



Cours de système

Les processus

Sébastien Paumier



L'idée

- faire croire à un programmeur C qu'il dispose de toute la machine, pour qu'il n'ait pas à changer sa façon de travailler
- pour donner cette illusion, le système doit disposer d'un type d'objet particulier: le processus



Définition

- concept=activité (exécution d'un programme)+données pour la gérer
- une partie est visible par l'utilisateur:
 - instructions, pile, tas
- une partie est réservée au système, le PCB (Process Control Block) stocké dans l'espace du noyau:
 - contexte du processus, table des descripteurs, table des ouvertures de fichiers, etc



Caractéristiques

- ne pas confondre processus et programme !
 - un même programme peut avoir plusieurs exécutions simultanées
- les processus sont protégés les uns des autres par le système
- pour communiquer, ils doivent passer par des appels système (IPC)



Le premier processus

- au boot du système, il n'y a qu'une seule chose qui s'exécute, le main du boot, considéré comme la tâche n°0
- initialisation du système puis création du premier processus: la tâche *init* n°1
- puis, démarrage du scheduler, avant de faire rentrer le main du boot dans une boucle infinie (*idle task*)

```
init/main.c: rest_init
```



Création d'un processus

- plutôt que d'avoir un constructeur très compliqué, on fonctionne par héritage à partir du processus courant

kernel/fork.c

```
p->state = TASK_RUNNING;
p->pid = last_pid;
p->father = current->pid;
p->counter = p->priority;
p->signal = 0;
p->alarm = 0;
p->leader = 0;
p->utime = p->stime = 0;
p->cutime = p->cstime = 0;
p->start_time = jiffies;
p->tss.back_link = 0;
p->tss.esp0 = PAGE_SIZE + (long) p;
p->tss.ss0 = 0x10;
p->tss.eip = eip;
p->tss.eflags = eflags;
p->tss.eax = 0;
p->tss.ecx = ecx;
p->tss.edx = edx;
p->tss.ebx = ebx;
p->tss.esp = esp;
...
```



Création d'un processus

- `pid_t fork(void);`
- crée une copie du processus courant et renvoie:
 - -1 en cas d'erreur
 - 0 si on est dans le processus fils
 - le PID du fils (>0) si on est dans le père
- on ne sait pas si c'est le père ou le fils qui va être ordonnancé en premier après, ça dépend du scheduler !



Création d'un processus

- un processus peut avoir zéro ou plusieurs fils, mais un seul parent
- PID du processus: `pid_t getpid(void);`
- PID du père: `pid_t getppid(void);`
- usage prototypique:

```
switch (n=fork()) {  
  case -1: perror("fork"); ...  
  case 0: /* code du fils */  
  default: /* code du père */  
}
```



Données dupliquées

- après **fork**, le fils est la copie du père
- conséquence: ce qui a été alloué ou ouvert avant **fork**, doit être libéré ou fermé dans le père ET dans le fils
- copy-on-write:
 - tant qu'ils ne font que lire, la mémoire est partagée
 - à la première écriture, il y aura une copie réelle de la mémoire pour que chacun puisse avoir son espace propre

fork_valgrind.cpp



Données dupliquées

- exemples:
 - fork et la libc:

`fork_printf.cpp`

`fork_write.cpp`

- descripteurs de fichiers:

`fork_creat.cpp`

`fork_creat2.cpp`



Divergences

- le père et le fils ne partagent pas tout:
 - leurs PID et PPID sont différents
 - les compteurs de temps et d'utilisation CPU du fils sont à zéro
 - le fils n'hérite pas des verrouillages mémoire de son père, de ses sémaphores, des signaux en attente, des opérations d'E/S asynchrones, etc (cf. man fork)



Groupes

- les processus sont rassemblés en *groupes*
- permet de contrôler la distribution de signaux:
 - un signal envoyé à un groupe est envoyé à tous les membres du groupe



Sessions

- les groupes sont rassemblés en *sessions*, caractérisées par un terminal de contrôle commun aux différents processus
 - terminal partagé par le shell et ses processus fils
- on hérite la session de son père
- si on n'est pas déjà leader de groupe, on peut en créer une avec **setsid** (existe en appel système et en commande shell)



Sessions

- quand le leader de session meurt, tous les membres de la session reçoivent un signal SIGHUP qui les tue par défaut:
 - **firefox &**: meurt si on tue le shell
 - **setsid firefox**: survit parce qu'il est leader d'une nouvelle session
 - **nohup firefox**: survit parce qu'il ignore le signal SIGHUP, mais il reçoit le message, d'où le warning de gnome:





Mort

- un processus meurt si:
 - il exécute `exit`, `_exit`, ou `exit_group`
 - sa fonction `main` se termine
 - il reçoit un signal mortel du système:
 - division par zéro
 - erreur de segmentation
 - mort du leader de session
 - etc
- ses descripteurs sont fermés et il est retiré de la mémoire



waitpid

- pour savoir comment un processus est mort, son père doit invoquer:
- `pid_t waitpid(pid_t pid, int *status, int options);`
- attend la mort d'un processus:
 - `pid < -1`: du groupe d'ID `-pid`
 - `pid = -1`: un fils
 - `pid = 0`: un fils du groupe du père
 - `pid > 0`: le processus qui a ce PID



waitpid

- **status**: si non **NULL**, utilisé pour stocker des informations sur les conditions de la mort, que l'on peut connaître avec des macros:
 - **WIFEXITED**: mort normale ?
 - **WEXITSTATUS**: si oui, code de retour
 - **WIFSIGNALED**: mort par signal ?
 - **WTERMSIG**: si oui, n° du signal
 - etc



waitpid

- **options**: si non **WNOHANG**, permet de tester si un processus est mort de façon non bloquante
- **waitpid** renvoie le PID du processus mort, -1 en cas d'échec, ou 0 s'il n'y a pas de processus mort et qu'on est en mode non bloquant



Les zombies

- quand un processus meurt, le système garde les infos sur sa mort dans la table des processus jusqu'à ce que son père les ait lues avec `waitpid`

`zombie.cpp`

```
$>./zombie &
$>ps
  PID TTY          TIME CMD
 3084 pts/0    00:00:00 bash
 4640 pts/0    00:00:00 zombie
 4641 pts/0    00:00:00 zombie <defunct>
 4642 pts/0    00:00:00 zombie <defunct>
 4643 pts/0    00:00:00 zombie <defunct>
 4644 pts/0    00:00:00 zombie <defunct>
```

...



Les orphelins

- quand un processus meurt, ses fils sont adoptés par *init* qui attend la mort de tous ses enfants
- ça évite de saturer la table des processus avec des zombies
- c'est aussi un moyen de lancer une tâche de fond qui doit survivre à son père

`init_adoption.cpp`



times

- `clock_t times(struct tms *buf);`
- donne les consommations en temps système et utilisateur du processus et de ses fils, en tops d'horloge
- pour un temps en secondes, diviser par `sysconf(_SC_CLK_TCK)`
- ne pas utiliser la valeur de retour ! (cf. `man times`)
- ne pas confondre avec `time`



UID

- UID d'un processus=identifiant de l'utilisateur qui l'exécute
- 3 sortes d'UID:

`fs/namei.c: permission`

- euid: UID effective, utilisée pour les tests de droits d'accès
- ruid: UID réelle, à qui sont facturés les temps de calcul et quotas disques
- suid: UID sauvegardée, pour pouvoir revenir en arrière



UID

- `uid_t getuid(void);`
- `uid_t geteuid(void);`
- renvoient l'utilisateur réel/effectif du processus
- se base sur le set-uid bit qu'on positionne avec `chmod u+s`

```
$>sudo chown root ./getuid
$>sudo chmod u+s ./getuid
$>./getuid
Real user=paumier
Effective user=root
```

getuid.cpp



setuid/setreuid

- `int setuid(uid_t uid);`
- `int setreuid(uid_t ruid, uid_t euid);`
- fixe les UID réelle et effective du processus en sauvant les anciens, pour pouvoir retrouver les anciens privilèges après
- pour des raisons de sécurité, un programme root qui abandonne ses droits ne peut les retrouver ensuite



GID

- `uid_t getgid(void);`
- `uid_t getegid(void);`
- renvoient le groupe réel/effectif du processus
- sur un répertoire: avec le set-gid bit (`chmod g+s`), tout fichier créé dans le répertoire appartient au groupe du répertoire et non de celui qui crée le fichier



GID

- utile pour restreindre des permissions, comme pour la commande **mail**
- avec des droits appropriés, tout le monde peut écrire, mais à part le créateur d'un fichier, seuls les utilisateurs du bon groupe peuvent y accéder

```
$>mkdir toto
$>sudo chgrp root toto
$>sudo chmod g+s toto
$>touch toto/titi
$>ls -l toto/titi
-rw-r--r-- 1 paumier root 0 2010-09-27 13:04 toto/titi
```



Bit de suppression restreinte

- ne fonctionne que sur les répertoires
- le "sticky bit" (`chmod a+t`) interdit de supprimer ou renommer un fichier, à moins d'être le possesseur, le possesseur du répertoire ou un utilisateur privilégié
- sert à protéger les répertoires publics comme `/tmp`



Valeur de courtoisie

- valeur nice entre -20 (priorité maximum) et 19 (priorité minimum)
- `int nice(int inc);`
- `inc` est ajouté à la valeur courante du processus courant
- seul root peut donner un `inc` négatif
- on peut aussi fixer la valeur pour une tâche avec la commande shell `nice`



Valeur de courtoisie

- pour lire/modifier la valeur d'un processus déjà lancé:
- commande shell **renice**
- **int setpriority(int which, int who, int prio);**
- **int getpriority(int which, int who);**
 - belle erreur de conception pour sa valeur de retour: cf. page man



chroot

- `int chroot(const char *path);`
- définit la nouvelle racine utilisée pour résoudre les chemins depuis /
- ne change pas le répertoire courant
- utilisée pour limiter des accès comme pour du ftp
- attention à la sécurité: plusieurs possibilités de s'évader d'un piège chroot



Limites d'un processus

- `int getrlimit(int res, struct rlimit *rlim);`
- `int setrlimit(int res, const struct rlimit *rlim);`
- manipulation des limites du processus définies par `res`:
 - `RLIMIT_AS`: taille de la mémoire virtuelle
 - `RLIMIT_CPU`: temps max en secondes
 - `RLIMIT_FSIZE`: taille max d'un fichier créé
 - ...



Limites d'un processus

- paires limite souple/limite dure
- la limite souple peut être franchie par le processus, la dure non
 - exemple: au-delà de la limite souple sur le temps CPU, le processus reçoit SIGXCPU, qu'il peut bloquer; après la limite dure, c'est SIGKILL
- la souple est modifiable dans [0;dure]
- la dure ne peut être que baissée (sauf si on est root)

```
limit_no_sighandler.cpp  
limit_with_sighandler.cpp
```



Capacités

- si l'UID effective vaut 0, le processus est *privilégié*, il contourne les tests de permissions du noyau (droits d'accès aux fichiers, droits d'envoyer des signaux à tous les processus, etc)
- depuis Linux 2.2, ces droits de root sont découpés en *capacités* qu'on peut donner individuellement aux autres processus et aux fichiers exécutables
- liste: cf. *man capabilities*



Capacités de threads

- au sein d'un processus, les capacités peuvent être définies par thread
- 3 ensembles de capacités de threads:
 - autorisées: sur-ensemble qui limite les capacités utilisables
 - héritables: capacités qui seront conservées à travers un exec
 - effectives: les capacités réellement utilisables (sous-ensemble des autorisées)



Capacités de fichiers

- 3 ensembles de capacités pour les fichiers exécutables:
 - autorisées: capacités accordables au thread quoi qu'il arrive par ailleurs
 - héritables: capacités héritées du parent travers un exec
 - effectives: bit qui, s'il est allumé, accorde effectivement au thread toutes ses capacités autorisées



Capacités

- pour lire/modifier les capacités du processus courant:
- `cap_t cap_get_proc(void);`
- `int cap_set_proc(cap_t cap_p);`
 - dans le package libcap2-dev
- mais, il faut que le processus soit déjà privilégié pour pouvoir utiliser `cap_set_proc`



Capacités

- pour attribuer des capacités à des exécutables, il faut utiliser **setcap**
 - nécessite le package libcap2-bin

```
$>touch toto
$>chmod 0 toto
$>./capabilities
opening forbidden file toto: -1
creating /bin/foo: -1
$>sudo setcap CAP_DAC_READ_SEARCH=eip ./capabilities
$>./capabilities
opening forbidden file toto: 3
creating /bin/foo: -1
```

capabilities.cpp

- mais, système pas tout à fait mûr, attention aux risques de sécurité !



exec

- si on n'avait que **fork**, on ne pourrait lancer que des commandes sous forme de fonctions, comme pour démarrer *init*:

```
if (!fork()) {      /* we count on this going ok */
    init();
}
```

- beaucoup trop compliqué à gérer
- l'excellente idée: recycler le processus courant en remplaçant son activité par une nouvelle

fs/exec.c: do_execve



execve

- `int execve(const char *filename, char *const argv[], char *const envp[]);`
- remplace le contenu du processus courant par l'exécution du programme désigné par `filename` qui doit être:
 - un exécutable binaire
 - un script commençant par `#!binious`
- on ne survit à `execve` qu'en cas d'erreur !



Les variantes

- `int execl(const char *filename, char *const argv[]);`
 - même chose, sans donner explicitement un tableau des variables d'environnement
- `int execlp(const char *file, char *const argv[]);`
 - même chose, mais si `file` ne commence pas par `/`, on explore `PATH`, ou `/bin:/usr/bin:` si `PATH` n'est pas défini



Les variantes

- `int execl(const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execl(const char *path, const char *arg, char * const envp[]);`
- même chose, mais les arguments sont variadiques au lieu d'être dans un tableau terminé par **NULL**



execve et héritage

- un processus remplacé par **execve** perd certaines choses:
 - signaux en attente, gestionnaires de signaux, sémaphores, mémoire mappée, verrouillages de mémoire, crochets d'arrêts, etc (cf. man execve)
- il garde tout le reste, y compris les descripteurs de fichiers, sauf ceux marqués avec l'attribut close-on-exec



Redirections

- comme on hérite des descripteurs, on peut rediriger les E/S d'un processus, puis faire un exec

my_cat.cpp

```
$>ls > toto  
$>./my_cat toto tutu  
$>diff toto tutu
```

swap_outputs.cpp

```
$>./swap_outputs ls dgf * > out 2> err  
$>cat out  
ls: ne peut accéder dgf: Aucun fichier ou dossier de ce type
```



Remplacement

- on garde les descripteurs de fichiers, mais on perd toute la bufferisation de la libc:

`exec_printf.cpp`

- mais ça change rien aux liens de parenté entre processus:

`exec_starwars.cpp`

