Shell

Gestion des processus et entrées-sorties

Ensimag, 2A, édition 2013-2014

5 septembre 2014

TP 2 : le shell

Opérations sur les processus
- Processus
- Création de processus
- Recouvrement de programme
- Attente de la terminaison

les IO
- Rappel sur les processus
- Le shell
- Le pipe

Shell

Le but est de faire un *shell* de commandes (interpréteur simpliste de commandes fourni)

- lancer les programmes demandés avec leurs arguments,
- gérer l'attente éventuelle et la terminaison des processus.

Il y a aussi : les redirections d'entrées-sorties (`<`, `>`, `|`) et quelques variantes (libreadlines, signaux, joker, libcurses, Ctrl-Z/fg/bg, etc.).

Le sujet est sur ensiwiki ! ! !

Attention : le sujet est sur ensiwiki !

De l'aide ?

- `man fork`
- `man execvp`
- `man 2 wait`
- ...
- En cas d'ambiguité :
  - `whatis` *commande* puis
  - `man` *N commande*
  - Exemple : `man 2 open` pour le open du C, `man 3 open` pour le open de perl ...)

Processus

Definition (Qu'est-ce qu'un processus ?)

Processus

Definition (Qu'est-ce qu'un processus ?)

Un processus est un *programme* en *exécution*

Programme

Qu'est-ce qu'un programme ?

Programme

Qu'est-ce qu'un programme ?

- du code
- des données

Exécution

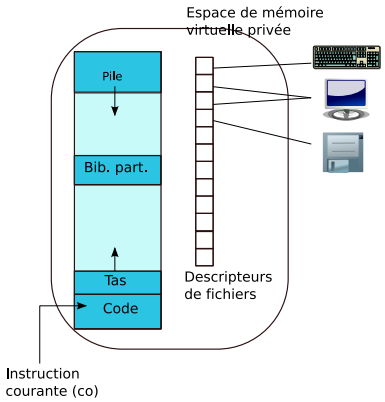
Qu'est-ce que l'exécution d'un programme ?

Exécution

Qu'est-ce que l'exécution d'un programme ?

- une instruction courante (compteur ordinal)
- un état courant (mot d'état : division par 0, masquage des interruptions, résultats de tests de branchements, etc.)
- de la mémoire (RAM + registres)
- des entrées-sorties

Un processus dans un OS moderne



Processus

- Un espace de mémoire virtuelle privée contenant :
  - le code,
  - la pile : variables locales des fonctions récursives,
  - le tas : malloc/free
  - des segments de mémoire partagée : bibliothèques de fonctions partagées
- Une instruction courante (compteur ordinal)
- Des registres
- Un ensemble de descripteurs de fichiers : entrées-sorties vers l'écran, le clavier, des fichiers, le réseau, etc.

Création de processus sous UNIX

Création par l'appel système `fork`

Mais que fait `fork` ? Comment savoir ?

Création de processus sous UNIX

Création par l'appel système `fork`

Mais que fait `fork` ? Comment savoir ?

```
FORK(2)          Linux Programmer's Manual          FORK(2)
NAME
       fork - create a child process
SYNOPSIS
       #include <unistd.h>
       pid_t fork(void);
DESCRIPTION
       fork()  creates  a  new  process by duplicating the
       calling process.  The new process, referred  to  as
       the  child,  is  an  exact duplicate of the calling
       process, referred to as the parent, except for  the
       following points:
```


Création de processus sous UNIX

`int fork()` : création par copie

La création se fait par copie à l'identique du processus qui appelle la fonction `fork`.

Création de processus sous UNIX

`int fork()` : création par copie

La création se fait par copie à l'identique du processus qui appelle la fonction `fork`.

La règle : copie *TOUT*, état de la mémoire (programme, données, pile, tas, la plupart des segments partagées), instruction courante, état des registres et des entrées sorties.

Création de processus sous UNIX

`int fork()` : création par copie

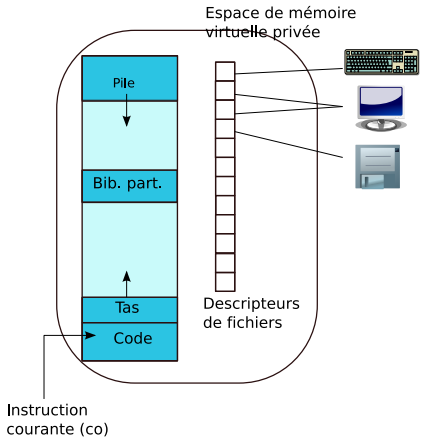
La création se fait par copie à l'identique du processus qui appelle la fonction `fork`.

La règle : copie *TOUT*, état de la mémoire (programme, données, pile, tas, la plupart des segments partagées), instruction courante, état des registres et des entrées sorties.

Les exceptions : *la valeur de retour de* `fork` *(0 dans le fils, PID du fils dans le père)*, l'identification du processus (numéro unique, PID), les verrous, les statistiques (`getrusage()`), les alarmes (`setitimer()`)

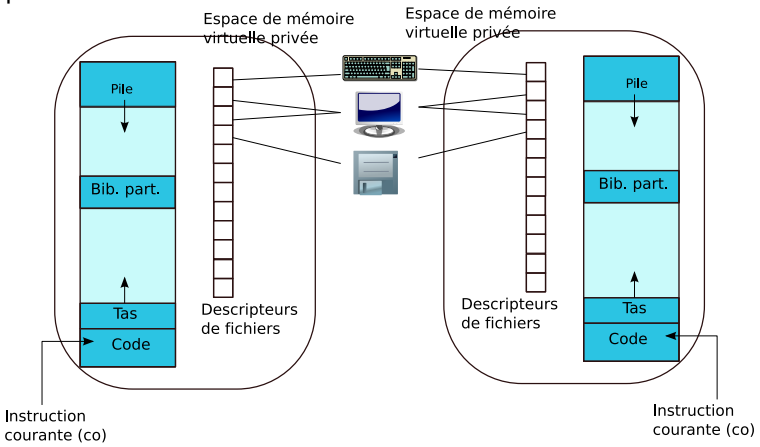
Création de processus avec fork

Le processus père commence l'exécution de fork.



Création de processus avec fork

Le processus père commence l'exécution de fork. Les deux processus terminent leur exécution de la fonction fork.



Création d'un processus avec fork

```
pid_t pid;
switch( pid = fork() ) {
case -1:
  perror("fork:"); break;
case 0:
  printf("Ahhh !!!!!\n"); break;
default:
  printf("%d, je suis ton père\n", pid);
  break;
}
```


Créations multiples

Comment créer *n* processus ?

Comment créer *n* processus avec la fonction fork ?

Créations multiples

Comment créer *n* processus ?

Comment créer *n* processus avec la fonction fork ?

Boucle simple

Le code suivant ne crée pas *n* processus !

```
for(i=0; i< n; i++) {
  fork();
}
```


Créations multiples

Comment créer *n* processus ?

Comment créer *n* processus avec la fonction fork ?

Boucle simple

Le code suivant ne crée pas *n* processus !

```
for(i=0; i< n; i++) {
  fork();
}
```

Il crée 2^n processus ! (CAUTION : fork bomb ! Rappel aux imprudents : les processus ont un propriétaire :-) !)

Créations multiples

La bonne solution : tester la valeur de retour de fork

```
for(i=0; i< n; i++) {
  int pidfils;
  pidfils = fork();
  if (! pidfils) // si pidfils est égale à 0
    break;
}
```


Filiation

ou bien

```
for(i=0; i< n; i++) {
  int pidfils;
  pidfils = fork();
  if (pidfils) // si pidfils est différent de 0
    break;
}
```

Quelle est la différence entre les deux codes ?

Quel impact sur la filiation ?

Le recouvrement

`execvp, execve, execlp, execle` : le recouvrement

L'appel à `exec` remplace le programme (code + données) du processus par un autre programme. Puis le processus recommence au début du nouveau programme (main).

Il est souvent utilisé après un fork.

Le recouvrement

`execvp, execve, execlp, execle` : le recouvrement

L'appel à `exec` remplace le programme (code + données) du processus par un autre programme. Puis le processus recommence au début du nouveau programme (main).

Il est souvent utilisé après un fork.

Pas de création !

`Exec` ne crée pas un nouveau processus ! Il remplace le programme d'un processus en cours de route

Le recouvrement

`execvp, execve, execlp, execle` : le recouvrement

L'appel à `exec` remplace le programme (code + données) du processus par un autre programme. Puis le processus recommence au début du nouveau programme (main).

Il est souvent utilisé après un fork.

Pas de création !

`Exec` ne crée pas un nouveau processus ! Il remplace le programme d'un processus en cours de route

Ne revient pas !

Un appel réussi à `exec` ne revient jamais (sauf erreur) !

L'attente

`wait, waitpid` : l'attente de la terminaison

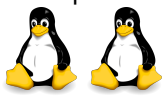
Les fonctions `pid_t wait(int *status)` ou `pid_t waitpid(pid_t pid, int *status, int options)` permettent à un processus parent d'attendre la fin d'un de ses fils directs. Le processus père récupère la valeur entière donnée en argument à `exit(int)` ou en retour du `main`.

En résumé

- `fork` permet la création d'un nouveau processus par copie

En résumé

- `fork` permet la création d'un nouveau processus par copie



En résumé

- `fork` permet la création d'un nouveau processus par copie

- `exec` change le programme exécuté par un processus

En résumé

- `fork` permet la création d'un nouveau processus par copie

- `exec` change le programme exécuté par un processus

En résumé

- `fork` permet la création d'un nouveau processus par copie

- `exec` change le programme exécuté par un processus

- `wait` : le processus père attend la fin de son fils.

La création de processus

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

La création de processus

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

Pour faire des modifications sur les entrées-sorties

La création de processus

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

Pour faire des modifications sur les entrées-sorties

1. Le nouveau processus créé par le fork a les mêmes entrées-sorties que son père,

3. l'execvp ne change pas les entrées-sorties

La création de processus

Création en deux étapes

- La création par copie à l'identique (fork),
- Le remplacement du programme en cours (exec).

Pourquoi séparer les opérations ?

Pour faire des modifications sur les entrées-sorties

1. Le nouveau processus créé par le fork a les mêmes entrées-sorties que son père,
2. **FAIRE LES CHANGEMENTS DES I/O ICI**
3. l'execvp ne change pas les entrées-sorties

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros.

Quelles sont les opérations classiques sur les entrées-sorties ?

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert
- la lecture (read(int fd, ...), recv(int fd, ...)),

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert
- la lecture (read(int fd, ...), recv(int fd, ...)),
- l'écriture (write(int fd, ...), send(int fd, ...)),

TP 2 : le shell Opérations sur les processus les IO

Les descripteurs de fichiers

Pour un processus, (presque) toutes les entrées-sorties passent par des descripteurs de fichiers. Un descripteur est accédé par son numéro. Toutes les opérations utilisent ces numéros.

Quelles sont les opérations classiques sur les entrées-sorties ?

- l'ouverture (int `open(...)`, int socket(...), pipe(...)) qui renvoie le numéro du descripteur ouvert
- la lecture (read(int fd, ...), recv(int fd, ...)),
- l'écriture (write(int fd, ...), send(int fd, ...)),
- la fermeture (close(int fd), shutdown(int fd,...)).

TP 2 : le shell Opérations sur les processus les IO

La gestion des entrées-sorties

Definition (Les entrées-sorties standards)

Par convention, chaque processus s'attend à avoir à son démarrage trois descripteurs d'entrées-sorties ouverts :

- l'entrée standard (stdin), dans le descripteur 0,
- la sortie standard (stdout), dans le descripteur 1,
- la sortie d'erreur standard (stderr), dans le descripteur 2.

Les fonctions des bibliothèques standards

Elles ne s'occupent pas de savoir vers quoi sont ouverts les descripteurs : terminal, fichiers, sockets réseaux, etc.
`printf("toto")` **réalise un** `write(1, "toto", 4)`

TP 2 : le shell Opérations sur les processus les IO

Le shell

Le shell peut donc rediriger à la demande les entrées-sorties standards des processus. Pour cela il faut :

1. ouvrir un nouveau descripteur γ (open) vers l'entrée-sortie voulue, par exemple un fichier,
2. fermer le descripteur standard (close),
3. dupliquer le descripteur γ dans le descripteur standard (dup ou dup2),
4. fermer le descripteur γ qui est en double (close),

Les tuyaux (pipe)

- Les tuyaux sont des outils de synchronisation de type producteur-consommateur qui connectent la sortie d'un processus avec l'entrée d'un autre.

  `ls -R | egrep "\.c$" | less`
- Comme le tuyau est de petite taille (quelques kilos), il synchronise les "vitesses" de production et de consommation : la puissance de calcul est répartie et l'occupation mémoire constante, indépendamment la longueur du flot.
- Un tuyau est un objet anonyme, donc il n'est connecté qu'avec les processus qui ont un descripteur ouvert sur lui.

`int pipe(int fds[2])`

1. L'appel `int pipe(int fds[2])` crée un tuyau et renvoie les numéros des deux descripteurs vers le tuyau (fds[0] pour lire, fds[1] pour écrire)
2. ensuite on peut faire des *fork* pour créer des processus connectés au pipe.

Détection de la fin de l'écriture dans un pipe

Une communication par tuyau est terminée quand :

- il est vide,
- aucun processus ne peut écrire dedans (tous les descripteurs en écriture sont maintenant fermés), y compris les descripteurs des processus bloqués en lecture dans le pipe.

`int pipe(int fds[2])`

```
int res;
char *arg1[]={"ls","-R", 0};
char *arg2[]={"egrep","\.c$", 0};
int tuyau[2];

pipe(tuyau);
if((res=fork())==0) {
  dup2(tuyau[0], 0);
  close(tuyau[1]); close(tuyau[0]);
  execvp(arg2[0],arg2);
}
dup2(tuyau[1], 1);
close(tuyau[0]); close(tuyau[1]);
execvp(arg1[0],arg1);
```


valgrind et gdb

Valgrind vérifie vos allocations et initialisations de Variables

L'utilisation systèmatique de valgrind permet de détecter certans bugs dès leur introduction (copie de chaîne de caractère, paramètres d'appels systèmes).

```
valgrind ./monshell
```

gdb permet de tester l'état d'un processus

On peut attacher gdb à un processus déjà en execution et inspecter son état.

```
gdb ./monshell 1234
```

pour un processus de PID 1234

valgrind et gdb

On peut même faire les deux en même temps

On peut accrocher gdb à un processus en exécution au moment de la détection d'un bug par valgrind.

```
valgrind --db-attach=yes ./monshell
```