



Systemes et reseaux

Chapitre 2 Mémoire et processus



Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/sr

Mémoire et processus

Qu'est-ce qu'un processus ?

- ▶ Un *processus* est un programme en cours d'exécution.
- ▶ Il est composé de plusieurs choses :
 - du code exécutable (une suite d'instructions),
 - d'un espace mémoire de travail,
 - d'entrées/sorties.
- ▶ Il est identifié par un nombre unique sur le système, son *process id* ou *pid*.

- Le cycle de vie d'un processus est composé de différents *états* :
- nouveau : l'état initial une fois le processus créé ;
 - exécution : l'état dans lequel le processus s'exécute ;
 - bloqué : l'état dans lequel se retrouve un processus *interrompu* ;
 - prêt : l'état d'attente d'un processus disponible pour être exécuté ;
 - terminé : le processus est fini.

Création d'un processus

- ▶ Au démarrage du système, le noyau lance un premier processus le pid 1, *init*.
 - Ce processus est lancé par l'ordonnanceur qui s'attribue lui même le pid 0.
 - Les quelques premiers pid sont aussi réservés à des services du noyau.
- ▶ Après ça la création de nouveaux processus se fait par *clonage*.
 - Regardons un peu l'appel système `clone`.
- ▶ Quand un processus se clone, il en existe ensuite deux versions :
 - le processus original devient le *parent* du nouveau processus ;
 - cela résulte en une *hiérarchie* (un arbre) de processus.

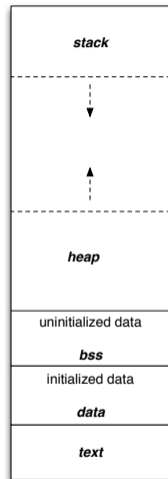
Lancement d'un programme

- ▶ Pour qu'un programme puisse en lancer un autre :
 - son processus doit d'abord se cloner, puis
 - se remplacer en mémoire par l'autre, ce que fait l'appel système `execve`.

- ▶ Au remplacement d'un processus par un nouveau programme :
 - un nouvel espace mémoire est alloué par le système,
 - le tas et la pile du programme sont initialisées,
 - la fonction `main` est appelée (avec ses arguments).

Organisation de la mémoire

- ▶ La mémoire d'un programme est organisée en cinq sections :
 - le segment de code,
 - le segment de données initialisées,
 - le segment de données non-initialisées,
 - le tas,
 - la pile.
- ▶ Les trois segments sont encodés statiquement dans l'exécutable.
- ▶ Le tas et la pile sont alloués dynamiquement par le système au lancement du programme.
 - La pile est gérée directement dans l'espace utilisateur.
 - Le tas est géré avec les appels systèmes `brk` et `sbrk`.



Exécution d'un processus

- ▶ L'exécution d'un processus peut se faire :
 - dans l'espace utilisateur, avec des contraintes et limites sur les instructions disponibles ;
 - dans l'espace noyau, sans ces contraintes.

- ▶ Comme on l'a déjà vu on utilise depuis l'espace utilisateur des *appels systèmes* pour faire les tâches qui nécessitent des privilèges.

Interruption d'un processus

- ▶ Pour différentes raisons, un processus peut être *interrompu* par le système :
 - prioriser une entrée/sortie matériel,
 - gérer un signal ou une tâche urgente et prioritaire,
 - ou tout simplement permettre le multitâches.
- ▶ Les premiers systèmes multitâches nécessitaient que les processus soient *coopératifs* :
 - les processus devaient activement rendre la main au système,
 - et donc gérer avant ça de sauvegarder leur état pour être capable de se remettre en route.
- ▶ Les systèmes modernes font du multitâche *préemptif* :
 - le système organise via son ordonnanceur une alternance rapide entre les processus,
 - cela crée l'illusion de l'exécution simultanée de plusieurs tâches,
 - l'ordonnanceur gère avec une file de priorité les différents processus en cours d'exécution.
- ▶ Le passage d'une tâche à une autre, c'est-à-dire de l'exécution d'un processus à un autre, s'appelle un *context switch*.

Terminaison d'un processus

- ▶ Un processus peut terminer de plusieurs façons :
 - dans un état de succès (volontaire),
 - dans un état d'erreur (volontaire),
 - dans un état d'erreur fatal (involontaire),
 - tué par un autre processus.

- ▶ Si un processus termine sans que ça mémoire puisse être libéré, il passe en mode *zombie*.
 - Par exemple, jusqu'à ce que son parent l'attende explicitement (appels systèmes `wait` et `waitpid`).

- ▶ Si un processus parent termine avant ses enfants, les processus orphelins sont automatiquement adoptés par le processus `init`.
 - Dans le systèmes modernes, il y a parfois un processus utilisateur qui sert de gestionnaire de session et qui adopte les orphelins à la place de `init`.

- ▶ Voyons ensemble quelques appels systèmes utiles autour des processus.

Lancer un nouveau processus

- ▶ **fork** : créer un processus enfant
 - `#include <unistd.h>`
 - `pid_t fork(void);`

- ▶ **execve** : exécuter un programme
 - `#include <unistd.h>`
 - `int execve(const char *pathname, char *const _Nullable argv[], char *const _Nullable envp[]);`
 - Il existe de nombreuses variantes bien pratiques.

- ▶ **wait** et **waitpid** : surveiller les changements d'états d'un processus
 - `#include <sys/wait.h>`
 - `pid_t wait(int *_Nullable wstatus);`
 - `pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);`

- ▶ **_exit** : terminer le processus appelant
 - `#include <unistd.h>`
 - `[[noreturn]] void _exit(int status);`
 - Voir aussi : `exit`, `atexit`.

Gestion du tas

- ▶ Gérer le tas avec `brk` et `sbrk` est compliqué.
- ▶ On préférera utiliser `malloc` et `free`.

Identité d'un processus

- ▶ `getpid` / `getppid` : récupérer l'identifiant du processus / de son parent
 - `#include <unistd.h>`
 - `pid_t getpid(void);`
 - `pid_t getppid(void);`

Utilisateur et groupe

- ▶ `getuid / geteuid` : récupérer l'identité de l'utilisateur
 - `#include <unistd.h>`
 - `uid_t getuid(void);`
 - `uid_t geteuid(void);`

- ▶ `getgid / getegid` : récupérer l'identité du groupe
 - `#include <unistd.h>`
 - `uid_t getgid(void);`
 - `uid_t getegid(void);`

- ▶ L'*effective* uid ou gid sont ceux qui donne ses droits effectifs au processus.

Changement d'utilisateur et groupe

- ▶ Les appels systèmes `setuid` / `seteuid` / `setreuid` (et idem pour `gid`) permettent de modifier les `uid` (et `gid`) effectifs du processus.
- ▶ Dans ce cas, les *effective* `uid` et `gid` peuvent différer des *real* `uid` et `gid`, qui sont ceux de l'utilisateur qui exécute le processus.
- ▶ Pourquoi peut-on modifier les valeurs effectives des identités d'utilisateur et de groupe ?

Changement d'utilisateur et groupe

- ▶ Les appels systèmes `setuid` / `seteuid` / `setreuid` (et idem pour gid) permettent de modifier les uid (et gid) effectifs du processus.
- ▶ Dans ce cas, les *effective* uid et gid peuvent différer des *real* uid et gid, qui sont ceux de l'utilisateur qui exécute le processus.
- ▶ Pourquoi peut-on modifier les valeurs effectives des identités d'utilisateur et de groupe ?
 - Pouvoir effectuer des opérations privilégiés (exemples : raw socket, ports < 1024) ;
 - sans pour autant garder les privilèges root en permanence (question de sécurité).

Changement d'utilisateur et groupe

- ▶ Les appels systèmes `setuid` / `seteuid` / `setreuid` (et idem pour `gid`) permettent de modifier les `uid` (et `gid`) effectifs du processus.
- ▶ Dans ce cas, les *effective* `uid` et `gid` peuvent différer des *real* `uid` et `gid`, qui sont ceux de l'utilisateur qui exécute le processus.
- ▶ Pourquoi peut-on modifier les valeurs effectives des identités d'utilisateur et de groupe ?
 - Pouvoir effectuer des opérations privilégiés (exemples : raw socket, ports < 1024) ;
 - sans pour autant garder les privilèges root en permanence (question de sécurité).
- ▶ Un mot sur `sudo` : l'exemple de `sudo`.