

Chapitre 2 : Processus

- 2.1 Expliquez pourquoi, dans Linux, lorsqu'un processus exécute l'appel système « exit », ses ressources ne sont pas libérées tout de suite.

Le parent a toujours la possibilité de demander le résultat (statut de sortie) de ses enfants. Il faut donc conserver l'entrée dans la table de processus qui contient ce résultat. Conserver l'entrée empêche aussi dans l'intervalle que le numéro de processus de l'enfant soit recyclé, ce qui causerait de la confusion.

- 2.2 Dans le cas d'UNIX, la création de processus est réalisée par duplication. Citez un avantage et un inconvénient.

Le principal avantage est la simplicité pour activer cette fonctionnalité et la simplicité de l'implémentation sous-jacente. Dans le cas où on veut effectivement un clone du parent, c'est simple et efficace. L'inconvénient est lorsque le processus enfant est différent et doit faire un exec tout de suite après. Le combinaison fork + exec est moins intuitive et il y a une part d'effort perdu à avoir initialisé la table de page de l'enfant à partir de celle du parent.

- 2.3 Dans le système UNIX, est-ce que tout processus a un père? Que se passe-t-il lorsqu'un processus devient orphelin (mort de son père)? Quand est-ce qu'un processus passe à l'état Zombie?

Le processus Init récupère automatiquement tous les processus qui seraient orphelins. Un processus est zombie après sa sortie, en attente que son parent fasse un wait pour récupérer son statut de sortie.

- 2.4 Sur la plupart des systèmes, on retrouve pour les processus au moins les trois états: prêt, en exécution, et bloqué. Nous pourrions donc avoir six transitions possibles au maximum entre ces états. Dites lesquelles transitions ne sont pas logiquement possibles.

Un processus bloqué ne passe pas directement en exécution, il passe à l'état prêt lorsqu'il n'est plus bloqué. Un processus prêt est figé et ne peut donc pas devenir bloqué.

- 2.5 Le code qui s'occupe des interruptions dans les systèmes d'exploitation est presque toujours écrit en assembleur. Pourquoi?

D'une part, le code qui reçoit les interruptions doit être très efficace. Plus important, un traitement de bas niveau particulier est souvent requis, (e.g. activer ou désactiver les interruptions, changer le contenu de certains registres de contrôle), qui n'est pas accessible à partir des énoncés fournis par les langages de haut niveau.

2.6 Lors d'une interruption, le noyau utilise une pile séparée de celle du processus appelant. Pourquoi?

Premièrement, on ne peut pas se fier sur le processus pour laisser assez d'espace sur sa pile. Deuxièmement, les variables locales écrites par le noyau ne devraient pas pouvoir être vues par le processus.

2.7 Un processus a un descripteur de fichier connecté au clavier qui attend les entrées. Après un `fork()`, le nouveau processus a-t-il lui aussi accès au clavier? Est-ce que cela pose problème?

Tout d'abord, le processus ne peut pas être bloqué dans un appel de lecture du clavier s'il est en train d'appeler `fork()`! Autrement, si deux processus tentent en même temps de lire d'un même pointeur de fichier, chacun pourrait avoir une partie de ce qui est tapé, avec une répartition difficilement prévisible. Pour le clavier, ce n'est probablement pas le comportement désiré.

Les questions suivantes sont associées à un code source.

2.8 Combien de processus seront créés par ce programme?

```
// ex01.c
int main() {
    int cnt = 0;
    int p = 1;
    while (p > 0) {
        cnt++;
        p = fork();
    }
    if (p < 0) {
        printf("failed after %d fork\n", cnt);
    }
    execlp("sleep", "sleep", "1", (char *) NULL);
    return 0;
}
```

2.9 Quelle est la valeur finale de a pour chaque processus? Combien de millions de fois la variable a est-elle incrémentée?

```
// ex02.c
#define MAX 100000000 // 100 millions
static uint64_t a = 0;

void *count(void *arg) {
    volatile uint64_t *var = (uint64_t *) arg;
    volatile uint64_t i;
    for (i = 0; i < MAX; i++) {
        *var = *var + 1;
    }
    return NULL;
}

int main(int argc, char **argv) {
    int p;

    count(&a);

    if ((p = fork()) < 0)
        return EXIT_FAILURE;

    count(&a);
    wait(NULL);

    printf("pid=%d a=%" PRId64 "\n", getpid(), a);
    return EXIT_SUCCESS;
}
```

2.10 Quel est l'affichage sur stdout produit par ce programme?

```
// ex04.c
int main(int argc, char **argv) {
    int i = 2;
    int j = 10;
    int p;

    while(--i && (p = fork())) {
        if (p < 0)
            exit(EXIT_FAILURE);
    }
    j = j + 2;
    if (p == 0) {
        i = i * 3;
        j = j * 3;
    } else {
        i = i * 2;
        j = j * 2;
    }
    printf("pid=%d i=%d j=%d\n", getpid(), i, j);
    return EXIT_SUCCESS;
}
```

Les questions suivantes requièrent de la programmation

2.11 Lanceur de commandes

Écrivez un programme qui lance, l'un à la suite de l'autre, les fichiers exécutables passés en paramètre. Un fichier exécutable est lancé uniquement si tous ceux qui le précèdent sont terminés. Avant de se terminer, le programme affiche à l'écran le nombre de lancements de fichiers exécutables qui ont échoué (parce que le fichier n'existe pas ou n'est pas exécutable).

2.12 Recherche en parallèle

Considérez un fichier nommé « foo ». Pour accélérer la recherche du mot « bar » dans le fichier « foo », le processus de départ crée quatre processus. Chaque processus fils créé effectue la recherche dans une des quatre parties du fichier en appelant la fonction « search » suivante :

bool search(char * file, char * word, int section) où :

- file est le nom du fichier, c'est-à-dire « foo »,
- word est le mot recherché, c'est-à-dire « bar » et
- section est la partie 1, 2, 3 ou 4 du fichier.

Cette fonction retourne 1 en cas de succès et 0 sinon. Au retour de la fonction « search », le processus fils transmet, en utilisant l'appel système **exit**, au processus père le résultat de la recherche (**exit(0)** en cas de succès, **exit(1)** en cas d'échec). Lorsque le processus père est informé du succès de l'un de ses fils, il tue tous les autres fils. Écrivez un programme C qui réalise le traitement. N'écrivez pas le code de la fonction « **search** ».

2.13 Arbre généalogique

Considérez le programme C ex06.c suivant:

```
int main(int argc, char **argv) {
    int pid;
    int i;

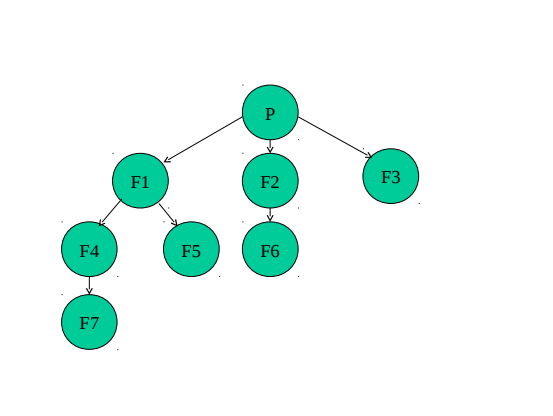
    // FIXME

    sleep(30);
    while (wait(NULL) >= 0);

    return EXIT_SUCCESS;
}
```

```
}
```

2.13.1 Complétez le code précédent de manière qu'il crée l'arborescence illustrée ci-dessus :



Les processus fils du dernier niveau se transforment en « sleep ». Les autres processus fils affichent à l'écran le message « continue » avant de poursuivre la création de processus.

2.13.2 Est-ce que le code complété engendre des processus zombies? Justifiez.

2.13.3 Tracez l'arborescence des processus créés par le programme si l'on supprime la ligne **while (wait(NULL) >= 0);** et à la fin du code des pères, l'on ajoute les lignes dans le « for » :

```
wait(NULL);  
exit(0);
```

2.14 Une grande famille

Considérez le programme C ex07.c suivant:

```
void fx(char *msg) {  
    printf("pid=%d %s\n", getpid(), msg);  
    exit(EXIT_SUCCESS);  
}  
  
void f1() {    fx("1"); }  
void f2() {    fx("2"); }  
void f3() {    fx("3"); }  
void f4() {    fx("4"); }
```

```

int main(int argc, char **argv) {
    pid_t p1, p2, p3, p4;

    if ((p1 = fork()) == 0) {
        if ((p2 = fork()) == 0) {
            f2();
        } else {
            f1();
        }
    } else {
        if ((p3 = fork()) == 0) {
            f3();
        } else {
            if ((p4 = fork()) == 0) {
                f4();
            }
        }
    }
    sleep(30);
    return EXIT_SUCCESS;
}

```

2.14.1 Tracez l'arborescence des processus créés par ce programme si les fonctions f1, f2, f3 et f4 se terminent par exit().

2.14.2 Est-ce que ce programme peut produire des processus zombies? Justifiez.

2.15 Jouer en équipe

Considérez le programme ex08.c suivant :

```

#define MAX 4

void play(int id) {
    printf("play id=%d\n", id);
    sleep(id);
}

int main(int argc, char **argv) {
    int i;
    int p;
    int status;

    // créer MAX processus fils
    // chaque processus execute la fonction play puis quitte

    while(1) {
        // attendre la fin d'un joueur
    }
}

```

```

        // creer un autre joueur de meme numero
    }
    return EXIT_SUCCESS;
}

```

2.15.1 Complétez le programme en ajoutant le code qui réalise exactement les traitements spécifiés sous forme de commentaires.

2.16 Rester vivant

Considérez le code ex09.c suivant :

```

void server() {
    FILE *log = fopen("server.log", "a");
    int d = 0;

    while(d < 10) {
        fprintf(log, "%d server is alive\n", d++);
        fflush(log);
        sleep(1);
    }
}

int main(int argc, char **argv) {
    int p;
    if ((p = fork()) == 0) {
        fclose(stdin);
        fclose(stdout);
        fclose(stderr);
        if ((p = fork()) == 0) {
            server();
        }
        return EXIT_SUCCESS;
    }
    wait(NULL);
    printf("server started\n");
    return EXIT_SUCCESS;
}

```

2.16.1 Quelle est le résultat du programme?

2.16.2 Quel est le parent du processus exécutant la fonction « server() »?