Les processus (suite)

Louis Mandel
louis.mandel@lri.fr
Laboratoire de Recherche en Informatique
Université Paris-Sud 11

IFIPS
Cycle ingénieur de la filière étudiant
année 2007/2008

Informations pratiques

- La semaine prochaine
  - mercredi 19 mars : TP à 8h15
  - jeudi 20 mars : Cours à 10h30
  - vendredi 21 mars : Cours à 8h45

La mutation de processus

- Mutation = remplacement du code à exécuter
  - c'est le même processus à exécuter
  - on parle de recouvrement
- Le Nouveau programme hérite de l'environnement système
  - même numéro de processus (PID, PPID)
  - héritage des descripteurs de fichiers ouverts (sauf ...)
  - pas de remise à zéro du temps d'exécution
  - etc.
- Famille de fonction :
  - appel système : `execve`
  - fonctions de bibliothèque `exec*` (`#include <unistd.h>`)

- Appel système
  - `int execve(const char *path,`
    `char *const argv[],`
    `char *const envp[]);`
  - Comportement
    - Exécution du programme `path`
    - avec les arguments `argv`
    - et les variables d'environnement `envp`
- Exemple :

```
int main (int argc, char **argv, char **envp) {
  execve(argv[1], argv, envp);
  perror("recouvrement");
  return 1;
}
```

La famille `exec*`

- Plusieurs fonctions de bibliothèque
  - écrites au dessus de `execve`
  - différents moyens de passer les paramètres : tableau, liste
  - spécification ou non des variables d'environnement
- Deux spécifications de la commande à exécuter
  - chemin complet (absolu ou relatif) : `path`
  - nom de commande : `file`
    - recherche de la commande dans `$PATH`
- ```
  #include <unistd.h>
  int execl(const char *path, const char *arg, ... /*, (char *)0 */);
  int execle(const char *path, const char *arg, ...
      /*, (char *)0, char *const envp[] */);
  int execv(const char *path, char *const argv[]);
  int execlp(const char *file, const char *arg, ... /*, (char *)0 */);
  int execvp(const char *file, char *const argv[]);
  ```
  ```

La famille `exec*` (`execs.c`)

- Exemples d'appels d'`exec*`

```
char *args[] = { "ls", "a.out", "/tmp", (char *)0 };

execv("/bin/ls", args);

execvp("ls", args);

execl("/bin/ls", "ls", "a.out", "/tmp", (char *)0);

execlp("ls", "ls", "a.out", "/tmp", (char *)0);
```

Combinaison de `fork` et `exec`

Combinaison de `fork` et `exec`

- Faire exécuter un programme par un nouveau processus
  - Réalisation en deux étapes :
    1. clonage : `fork`
    2. mutation : `exec*`
- Souplesse par la combinaison des deux primitives
  - réalisation d'opérations entre le `fork` et l'`exec`
  - exemple : redirection des entrées/sorties standard

Exemple du shell

- Le shell fonctionne par combinaison de `fork` et `exec`
- Exemple (`pid`) :

```
#include <stdio.h>
int main () {
  printf("Je suis %d le fils de %d", getpid(), getppid());
  return 0;
}
```

- Une erreur dans un programme n'affecte pas le shell
- changement d'attributs dans le programme n'affecte pas le shell
- sauf pour les commandes internes : `cd` ...

Héritage des descripteurs lors de la mutation (`cloexec.c`)

- Les descripteurs de fichiers sont hérités par le nouveau programme
  - largement utilisé par le shell pour assurer les redirections
  - exemple : `--> cmd > out`
    - le shell associe le descripteur 1 (`STDOUT_FILENO`) à un fichier `out` (cf. `dup`, `dup2`)
    - le programme `cmd` écrit sur `STDOUT_FILENO` qui référence `out`
- Il est possible de changer ce comportement avec l'option *close on exec* sur le descripteur
  - Positionnement de `FD_CLOEXEC` par un appel à `fcntl`
  -
```
int main(int argc, char **argv) {
  fcntl(STDOUT_FILENO, F_SETFD,
        fcntl(STDOUT_FILENO, F_GETFD, 0) | FD_CLOEXEC);
  execvp(argv[1], argv+1);
}
```

redirection des entrées/sorties : dup2

- Création d'un descripteur synonyme d'un autre
- Les deux descripteurs partagent la même entrée dans la table des fichiers ouverts
- `#include <unistd.h>`
  `int dup2(int oldd, int newd);`
  - force `newd` à devenir un synonyme de `oldd`
  - ferme préalablement le descripteur `newd` si nécessaire

```
void print_inode(char *msg, int fd) {
  struct stat st;
  fstat(fd, &st);
  printf(stderr, "\t%s : inode %d", str, st.st_ino);
}
```

dup2 : **exemple** (dup2pr.c)

```
int main (int argc, char ** argv) {
  int fdin, fdout;
  fdin = open(argv[1], O_RDONLY);
  fdout = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
  fprintf(stderr, "Avant dup2 : \n");
  print_inode("fdin", fdin); print_inode("fdout", fdout);
  print_inode("stdin", STDIN_FILENO); print_inode("stdout", STDOUT_FILENO);

  dup2(fdin, STDIN_FILENO);
  dup2(fdout, STDOUT_FILENO);

  fprintf(stderr, "Apres dup2 : \n");
  print_inode("fdin", fdin); print_inode("fdout", fdout);
  print_inode("stdin", STDIN_FILENO); print_inode("stdout", STDOUT_FILENO);
  return 0; }
```