Processus

Dans ce TP vous allez devoir utiliser différents appels systèmes. En pratique, un programme utilise rarement directement un appel système mais passe par des fonctions de la biliothèque `libc` qui se charge de le réaliser. Cela permet à la `libc` de proposer des interfaces plus agréables à l'utilisateur que celles souvent complexes proposées par l'OS.

Toutes les fonctions sont documentées dans les sections 2 et 3 des pages de manuel. Dans votre terminal, la commande "`man 3 exec`" vous permet par exemple de consulter la documentation des fonctions permettant de réaliser l'appel système exec. On peut lire que différentes interfaces sont proposées pour celui-ci ainsi que la manière de l'utiliser correctement et le fichier d'en-tête à inclure.

Pour les problèmes de ce TP vous aurez besoin des appels systèmes suivants, il vous est donc conseillé de consulter leur documentation :

`fork, exec, sleep, usleep, wait`

On rappel aussi qu'un programme peut prendre des arguments sur la ligne de commande. Afin d'accéder à ceux-ci, votre fonction `main` doit avoir le prototype suivant :

```
int main(int argc, char *argv[])
```

Le paramètre `argc` indique le nombre d'arguments donné au programme et le paramètre `argv` est un tableau de chaines de caractères contenant les arguments. Le premier argument donné au programme est son propre nom.

Exercice 1. Questions de cours Les questions de cours sont à destinées à vous permettre de vérifier votre compréhension du cours. Elles sont à travailler à l'avance et ne seront pas traitées en TD ou TP.

1. Quelle est la définition d'un processus ?
2. De quoi se compose un processus ? Expliquez.
3. Peut-on avoir plusieurs processus prêts dans le système ?
4. Le système d'exploitation est-il un processus ?

Exercice 2. Création de processus Dans cet exercice et les suivants, vous devez utiliser les appels systèmes vus en cours ainsi que d'autres qui sont documentés dans les pages de manuels des systèmes UNIX. (Dans un terminal utilisez la commande `man` suivie du nom de l'appel système)

1. Écrivez en C un programme qui affiche "Bonjour", attend 5 secondes et affiche "Au revoir" avant de se terminer.

   Correction: *Rien de compliqué ici, on reprend le code du TP précédent et on vérifie que la compilation se passe correctement.*

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

int main(void) {
    printf("Bonjour\n");
    sleep(5);
    printf("Au revoir\n");
    return 0;
}
```

2. Modifiez votre programme de manière à ce qu'il effectue un `fork` et que les deux processus préfixent leurs affichages par le numéro de processus. Le processus père attendra 5 secondes tandis que le fils attendra seulement 2 secondes avant de se terminer.

   Correction: *Une utilisation basique de `fork` sans grand interet. Attention de bien conserver la valeur de retour car, si elle n'est pas utile pour l'affichage, elle est indispensable pour distinguer le père de son fils.*

```c
int main(void) {
    pid_t pid = fork();
```

```
3     if (pid < 0) {
4         printf("Echec du fork\n");
5     printf("%d Bonjour\n", getpid());
6     if (pid == 0)
7         sleep(2);
8     else
9         sleep(5);
10    printf("%d Au revoir\n", getpid());
11    return 0;
12 }
```

Exercice 3. Explosion On considère le programme suivant : (qui n'est bien sûr pas un exemple de bonne programmation)

```
1 #include <stdlib.h>
2 #include <unistd.h>
3 int main(void) {
4     fork();
5     fork();
6     fork();
7     fork();
8     return EXIT_SUCCESS;
9 }
```

1. Combien de processus sont créés par l'exécution de se programme ?

Correction: *Le premier* `fork` *va créer un nouveau processus à partir du processus initial, chacun de ces deux processus va continuer l'exécution du code et donc le* `fork` *suivant va créer deux nouveaux processus pour un total de quatre processus en cours d'exécution. Le troisième va créer quatre nouveau processus et le cinquième en créera huit. Au total, le processus initial va (directement ou via un de ses enfant) créer* $1+2+4+8=15$ *nouveau processus.*

Une mauvaise utilisation de `fork`*, en particulier dans une boucle, peut rapidement amener à la création d'un nombre exponentiel de processus produisant une saturation et donc un ralentissement voir un arrêt du système.*

2. Que ce passe-t-il si l'on remplace le premier appel à `fork` par un appel à `exec` exécutant le programme lui-même ?

Correction: *Au moment de l'appel à* `exec` *le programme va être recouvert par une copie de lui-même mais qui reprend l'exécution au début, une boucle infinie qui ne fait que recouvrir le programme va donc ce produire. Cela consomme un peu de ressources pour rien mais sans gros effet néfastes.*

3. Même question mais pour le deuxième appel à `fork` ?

Correction: *Le même phénomène se produit mais cette fois-ci le processus va se dupliquer avant le recouvrement, il va donc produire un nouveau processus fils à chaque étape et chacun de ces processus fils fera de même. On a donc création d'un nombre exponentiel de processus qui va très rapidement surcharger le système.*

Exercice 4. Recherche parallèle La recherche des occurences d'une valeur dans un tableau non trié nécessite de parcourir tout le tableau. Si le tableau est grand ou si la comparaison des valeurs est coûteuse, cette recherche peut être longue. Si l'ordinateur exécutant le programme dispose de plusieurs processeurs ou cœurs, il est possible de réaliser la recherche simultanement sur chacun d'entre eux afin de l'accelérer.

Dans ce problème, vous devez écrire un programme qui alloue un grand tableau d'entiers aléatoires, puis recherche toutes les occurences d'une valeur et affiche leurs indices en utilisant deux processus.

Les ordinateurs modernes sont très rapides et si l'on souhaite pouvoir observé des choses intéressantes ici il faudrait travailler sur un tableau vraiment immense ou utiliser une fonction de comparaison complexe. Pour garder un programme simple, le plus facile est d'utiliser un tableau de taille 1000 contenant des entiers inférieurs à 100 mais d'ajouter un appel à `usleep(10000)` *; à l'interieur de la boucle de recherche pour simuler une fonction de comparaison coûteuse.*

1. Commencez par écrire un premier programme non-parallèle qui alloue un grand tableau, le remplis d'entiers aléatoires et recherche toutes les occurences de la valeur 0. Pour chaque occurence trouvée, affichez un message indiquant son indice.

Correction:

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

#define N 1000

int main(void) {
    int *T;
    T = malloc(sizeof(int) * N);
    for (int i = 0; i < N; i++)
        T[i] = rand() % 100;
    for (int i = 0; i < N; i++) {
        usleep(10000);
        if (T[i] == 0)
            printf("0 trouvé à l'index %d\n", i);
    }
    return EXIT_SUCCESS;
}
```

2. Il s'agit maintenant de parallèliser ce programme sur deux processus, pour cela le programme doit réaliser un `fork`, puis un des processus doit chercher les valeurs 0 dans la première moitié du tableau pendant que l'autre cherche dans la deuxième moitié.

Correction: *On commence par ajouter, après l'initialisation du tableau, l'appel système* `fork`*. Chaque processus doit aussi déterminer les bornes de la portion du tableau qu'il doit explorer.*

```
    pid_t pid = fork();
    int deb, fin;
    if (pid < 0) {
        perror("Echec du fork");
        return EXIT_FAILURE;
    } else if (pid > 0) {
        deb = 0; fin = N / 2;
    } else {
        deb = N / 2; fin = N;
    }
```

Une fois cela fait, la boucle de recherche ne change quasiment pas, il suffit juste d'utiliser les bornes qui viennent d'être déterminées et chaque processus explorera sa moitié du tableau.

```
    for (int i = deb; i < fin; i++) {
        usleep(10000);
        if (T[i] == 0)
            printf("0 trouvé à l'index %d\n", i);
    }
```

Le shell va reprendre la main lorsque le processus père se termine. Le soucis est qu'à ce moment-là, il est possible que le processus fils n'ai pas terminé sa recherche. S'il trouve un nouveau 0, son affichage va se mélanger à l'affichage du shell et perturber l'interaction avec l'utilisateur. Pour éviter cela, on demande au processus père d'attendre la fin de l'exécution de son fils avant de se terminer lui même.

```
    if (pid > 0) {
        int status;
        wait(&status);
    }
    return EXIT_SUCCESS;
```

3. Comparez les temps d'exécution et les affichages des programmes des deux questions précédentes. Que remarquez-vous ?

Correction: *La version parallèle s'execute environ deux fois plus rapidement. La boucle de recherche représente à peu près tout le temps d'exécution du programme et elle est réalisée en parallèle sur deux cœur dans cette version, c'est donc normal. Par contre l'affichage des 0 trouvés ne se fait plus dans l'ordre, en effet chaque processus affiche les index où il trouve les zéros dans l'ordre mais les deux affichages sont intercalés car fais en parallèle.*

Exercice 5. Signaux Les signaux permettent à l'OS d'informer immédiatement un processus d'un évennement. Ils sont utilisés principalement pour les situations d'erreurs mais peuvent aussi être utilisés dans d'autre circonstances telles qu'une alarme ou le redimensionement du terminal. Dans cet exercice, nous allons utiliser le signal `SIGINT` qui à l'avantage de pouvoir être envoyé simplement à un processus par la combinaison de touches `ctrl-C`.

1. Écrivez un programme qui simplement attend 10 secondes avant de quitter. Lorsqu'il reçoit le signal `SIGINT` affiche "Au revoir" sur le terminal et quitte sans attendre la fin des 10 secondes.

Correction: *Nous avons ici la forme la plus simple de gestion de signal. Il suffit d'enregistrer le gestionaire de signal avant l'appel système* `sleep`.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <signal.h>
4  #include <unistd.h>
5
6  void sigint(int sig) {
7      printf("Au revoir\n");
8      exit(0);
9  }
10
11 int main(void) {
12     signal(SIGINT, sigint);
13     sleep(10);
14     return 0;
15 }
```

Un test de se programme montre que si l'on patiente 10s, aucun message n'est affiché sur le terminal tandis que si l'on interrompt le programme le message est affiché.

2. Modifiez votre programme afin qu'avant de patienter 10s il crée un processus fils. Ce processus fils devra attendre 5s puis envoyer le signal `SIGINT` à son processus père.

Correction: *Il suffit de remplacer le* `sleep(10)` *par le code suivant qui va réaliser la création du processus fils. Ce fils utilise l'appel système* `kill` *afin d'envoyer le signal à son père.*

```
1      pid_t pid = fork();
2      if (pid < 0)
3          perror("Echec du fork");
4      if (pid != 0) {
5          sleep(10);
6      } else {
7          sleep(5);
8          kill(getppid(), SIGINT);
9      }
```

Exercice 6. Commande shell Le shell de base sous UNIX, `sh`, offre une commande `if` permettant d'exécuter une commande de manière conditionelle en fonction du résultat de l'exécution d'une autre commande. Dans cet exercice, ils vous est demandé d'implémenter une version simplifiée de cette commande qui s'utilise de la manière suivante :

```
si <commande1> alors <commande2> sinon <commande3>
```

Le programme `si` exécute la première commande, attend qu'elle se termine et si son exécution s'est bien passé exécute la deuxième commande, sinon il exécute la troisième commande. Chaque commande peut avoir des arguments et la troisième est optionelle. (si elle n'est pas présente le mot clé `sinon` ne doit pas être présent non plus)

Afin de simplifier les aspects purement programation en C et se concentrer sur les aspects système, on considère que l'utilisateur utilise correctement la commande, vous n'avez pas à gérer les cas ou les arguments donnés sont faux. Il n'est pas, par exemple, nécessaire de gérer le cas ou le mot clé `alors` *est absent ou le cas ou le mot clé* `sinon` *apparait avant le mot clé* `alors`.

1. Commencez par une version basique du programme qui ne prend que deux arguments, la commande à exécuter pour le test et celle à exécuter si la première se termine correctement. (Avec un code de retour de 0) Les deux commandes n'acceptent donc pas d'arguments et sont respectivement dans `argv[1]` et `argv[2]`.

Correction: *Rien de complexe ici, on utilise l'appel à* `wait` *pour récupérer le code de retour du test et on exécute éventuellement la deuxième commande. Attention de bien lire la documentation de l'appel système pour décoder correctement le code de retour. Pour l'exécution de la deuxième commande, il n'est pas nécessaire de faire un fork puisque le processus père n'a plus rien à faire ensuite, il peut donc disparaitre sans soucis.*

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    /* On réalise un fork+exec pour exécuter le test et on attend la fin de
     * l'exécution de celui-ci afin de récupérer son code de retour.
     */
    pid_t pid = fork();
    if (pid < 0) {
        perror("Echec du fork pour le test");
        return EXIT_FAILURE;
    } else if (pid == 0) {
        execl(argv[1], argv[1], NULL);
        perror("Echec du exec pour le test");
        return EXIT_FAILURE;
    }
    int status;
    wait(&status);
    /* On vérifie que le test s'est terminé correctement et on exécute la
     * deuxième commande en fonction du résultat.
     */
    if (!WIFEXITED(status)) {
        fprintf(stderr, "Le test n'a pas terminé normalement.\n");
        return EXIT_FAILURE;
    }
    if (WEXITSTATUS(status) == 0) {
        execl(argv[2], argv[2], NULL);
        perror("Echec du exec pour le alors");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

2. Modifier votre programme afin qu'il accepte des commandes avec arguments. Le mot clé `alors` doit apparaitre parmi les arguments du programme, tout ce qui est avant constitue le test et tout ce qui est après constitue la commande à exécuter.

Correction: *Il faut commencer par traiter les arguments, rien à voir avec l'OS ici, juste de la programation en C classique. Une des variantes de la fonction* `exec` *accepte que l'on spécifie le programe et ses arguments sous la forme d'un tableau de chaines de caractères ce qui est parfait ici.*

```c
    char **test = NULL, **alors = NULL;
    test = argv + 1;
    for (int i = 2; i < argc; i++) {
        if (strcmp(argv[i], "alors") == 0) {
            argv[i++] = NULL;
            alors = argv + i;
        }
    }
```

Il reste juste à changer les deux appels systèmes pour prendre en compte les arguments.

```c
    execvp(test[0], test);
    execvp(alors[0], alors);
```

Il est intéressant de noter ici le fait que les différentes variantes de `exec` *sont utiles dans différentes circonstances. L'appel système fourni par le système ne propose qu'une seule interface relativement complexe, la libc réalise pour nous le travail de fournir les différentes variantes simples à utiliser.*

3. Enfin, ajoutez la gestion du mot clé `sinon` afin d'obtenir la commande complète.

Correction: *On commence par modifier le code qui traite les arguments. On gère la partie* `sinon` *de la même manière que les autres et l'on obtient donc un troisième sous-tableau s'il est présent.*

```
1     char **test  = NULL, **alors = NULL, **sinon = NULL;
2     test = argv + 2;
3     for (int i = 2; i < argc; i++) {
4         if (strcmp(argv[i], "alors") == 0) {
5             argv[i++] = NULL;
6             alors = argv + i;
7         } else if (strcmp(argv[i], "sinon") == 0) {
8             argv[i++] = NULL;
9             sinon = argv + i;
10        }
11    }
```

Ensuite, il suffit de rajouter un `else` *au* `if` *qui test le code de retour du test. Si l'utilisateur à spécifié une commande* `sinon`*, on l'exécute de la même manière que la commande* `alors`*.*

```
1     } else if (sinon != NULL) {
2         execvp(sinon[0], sinon);
3         perror("Echec du exec pour le sinon");
4         return EXIT_FAILURE;
5     }
```

Exercice 7. Redirections *Attention, cet exercice utilise le concept des descripteurs de fichiers qui seront étudiés plus tard dans le cours. Il demande peu de code mais plus de réflexion que les autres, vous pourrez y revenir une fois le cours sur les fichiers assimilé.*

Contrairement à d'autre systèmes tels que Windows, les systèmes UNIX décomposent l'exécution d'un programme en deux étapes :

- la création du processus par duplication d'un processus existant ;
- son recouvrement par le programme à exécuter.

Cette séparation permet au processus qui souhaite exécuter un autre programme de réaliser certaines opérations de configuration du processus entre les deux étapes. C'est par exemple à ce moment qu'il est possible de mettre en place la redirection des entrées et sorties standard comme le fait le shell. La sortie standard est par exemple fournie par le système sous la forme d'un fichier virtuel dont le descripteur est donné par la constante `STDOUT_FILENO`. Il est possible de changer ce descripteur afin de diriger la sortie vers un fichier ou vers un autre processus.

Dans cet exercice, vous devez écrire un programme exécutant l'équivalent de la commande shell "`ls . > stdout.txt`". C'est-à-dire l'exécution de la commande `ls` afin de lister le répertoire courant et d'écrire le résultat dans le fichier `stdout.txt`.

1. Commencez par écrire un programme simple qui execute la commande `ls` dans le répertoire courant à l'aide de `fork+exec`. Le processus père devra attendre la fin de l'exécution du processus fils puis afficher si l'exécution s'est bien passée.

Correction: *Rien de bien compliqué ici si les exercices précédent ont été compris.*

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <errno.h>
4  #include <unistd.h>
5
6  int main(void) {
7      pid_t pid = fork();
8      if (pid < 0) {
9          perror("Echec du fork");
10         return EXIT_FAILURE;
11     } else if (pid == 0) {
```

```
12          execl("/bin/ls", "ls", ".", NULL);
13          return EXIT_FAILURE;
14      }
15      int status;
16      wait(&status);
17      if (WIFEXITED(status) && WEXITSTATUS(status) == 0)
18          printf("Tout s'est bien passé.\n");
19      else
20          printf("Un problème est arrivé.\n");
21      return EXIT_SUCCESS;
22  }
```

2. Maintenant modifier le code exécuté par le processus fils afin qu'il ouvre en écriture le fichier `stdout.txt` et le réassigne à la sortie standard. Pour cela, vous aurez besoin des appels système open et dup2.

Correction: *Il faut ajouter le code suivant just avant l'appel à* `execl`.

```
1       int fd = open("stdout.txt", O_WRONLY | O_CREAT, 0660);
2       if (fd < 0) {
3           perror("Ne peut ouvrir le fichier de sortie");
4           return EXIT_FAILURE;
5       }
6       if (dup2(fd, STDOUT_FILENO) < 0) {
7           perror("Ne peut rediriger la sortie standard");
8           return EXIT_FAILURE;
9       }
10      close(fd);
```

L'appel à open *permet d'ouvrir le fichier, on obtient un descripteur qui permet de travailler dessus. Le soucis est que la sortie standard est associée à un autre descripteur. L'appel system* dup2 *demande au système de rendre le deuxième descripteur donné équivalent au premier, ici on va donc rendre la sortie standard équivalent au fichier que l'on vient d'ouvrir. L'appel à* `close` *permet de ne pas laisser trainer un descipteur inutile, le fichier ne sera pas fermé car un autre descripteur pointe vers lui.*

Le shell utilise ce méchanisme pour les redirections vers des fichiers. Pour la connection de deux programmes à l'aide d'un tube comme "`ls | more`" les choses sont un peu plus complexes. L'appel système `pipe` demande au système d'allouer deux descripteurs de fichiers connectés de telle manière que ce qui est écris dans le premier peut être lus dan sle deuxième. Le shell l'utilise puis réalise deux fork afin de créer les deux processus fils. Ensuite, chaque fils va modifier son entrée ou sa sortie standard avant de réaliser l'appel à `exec`.