université PARIS-SACLAY
IUT D'ORSAY

BUT Informatique
R305 Système S3
2024 / 2025

Travaux Pratiques 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.

Exercice 0 : Hello world. Voici un exemple minimal de programme C qui affiche la chaîne de caractères *hello, world* (cet exemple célébrissime repris dans tous les langages a d'ailleurs été fait à l'origine en C en 1978 par les créateurs du langage, Brian Kernighan et Dennis Ritchie).

```
#include <stdio.h>
int main() {
  printf("hello, world\n");
  return 0;
}
```

Vous devrez utiliser un éditeur pour créer vos fichiers sources, qui auront l'extension `.c` pour les fichiers contenant le code C (et non `.cpp` comme en C++) ou `.h` dans le cas des fichiers d'entête (*header files*). Pour compiler vos fichiers sources (par exemple, `hello.c`) et créer un fichier exécutable (par exemple, `hello`), placez-vous dans le répertoire dans lequel vous avez enregistré le source et utilisez la commande `gcc -Wall -Werror hello.c -o hello`

Vous pourrez alors lancer l'exécution de votre programme en utilisant la commande `./hello`

Créez un fichier `hello.c` contenant le code ci-dessus, compilez-le et exécutez-le.

Rappel : Sorties en C. Le langage C ne dispose pas du flot de sortie `cout` bien connu en C++. À sa place, on utilise une fonction spécifique nommée `printf()`. L'instruction

```
printf("La moyenne des %d entiers est %f.\n", i, moyenne);
```

affiche sur la sortie standard (l'écran) la chaîne de caractères comprise entre les guillemets (appelée un *format*) mais où `%d` a été remplacé par la valeur de la variable entière `i`, `%f` a été remplacé par la valeur de la variable réelle `moyenne` et `\n` n'a pas été affiché mais un retour à la ligne a été effectué. `%d` et `%f` sont appelés des *codes*. Ils définissent la manière dont une variable d'un certain type doit être affichée. Le $n^{\text{ième}}$ code est destiné au $n^{\text{ième}}$ paramètre après la chaîne de caractères. La carte de référence vous donne plus de détails sur les codes. Le caractère spécial `\n` désigne le retour à la ligne.

Reprenez `hello.c` pour tester différents formats à l'aide de la fiche de référence.

Que se passe-t-il s'il y a trop de codes pour pas assez de paramètres ? Et inversement ?
Que se passe-t-il si on applique un code d'un type pour un paramètre d'un autre type ?
(Pensez à ne pas utiliser l'option `-Werror` pour cet exercice.)

Rappel : Pointeurs. Dans le langage C, on peut accéder directement à des zones de la mémoire par l'intermédiaire de leurs *adresses*. Pour connaître l'adresse d'une variable, on utilise l'opérateur « & » : par exemple, si `num` est une variable de type entier, alors `&num` est l'adresse à laquelle se trouve cette variable. Pour connaître la taille en octets de la zone mémoire allouée à une variable, on utilise l'opérateur **`sizeof`**`()`, par exemple, la place occupée par `num` est donnée par **`sizeof`**`(num)` ou encore **`sizeof`**`(int)`. Pour afficher le résultat de **`sizeof`**`()` avec `printf()`, on utilise le code « `%zu` ».

On appelle les variables qui contiennent les adresses les *pointeurs* (on peut s'en servir pour stocker `&num`, par exemple). On déclare un pointeur en ajoutant le caractère étoile « ∗ » devant le nom de la variable. Par exemple, l'instruction suivante :

```
int *ptr;
```

déclare une variable nommée `ptr` destinée à contenir l'adresse de début d'une zone de mémoire contenant un entier. On dit que `ptr` est un *pointeur sur entier*. Dans le reste du programme, on utilisera `ptr` pour désigner l'adresse d'un entier (par exemple, `ptr` aurait pour valeur l'adresse de `num` si on effectuait l'affectation `ptr = &num`), et `*ptr` pour désigner l'entier qui se trouve à cette adresse (ainsi, pour changer la valeur de `num`, on pourrait exécuter `*ptr = 2`). Soit le code suivant :

```
#include <stdio.h>

int main() {
  double valeur = 10.0;
  double *pv = &valeur;
  int nombre = 1, *pn;
  pn = &nombre;
  valeur = *pv + *pn;
  printf(" valeur = %f\n", valeur);
  printf("&valeur = %p\n", &valeur);
  return 0;
}
```

Que représente `valeur` ? Qu'affiche le premier `printf()` ?

Que représente `&valeur` ? Qu'affiche le second `printf()` ?

Que représente `pv` ? Que représente `&pv` ? Que représente `*pv` ?

Quelle est la taille de la zone mémoire réservée pour `valeur` ? Pour `pv` ? Pour `nombre` ? Pour `pn` ? Pourquoi les tailles sont-elles différentes pour `valeur` et `nombre`, mais identiques pour `pv` et `pn` ?

En C, tous les paramètres des fonctions sont des *copies* des arguments passés lors des appels à ces fonctions. Cela signifie que l'on ne travaille *jamais* directement sur les éléments passés lors de l'appel, mais avec d'autres variables qui contiennent, par contre, exactement les mêmes valeurs que les originales. Soit le programme suivant :

```
#include <stdio.h>

void echange1(float a, float b) {
  float temp = a;
  a = b;
  b = temp;
}

int main() {
  float pi = 2.71828, e = 3.14159;
  printf("Avant echange : pi = %f, e = %f.\n",pi,e);
  echange1(pi,e);
  printf("Apres echange : pi = %f, e = %f.\n",pi,e);
  return 0;
}
```

Qu'affiche ce programme ? Expliquez.

Écrivez une fonction `echange2()` qui réalisera effectivement l'échange des valeurs grâce à des pointeurs. Par ailleurs, vous ferez en sorte que les valeurs ne s'affichent qu'avec deux décimales. **Vous joindrez le code complet commenté à votre compte rendu.**

Rappel : Entrées en C. À la place du flot d'entrée `cin` de C++, on utilise la fonction `scanf()`. L'instruction

```
scanf("%f %d", &x, &i);
```

lit des informations sur le fichier standard d'entrée (le clavier). Comme `printf()`, `scanf()` possède en premier argument un format exprimé sous la forme d'une chaîne de caractères (ici `"%f %d"`). Les arguments suivants sont les adresses où il faudra « ranger » les informations lues. Cette instruction lira donc un flottant et un entier qu'elle rangera respectivement dans les variables `x` et `i`. Pour les chaînes de caractères, l'emploi de `scanf()` et du code « `%s` » est peu pratique car le caractère *espace* est considéré comme un délimiteur (pas de chaînes avec des espaces). On préférera utiliser la fonction `fgets()` (appel `fgets(buffer,taille,stdin)`) qui permet de lire une ligne du fichier standard d'entrée. La fonction `fgets()` place la ligne lue (avec le(s) caractère(s) de fin de ligne) dans le tampon préalloué `buffer`, la lecture s'arrête après `taille-1` caractères, et le caractère `'\0'` est toujours ajouté dans `buffer` après le dernier caractère lu.

Exercice 1 : Aiguille dans le foin. Le but de ce problème est de détecter la présence d'un zéro dans un tableau de **`unsigned char`** de taille `TABSIZE` (par exemple, 10000) en partageant le travail entre plusieurs processus. Nous allons utiliser le code suivant pour initialiser le tableau :

```
unsigned char arr[TABSIZE];
srandom(time(NULL));
// entasser du foin
for (i = 0; i < TABSIZE; i++)
  arr[i] = (unsigned char) (random() % 255) + 1;
// cacher l'aiguille
printf("Enter a number between 0 and %d: ", TABSIZE);
scanf(" %d", &i);
if (i >= 0 && i < TABSIZE) arr[i] = 0;
```

Consultez les pages `man 3 random` et `man 2 time` pour vous renseigner sur le fonctionnement de ces primitives (et les fichiers entêtes à inclure dans votre code source).

À quoi sert l'instruction `srandom(time(NULL))` ?

Combien de zéros, au minimum et au maximum, peuvent apparaitre dans le tableau à la fin de ce fragment de code ? Justifiez la réponse.

Une première version de votre programme doit générer un processus fils et lui confier une moitié du tableau. Le processus père devra fouiller l'autre moitié. À la fin de son travail, le fils communiquera au père le résultat de ses recherches : 1 si zéro est trouvé, sinon 0. Le père doit alors combiner les résultats et afficher le verdict final :

```
if (found) printf("Got a needle!\n"); else printf("No needles.\n");
```

De quel mécanisme peut-on se servir pour passer l'information du fils au père ?

Écrivez le programme C correspondant et joignez le code source à votre compte rendu.

Énumérez les bonnes valeurs d'indice à donner à votre programme pour le tester.

Exercice 2 : Des foules pour des fouilles. On modifie le programme de l'exercice précèdent pour qu'il prenne sur la ligne de commande un nombre `N` entre 1 et 100 qui désigne le nombre de processus entre lesquelles on partage le tableau pour faire la recherche :

```
if (argc > 1) N = atoi(argv[1]);
if (N < 1 || N > 100) N = 1;
```

Nous allons maintenant modifier la structure de notre algorithme : votre programme devra créer `N` processus fils pour les recherches, et laisser le père juste récupérer et combiner leurs résultats.

Écrivez le programme C correspondant et joignez le code source à votre compte rendu.

Exercice 3 : Tous dehors. Reprenons le programme de l'exercice 2. Supposons qu'un des processus fils trouve zéro très vite et en avertit le père. Dans ce cas, le père peut afficher la réponse et terminer sans attendre la fin de calcul des autres fils.

Modifiez le code de votre programme à cet effet. Pour mieux comprendre le comportement du programme, ajoutez deux commandes suivantes dans la boucle qui parcourt le tableau :

```
putc('.', stdout); fflush(stdout);
```

Lisez les pages de `man` correspondantes et expliquez ce que font ces commandes.

Faites plusieurs tests avec le programme modifié. Que constatez-vous ?

Il nous faut un moyen d'arrêter les processus fils encore vivants dès l'instant que le père a reçu le résultat positif (et même avant de l'afficher !). Heureusement, la primitive `kill()` nous permet d'envoyer un signal à tous les processus dans le groupe de l'appelant, ce qui dans notre cas correspond à tous les processus engendrés au cours d'exécution du programme :

```
kill(0, SIGTERM);
```

Modifiez le code de votre programme à cet effet. Pour mieux comprendre le comportement du programme, faites afficher leur PID au processus père et à tous les processus fils au début de l'exécution. Outre cela, pensez à installer un gestionnaire de `SIGTERM` qui affiche le PID du processus courant et le termine par un appel de `exit()`.

Faites plusieurs tests avec le programme modifié. Que constatez-vous ? Proposez une solution.

Finissez le programme et joignez le code source à votre compte rendu.

Exercice 4 : Mais on l'a déjà fait en TD ! Implémentez et testez l'exercice 5 du TD1. Utilisez la commande `time` pour savoir combien de temps il faut à votre machine pour compter jusqu'à 0. Vérifiez avec la commande `kill` que votre code gère correctement les signaux `SIGUSR1` et `SIGUSR2`.