Systèmes et réseaux

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*.

La vie d'un processus

- 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

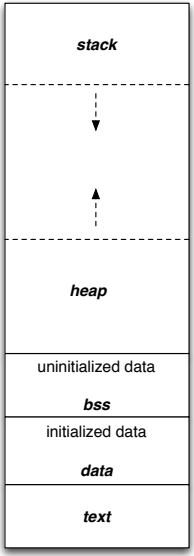
- Au démarrage du système, le noyau lance un premier processus le pid 1, `init`.
  - Ce processus est lancé par l'ordonnanceur qui s'attribue lui même le pid 0.
  - Les quelques premiers pid sont aussi réservés à des services du noyau.
- Après ça la création de nouveaux processus se fait par *clonage*.
  - 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`.

En pratique

- 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`.

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

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

- `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).

- 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 suid : l'exemple de **`sudo`**.

Processus légers (*threads*)

- À la différence d'un `fork`, un *thread* (ou *processus léger*), ne duplique pas la mémoire du processus.
  - En fait, un processus est un ensemble de threads (souvent qu'un seul, le thread *main*).
  - Les threads d'un processus partagent presque tout : mémoire, pid et ppid bien sûr, descripteurs de fichiers, utilisateur et groupe, répertoire courant, et quelques autres trucs qu'on verra plus tard (ex: signaux).
  - Ils ne partagent pas : leur identifiant, leur pile, et quelques autres trucs qu'on verra plus tard (ex: signaux).
  - Création et switch sont très rapides !
- Cela permet d'exprimer au niveau de la logique du programme des calculs *concurrents* (multitâches).
  - Concurrence : *logiquement* en même temps mais pas forcément *physiquement*.
  - Parallélisme : *physiquement* en même temps (nécessite plusieurs processeurs/cœurs).
- Attention, les accès à une même zone mémoire depuis plusieurs threads peuvent poser problème !
  - La synchronisation entre threads n'est pas gérée par le système mais par l'utilisateurice.
  - Il faut utiliser des fonctions "MT-safe" (cf les pages de manuel) !
  - Demande plus de rigueur pour éviter les inter-blocages (*deadlock*) et d'autres types de bugs (*race condition*, *starvation*, etc).

- La libraire `pthread` implémente les threads POSIX (compiler avec `-lpthread`).
  - `#include <pthread.h>`
  - ```
    int pthread_create(pthread_t *restrict thread,
                       const pthread_attr_t *restrict attr,
                       void *(*start_routine)(void *),
                       void *restrict arg);
    ```
  - `pthread_t pthread_self(void);`
  - `int pthread_equal(pthread_t t1, pthread_t t2);`
  - `int pthread_join(pthread_t thread, void **retval);`
  - `[[noreturn]] void pthread_exit(void *retval);`
- Attention :
  - le thread *main* est particulier : s'il termine, le processus termine, et donc ses threads avec ;
  - appeler `exit` depuis n'importe quel thread termine tout le processus ;
  - le comportement des threads en interaction avec `fork` et `exec` est… potentiellement complexe.
- Si besoin, lisez attentivement les pages de manuel `pthread_*` qui sont très bien faites.
  - `man pthreads`
    ```

Annulation

- Il est possible d'annuler un thread en cours d'exécution, si celui-ci le veut bien !
  - `#include <pthread.h>`
  - `int pthread_cancel(pthread_t thread);`
  - `int pthread_setcancelstate(int state, int *oldstate);` (`PTHREAD_CANCEL_ENABLE` ou `PTHREAD_CANCEL_DISABLE`)
  - `int pthread_setcanceltype(int type, int *oldtype);` (`PTHREAD_CANCEL_DEFERRED` ou `PTHREAD_CANCEL_ASYNCHRONOUS`)
- Les fonctions pouvant servir de points d'annulation (si "deferred") sont listées dans `man pthreads`.
  - `void pthread_testcancel(void);`
- Lors du join, un thread annulé renvoie `PTHREAD_CANCELED`.

Semaphores

- Historiquement, un des premiers outils de synchronisation de threads sont les *sémaphores*, inventées en 1962 par Dijkstra.
  - Il s'agit d'une variable partagée qui correspond à un entier représentant le nombre de ressources disponibles.
  - On peut effectuer dessus une opération "attendre" qui décrémente la variable et bloque jusqu'à ce qu'au moins une ressource soit disponible.
  - L'autre opération possible est "poster" qui incrémente la variable pour signifier la libération d'une ressource.
- Dans POSIX :
  - `#include <semaphore.h>`
  - `int sem_init(sem_t *sem, int pshared, unsigned int valeur);`
  - `int sem_wait(sem_t *sem);`
  - `int sem_post(sem_t *sem);`
  - `int sem_destroy(sem_t *sem);`

Exclusion mutuelle

- Un objet d'*exclusion mutuelle* permet d'empêcher deux threads de modifier des données partagées en même temps, par exemple.
  - `#include <pthread.h>`
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);`
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
- On peut être plus subtil avec les *verrous de lecture-écriture* (`pthread_rwlock_t`) qui permettent de traiter différemment les accès en lecture et en écriture (notamment pour permettre l'accès simultané en lecture seule).
- On peut être arbitrairement subtil avec les *variables de condition* (`pthread_cond_t`), qui permettent d'attendre la réalisation d'une condition arbitraire tant que celle-ci peut s'exprimer dans notre langage de programmation.

- Les *barrières* permettent de synchroniser plusieurs threads en les faisant s'attendre les uns les autres.
  - `#include <pthread.h>`
  - ```
    int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                             const pthread_barrierattr_t *restrict attr, unsigned count);
    ```
  - `int pthread_barrier_wait(pthread_barrier_t *barrier);`
  - `int pthread_barrier_destroy(pthread_barrier_t *barrier);`
    ```