Séance machine 3 - fork / exec

S3 - M3101

2017-2018

1. Que fait `system()` ?

Fonction de bibliothèque, combine 3 appels système

- `fork()`, qui crée un nouveau processus
- `exec()`, qui exécute un fichier (exécutable)
- `waitpid()`, qui attend la fin d'un processus

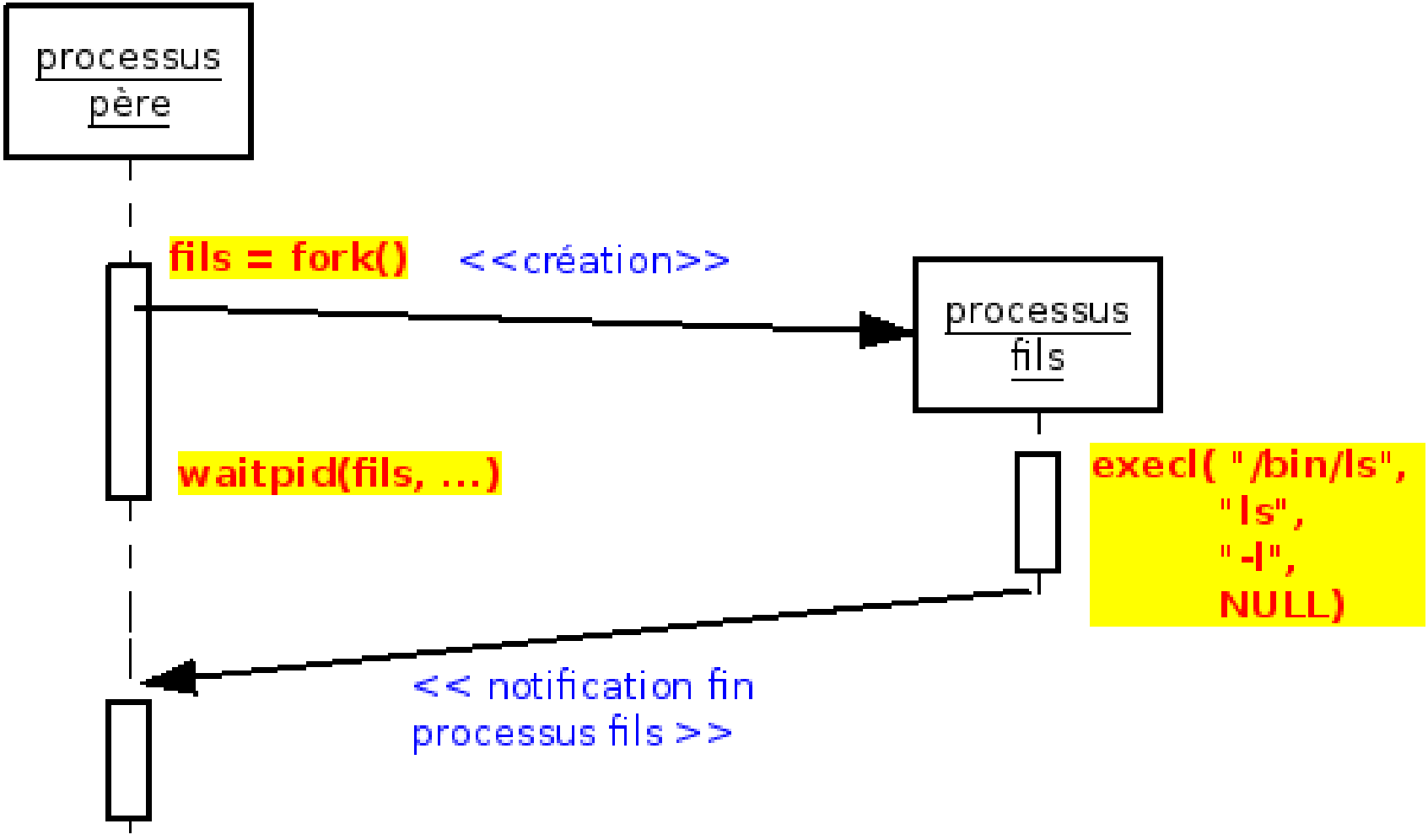
Déroulement de `system("une commande")`

1. créer un nouveau processus (fils)

2. le processus fils
   - cherche le fichier exécutable
   - le copie dans son espace mémoire
   - lance son exécution

3. le processus père
   - attend que le fils se termine

Illustration :



2. L'appel `fork()`

- Demande au système de créer un nouveau processus (fils)
- copie presque identique du processus appelant (père) :
  - même contenu de la mémoire,
  - mêmes fichiers ouverts, etc.

Différence : la fonction retourne

- 0 au processus fils
- le numéro du fils au père

Note : Il se peut aussi que le `fork()` échoue (retourne -1 au père).

Illustration

```
pid_t p = fork();
if (p == 0) {
    cout << "je suis le processus fils " << endl;
    ...
    exit (EXIT_SUCCESS);
}

cout << "je suis le processus père" << endl;
cout << "le processus fils a le numéro " << p << endl;
....
}
```

Travail avec `fork()`

Ecrire un programme C++ qui

- affiche 5 fois “tip”, avec un délai (`sleep`) de 3 secondes,
- après avoir lancé un processus fils qui affiche 10 fois “top” avec un délai de 2 secondes.

Application de `fork()`

Ajoutez au “shell” une commande qui affichera un message de rappel dans un délai indiqué (en secondes)

```
> rappel 30 aller manger
```

```
...
RAPPEL : aller manger
```

Remarquez (ps) l'apparition de *zombies*.

3. L'appel système `wait()` / `waitpid()`

La fonction `waitpid()`

- attend qu'un processus fils se termine,
- récupère un `int` qui combine plusieurs informations sur l'exécution du fils. `WEXITSTATUS` extrait le code de retour

```
pid_t fils = fork();
...
int status;
waitpid (fils, &status, 0);      // passage par adresse
cout << "le processus " << fils
     << " s'est terminé avec le code de retour "
     <<  WEXITSTATUS(status) << endl;
```

wait

Le troisième paramètre est une combinaison d'options. Voir la doc.

Il existe également un appel `wait` qui permet d'attendre un processus fils non spécifié. Il équivaut à `waitpid(-1, &status, 0)`.

4. Exercice `fork + wait` : la course de haies (TP + Maison)

On simule une course de haies 4x100 m entre 6 équipes.

Chaque équipe est simulée par un processus, avec tirage aléatoire de la durée

```
pour j de 1 à 4
| afficher "le coureur j de l'équipe n est parti"
| attendre de 8 à 11 secondes
afficher "l'équipe n est arrivée"
```

Version 1

Dans un premier temps, équipes identifiées par leur numéro de processus. Le `wait()` fera afficher les équipes avec leur rang

```
1. équipe #1234
2. équipe #1236
...
```

Éléments techniques nécessaires

- appel `getpid()` pour connaitre le numéro d'un processus
- nombres aléatoires entiers entre a et b :

```
srandom(time());
...
int r = a+ random() % (b-a+1);
```

A rendre la semaine suivante.

Version 2 : chutes

Chaque coureur a une chance sur 10 de **tomber**.

- Dans ce cas, le processus de l'équipe se termine avec `EXIT_FAILURE`.
- Et l'équipe ne figure pas dans la liste finale.

Version 3 : équipes avec noms

Le programme prend en paramètre les noms des équipes (paramètres 1 à `argc-1` de l' `argv` du `main`) :

```
$ course FRA USA ITA CHN
- le coureur 1 de FRA est parti
- le coureur 1 de USA est parti...

1. FRA
2. CHN
...
```

une table de correspondance permettra d'afficher le nom à partir du numéro de processus retourné par `wait()`.

Résumé

Dédoubler un processus

- `pid = fork()`

Attendre la fin d'un processus

- `waitpid(pid, &status, options)`
- `pid = wait(&status)`

Consultation du code de retour

- `retcode = WEXITSTATUS(status);`

5. Les appels “exec”

Exemple “execl”

```
if (fork()==0) {
    execl("/bin/ls", "ls", "-l", NULL);
}
int s;
wait( &s );
```

Déroulement du processus fils

- charge et exécute le programme `/bin/ls`,
- “`ls` et”`-l"` dans `argv[0]` et `argv[1]`,
- 2 dans `argc`

Fonctionnement de “exec”

- Le code du programme chargé **remplace** celui du processus
- quand le programme s'arrête, fin du processus
- la valeur transmise par `exit()` est le code de retour du processus fils

La famille exec

Par commodité, plusieurs fonctions avec un rôle similaire

- execl, execv, execvl, execvp

Paramètres transmis

- sous forme d'une liste : execl
- sous forme d'un tableau : execv

Programme indiqué

- par un chemin absolu
- par un chemin relatif (résolu par le PATH)

Exemples (C)

Liste de paramètres terminée par NULL

```
execl ("/bin/ls", "ls", "-l", NULL);
execlp(     "ls", "ls", "-l", NULL);
```

Tableau de paramètres

```
char * arg[] = { "ls", "-l", NULL};

execv ("/bin/ls", arg);
execvp(     "ls", arg);
```

Exemples (C++)

- Les fonctions exec attendent des chaines constantes
- C++ est moins permissif que C
- ajouter des conversions

```
// C++11
const char * a[] {
    "ls", "-l", nullptr
};

execv("/bin/ls",
      const_cast<char **>(a) );
```

6. Application : lancement de commandes par le shell

Intégrer le lancement direct d'un programme par le shell.

```
si premier mot est exit, help, cd ... (commande interne)
   l'exécuter
sinon {
   faire un fork
    - le fils construit un tableau d'arguments
        et appelle execvp(premier mot, tableau)
    - le pere attend la fin (waitpid)
    }
```

7. Application : lancement de commandes en arrière plan

Si le dernier “mot” de la commande est “&”,

- ne pas attendre le processus fils
- afficher son numero

```
12> emacs truc.txt &
[1234] emacs truc.txt
13>
```

Ajouter aussi une commande `wait nnn` qui se bloque en attendant la fin d'un processus

8. Application : gestion des travaux

Ajouter une table de correspondance entre

- numéros des processus fils lancés en arrière-plan
- commande correspondante

La commande “`jobs`” permettra de faire afficher cette table

```
18>jobs
[1234] emacs truc.txt
[1236] firefox
19>
```

Gestion des travaux (suite)

Objectif : quand un processus fils se termine,

- prévenir l'utilisateur
- mettre à jour la table des "jobs"

Moyen : quand un processus se termine

- le système envoie un **signal** `SIGCHLD` à son père
- le père peut associer une action à ce signal

Appel système

```
signal(SIGCHLD, une_fonction);
```

Exemple

```
void fin_fils(int sig) {
    pid_t p = wait(nullptr);
    cout << "fin de " << p << " !";
}

int main()
{
    signal(SIGCHLG, fin_fils);

    if (fork() == 0) {
        ...
    }
}
```

Mise en oeuvre sur l'exemple

Par exemple, la réception d'un signal

- fera afficher la ligne de commande concernée
- retirera le processus de la liste de travaux en arrière plan

9. Signaux

Signal :

- mécanisme de communication primitif, ne transmet qu'un numéro de signal
- permet à un processus de réagir à un évènement.
  - `SIGQUIT`,
  - `SIGCHLD`,
  - ‘SIGSTOP,
  - `SIGCONT`,
  - `SIGSEGV`,
  - `SIGUSR1`, ...

Déclenchement/réception d'un signal

Déclenchement par

- evènement matériel (violation d'accès mémoire, ...)
- evènement du système (un processus s'est terminé, ...)
- appel à la fonction `kill(pid,sig)`

Réception par un processus

- comportement par défaut : dépend du signal
  - ignorer, mettre fin au processus
  - le stopper (pause), le relancer
- modification par : `signal(sig, handler)`

Exemple : SIGINT

On tape controle-C pendant l'exécution d'un programme :

- le shell l'intercepte
- il envoie le signal SIGINT au processus qui tourne

Lorsque le "handler" est lancé,

- le comportement par défaut est rétabli
- réarmer le signal ?

10. Travail en groupe

Faire un programme qui

- incrémente et affiche un compteur quand il reçoit SIGUSR1
- s'arrête de lui-même au bout de 30 secondes

À découvrir :

- la commande "`kill`",
- fonction `alarm()`
- attendre à ne rien faire : `while(true} sleep(1);` ou mieux `pause();`

Exercice à faire

A partir d'une fonction qui teste si un nombre est premier, programme qui

- prend en paramètre une durée (conseil `atoi(argv[1])`)
- regarde si les nombres 2,3,4 . . . sont premiers
- afficher le plus grand qu'il a trouvé au bout de la durée.