SYSTEMES D'EXPLOITATION && PROGRAMMATION CONCURRENTE

CPE Lyon – 4IRC

TRAVAUX PRATIQUES SEANCES 2,3

fork() - exec() – exit() – wait()

Exercice 1 *- Introduction au* ***fork()*** *et* ***exec()*** *:* Ecrire un programme qui de crée et lance deux processus, chacun réalisant son propre traitement. Tester l'appel **execlp()** en écrivant un programme qui lance un autre programme (**exécutable**).

Exercice 2 *- L'appel à* ***fork()*** *dans une boucle* : Ecrire un programme qui fait appel à la fonction **fork()** dans une boucle **for (k=0 ; k<3 ; k++)**.

A chaque itération le programme affichera les informations suivantes :

(k = valeur de k) : je suis le processus : pid, mon pere est : ppid , retour : retour

où

- ***pid*** est le PID du processus courant,
- ***ppid*** est le PID du processus père du processus courant,
- ***retour*** est la valeur du code retour de l'appel à la fonction **fork()**.

Dessiner l'arbre des processus correspondant à l'exécution de ce programme.

Exercice 3 *- Effet du* ***fork()*** : Le programme suivant ouvre deux fichiers, le premier [entrée] en mode LECTURE, le second [sortie] en mode ECRITURE, puis créé un processus fils et recopie le fichier d'entrée dans le fichier de sortie. Les descripteurs df1 et df2 désignent les 2 mêmes fichiers dans les deux processus, car l'ouverture des 2 fichiers correspondant est faite avant la création du processus fils (héritage). Exécutez plusieurs fois ce programme (sur un gros fichier texte d'entrée) et commentez son résultat. Vous pouvez également examiner la trace d'exécution dans le fichier trace.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main( ) {
        int pid , df1 , df2 ;
        char carac ;
        df1 = open("entree" ,O_RDONLY);
        df2 = creat("sortie" , 0666);
        FILE *trace = fopen("trace" , "w" ) ;
        fprintf(trace, "Le caractère '#' indique que c'est le processus fils qui s'exécute \n") ;
        fprintf(trace, "Le caractère '$' indique que c'est le processus père qui s'exécute \n") ;
        fflush(trace) ;
        pid  = fork() ;
        printf(("Création de Processus\n") ;
        while (read (df1 , & carac , sizeof(char)) > 0) {
                if (pid  == 0) fprintf(trace, "#%c ", carac ) ;
                else fprintf(trace, "$%c", carac ) ;
                fflush(trace) ;
                write( df2, & carac , sizeof(char)) ;
        }
        printf("Sortie de la boucle while\n") ;
        close(df1) ; close(df2) ;
        return 0;
}
```

Exercice 4 - fork() & exec : Ecrire un programme **C** équivalent aux commandes *shell* suivantes :

- **who & ps & ls –l** Les commandes séparées par **&** s'exécutent *simultanément*.
- **who ; ps ; ls -l** Les commandes séparées par **;** s'exécutent *successivement*.

Exercice 5 - fork() & exec : Ecrire un programme **C** qui prend en paramètre une série de fichiers source **.c**, les compile en simultané puis édite les liens pour produire un exécutable. Ce programme doit :

- lancer un **processus fils** pour chacun des fichiers passés en paramètre ;
- chaque processus fils doit exécuter la commande **gcc -c** sur le fichier dont il s'occupe ;
- le père doit **attendre la terminaison** de tous ses fils [*de toutes les compilations*] ;
- si l'ensemble des fils ont terminés sans erreur, le père génère un exécutable - Phase d'édition de liens en exécutant **gcc -o** sur les **fichiers.o** produits par les fils.

Exercice 6 – Questions cours : Soit un système qui exécute le programme suivant :

```
#include <sys/types.h>
#include <unistd.h>
int main( ) {
        int i, n=0;
        pid_t pid;
        for (i=1; i< 5; i++) {
                pid = fork();              /*1*/
                if (pid > 0) {             /*2 */
                        wait(NULL);        /*3*/
                        n = i * 2;
                        break; /*sortie de la
                boucle*/
                }
        }
        printf("%d\n", n);       /* 4 */
}
```

- Après la ligne étiquetée **/*2*/**, dans le bloc d'exécution du **if**, on se retrouve dans quel processus, le père ou le fils ? Pour qui la valeur de **pid** vaut 0 ?
- Ce programme est-il déterministe? (*Justifiez*)
- Même question si l'on supprime la ligne étiquetée **/*3*/** - justifiez.
- Si le programme est déterministe tel quel, indiquez exactement ce qui sera affiché à l'écran lors de son exécution. S'il n'est pas déterministe, donnez un des affichages possibles.
- L'appel à **fork()**, ligne étiquetée **/*1*/**, peut-il échouer? Pourquoi?

Exercice 7 : Ecrire un programme C dont le fonctionnement est le suivant :

- ✓ il lit sur la ligne de commande (utiliser **argc** et **argv**) le nombre **N** de processus à créer.
- ✓ il crée ces **N** processus en faisant **N** appels à **fork()**.
- ✓ il se met en attente (appel à **pid_fils = wait(&Etat)**) de ces **N** processus fils et visualise leur identité (**pid_fils** et valeur de Etat) au fur et à mesure de leurs terminaisons. Pour attendre la fin de tous les fils, utiliser le fait que **wait** renvoie la valeur **-1** quand il n'y a plus de processus fils à attendre.

Chacun des processus fils **Pi** réalise le traitement suivant :

- il visualise son **PID** (**getpid()**) et celui de son père (**getppid()**),
- il se met en attente pendant **2*i** secondes (**sleep (2*i**)), visualise la fin de l'attente,
- il se termine par **exit (i)**.

Exercice 8 : Considérons le programme C suivant :

```
int main( ) {
1.  int i, delai ;
2.  for (i=0 ; i<4 ; i++) if ( fork() ) break ;
3.  srand(getpid()) ;
4.  delai = rand()%4 ;
5.  sleep(delai) ;
6.  printf("Mon nom est %c, j'ai dormi pendant %d secondes\n", 'A'+i , delai) ;
7.  exit(0) ;
}
```

1. Donnez l'arbre généalogique des processus engendrés par ce programme.
2. Quels sont les affichages possibles ?
3. Sans modifier les lignes de *2* à *5*, modifiez ce programme de façon à ce que les processus fassent leur affichage par ordre alphabétique inversé du nom.

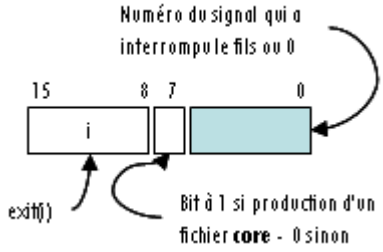
Exercice 9 : Ajoutez à l'endroit indiqué dans le programme ci dessous des instructions permettant de gérer les processus nécessaires pour avoir **4^N fois** l'affichage du message «**Bonjour**». Vous ne pouvez pas insérer du code supplémentaire ailleurs qu'à l'endroit indiqué (sauf si vous voulez ajouter des déclarations de variables), vous ne pouvez plus ajouter d'appels **printf()**. Vous devez donner une solution utilisant des **fork( )**, avec éventuellement des **wait( )**. Donnez toutes les justifications nécessaires.

```
#define N 3
int main(int argc, char *argv[]) {
   int i , pid1 , pid2;
   for (i = 0; i<N; i++) {
   /* _____debut des ajouts_____ */
   /* _____fin des ajouts_____ */
   }
   printf("Bonjour\n") ;
   return 0 ;
}
```

ANNEXE
SYNCHRONISATION DE PROCESSUS PERE ET FILS

- ✓ **exit(i)** : termine un processus, **i** est un octet (donc valeurs possibles : **0** à **255**) retourné dans une variable du type **int** au processus père.
- ✓ **wait(&Etat)** : met le processus en attente de la fin de l'un de ses processus fils.

Quand un processus se termine, le signal **SIGCHILD** est retourné à son père. La réception de ce signal fait passer le processus père de l'état BLOQUE à l'état PRET. Le processus père sort donc de la fonction **wait()**. La valeur retournée par **wait()** est le numéro du processus fils venant de se terminer. Lorsqu'il n'y a plus (ou pas) de processus fils dont il faut attendre la fin, la fonction **wait()** retourne **-1**. Chaque fois qu'un fils se termine le processus père sort de **wait()**, et il peut consulter la variable **Etat** pour obtenir des informations sur le fils qui vient de se terminer.



L'octet de poids fort de la variable **Etat** contient la valeur retournée par le fils (**i** de la fonction **exit(i)**). L'octet de poids faible contient **0** dans le cas général. En cas de terminaison anormale du processus fils, cet octet de poids faible contient la **valeur** du **signal** reçu par le fils. Cette valeur est augmentée de **80** en hexadécimal (**128** en décimal), si ce signal a entrainé la sauvegarde de l'image mémoire du processus dans un fichier *core*.

La fonction **exec()** charge un fichier exécutable dans le segment de code du processus qui l'appelle, remplaçant [recouvrant] ainsi le code courant par cet exécutable. Une des formes de cette fonction est :

int execl (char *fic, char * arg0, [arg1, … argn,])

fic est le nom du fichier exécutable qui sera chargé dans le segment de code du processus qui appelle la fonction **execl()**. Si ce fichier n'est pas dans le répertoire courant, il faut donner son nom complet (chemin absolu). Les paramètres suivants sont des pointeurs sur des chaines de caractères contenant les arguments passés à cet exécutable (cf. **argv** en *C*). La convention UNIX impose que la première chaîne soit le nom de l'exécutable lui-même et que le dernier soit un pointeur **NULL**. Par exemple, si on veut charger le fichier **prog** qui est stocké dans le répertoire courant et qui n'utilise aucun argument passé sur la ligne de commande, on utilisera

execl ("prog", "prog", NULL)