Systèmes d'exploitation — Exercices
Processus

Seweryn Dynerowicz (Seweryn.DYNEROWICZ@umons.ac.be)

1 Création et terminaison

Dans un système GNU/Linux, la dynamique des processus passe par une interface de programmation comportant deux appels systèmes importants ; `fork()` et `exit(..)`. Le *listing* suivant reprend les éléments essentiels que nous détaillons plus bas.

```
#include <stdio.h>  // printf()
#include <stdlib.h> // exit(..), EXIT_SUCCESS
#include <unistd.h> // fork(), getpid()

int main(int argc, char* argv[]) {
  pid_t fvalue = fork();

  if(fvalue != 0) { // Parent process
    printf("[%ld] I am your father , %ld !\n", getpid() , fvalue);
  } else { // Child process
    printf("[%ld] Noooooooooooooo ! (%ld)\n", getpid() , fvalue);
  }

  exit(EXIT_SUCCESS);
}
```

FIGURE 1 – Création et terminaison

La création de processus ne peut être demandée que par un processus existant. Ceci a pour conséquence de lier tous les processus du système à un moment donné dans une arborescence. Deux exemples d'arborescences sont présentés dans la figure ci-dessous. L'appel système `fork()`, qui permet de réaliser cette opération (*cfr.* ligne 6), réalise une copie du processus appelant qui disposera du même contexte d'exécution (*i.e.* états de registres, *stack*, pointeur d'instruction courante). Au terme de l'appel, il existe deux processus parfaitement identique ; l'appelant est considéré comme le parent, la copie comme l'enfant. D'autre part, tous les fichiers qui étaient ouverts avant l'appel `fork()` sont partagés entre les deux processus ; une lecture ou une écriture altère le fichier partagé concerné [1].

Les deux processus poursuivent au retour de l'appel `fork()` et chacun reçoit une valeur de retour différente. Le processus parent reçoit le `PID` du processus enfant qui a été crée tandis que le processus enfant reçoit une valeur de `0`. Il est possible, sur base de cette valeur de retour, d'encoder les comportements des processus dans un seul programme (*cfr.* ligne 8). Il est également possible à tout moment pour un processus d'obtenir le `PID` qui lui est attribué à l'aide de la fonction `getpid()` (*cfr.* ligne 9 et 11) ainsi que le `PID` de son parent en utilisant la fonction `getppid()`.

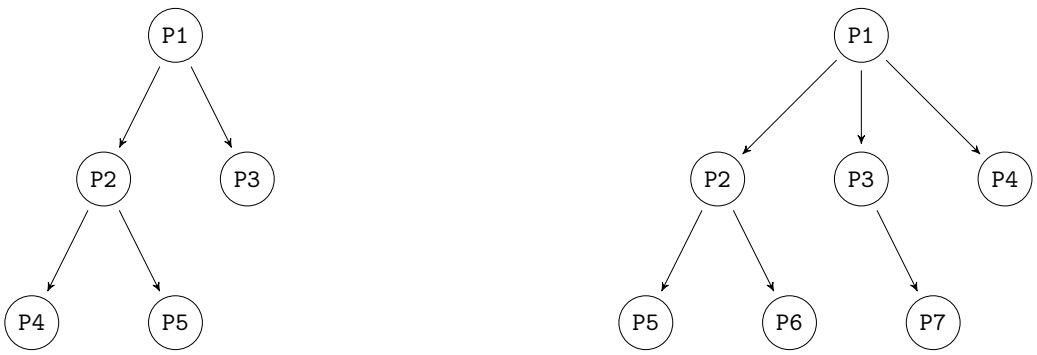

FIGURE 2 – Exemples d'arborescence de processus.

1. Trois fichiers importants dont dispose tout processus ; `stdin`, `stdout` et `stderr` sont également partagés.

Lorsqu'un processus arrive à son terme, la fonction `exit` doit être utilisée pour terminer proprement son exécution [2]. La valeur passée en paramètre constitue la valeur de retour du processus (*cfr.* état `ZOMBI`). Dans le terminal, il est possible d'obtenir la valeur de retour du processus principale d'un programme avec la commande `echo $?`

Il est possible d'observer l'ensemble des processus existants à un moment donné en utilisant la commande `ps` ou de surveiller en continu cet ensemble de manière plus détaillées via la commande `top` [3].

```
top - 20:14:49 up 5 min,  1 user,  load average: 1.20, 0.65, 0.28
Tasks: 295 total,   1 running, 294 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15478.7 total,  13662.1 free,    880.9 used,    935.7 buff/cache
MiB Swap:  16384.0 total,  16384.0 free,      0.0 used.  14319.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    784 root      20   0 1789568 127972  75528 S   1.0   0.8   0:05.84 Xorg
   3147 sdy       20   0   11.9g 445572 230896 S   1.0   2.8   0:10.76 firefox
      7 root      20   0       0      0      0 I   0.3   0.0   0:00.26 kworker/u32:0-events_unbound
    905 sdy       20   0  243696  15848  12352 S   0.3   0.1   0:00.60 i3bar
   1004 sdy       20   0   26208  16872  12312 S   0.3   0.1   0:00.23 urxvt
   2036 sdy       20   0   10836   4384   3496 R   0.3   0.0   0:00.52 top
   3489 sdy       20   0   10.4g 168148 105512 S   0.3   1.1   0:01.05 Isolated Web Co
   3665 sdy       20   0   54008  10388   9028 S   0.3   0.1   0:00.01 import
```

FIGURE 3 – Liste des processus existants via `top`.

2 Commutation et blocage

Les processus sous GNU/Linux sont soumis à la dynamique habituelle des commutations (*i.e.* perte du CPU), blocages et déblocages. Les appels systèmes sont susceptibles de provoquer une commutation ou un blocage du processus qui les invoquent. Dans le premier cas, on parle d'appels *non-bloquants* [4]. Dans le second cas, on qualifie de tels appels systèmes de *bloquants*. À titre d'exemple, les appels concernant les manipulations de fichiers (*i.e.* ouverture, lecture et écriture) seront des appels *bloquants*. Une catégorie particulière d'appels *bloquants* qui va nous intéresser sont ceux permettant la mise en sommeil du processus qui les invoquent.

La fonction `wait(..)` permet de mettre en attente le processus appelant jusqu'à un changement d'état d'un de ses processus enfants. La fonction `waitpid(..)` est plus générale car elle permet de spécifier le `PID` de n'importe quel processus dont on souhaite attendre un changement d'état. La *macro* `WEXITSTATUS` peut être utilisée pour extraire la valeur de retour qui a été récupérée. L'utilisation de ces fonctionnalités est illustrée dans la Figure 4.

```c
#include <stdio.h>     // printf(..)
#include <stdlib.h>    // exit(..), EXIT_SUCCESS
#include <unistd.h>    // fork(), getpid()
#include <sys/wait.h> // wait(..), WEXITSTATUS

int main(int argc, char* argv[]) {
  pid_t fvalue = fork();

  if (fvalue != 0) { // Parent process
    int wstatus;
    wait(&wstatus);
    printf("Deep Thought: %i.\n" , WEXITSTATUS(wstatus));
  } else { // Child process
    printf("Arthur: What do you get if you multiply six by nine ?\n");
    exit(42);
  }

  exit(EXIT_SUCCESS);
}
```

FIGURE 4 – Attente de processus

2. Si vous n'utilisez pas `exit` à la fin de la fonction `main`, la fonction constituant le vrai point d'entrée de votre programme s'en charge.
3. Vous pouvez également utiliser la commande `htop` si celle-ci est installée sur votre système.
4. Notez qu'un appel système donnant lieu à une continuation est également considéré comme *non-bloquant*.

Les fonctions de sommeil `sleep(..)`, `usleep(..)` et `nanosleep(..)` peuvent être utilisées pour mettre le processus appelant en attente jusqu'à ce qu'un délai spécifié se soit écoulé. Le délai que ces fonctions prennent en paramètre représente respectivement un nombre de secondes, de microsecondes (10^{-6}) ou de nanosecondes (10^{-9}). Le choix de la fonction à utiliser dépend de la précision requise dans l'application considérée. De plus la précision de ces fonctions [5] dépendra de plusieurs facteurs ; charge actuelle du système, longueur du code de la fonction de mise en sommeil utilisée. Il est dès lors nécessaire d'expérimenter par soi-même pour identifier si la durée fournie satisfait aux exigences de votre application. Ces trois fonctions sont susceptibles d'être interrompues par l'occurrence d'un signal pendant le sommeil. La valeur de retour de ces fonctions permet de savoir si le sommeil s'est fait complètement ou bien à été interrompu. La fonction `nanosleep(..)` se démarque par les paramètres qu'elle prend ; deux structures de type `timespec` qui encode le temps demandé et le temps résiduel en cas d'interruption. Nous décrivons cette structure dans la section suivante.

3 Mesure du temps

Afin de pouvoir mesurer le temps que prend une exécution, deux options existent ; la commande `time` qui permet de mesurer le temps complet [6] pris par l'exécution d'un processus ou bien l'appel système `clock_gettime(..)` qui peut être utilisé pour mesurer le temps écoulé entre deux points dans le code. Le *listing* suivant présente un exemple de mesure du temps pris pour effectuer un certain calcul (ligne 15).

```
1  #include <stdio.h>  // printf(..)
2  #include <stdlib.h> // exit(..), EXIT_SUCCESS
3  #include <time.h>   // struct timespec, clock_gettime(..)
4
5  long int nanos_between(struct timespec *final, struct timespec *start) {
6    time_t   seconds     = final->tv_sec  - start->tv_sec;
7    long int nanoseconds = final->tv_nsec - start->tv_nsec;
8    return seconds * 1e9 + nanoseconds;
9  }
10
11 int main(int argc, char* argv[]) {
12   struct timespec start, final;
13   clock_gettime(CLOCK_REALTIME, &start);
14
15   // Computation under measurement ...
16
17   clock_gettime(CLOCK_REALTIME, &final);
18   printf("Time elapsed : %ld ns\n", nanos_between(&final, &start));
19
20   exit(EXIT_SUCCESS);
21 }
```

FIGURE 5 – Mesure d'un temps de calcul.

La structure de type `timespec` (*cfr.* ligne 12) encode deux valeurs ; un nombre de secondes (*i.e.* `tv_sec`) et un nombre de nanosecondes [7] (*i.e.* `tv_nsec`). Une telle structure peut être passée par adresse à la fonction `clock_gettime(..)` (*cfr.* lignes 13 et 17) pour obtenir le temps écoulé depuis le démarrage du système. La premier paramètre spécifie la source à partir de laquelle la mesure doit être prise (*e.g.* `CLOCK_REALTIME` pour l'horloge temps-réel) tandis que le second paramètre contient la référence à l'endroit où sauvegarder la mesure réalisée. La fonction `nanos_between(..)` (lignes 5 à 9) permet de calculer le nombre de nanosecondes qui se sont écoulées entre deux instants de mesure passés en paramètres. Celle-ci réalise l'agrégation du nombre de secondes avec le nombre de nanosecondes exprimées par la différence entre les deux structures.

Attention : il est important de garder à l'esprit que de nombreux facteurs peuvent affecter la qualité d'une mesure effectuée de cette manière [8]. Si le calcul incorpore des opérations d'entrées/sorties (*i.e.* accès vers fichier), les temps d'accès associés peuvent être variables selon le périphérique ciblé. Le calcul dont on cherche à mesurer le temps peut faire l'objet d'une ou plusieurs suspension(s) pour cause d'ordonnancement (*cfr.* type *Round-Robin*). Les interruptions qui surviennent au cours du temps peuvent également introduire des délais dans le code mesuré si le processeur qu'il occupe est dédié à leurs traitements. Selon la charge du système (*i.e.* nombre de processus `READY`/`RUNNING`), l'état de la mémoire physique (*i.e.* combien de défauts de page vont survenir) ou le nombre de processeurs, ces temps additionnels peuvent être plus ou moins importants et introduire une erreur significative dans la mesure réalisée.

5. Une demande de mise en sommeil pendant $100\mu s$ ne va pas nécessairement durer **exactement** autant.
6. Ce temps est, par défaut, décomposé en temps utilisateur et temps système.
7. Une seconde contenant 10^9 nanosecondes, la valeur de ce champ est toujours comprise entre 0 et $10^9 - 1$.
8. Quelque soit la granularité considérée ; secondes, millesecondes, microsecondes ou nanosecondes.

4 Programmation *multi-threads*

Les possibilités de la programmation *multi-threads* sont rendues possibles par l'utilisation de la librairie `pthread` sous GNU/Linux. Afin de pouvoir utiliser cette librairie, la directive d'inclusion `#include <pthread.h>` doit être utilisée et l'option `-pthread` doit être passée lors de la compilation. Le *listing* ci-dessous reprend les éléments essentiels de la gestion des *threads* dans un exemple basique.

```
1  #define _GNU_SOURCE
2  #include <stdio.h>   // printf(..)
3  #include <stdlib.h>  // EXIT_SUCCESS
4  #include <unistd.h>  // gettid()
5  #include <pthread.h> // pthread: create(..), exit(), join(..), self()
6
7  int thread_retval = 0;
8  void* behavior(void* argument) {
9    thread_retval = 42;
10   printf("[%ld] Secondary thread\n", gettid());
11   pthread_exit(&thread_retval);
12 }
13
14 int main(int argc, char* argv[]) {
15   pthread_t primary = pthread_self();
16   pthread_t secondary;
17
18   pthread_create(&secondary, NULL, behavior, NULL);
19
20   int* retval = NULL;
21   pthread_join(secondary, (void*) &retval);
22
23   printf("[%ld] Primary thread received %i\n", gettid() , *retval);
24
25   pthread_exit(EXIT_SUCCESS);
26 }
```

FIGURE 6 – Gestion élémentaire des *threads*.

Le type *opaque*[9] `pthread_t` (*cfr.* lignes 12 et 13) représente un descripteur qui peut être utilisé pour se référer à un *thread* particulier, celui-ci faisant office d'identifiant au sein d'un processus. Un processus admet toujours un *thread* principal au démarrage qui peut engendrer d'autre *threads*. Pour obtenir l'identifiant d'un *thread* au niveau du système, la fonction `gettid()` [10] peut être utilisée pour obtenir un `PID` qui lui correspond.

La fonction `pthread_create(..)` (*cfr.* ligne 15) permet de réaliser la création d'un *thread* qui démarrera directement son exécution dans la fonction de traitement passée en paramètre (*cfr.* `behavior(..)`, lignes 6 à 9). Par contraste avec l'appel `fork()`, il n'existe pas de relation de filiation entre les *threads* ; tous existent au sein du même processus et se partagent toutes ses ressources. Tous partagent un destin commun ; si l'un d'entre eux provoque une exception fatale, le processus fera l'objet d'une terminaison, entraînant celle de tous ses *threads*.

Lorsqu'un *thread* arrive au terme de son traitement, la fonction `pthread_exit(..)` peut être utilisée pour effectuer sa terminaison et fournir une valeur de retour (*cfr.* lignes 8 et 21). L'utilisation d'un pointeur [11] permet de passer une valeur plus complexe qu'un simple entier (*e.g.* `struct`). Alternativement, il existe une fonction (*i.e.* `pthread_detach()`) qui permet de terminer le *thread* appelant sans fournir de valeur de retour, ce qui contourne le passage par un état `ZOMBI` pour ce *thread*. **Attention :** l'invocation de la fonction `exit(..)` dans le corps d'un *thread* entraîne la terminaison du processus.

Il est fréquemment utile dans la gestion d'un ensemble de *threads* réalisant un traitement parallélisé de pouvoir attendre leur terminaison. La fonction `pthread_join(..)` met en attente le *thread* appelant jusqu'à ce que le *thread* spécifié soit terminé. Une organisation classique en *multi-threads* consiste à effectuer la création d'un ensemble de *threads* par le *thread* principal qui se met ensuite en attente de la terminaison de ceux-ci avant de poursuivre ses propre traitements.

9. Il ne faut pas se reposer directement sur la valeur d'un type *opaque* dans la logique des traitements.
10. Pour être visible, il est nécessaire d'ajouter la directive `#define _GNU_SOURCE` au sommet de votre code.
11. L'emplacement pointé ne doit pas être local à la fonction du *thread*.

5 Énoncés

Exercice 1

Dessinez l'arborescence de processus résultant de deux appels `fork` successifs et de trois appels `fork` successifs. Écrivez deux programmes qui produisent les arborescences de processus de la Figure 2. Utilisez les fonctions `getpid` et `getppid` pour identifier les relations de filiations.

Astuce : tracez l'arborescence progressivement au fur et à mesure que vous écrivez votre programme.

Exercice 2

Écrivez un programme qui vous permet d'observer, grâce à la commande `top`, un processus existant à l'état `ZOMBI` et un processus existant à l'état `BLOQUÉ`.

Exercice 3

Écrivez un programme qui engendre un processus enfant. Une fois crée, le processus parent se met en attente de la terminaison du processus enfant qui génère deux nombres pseudo-aléatoires ; `delay` (compris entre 1 et 10) et `value` (compris entre 0 et 255). Après `delay` secondes, le processus enfant se termine en renvoyant comme valeur de retour `value`. Le processus parent récupère la valeur de retour de l'enfant et l'affiche.

Astuce : vous pouvez utiliser la fonction `rand()` pour générer un nombre aléatoire [12] et la fonction `srand(..)` pour que les nombres générés soient différents à chaque exécution.

Exercice 4

Écrivez un programme pour évaluer la précision du sommeil produit à l'aide de la fonction `usleep`. Implémentez des mesures de l'écart du temps de mise en sommeil du processus et calculer la moyenne et la médiane de cette valeur sur base de nombreuses mesures successives. Évaluez pour différentes valeurs de durées de sommeil.

Exercice 5

Évaluez la précision du sommeil produit par la fonction `nanosleep` en suivant les contraintes de l'exercice précédent.

Exercice 6

Écrivez un programme dont le *thread* principal engendre 9 autres *threads*. Chacun de ces *threads* génère deux nombres aléatoires ; `delay` (compris entre 0 et 5) et `value` (compris entre 0 et 255). Après `delay` secondes, un *thread* se termine et retourne `value` comme valeur de retour. Le *thread* principal accumule toutes ces valeurs dans un tableau qu'il trie avant d'afficher les identifiants des *threads* dans l'ordre décroissant des valeurs reçues. Dans le cas où deux valeurs identiques ont été fournies par deux *threads* différents, affichez l'identifiant le plus bas en premier lieu.

Exercice 7

Écrivez des versions *multithread* de programmes pour (1) identifier l'élément minimal, (2) calculer la moyenne et l'écart-type d'un tableau d'entiers.

Astuce : récupérez les fichiers `arrays.h` et `arrays.c` sur Moodle afin de disposer d'un jeu de tableaux pour tester votre programme.

12. **Attention :** les nombres pseudo-aléatoires ne sont pas adaptés pour faire de la cryptographie.

Travail de groupe

Implémentez une version *multi-thread* de l'algorithme de triage *merge-sort* de tableaux d'entiers. Ajoutez une mesure de temps d'exécution dans votre code pour évaluer combien de *threads* produisent les meilleures performances.

Astuce : dessinez le graphe d'appel pour un petit tableau (*e.g.* []).

```
typedef unsigned int index;
typedef unsigned int length;

void merge(int array[], index start, index middle, index final);
void merge_sort(int array[], index start, index final);

void merge_sort(int array[], index start, index final) {
  if(start < final) {
    index middle = floor((start + final) / 2);
    merge_sort(array, start, middle);
    merge_sort(array, middle+1, final);
    merge(array, start, middle, final);
  }
}

void merge(int array[], index start, index middle, index final) {
  length countL = middle - start + 1;
  int *arrayL = malloc(countL * sizeof(int));
  index currentL, currentR;
  for(currentL = 0; currentL < countL; currentL++)
    arrayL[currentL] = array[start + currentL];

  length countR = final - middle;
  int* arrayR = malloc(countR * sizeof(int));
  for(currentR = 0; currentR < countR; currentR++)
    arrayR[currentR] = array[middle + 1 + currentR];

  currentL = 0;
  currentR = 0;
  index current;
  for(current = start; current <= final && currentL < countL && currentR < countR; current++) {
    if(arrayL[currentL] <= arrayR[currentR]) {
      array[current] = arrayL[currentL];
      currentL++;
    } else { // arrayL[currentL] > arrayR[currentR]
      array[current] = arrayR[currentR];
      currentR++;
    }
  }

  // If <arrayL> was completely consumed, copy the remainder of <arrayR> over the remainder of <array>
  if(currentL >= countL)
    while(currentR < countR) {
      array[current] = arrayR[currentR];
      current++;
      currentR++;
    }

  // If <arrayR> was completely consumed, copy the remainder of <arrayL> over the remainder of <array>
  if(currentR >= countR)
    while(currentL < countL) {
      array[current] = arrayL[currentL];
      current++;
      currentL++;
    }

  free(arrayL);
  free(arrayR);
}
```