

# Système d'Exploitation Unix

TP

## Les appels système : gestion de processus (suite)

Tous les programmes devront être développés avec passage de leurs éventuels paramètres à la fonction `main (int argc, char *argv [])`. Les valeurs de retour des appels aux primitives devront être testées et les messages d'erreurs affichés avec `perror`. Les messages d'erreurs à destination de l'utilisateur se feront sur le fichier standard des erreurs `stderr`.

### Question 1 Le recouvrement : les primitives `exec ()`

Les primitives de la famille `exec()` permettent de charger en mémoire de nouveaux programmes binaires en vue de leur exécution.

Les primitives de la famille `exec()` se différencient par la manière dont les arguments sont transmis. Ces arguments sont transmis soit sous forme d'un tableau (famille `execv()`), soit sous forme de liste (famille `execl()`) selon que la primitive utilisée a un nombre fixe ou variable de paramètres.

- a) Ecrire un programme qui charge un nouveau programme binaire et dont les arguments sont transmis selon les deux modes précédents.

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

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

    if (argc < 2){
        fprintf(stderr, "Usage: %s chemin\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    execl(argv[1],argv[1],NULL);
    perror(argv[1]);
    exit(EXIT_FAILURE);

} // main
```

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

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

```
    if (argc < 2){
        fprintf(stderr, "Usage: %s chemin commande\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    execv(argv[1],&argv[1]);
    perror(argv[1]);
    exit(EXIT_FAILURE);
}
```

Les primitives de la famille `exec()` se différencient également par la manière dont le programme à charger est recherché dans le système de fichiers. Soit la recherche est relative au répertoire courant, soit elle l'est par rapport aux répertoires spécifiés via la variable `PATH`.

- b) Ecrire un programme qui charge un nouveau programme binaire qui est recherché selon les deux modes précédents.

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

main (int argc, char *argv[]){
    char *cde;

    if (argc < 2){
        fprintf(stderr, "Usage: %s chemin\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    execlp(argv[1],argv[1],NULL);
    perror(argv[1]);
    exit(EXIT_FAILURE);
} // main
```

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

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

    if (argc < 2){
        fprintf(stderr, "Usage: %s commande\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    execvp(argv[1],&argv[1]);
    perror("ERREUR execvp");
    exit(EXIT_FAILURE);
} // main
```

Les primitives de la famille `exec()` se différencient enfin par l'environnement conservé par le processus après recouvrement. Soit l'environnement reste inchangé, soit un nouvel environnement est transmis en paramètre.

- c) Ecrire un programme qui charge un nouveau programme binaire et qui dans un premier cas conserve le même environnement et dans un second cas acquière un nouvel environnement.

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

main (int argc, char *argv[], char **arge){
    int i;

    for (;*arge != NULL ; *arge++){
        printf("%s\n", *arge);
    }

} // main

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

main (int argc, char *argv[], char **arge){
    char **arge_sauv;

    arge_sauv=arge;

    // affichage de l'environnement avant l'execle
    printf("<<< affichage de l'environnement avant l'execle >>>\n");
    for (;*arge != NULL ; *arge++){
        printf("%s\n", *arge);
    }

    printf("\n<<< affichage de l'environnement apres l'execle >>>\n");
    execle("./affich_env","affich_env", NULL, arge_sauv);

    perror("affich_env");
    exit(EXIT_FAILURE);
}
```

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

main (int argc, char *argv[], char **arge){
    char *env[3];

    // affichage de l'environnement avant l'execle
    printf("<<< affichage de l'environnement avant l'execle >>>\n");
    for (;*arge != NULL ; *arge++){
        printf("%s\n", *arge);
    }

    // nouvel environnement pour le programme qui va être chargé
    env[0]="VARENV1=BIDON1";
    env[1]="VARENV2=BIDON2";
    env[2]=NULL;

    printf("\n<<< affichage de l'environnement apres l'execle >>>\n");
    execle("./affich_env","affich_env", NULL, env);

    perror("affich_env");
    exit(EXIT_FAILURE);
}
```

Au cours d'un recouvrement, les caractéristiques suivantes ne sont pas conservées selon les conditions spécifiées :

- **propriétaire effectif** : si le *set-uid bit* est positionné sur le fichier exécutable chargé, le propriétaire de ce fichier devient propriétaire effectif du processus,
- **groupe effectif** : si le *set-gid bit* est positionné sur le fichier exécutable chargé, le groupe propriétaire de ce fichier devient groupe propriétaire effectif du processus,
- **descripteur de fichier ouvert** : si le bit `FD_CLOEXEC` d'un descripteur a été positionné à l'aide de la primitive `fcntl()`, ce descripteur est fermé après un recouvrement.

- d) Ecrire des programmes qui vérifient les affirmations précédentes.

Voir e)

Au cours d'un recouvrement, les attributs suivants sont hérités par le nouveau programme :

- identifiant de processus (PID) et identifiant du processus parent (PPID),
- identifiant utilisateur réel (UID) et identifiant de groupe réel (GID),
- répertoire de travail courant,
- masque de création de fichier,
- verrous sur les fichiers,
- masque des signaux (cf. partie suivante),
- signaux pendants (cf. partie suivante).

- e) Ecrire des programmes qui vérifient les affirmations précédentes.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

char dir[200];
char *d;
mode_t mode;
int fd;

main (int argc, char *argv[]){
    printf("\n===== avant exec =====\n");
    printf("mon pid:\t\t%d\n",getpid());
    printf("le pid de mon pere:\t%d\n",getppid());
    printf("mon UID réel:\t\t%d\n",getuid());
    printf("mon UID effectif:\t%d\n",geteuid());
    printf("mon GID réel:\t\t%d\n",getgid());
    printf("mon GID effectif:\t%d\n",getegid());
    mode=umask(mode); /* retourne le masque actuel et le modifie */
    printf("mon masque(octal):\t%o\n",mode);
    mode=umask(mode); /* restauration du masque */
    d=getcwd(dir, 200);
    if (d!=NULL)
        printf("mon repertoire courant:\t%s\n",dir);
    else perror(dir);
    execl ("apres_exec", "apres-exec", NULL);
    perror("apres_exec");
} // main

```

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

```

```

char dir[200];
char *d;
int r;
mode_t mode;

```

```

main (int argc, char *argv[]){
    printf("\n===== apres exec =====\n");

    printf("mon pid:\t\t%d\n",getpid());
    printf("le pid de mon pere:\t%d\n",getppid());
    printf("mon UID réel:\t\t%d\n",getuid());
    printf("mon UID effectif:\t%d\n",geteuid());
    printf("mon GID réel:\t\t%d\n",getgid());
    printf("mon GID effectif:\t%d\n",getegid());
    printf("mon masque(octal):\t%o\n",umask(mode));
    d=getcwd(dir, 200);
    if (d!=NULL)
        printf("mon repertoire courant:\t%s\n",dir);
    else perror(dir);
} // main

```

Pour vérifier la modification des UID effectif et GID effectif vous pouvez utiliser l'exécutable du programme apres-exec qui se trouve à :

```

/users/ufr/prof/soto/Public
-rwsr-sr-x 1 soto ufr 6209 oct. 11 16:35 apres_exec

```

et sur lequel le *set-uid bit* le *set-gid bit* sont positionnés.

## Question 2 Le shell

Ecrire le programme d'un shell simplifié capable d'exécuter, en premier plan ou en arrière plan, n'importe quelle commande entrée par l'utilisateur avec un nombre arbitraire de paramètres.

Fonctions utiles:

```

fork, wait, exit,execvp, fgets, strtok, strchr, strlen, strcpy,
malloc.

```

```

#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>

#define TAILLE_LIGNE 250
#define TAILLE_ARG 40
#define EVER (;;)

char commande [TAILLE_LIGNE]; /* Commande tapee par l'utilisateur */
char *arg[TAILLE_ARG]; /* Tableau de pointeurs vers chaque "mot" */
/* de la ligne de commande tapee par */
/* l'utilisateur */

char *bg;
char *s;
int i;

/*=====*/
main(){
/*=====*/
int pid, CodeRetour;

for EVER { /* => Boucle infinie */
printf ("mishell> "); /* => Affichage de l'invite */

/* = > lecture de la ligne de commande */
fgets(commande, TAILLE_LIGNE, stdin);

/* ===== Nettoyage de la ligne de commande ===== */
commande[strlen(commande)-1]='\0'; /* => suppression du retour chariot */

/* = > suppression du & eventuel de la ligne de commande */
if (bg=strchr(commande, '&')) *bg='\0';

/* ===== Analyse de la ligne de commande ===== */
for(i=0,s=strtok(commande, " ");// Recherche des mots separes par un blanc
s!=NULL;
s=strtok(NULL, " "),i++){

/* === Preparation des parametres du execvp === */
arg[i]=(char *)malloc(strlen(s)+1);
strcpy(arg[i],s);

} /*for */

arg[i]=NULL; /* => Le tableau de pointeur doit se terminer */
/* par un pointeur NULL. */

```

```

if (i>0) /* => la ligne de commande n'est pas vide */
if (fork()==0) { /* ===== CODE DU FILS ===== */
printf("\n");

execvp (arg[0], arg);

/* == si on est la, c'est qu'il y a un probleme: == */
/* == affichage d'un message d'erreur == */
perror ("execvp");
}
else { /*===== CODE DU PERE =====*/
if (bg!=NULL) {
/* => commande executee en arriere plan: inutile d'attendre */
}
else {
pid=wait(&CodeRetour);
}/*if (bg!=NULL) */
} /* if (fork() ==0) */
}/* for EVER */
} /* main() */

```