
Les processus

Polytech Paris-Sud
Cycle ingénieur de la filière étudiant

Louis Mandel
Université Paris-Sud 11
Louis.Mandel@lri.fr

année 2011/2012

Processus

- ▶ Processus = Instance d'un programme en cours d'exécution
 - ▷ plusieurs exécutions de programmes
 - ▷ plusieurs exécutions d'un même programme
 - ▷ plusieurs exécutions « simultanées » de programmes différents
 - ▷ plusieurs exécutions « simultanées » du même programme
- ▶ Ressources nécessaire à un processus
 - ▷ Ressources matérielles : processeur, périphériques ...
 - ▷ Ressources logicielles :
 - ▷ code
 - ▷ contexte d'exécution : compteur ordinal, fichiers ouverts ...
 - ▷ mémoires
- ▶ Mode d'exécution
 - ▷ utilisateur
 - ▷ noyau (ou système ou superviseur)

Processus

- ▶ Information sur les processus
 - ▷ Commandes shells
 - ▷ ps
 - ▷ pstree
 - ▷ Système de fichier
 - ▷ /proc

Attributs d'un processus

- ▶ Identification
 - ▷ numéro du processus (*process id*) : `pid_t getpid(void);`
 - ▷ numéro du processus père : `pid_t getppid(void);`
- ▶ Propriétaire réel
 - ▷ Utilisateur qui a lancé le processus et son group
 - `uid_t getuid(void);`
 - `gid_t getgid(void);`
- ▶ Propriétaire effectif
 - ▷ Détermine les droits du processus
 - `uid_t geteuid(void);`
 - `gid_t getegid(void);`
 - ▷ Le propriétaire effectif peut être modifié
 - `int setuid(uid_t uid);`
 - `int setgid(gid_t gid);`

Attributs d'un processus

- ▶ Répertoire de travail

- ▷ Origine de l'interprétation des chemins relatifs

- `char *getcwd(char *buf, size_t size);`

- ▷ Peut être changé

- `int chdir(const char *path);`

► Masque de création des fichiers

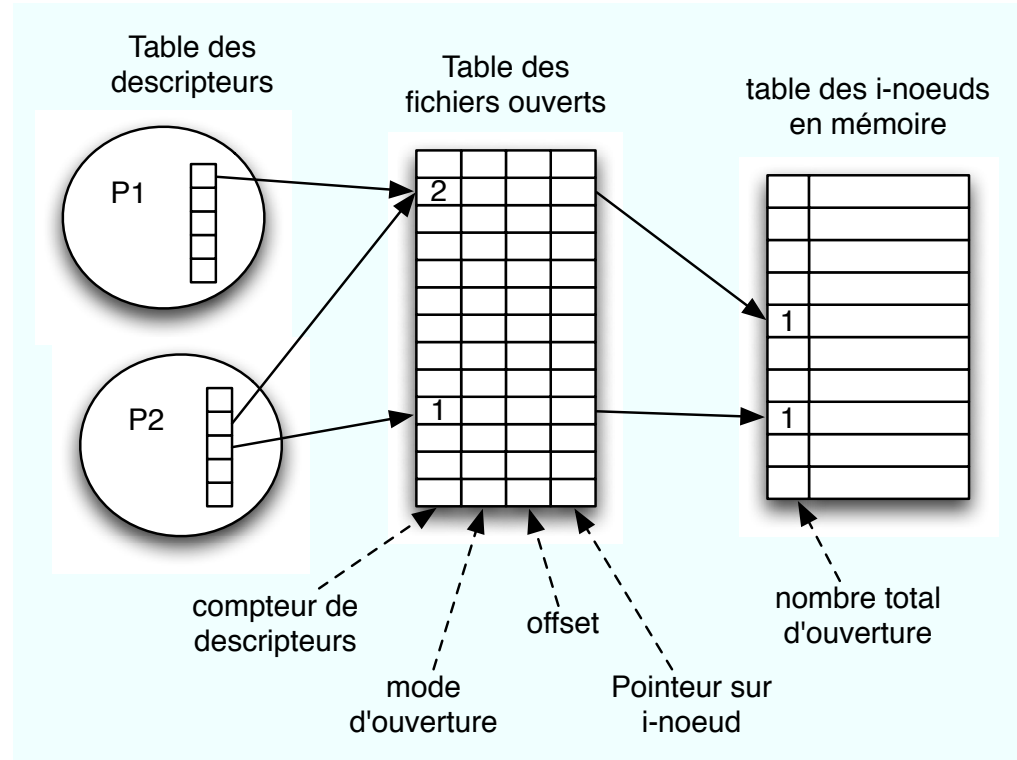
- ▷ `mode_t umask(mode_t mask);`
- ▷ fixe le masque de création de fichiers à la valeur `mask & 0777`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv) {
    if (argc != 3) return 1;
    if (-1 == creat(argv[1], S_IRWXU|S_IRWXG|S_IRWXO )) {
        perror("creat"); return 1;
    }
    umask(S_IRWXG|S_IRWXO);
    if (-1 == creat(argv[2], S_IRWXU|S_IRWXG|S_IRWXO )) {
        perror("creat"); return 1;
    }
    exit(EXIT_SUCCESS);
}
```

Attributs d'un processus

► Table des descripteurs de fichiers



- ▷ Limite : `OPEN_MAX`
- ▷ Acquisition : `open`, `creat`, `dup`, `dup2`, `fcntl`, `pipe` et `socket`
- ▷ Libération : `close`

Attributs d'un processus

► Variables d'environnement

▷ `extern char **environ`

▷ `ou int main (int argc, char **argv, char **envp);`

▷ Exemple :

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

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

```
int main (int argc, char **argv, char **envp) {
```

```
    while (*envp != NULL) { printf("%s\n", *envp); envp++; }
```

```
    exit(EXIT_SUCCESS);
```

```
}
```

▷ Récupérer la valeur d'une variable :

```
char *getenv(const char *var_name);
```

▷ Changer la valeur d'une variable :

```
int putenv(const char *string);
```


Attributs d'un processus

► Consultation et changement d'attributs variables

- ▷ `int getrlimit(int resource, struct rlimit *rlim);`
- ▷ `int setrlimit(int resource, const struct rlimit *rlim);`
 - ▷ Exemple de ressources : `RLIMIT_CORE`, `RLIMIT_CPU`, etc.
 - ▷ `struct rlimit {`
 - `rlim_t rlim_cur; /* limite souple */`
 - `rlim_t rlim_max; /* limite stricte (plafond de rlim_cur) */`
 - `};`
- ▷ `long sysconf (int name);`
 - ▷ Exemple de noms : `_SC_CLK_TCK`, `_SC_OPEN_MAX`, etc.

Gestion de processus

► Parallélisme asynchrone

- ▷ Plusieurs processus tournent en parallèle sur la machine.
- ▷ Sur une machine mono-processeur, le système se charge d'entrelacer les processus en donnant un peu de temps à chacun (ordonnancement) : donne l'illusion du parallélisme.
- ▷ L'entrelacement est imprévisible et non reproductible : l'exécution d'un programme peut être interrompue de façon autoritaire par le système pour donner la main à un autre.
- ▷ *Race condition* : Deux entrelacements possibles de deux processus changent le sens du programme (de façon non voulue) : accès aux ressources, lecture/écriture d'une même donnée, etc.
- ▷ Changer la priorité (ajoute une valeur de gentillesse, i.e. diminue la priorité. Valeurs négatives possibles avec privilège).
 - ▷ `int nice(int inc);`

Création de processus

- La création de processus se fait par clonage

- `#include <sys/types.h>`

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

```
pid_t fork(void);
```

- Exemple :

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

```
#include <sys/types.h>
```

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

```
int main() {
```

```
    printf("avant fork\n");
```

```
    fork();
```

```
    printf("apres fork\n");
```

```
    return 0;
```

```
}
```

- ▶ Le processus est une copie du processus père
- ▶ à l'exception de
 - ▷ la valeur de retour de fork
 - ▷ son identité pid et de celle de son père ppid
- ▶ Exemple :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t id;
    id = fork();
    printf("id = %d, pid = %d, ppid = %d\n",
           id, getpid(), getppid());
    return 0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t status;
    status = fork();
    switch (status) {
    case -1 :
        perror("Creation de processus");
        return 1;
    case 0 : // Code du fils
        printf("[%d] Je viens de naitre\n", getpid());
        printf("[%d] Mon père est %d\n", getpid(), getppid());
        break;
    default : // Code du pere
        printf("[%d] J'ai engendre\n", getpid());
        printf("[%d] Mon fils est %d\n", getpid(), status);
    }
    printf("[%d] Je termine\n", getpid());
    exit(EXIT_SUCCESS);
}
```

- ▶ Comme la mémoire est copiée :
 - ▷ les données sont copiées

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int glob = 1;
int main() {
    int loc = 1;
    switch (fork()) {
        case -1 : perror("Creation de processus"); return 1;
        case 0 :
            glob++; loc++;
            printf("Fils : (%d, %d)\n", glob, loc);
            break;
        default :
            sleep(1);
            printf("Pere : (%d, %d)\n", glob, loc);
    }
    printf("[%d] Je termine\n", getpid());
}
```

- ▶ Comme la mémoire est copiée :
 - ▷ les buffers d'écriture de la bibliothèque standard d'entrées/sorties sont dupliqués

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    printf("avant ");
    fork();
    printf("apres\n");
    return 0;
}
```

- ▷ Il faut vider les buffers avant fork (par un appel à `fflush`)

- ▶ Comme la mémoire est copiée :
 - ▷ les références vers les ressources systèmes sont copiées
 - ▷ les ressources systèmes sont partagées.

```
int main(int argc, char *argv[]) {
    int desc;  char buf [10];
    if (-1 == (desc = open(argv[1], O_RDONLY))) { perror("open"); exit(1); }
    switch (fork()) {
    case -1 : perror("Creation de processus"); return 1;
    case 0 :
        if (-1 == read(desc, buf, 3)) { perror("read"); exit(1); }
        buf[3] = '\0'; printf("Fils : '%s'\n", buf);
        break;
    default :
        sleep(1);
        if (-1 == read(desc, buf, 3)) { perror("read"); exit(1); }
        buf[3] = '\0'; printf("Pere : '%s'\n", buf);
    }
    close(desc); exit(0);
}
```

- ▶ Attributs non copiés :
 - ▷ Numéro de processus
 - ▷ Numéro de processus du père
 - ▷ Temps d'exécution
 - ▷ Priorité du processus
 - ▷ Verrous sur les fichiers
- ▶ Coût de la copie mémoire ?
 - ▷ Sémantique = copie
 - ▷ Performance = pas de copie systématique
 - ▷ *copy on write* des pages mémoires

Terminaison des fils : `wait`

- ▶ Terminaison d'un processus
 - ▷ Appel système : `_exit`
 - ▷ Appel à la fonction de bibliothèque : `exit`
 - ▷ Positionnement d'une valeur de retour
- ▶ Le processus père peut consulter la valeur de retour
- ▶ Attente de la terminaison d'un fils
 - ▷ `pid_t wait(int *pstatus);`
 - ▷ retourne le PID de fils ou -1 en cas d'erreur (n'a pas de fils)
 - ▷ bloquant si aucun fils n'a terminé
 - ▷ `*pstatus` renseigne sur la terminaison du fils

Terminaison des fils : `wait`

- ▶ Renseignements concernant la terminaison d'un fils
 - ▷ Rangés dans l'entier `status` pointé par `pstatus`
 - ▷ Raison de la terminaison
 - ▷ le processus a fait un `exit` : `WIFEXITED(status)`
 - ▷ le processus a reçu un signal : `WIFSIGNALED(status)`
 - ▷ le processus a été stoppé : `WIFSTOPPED(status)`
 - ▷ valeur de retour (si `WIFEXITED(status)`)
 - ▷ 8 bits de poids faibles
 - ▷ accessible par la macro `WEXITSTATUS(status)`
 - ▷ numéro du signal ayant provoqué la terminaison (si `WIFSIGNALED(status)`) / l'arrêt (si `WIFSTOPPED(status)`)
 - ▷ accessible par les macros `WTERMSIG(status)` / `WSTOPSIG`

```
int main(int argc, char **argv) {
    int status; pid_t pidz;
    switch (fork()) {
    case -1 : perror("Creation de processus"); exit(EXIT_FAILURE);
    case 0 :
        printf("[%d] fils eclaire\n", getpid());
        exit(42);
    default :
        if (-1 == (pidz = wait(&status))) { perror("wait"); exit(EXIT_FAILURE); }
        if (WIFEXITED(status)) {
            printf("[%d] mon fils %d a termine normalement\n", getpid(), pidz);
            printf("[%d] code de retour : %d\n", getpid(), WEXITSTATUS(status));
        } else {
            printf("[%d] mon fils %d a termine anormalement\n", getpid(), pidz);
        }
    }
    exit(EXIT_SUCCESS);
}
```

Processus orphelins

- ▶ La terminaison d'un processus parent ne termine pas ses processus fils
 - ▷ les processus fils sont orphelins
- ▶ Le processus initial (PID 1) récupère les processus orphelins

```
int main(int argc, char **argv) {
    switch (fork()) {
        case -1 : perror("Creation de processus"); return 1;
        case 0 :
            printf("[%d] Pere : %d\n", getpid(), getppid());
            sleep(2);
            printf("[%d] Pere : %d\n", getpid(), getppid());
            exit(0);
        default :
            sleep(1);
            printf("[%d] fin du pere\n", getpid());
            exit(0);
    }
}
```

Processus zombies

- ▶ Zombi = état d'un processus
 - ▷ ayant terminé
 - ▷ non réclamé par son père
- ▶ Il faut éviter les zombis
 - ▷ Le système doit garder des informations relatives aux processus pour les retournées aux pères.
 - ▷ Encombre la mémoire
- ▶ Comment éviter les zombis si le père ne s'intéresse pas à la terminaison de ses fils ?

Mutation de processus

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 fonctions :
 - ▷ 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+1, 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 execlp(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[]);
```

► Exemples d'appels d'exec*

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

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

```
execvp("ls", args);
```

```
execl("/bin/ls", "ls", "a.out", "/tmp", NULL);
```

```
execlp("ls", "ls", "a.out", "/tmp", NULL);
```

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 ...

- ▶ 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", msg, st.st_ino);  
}
```

```
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; }
```