perror("execl");
    return 1;
  } else {
    printf("l'execl a fonctionne, ce message ne sera jamas affiche\n");
    return 0;
  }
}
```

On peut le compiler et l'exécuter :

```
$ gcc -g -Wall cg-exec-simple.c
$ a.out
arg1 arg2
$
```

On voit dans la sortie que la commande `echo` a bien été exécutée ; en revanche,le message annonçant la réussite n'apparaît pas. C'est normal : puisque l'`exec` a fonctionné, le processus n'est plus en train d'exécuter ce programme mais celui qui se trouve dans le fichier indiqué par le premier argument de `execl`.

Parce que `exec` remplace le programme en cours d'exécution, c'est (avec `exit`) un des rares appels système pour lequel il n'y a pas besoin de tester la valeur renvoyée. Si le programme qui appelle `execl` continue ensuite son exécution, c'est que l'exec n'a pas fonctionné ; il y a donc eu une erreur. Dans un programme, on peut écrire :

```
execl(fichier, nom, arg1, 0);
perror("exec");
exit(1);
```

3.5.1 La combinaison fork + exec

La combinaison des deux appels système `fork` et `exec` est la façon normale sous Unix de créer un nouveau processus en train d'exécuter un nouveau programme.

Ce qui est particulièrement intéressant dans la combinaison de ces deux appels, c'est la manière dont elle permet d'effectuer les modifications des caractéristiques du processus enfant : en général, c'est le processus parent qui sait de quelle manière l'environnement doit être modifié. Après le `fork`, le processus enfant exécute le programme du processus parent et peut donc modifier les caractéristiques du processus avant de lancer avec `exec` l'exécution du nouveau programme. Le schéma général est donc

```
t = fork();
assert(t != -1);
if (t == 0){
  // enfant
  modifier les caractéristiques de l'enfant
  execl(...);
  traiter l'erreur lors du exec
}
// parent
...
```

On verra un exemple pratique de ce modèle dans les redirections d'entrée-sortie.

Une fonction `spawn1` qui lance un nouveau processus en train d'exécuter un nouveau programme avec un argument peut s'écrire simplement :

```
/* ch-spawn.c
 Une fonction pour lancer un programme dans un nouveau proc.
*/
# include <stdio.h>
# include <unistd.h>
# include <sys/types.h>
# include <sys/wait.h>
# include <stdlib.h>

/* spawn1  --  lance un processus qui execute le fichier */
```

```
int
spawn1(char * fichier, char * commande, char * arg){
  int t;

  t = fork();
  if (t < 0)                    // erreur
    return -1;

  if (t == 0){                  // enfant
    execl(fichier, commande, arg, (void *)0);
    exit(1); // erreur dans l'enfant

  }
                                // parent
  return t;
}
```

Il y a deux étapes : on crée le processus avec `fork` puis on lui fait exécuter le programme avec `exec`.

On peut l'utiliser simplement comme dans l'exemple suivant, où on lance 10 fois la commande `mon-echo` que nous avons étudiée dans le chapitre sur le shell.

```
/* ci-spawn-main.c
 Un exemple d'utilisation de la fonction spawn1
*/
# include <stdio.h>
# include <assert.h>
# include <sys/types.h>
# include <sys/wait.h>

int spawn1(char *, char *, char *);

int
main(){
  int i, t, tt, etat;

  for(i = 0; i < 10; i++){
    t = spawn1("mon-echo", "nom bizarre", "ceci est un argument");
    if (t < 0){
      perror("fork"); // echec lors de la creation
      break;
    }
    tt = wait(&etat);
    assert(tt == t);
    if (etat != 0){
      perror("exec mon-echo"); // erreur lors de l'exec()
                               // ou dans la commande
      break;
    }
  }
  return i != 10;
```

```
}
```

Les trois arguments de `spawn1` sont le nom du fichier exécutable (ici `mon-echo`), un nom de commande et un argument. Un exemple d'utilisation :

```
$ gcc -g -Wall ba-echo.c -o mon-echo
$ gcc -g -Wall ch-spawn.c ci-spawn-main.c
$ a.out
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
$
```

Exercice 3.10 — Modifier le programme précédent pour donner à `spawn1` un nom de fichier qui n'existe pas ou qui n'est pas exécutable. Quel message d'erreur obtient-on ? Expliquer.

3.5.2 La variante `execv`

La fonction execl n'est pas très pratique si on ne connaît pas à l'avance le nombre d'arguments avec lequel il faut lancer la commande. La fonction `spawn1` de la section précédente ne permet de lancer que des commandes avec un argument. Pour les commandes avec deux arguments, il faudrait faire une autre fonction `spawn2` et ainsi de suite pour tous les nombres d'arguments possibles.

Une meilleure solution consiste à utiliser la fonction `execv` (avec 'v' comme *variable*, par opposition au 'l' comme *liste* de `execl`) dans laquelle on passe le nom de la commande et les arguments dans un tableau.

Nous pouvons réécrire notre programme qui utilisait `execl` avec `execv` comme dans :

```
/* cj-execv.c
 Une utilisation simple de execv
 */
# include <stdio.h>
# include <unistd.h>
```

```
char * tab[] = { "commande", "arg1", "arg2", 0, };

int
main(){
  execv("/bin/echo", tab);
  perror("en essayant de faire execv");
  return 1;
}
```

Exercice 3.11 — Écrire et tester une fonction `spawn` qui permet de lancer une commande avec un nombre variable d'arguments.

Exercice 3.12 — (assez difficile) Quelle est le nombre ou la taille maximum des arguments qu'on peut spécifier dans un appel à exec.

Exercice 3.13 — 3.13 Ecrire une commande `mon-if` qui lancera un processus, puis en lancera un second si le premier a réussi. Par exemple `mon-if 'true' 'echo reussi'` affichera `reussi` alors que `mon-if 'false' 'echo raté'` n'affichera rien. Prévoir le cas où les commandes contiennent plusieurs mots, comme dans `mon-if 'echo foo' 'echo bar'`.

3.6 Le `PATH`, les fonctions `execlp` et `execvp`

Nous avons vu dans le chapitre sur le shell que la variable d'environnement `PATH` contient la liste des répertoires où le shell va chercher un fichier exécutable. Quand on donne une commande, le shell va donc tenter d'exécuter un fichier qui porte le nom de la commande successivement dans chacun des répertoires.

Cette opération est tellement courante qu'il existe des fonctions de bibliothèque, `execlp` et `execvp`, qui font ce travail automatiquement.

Une réalisation de la fonction execvp pourrait être la suivante

```
/* cn-execvp.c
Une re-implementation de la fonction execvp
 */
# define _GNU_SOURCE // pour asprintf
# include <stdio.h>
# include <unistd.h>
# include <string.h>
# include <assert.h>
# include <stdlib.h>

int
monexecvp(char * file, char * av[]){
  char * p;
  char * path;
  int t;

  p = getenv("PATH");
  if (p == 0){
    execv(file, av);
```

```
    return -1;
  } else {
    p = strdup(p);
    assert(p != 0);
    for(p = strtok(p, ":"); p != 0; p = strtok(0, ":")){
      t = asprintf(&path, "%s/%s", p, file);
      assert(t != 0);
      execv(path, av);
      free(path);
    }
    free(p);
  }
  return -1;
}
```

Exercice 3.14 — (assez difficile) Un exercice d'un chapitre précédent demandait comment faire pour ajouter au `PATH` un nom de répertoire qui contient le caractère deux points (`':'`); réécrire la fonction `execvp` pour répondre correctement à cette question.

3.6.1 La fonction de bibliothèque `system`

Il existe une fonction `system` dans la bibliothèque usuelle qui permet de lancer une commande. Cette fonction lance (avec `fork` et `exec`) un shell et lui passe la commande en argument. Le shell à son tour lance, avec `fork` et `exec` dans la plupart des cas, la commande qu'on a passé comme argument à `system`.

L'intervention du shell signifie que les commandes lancées avec `system` peuvent utiliser toutes les fonctionnalités du shell, notamment le globbing et les redirections d'entrées-sorties; pour cette raison, elle est la source de nombreux problèmes de sécurité.

Je répète : *la fonction* `system` *n'est pas un appel système pour créer un processus : elle utilise en fait trois appels systèmes pour lancer un shell qui lancera lui-même le processus avec d'autres appels système.*

Exercice 3.15 — Mesurer la différence entre les temps d'exécution de `system("/bin/echo x")` et `execl("/bin/echo", "echo", "x", 0)`. (Attention, pour obtenir des mesures de temps significatives, il faudra l'exécuter de nombreuses fois; j'ai utilisé 10000 tours, qui prend quelques secondes sur ma machine. Pour éviter de mesurer le délai du à l'affichage, vous pouvez rediriger la sortie du programme vers un fichier, par exemple avec `a.out > toto`. Pour vérifier que vous n'avez pas oublié le `fork()` avant l'`execl()` dans le second cas, vous pouvez compter le nombre de `x` dans le fichier `toto` avec `wc toto`.)

Exercice 3.16 — Comparer l'effet de `system("echo * > toto")` avec celle de
`execlp("/bin/echo", "echo", "*", ">", "toto", 0)`. Expliquer.

Supplément : une réimplémentation de la fonction `system`

On peut à peu près re-définir la fonction system avec le code suivant

```
/* cl-monsystem.c

 Une re-implementation de la fonction system
*/
```

```
# include <unistd.h>
# include <stdio.h>
# include <sys/types.h>
# include <sys/wait.h>

int
monsystem(char * command){
  int pid, x, etat;

  pid = fork();
  if (pid < 0){
    perror("fork");
    return -1;
  }
  if (pid != 0){ // parent
    for(;;){
      x = wait(&etat);
      if (x == pid)
        return etat;
    }
  }
  // enfant
  execl("/bin/sh", "sh", "-c", command, NULL);
  perror("execl");
  return -1;
}

# ifdef TEST
int
main(int ac, char * av[]){
  int i;

  for(i = 1; i < ac; i++)
    printf("la commande %d donne %d\n", i, monsystem(av[i]));
  return 0;
}
# endif
```

3.6.2 Passage de l'environnement aux processus enfants

(section facultative)

En fait, lors d'un `exec`, on passe un troisième paramètre qui décrit l'environnement, tel que nous l'avons présenté dans le chapitre sur le shell. Il existe pour cela des variantes de `execl` et `execv` qui s'appellent `execle` et `execve` et qui prennent la description de l'environnement comme argument supplémentaire.

L'argument est décrit, dans la mémoire d'un programme, sous la forme d'un tableau de chaînes de caractères, comme `argv`. Chaque chaîne contient la définition d'une variable, comme imprimé par la commande `printenv`.

Quand un programme commence à s'exécuter, il reçoit l'environnent sous

la forme d'un troisième paramètre de la fonction main, rarement représenté, qu'on appelle communément `envp`. On peut facilement réécrire la commande `printenv` comme dans le programme suivant :

```
/* cl-monprintenv.c
 Refaire le travail de printenv
*/
# include <stdio.h>

int
main(int ac, char * av[], char * envp[]){
  int i;

  for(i = 0; envp[i] != 0; i++)
    printf("%s\n", envp[i]);
  return 0;
}
```

3.7 Récapitulation : un shell simple

Pour donner un exemple complet d'utilisation de ces appels système, Je présente ici un shell simple, qui permet d'entrer au clavier et de faire exécuter des commandes. Les exercices qui suivent proposent quelques extensions.

3.7.1 Découper une chaîne de caractères

Je commence par définir une fonction `decouper` qui permet de découper une chaîne de caractères en sous-chaînes. Elle utilisera la fonction de bibliothèque `strtok`.

```
/* cn-decouper.c
 Un wrapper autour de strtok
*/
# include <stdio.h>
# include <string.h>

/* decouper  --  decouper une chaine en mots */
int
decouper(char * ligne, char * separ, char * mot[], int maxmot){
  int i;

  mot[0] = strtok(ligne, separ);
  for(i = 1; mot[i - 1] != 0; i++){
    if (i == maxmot){
      fprintf(stderr, "Erreur dans la fonction decouper: trop de mots\n");
      mot[i - 1] = 0;
      break;
    }
    mot[i] = strtok(NULL, separ);
  }
```

```
  return i;
}

# ifdef TEST
int
main(int ac, char * av[]){
  char *elem[10];
  int i;

  if (ac < 3){
    fprintf(stderr, "usage: %s phrase separ\n", av[0]);
    return 1;
  }

  printf("On decoupe '%s' suivant les '%s' :\n", av[1], av[2]);
  decouper(av[1], av[2], elem, 10);

  for(i = 0; elem[i] != 0; i++)
    printf("%d : %s\n", i, elem[i]);

  return 0;
}
# endif
```

La fonction reçoit une ligne de texte (en fait une chaîne de caractères) et la découpe en mots qui sont placés dans le tableau qu'on lui passe comme troisième argument. Le deuxième argument est la liste des caractères qui peuvent servir de séparateurs et le quatrième est le nombre d'emplacements disponibles dans le tableau de mots.

La définition de la fonction est accompagnée dans le fichier, par une fonction `main` qui permet de la tester. Grâce au `#ifdef` ··· `#endif`, la fonction main ne sera compilée que si on appelle le compilateur avec l'option `-DTEST`. Ce genre de *test unitaire* est utile pour s'assurer que chaque partie d'un programme fonctionne bien, indépendamment des autres. De plus, le fait de réfléchir à la manière dont on peut tester un module conduit en général à une meilleure répartition des fonctions entre les modules.

Compilons et faisons tourner le programme de test :

```
$ gcc -g -Wall -DTEST cn-decouper.c      compiler
$ a.out                                  un test du message d'erreur
usage: a.out phrase separ
$ a.out 'x et y' ' '                     test simple
On découpe 'x et y' suivant les ' ' :
0 : x
1 : et
2 : y
$ a.out '' ' '                           découper une chaîne vide ?
On découpe '' suivant les ' ' :          ok
$ a.out 'x.....y' '.'                    plusieurs fois le séparateur ?
On découpe 'x.....y' suivant les '.' :
0 : x
1 : y                                    ok
$ a.out '...x.y....' '.'                 séparateur au début et à la fin
On découpe '...x.y....' suivant les '.' :
0 : x
1 : y                                    ok
$ a.out '..ceci.,,et,.cela,' '.,'        plusieurs séparateur ?
On découpe '..ceci.,,et,.cela,' suivant les '.,' :
0 : ceci
1 : et
2 : cela
```

Pour une fonction plus complexe, ça pourrait valoir la peine d'automatiser ces tests, de façon à pouvoir vérifier que ces cas restent correctement traités après une modification éventuelle de la fonction. On appelle ceci des *tests de (non) régression.*

3.7.2 La boucle principale

Le corps de la boucle principale lit une ligne de commande, la découpe (avec la fonction `découper` de la section précédente), fabrique un nouveau processus (avec `fork`) et tente d'y lancer la commande en allant chercher un fichier du même nom dans chacun des répertoires indiqué par `PATH`.

```
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <sys/types.h>
# include <sys/wait.h>
# include <assert.h>

enum {
  MaxLigne = 1024,                  // longueur max d'une ligne de commandes
  MaxMot = MaxLigne / 2,            // nbre max de mot dans la ligne
  MaxDirs = 100,                    // nbre max de repertoire dans PATH
  MaxPathLength = 512,              // longueur max d'un nom de fichier
};
```

```
void decouper(char *, char *, char **, int);

# define PROMPT "? "

int
main(int argc, char * argv[]){
  char ligne[MaxLigne];
  char pathname[MaxPathLength];
  char * mot[MaxMot];
  char * dirs[MaxDirs];
  int i, tmp;

  /* Decouper UNE COPIE de PATH en repertoires */
  decouper(strdup(getenv("PATH")), ":", dirs, MaxDirs);

  /* Lire et traiter chaque ligne de commande */
  for(printf(PROMPT); fgets(ligne, sizeof ligne, stdin) != 0; printf(PROMPT)){
    decouper(ligne, " \t\n", mot, MaxMot);
    if (mot[0] == 0)              // ligne vide
      continue;

    tmp = fork();                 // lancer le processus enfant
    if (tmp < 0){
      perror("fork");
      continue;
    }

    if (tmp != 0){               // parent : attendre la fin de l'enfant
      while(wait(0) != tmp)
        ;
      continue;
    }

                                  // enfant : exec du programme
    for(i = 0; dirs[i] != 0; i++){
      snprintf(pathname, sizeof pathname, "%s/%s", dirs[i], mot[0]);
      execv(pathname, mot);
    }
                                  // aucun exec n'a fonctionne
    fprintf(stderr, "%s: not found\n", mot[0]);
    exit(1);
  }

  printf("Bye\n");
  return 0;

}
```

On peut effectuer des tests simples de ce programme :

```
$ ls                              liste des fichiers avec le shell ordinaire
cn-decouper.c co-main.c
$ gcc -g -Wall *.c                compilation
$ a.out                           lancement du nouveau shell
?                                 on voit le prompteur
? ls                              lancement d'une commande simple
a.out cn-decouper.c co-main.c     ok
? echo *                          y a-t-il quelqu'un pour faire le globbing ?
*                                 non !
? pwd                             une autre commande ?
/tmp/essai                        ok
? exit                            sortir ?
exit: not found                   il n'y a pas de commande exit !
? ^D                              taper à la fois sur les touches CTRL et D.
Bye
$
```

Ce shell élémentaire est une occasion d'identifier finement la répartition du travail entre le shell, le noyau et les commandes. Ainsi `echo *` a simplement affiché une étoile au lieu de la remplacer par la liste des fichiers du répertoire courant, parce que mon shell ne remplace pas les modèles par les noms de fichiers qui correspondent (il ne fait pas de globbing). De même, la commande `exit` est introuvable, parce qu'il s'agit d'une commande *interne* au shell (qui lui indique de terminer le travail) et non pas d'une commande lancée via fork et exec.

Nous utiliserons ce shell comme base pour d'autres exercices dans les chapitres suivants.

3.7.3 Exercices

Exercice 3.17 — (assez facile) Rajouter la possibilité de lancer des processus en arrière-plan quand la ligne de commande se termine par un &. (indication : il suffit de ne pas faire le `wait` dans le parent).
La commande `cd` n'existe pas dans notre shell; pourtant il existe un appel système `chdir` qui permet de changer le répertoire courant. On pourrait l'utiliser dans une commande comme la suivante (exécutable à placer dans le fichier `moncd`) :

```
/* cp-moncd.c
 Pour faire un chdir (pas trop utilisable)
*/
# include <stdio.h>
# include <unistd.h>
# include <stdlib.h>

int
main(int ac, char * av[]){
  char * dir;
  int t;
                                  // traiter les arguments
  if (ac < 2){
```

```
    dir = getenv("HOME");
    if (dir == 0)
      dir = "/tmp";
  } else if (ac > 2){
    fprintf(stderr, "usage: %s [dir]\n", av[0]);
    return 1;
  } else
    dir = av[1];
                                  // faire le boulot
  t = chdir(dir);
  if (t < 0){
    perror(dir);
    return 1;
  }
  return 0;
}
```

On se contente de faire un `chdir` dans le répertoire donné en argument ou dans celui indiqué par `HOME` dans l'environnement comme la commande `cd` usuelle.

Exercice 3.18 — (facile) Pourquoi la commande `moncd` ne fonctionne-t-elle ni sous notre shell, ni sous `bash` ?

Exercice 3.19 — (moyen) Faire une commande interne `moncd` dans notre shell qui fonctionne.

Exercice 3.20 — (facile si l'exercice précédent a été bien fait) Rajouter une commande interne `exit`.

3.8 Les threads

Les processus présentent deux inconvénients majeurs : d'une part ils sont assez coûteux à créer (on a vu qu'on ne pouvait en créer que quelques milliers par secondes), d'autre part la communication entre les processus est assez complexe et délicate (comme nous le verrons dans le chapitre sur la communication entre les processus).

(Pour rendre le `fork` moins coûteux, certains systèmes Unix ont une variante, appelée `vfork` qui est plus rapide. Le système ne recopie que ce qui est indispensable des caractéristiques du processus parent, parce que l'enfant obtenu avec `vfork` va très prochainement effectuer un `exec` ; le parent reste bloqué jusqu'à l'`exec` de l'enfant.)

Pour pallier à ces inconvénients, il existe un autre niveau, entre programme et processus, qu'on appelle *processus légers* ou *threads*. Dans une famille de threads, les membres partagent (presque) toutes les caractéristiques d'un processus sauf le point d'exécution et les données sauvées dans la pile (donc les variables locales aux fonctions). En revanche le code, les variables globales, les fichiers ouverts sont partagés par tous les threads d'une même famille.

Un inconvénient des threads, c'est qu'il existe de nombreuses manières de les mettre en place, qui sont bien évidemment incompatibles entre elles. Une question importante est notamment de savoir jusqu'à quel point le noyau participe

à l'existence des threads (il est possible de les réaliser sans aucune modification du noyau).

Je présente ici sommairement la variante la plus répandue, qu'on appelle les `pthreads` (le 'p' vient de ce qu'ils ont été spécifiés par un comité de normalisation d'Unix qui s'appelle *Posix*).

3.8.1 Deux visions des threads

Il existe (au moins) deux manières de voir les threads : soit comme des sous-processus soit comme des fonctions parallèles.

Dans la vision comme des *fonctions parallèles*, chaque thread fonctionne comme une fonction ordinaire appelée par un processus. La fonction à accès, comme d'ordinaire, à ses variables locales et aux variables globales. Les threads ont cependant ceci de particulier qu'il est possible d'avoir, en même temps, plusieurs appels de fonctions ; ils se déroulent en parallèle. Le degré de parallélisme dans l'exécution dépend de la machine et de la façon dont les threads sont implémentés. (Sur une machine normale à l'heure actuelle (2014) il y a plusieurs cœurs ; on peut s'attendre que chaque cœur exécute un thread et donc que les threads soient réellement parallèles.)

Dans la vision comme des sous-processus, on voit les threads d'une même famille comme des processus indépendants, qui ont en plus la particularité de partager de nombreuses caractéristiques entre eux, notamment la mémoire qui contient les variables globales.

3.8.2 Les threads : naissance, vie et mort

Voici un programme simple qui crée quelques threads :

```
/* cq-thread.c
 Simple creation de threads
*/
# include <stdio.h>
# include <assert.h>
# include <pthread.h>

enum {
  Nthread = 3,
  Nfois = 10,
};

static void *
foo(void * arg){
  int i, t;
  char * name = arg;

  for(i = 0; i < Nfois; i++){
    t = rand() % 5;
    printf("%s: je dors %d secondes\n", name, t);
    sleep(t);
  }
```

```
  printf("%s: j'ai fini, merci\n", name);
  return arg;
}

int
main(){
  static char * name[Nthread] = { "thread 0", "thread 1", "thread 2" };
  pthread_t id[Nthread];
  void * junk[Nthread];
  int i;

  printf("main: creation de %d threads\n", Nthread);
  for(i = 0; i < Nthread; i++)
    pthread_create(&id[i], 0, foo, name[i]);

  printf("On attend la fin des trois threads\n");
  for(i = 0; i < Nthread; i++){
    pthread_join(id[i], &junk[i]);
    printf("main: %s est termine\n", junk[i]);
  }
  return 0;
}
```

Il a fallu inclure le fichier `pthread.h` pour avoir les prototypes de fonction et les types de données spécifiques des threads et il faut explicitement indiquer la bibliothèque qui contient le code de gestion des threads en compilant avec :

```
$ gcc -g -Wall cq-thread.c -lpthread
```

Un thread exécute une fonction principale comme un processus exécute la fonction `main`. Ici c'est la fonction `foo` pour tous les threads. Cette fonction reçoit un argument et renvoie une valeur qui est un pointeur sur n'importe quoi (en C, c'est un `void *`).

Ici la fonction `foo` se contente de dormir quelques fois, pendant des intervalles de temps pris au (pseudo-)hasard.

On crée un thread avec la fonction `pthread_create` dont les arguments sont d'une part un emplacement où placer les informations sur le thread créé (du type `pthread_t`, il joue le même rôle que la valeur renvoyée par `fork` dans le processus parent), des paramètres facultatifs sur le thread avec lesquels on peut jouer un peu sur ses caractéristiques, la fonction principale du thread (ici `foo`) et son argument.

On attend la fin du thread avec la fonction `pthread_join` (qui joue un peu le même rôle que `wait` pour un processus) : on lui passe comme arguments l'identité du thread dont on veut attendre la fin et un emplacement où sera placée la valeur renvoyée.

Exercice 3.21 — Écrire un programme pour mesurer le temps de création d'un thread sur votre ordinateur, comme nous l'avons fait pour fork. Comparer les résultats.

Exercice 3.22 — Que se passe-t-il quand un thread appelle la fonction `exit` ? Est-ce que tous les threads du processus s'arrêtent ou bien seulement celui qui a

appelé `exit` ? (indication : le plus simple est d'écrire un programme qui permette de répondre à la question).

Implémentation des threads dans les processus et le noyau

On peut implémenter les threads dans le noyau et il seront alors gérés comme des processus mais le noyau va se compliquer.

On peut les implémenter dans les processus mais il faut remplacer les appels système bloquants (comme celui utilisé par la commande `sleep`) par quelque chose qui active un autre thread pendant que celui qui effectue l'appel système est bloqué. D'autre part cela ne permet pas de tirer parties des processeurs multi-core.

Les threads : version simple

(section très facultative)

On peut définir des threads d'une façon qui n'est pas si compliquée mais il faut aller manipuler des choses de bas niveau. Je présente dans cette section une implémentation simple de threads minimaux.

Le code en C est le suivant :

```
/* cv-thread.c
 Des threads simplifies a l'extreme
*/
# include <stdio.h>
# include <stdlib.h>

typedef struct Ctxt Ctxt;

enum {
  StackSize = 1024,             // taille de la pile
};

/*
  Le contexte d'un thread
 */
struct Ctxt {
  int ebp;                      // frame pointer
  int esp;                      // stack pointer
  int edi, esi, ebx;            // registres a sauver : edi, esi, ebx
                                // eax, ecx edx ont ete sauves par l'appelante
  char * name;                  // purement informatif
  int stack[StackSize];         // de l'espace pour la pile
};

Ctxt ctxt[2];                   // deux threads
Ctxt * curthread;               // le thread courant
Ctxt original;                  // le thread original

void save(Ctxt*);               // en assembleur dans swtch.s
void restore(Ctxt*);
```

```
void sched(void);                // dans la suite du code
void createthread(int num, void (*fun)(void), char * name);
void run(void);
static void endthread(void);

int globi = 0;

/* principal  --  un thread elementaire */
void
principal(){
  int i;

  for(i = 0; i < 5; i++){
    printf("Je suis %s, i vaut %d, globi %d\n",
           curthread->name, i, globi++);
    sched();
  }
}

int
main(){
  createthread(0, principal, "thread 0");
  createthread(1, principal, "thread 1");
  run();
  return 0;
}


/* sched  --  elementaire : round robin avec deux taches */
void
sched(){
  static int tour;

  save(curthread);
  curthread = &ctxt[tour ^= 1];
  restore(curthread);
}

/* endproc  --  fin d'un processus : on arrete tout */
static void
endthread(){
  printf("endthread: on rentre dans la fonction\n");
  restore(&original);
  printf("endthread: impossible\n");
}

/* run  --  lance le premier thread */
void
run(){
```

```
  curthread = &original;
  original.name = "Original";
  save(&original);

  curthread = &ctxt[0];
  restore(&ctxt[0]);
  printf("run: c'est termine\n");
}

/* createthread  --  fabrique un thread : numero, fonction, nom */
void
createthread(int num, void (*fun)(void), char * name){
  int * p;

  p = &ctxt[num].stack[StackSize - 1];  // p sur la fin de la pile
  *p-- = (int)endthread;
  *p = (int)fun;                  // la fonction comme adresse de retour
  ctxt[num].esp = (int)p;
  ctxt[num].name = name;
}
```

Le fichier définit une structure `Ctxt` (comme *Contexte*) qui définit le contexte d'un thread. Elle contient de l'espace pour sauver les registres, le nom du thread et de la mémoire pour contenir la pile.

Dans notre exemple, il y a trois contextes : celui des deux threads que j'utilise dans le programme et le contexte original, qui sera utilisé pour sortir.

La fonction `principal` est la fonction avec laquelle les threads sont lancés. Dans notre exemple, elle se contente de faire quelques tours dans une boucle en affichant la valeur de deux variables, l'une locale et l'autre globale, qu'elle incrémente à chaque tour. Notre fonction appelle explicitement la fonction `sched` qui va activer le thread suivant.

Finalement la fonction `main` fabrique les deux threads (avec `createproc`), les lance (avec `run`). Quand les threads ont terminé leur travail, elle reprend la main et s'arrête proprement.

Les fonctions suivantes pourraient être placées dans une bibliothèque. La fonction `sched` sauve l'état du thread courant, en choisit un autre et l'active (comme on n'a que deux threads, le choix est facile). La fonction `endthread` n'est pas utilisée mais elle permettrait de terminer tous les threads. La fonction `run` est appelée pour le lancement des threads : elle sauve le contexte du processus (pour `endproc`) et installe le contexte du premier thread.

La fonction `createthread` sert à initialiser le contexte d'un thread. Elle se livre à des manipulations de pile peu orthodoxes qui ne sont pas détaillées ici.

Il manque les deux fonctions `save` et `restore` qui sont utilisées pour sauver et réinstaller le contexte d'un thread ; ces deux fonctions sont écrites en assembleur, comme dans ce qui suit :

```
#  %ebp (pointeur de frame)
#  %esp (pointeur de pile)
#  %edi, %esi, %ebx (qui peuvent avoir des valeurs a conserver)
#  %eax, %ecx, %edx ont ete sauves par l'appelante si necessaire
        .text
```

```
        .globl save, restore

# a appeler avec "save(ctxt)"
save:
        movl    4(%esp),%eax    # eax pointe sur le contexte
        movl    %ebp,0(%eax)    # sauve les registres dans old
        movl    %esp,4(%eax)
        movl    %edi,8(%eax)
        movl    %esi,12(%eax)
        movl    %ebx,16(%eax)
        ret
# a appeler avec "restore(ctxt)"
restore:
        movl    4(%esp),%eax    # eax pointe sur le contexte
        movl    0(%eax),%ebp    # remplit les registres avec new
        movl    4(%eax),%esp
        movl    8(%eax),%edi
        movl    12(%eax),%esi
        movl    16(%eax),%ebx
        ret
```

Le rôle de ces deux fonctions est simplement de copier le contenu des registres du processeur dans la mémoire (pour `save`) ou de remettre la copie qui est dans la mémoire à l'intérieur des registres (pour `restore`)/

On peut compiler le programme qui se trouve dans les deux fichiers avec `gcc -g -Wall cu-swtch.s cv-thread.c`; le compilateur se charge de combiner le C et l'assembleur (comme indiqué par le nom des fichiers). On peut ensuite lancer le programme :

```
$ gcc -g -Wall cu-swtch.s cv-thread.c
$ a.out
Je suis thread 0, i vaut 0, globi 0
Je suis thread 1, i vaut 0, globi 1
Je suis thread 0, i vaut 1, globi 2
Je suis thread 1, i vaut 1, globi 3
Je suis thread 0, i vaut 2, globi 4
Je suis thread 1, i vaut 2, globi 5
Je suis thread 0, i vaut 3, globi 6
Je suis thread 1, i vaut 3, globi 7
Je suis thread 0, i vaut 4, globi 8
Je suis thread 1, i vaut 4, globi 9
endthread: on rentre dans la fonction
run: c'est terminé
$
```

On voit que les deux fonctions `principal` tournent à tour de rôle. Chacune possède sa propre variable locale `i` et elles partagent la variable globale `globi`.

Exercice 3.23 — (difficile)
Ce programme ne fonctionne que sur les processeurs Intel 32 bits. Le modifier pour qu'il tourne sur les processeurs Intel 64 bits. (Il n'y a pas de corrigé pour cet exercice car je n'ai pas encore fait ce travail.)

3.8.3 L'appel système `clone` de Linux

Sous Linux, le noyau connaît les threads, qui sont créés avec un appel système appelé `clone`. Cet appel système permet de spécifier quelles caractéristiques du processus sont partagées entre le parent et l'enfant, lesquelles seront recopiées et lesquelles seront réinitialisées.