Processus

LIFASR5 - Systèmes d'exploitation

L. Gonnord, N. Louvet

14 mars 2022

Notion de processus

Système d'exploitation multitâche

C'est un système qui permet d'exécuter, de façon apparemment simultanée, plusieurs programmes.

Problèmes :

- un processeur est prévu pour exécuter un seul programme à la fois,
- on veut exécuter plus de programmes qu'il n'y a des processeurs,
- l'exécution d'un programme ne doit pas perturber celle des autres.

Exécution d'un programme

- Le système doit charger le programme en mémoire,
- trouver son point d'entrée (fonction `main()`),
- suivre son déroulement (pointeur d'instruction),
- à la fin du programme, libérer les ressources qu'il occupait.

Notion qui définit une exécution d'un programme : le *processus*.

Définition (Processus)

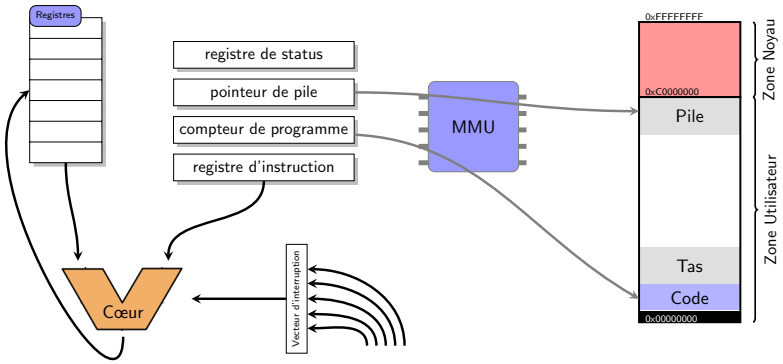
Un *processus* est un programme en cours d'exécution. Un processus est géré par le noyau comme une structure de donnée qui contient toutes les informations nécessaires afin de suivre le déroulement du programme, d'en stopper ou d'en reprendre l'exécution.

Cette structure de données doit permettre :

- d'identifier le programme dont elle est une instance ;
- d'isoler ce programme de tous les autres ;
- d'accéder aux fichiers et aux ressources matérielles ;
- de vérifier que seules les ressources autorisées sont accédées ;
- de libérer les ressources (fermer les fichiers, libérer la mémoire, *etc*).

Dans le processeur, à un instant donné, un *processus* est décrit par :

- les valeurs contenues dans les registres : compteur de programme, registres généraux, registre de status, pointeur de pile...
- l'état de la zone mémoire qui lui a été attribuée par le système.



Un processus n'accède jamais directement aux adresses de la mémoire physique, mais à une *mémoire virtuelle*.

Chaque processus dispose ainsi de son propre espace mémoire (4 Gio pour une installation 32 bits), et *ne peut pas interférer avec les autres processus*.

Chaque processus est représenté dans le système par un *bloc de contrôle de processus* (BCP ; on parle aussi de *contexte*) qui est une structure de donnée qui regroupe les informations précédentes (et d'autres. . .) :

bloc de contrôle de processus
identifiant unique du processus
compteur de programme
registre d'instruction
pointeur de pile
contenu des registres généraux
. . .

L'idée est que le système peut :

- à partir du BCP d'un processus, faire reprendre son exécution,
- sauvegarder le BCP d'un processus, et arrêter son exécution.

Commutation de processus

L'idée est de pouvoir faire progresser l'exécution de plusieurs processus sur un même processeur. Le système doit organiser le passage de l'exécution d'un processus à celle d'un autre : c'est la *commutation*.

En gros, lors de la commutation d'un processus A à un processus B :

- l'exécution de A est interrompue,
- le système reprend la main, et sauvegarde l'état actuel de A,
- le système restaure l'état du processus B dans le processeur,
- le processus B reprend son exécution.

La *commutation* permet notamment :

- d'exploiter le temps requis pour effectuer des entrées/sortie,
- l'exécution concurrente de plusieurs processus.

À chaque fois le noyau reprend la main, il choisit le prochain processus.

État d'un processus

Rappel : Le système doit

- gérer l'accès au processeur
- gérer l'occupation de la mémoire

Définition (état d'un processus)

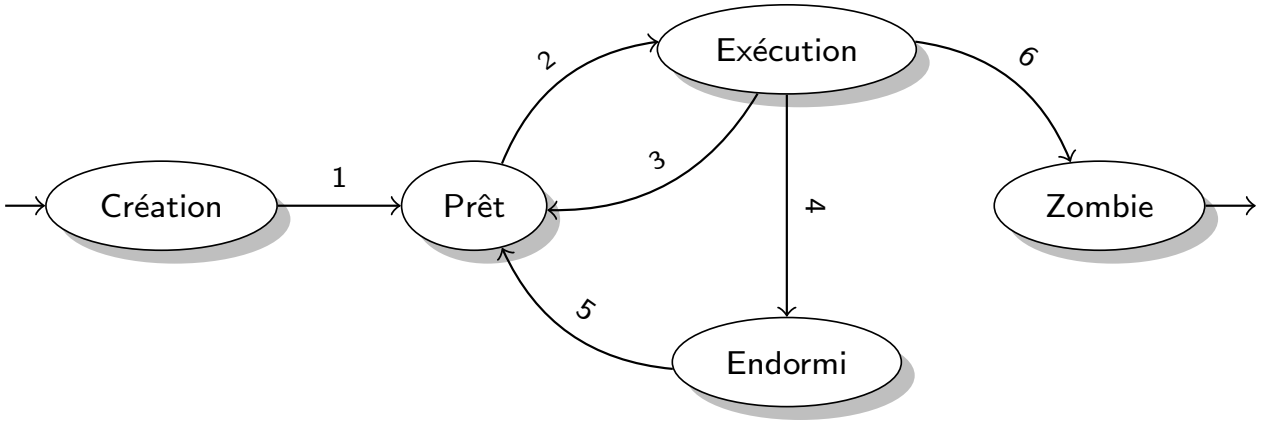
Le système doit donc gérer plusieurs files d'attentes de processus :

- ceux qui peuvent s'exécuter,
- ceux qui sont en mémoire,
- ceux qui sont bloqués car demandent une ressource occupée,
- ...

On parle *d'état du processus*.

Quand le système reprend la main, il met à jour ces files d'attentes, et détermine le prochain processus à s'exécuter : c'est *l'ordonnancement*.

Exemple : automate d'évolution de l'état d'un processus.



1 - le processus est prêt, il est pris en compte par l'ordonnanceur
2 - l'ordonnanceur choisit le processus pour être exécuter sur le processeur
3 - l'ordonnanceur choisit un autre processus pour être exécuter
4 - le processus décide de s'endormir pour une certaine durée (avec `sleep()`)
5 - l'ordonnanceur repasse la processus à l'état « prêt » à la fin de cette durée
6 - le processus décide de se terminer (avec `exit()` par exemple)

Rappel. Pour voir l'état des processus : `ps -l` ou `top` (`q` pour quitter).

Observation des processus

Sous Unix, les processus sont organisés de façon *hiérarchique* :

- Chaque processus est identifié de façon unique par un entier : c'est son *PID*, pour *Process Identifier*.
- Chaque processus peut créer des processus appelés *fils*.
- Tous les processus ont accès à l'identifiant de leur *père* : c'est leur *PPID*, pour *Parent Process Identifier*.

A la racine de la hiérarchique il y a l'*ancêtre* de tous les processus :

- son PID est égal à 1 ;
- sous GNU/Linux, c'est `systemd` (avant c'était `init` avec System V).

Deux commandes utiles pour observer les processus :

- `ps` avec notamment les options `-e` et `-l` ;
- `pstree` pour avoir une vue de l'arborescence des processus.

Un exemple :

```
nlouvet:~$ xclock & xcalc
nlouvet:~$ bash
nlouvet:~$ xcalc &
nlouvet:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  5503  4261  0  80   0 -  6317 wait   pts/3    00:00:00 bash
0 S  1000  5555  5503  0  80   0 - 18513 poll_s pts/3    00:00:00 xclock
0 S  1000  5556  5503  0  80   0 - 13683 poll_s pts/3    00:00:00 xcalc
0 S  1000  5558  5503  0  80   0 -  6065 wait   pts/3    00:00:00 bash
0 S  1000  5566  5558  0  80   0 - 13683 poll_s pts/3    00:00:00 xcalc
4 R  1000  5567  5558  0  80   0 -  7558 -      pts/3    00:00:00 ps
nlouvetk:~$ pstree -p 5503
bash(5503)──┬─bash(5558)──┬─pstree(5568)
            │             └─xcalc(5566)
            ├─xcalc(5556)
            └─xclock(5555)
```

On retrouve bien :

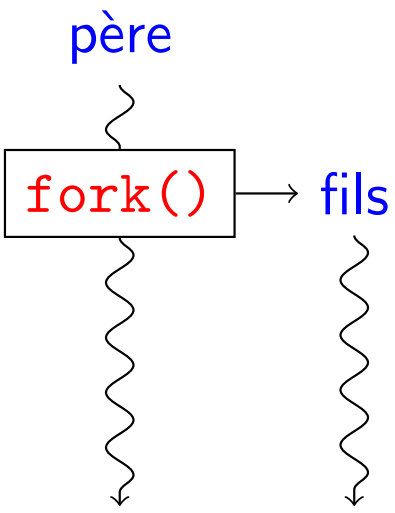
- les informations PID, PPID, état (colonne S)...
- l'organisation hiérarchique des processus

Création de processus

Sous Unix, un nouveau processus est forcément la copie d'un processus existant :

- le processus déjà existant est appelé le père,
- le nouveau processus créé est appelé le fils.

`fork()` est l'appel système utilisé pour créer un nouveau processus :



Au retour de `fork()`, *le père et le fils exécutent tous les deux le même code* : celui présent juste après l'appel à `fork()` dans le programme !

pid_t fork(void)

Que dit le manuel ?

- `fork()` creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following...
- On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, ..., and errno is set appropriately.

Après un appel réussi à `fork()` :

- le fils est une copie du père (même code, même mémoire),
- seule la valeur de retour de `fork()` permet de les distinguer.

Il faut tester la valeur de retour pour différentier leurs exécutions.

Fin d'un processus

Dans le cas général, un processus se termine en

- exécutant un `return()` dans sa fonction `main()`
- ou en faisant appel à la fonction `void exit(int status)`.

Lorsqu'il se termine, un processus passe forcément en état zombie :

- il n'est plus jamais prêt mais ses ressources ne sont pas toutes libérées : la valeur de retour de la fonction `main()` est en attente.
- le père doit lire ce résultat par l'appel `waitpid()` :
  le processus fils disparaît alors de la table des processus.
- si le père se termine avant le fils, le fils est orphelin : il est adopté par `systemd`, qui se charge d'en libèrer les ressources.

Un zombie occupe des ressources tant que son père n'a pas fait un appel à `waitpid()` : pour des processus de longue durée, il est important de ne pas laisser leurs fils dans cet état lamentable !

`pid_t waitpid(pid_t pid, int *wst, int opt);`

L'appel permet d'attendre qu'un fils se termine :

- `pid` est le PID du fils attendu, ou 0 si l'on attend n'importe lequel ;
- `wst` pour récupérer les infos de retour du fils (NULL pour ignorer) ;
- si opt = 0 l'attente est bloquante (le cas important pour nous) ;
- si opt = `WNOHANG`, l'appel retourne immédiatement.

Que dit le manuel ?

In the case of a terminated child, performing a wait allows the system to release the resources associated with the child ; if a wait is not performed, then the terminated child remains in a "zombie" state.

Valeur de retour :

- en cas d'échec, -1 est retourné et `errno` mise à jour ;
- si un fils s'est terminé, le PID de ce fils ;
- si l'appel est non-bloquant et qu'aucun fils n'est terminé, alors 0.

Un exemple :

```
int main(void) {
    int ret = fork();

    if(ret > 0) { // processus père
        cout << "(père) j'attens..." << endl;
        waitpid(ret, NULL, 0);
        cout << "(père) pid : " << getpid() << endl;
        cout << "(père) ppid : " << getppid() << endl;
    }
    else { // processus fils
        cout << "(fils) pid : " << getpid() << endl;
        cout << "(fils) ppid : " << getppid() << endl;
    }

    cout << "Fin du processus de pid " << getpid() << endl;

    return 0;
}
```

Recouvrement d'un processus

En fait, il existe toute une famille de fonctions, listées dans `man exec`.

Que dit le manuel ?

The exec() family of functions replaces the current process image with a new process image. The functions [listed here] are front-ends for execve.

The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

Le processus appelant est *recouvert* (remplacé) par la commande en argument d'`exec()`. Par exemple, la primitive

```
int execvp(const char *file, char *const argv[]);
```

recouvre le processus appelant par le commande dont le chemin est donné par `file`, en lui passant les arguments de qui sont dans `argv`.

Un exemple :

```
int main(void) {
    pid_t pid = fork();

    if(pid == 0) { // processus fils
        execl("/bin/ls", "ls", "-al", NULL);
        // Le code qui suit ne sera jamais exécuté
        // par le fils, sauf en cas d'erreur
        cerr << "Erreur : " << strerror(errno) << endl;
        return EXIT_FAILURE;
    }
    else { // processus père
        waitpid(pid, NULL, 0);
        cout << "Le fils est terminé, le père se termine." << endl;
        return EXIT_SUCCESS;
    }
}
```

La communication entre processus

Le système sépare les processus pour éviter les perturbations, assurer la protection des données, pour faciliter la gestion.

Sans faire appel au système, un processus ne peut pas agir sur un autre.

Mais la communication entre processus est nécessaire

- certains processus doivent communiquer, *e.g.*, serveur
- tout processus nécessite un moyen d'être contacté, ne serait-ce que pour pouvoir l'arrêter !

Il existe différentes manières de faire communiquer des processus entre-eux : les paramètres de la fonction `main()`, les variables d'environnement, les signaux, les pipes...

Paramètres de la fonction main()

Ce sont les paramètres passés aux programmes lancés depuis la ligne de commande. On rappelle leur utilisation avec le programme suivant :

```
int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    cout << "paramètre " << i << " : " << argv[i] << endl;
  return 0;
}
```

On compile se programme en l'exécutable paraldc. Par exemple :

```
~/$ ./paraldc.exx riri fifi
paramètre 0 : ./paraldc.exx
paramètre 1 : riri
paramètre 2 : fifi
```

Retenez bien que :

- argc compte tous les paramètres, y compris le chemin de l'exécutable
- argv[i] (pour i entre 0 et argc−1) est une chaîne de caractères pointée par un char* et qui se termine par un '\0'

Les variables d'environnement

Les processus sont séparés, mais fonctionnent dans un environnement :

- ils sont exécutés depuis un working directory,
- ils appartiennent à un utilisateur et héritent de ses droits, ...

Ces informations sont transmises par des variables héritées de leur processus père : les *variables d'environnement*.

Exemples :

- `PATH` : liste des répertoires où sont cherchés les exécutables.
- `USER` et `HOME` : nom et répertoire personnel de l'utilisateur.

Commandes utiles (Bash) :

- `echo $VAR` pour afficher la variable `VAR` !
- `env` pour lister toutes les variables d'environnement.
- `export VAR=valeur` pour définir une variable d'environnement.

Les signaux

Définition

Un *signal* est une alerte qui stoppent l'exécution du processus pour lui faire exécuter un gestionnaire de signal (pour simplifier, disons une fonction) :

- c'est un message simple, juste un entier
- il existe un petit nombre de messages prédefinis :
  - `SIGINT` = 2 (touche Ctrl+c),
  - `SIGSTOP` = 19 (touche Ctrl+z),
  - `SIGKILL` = 9 (kill),
  - `SIGSEGV` = 11 (erreur de segmentation),
  - ...
- c'est un évènement asynchrone (il peut arriver n'importe quand)
- il peut provoquer le déroutement du processus vers un gestionnaire
- la communication est très limitée (type du signal, émetteur, ...

Par défaut, un signal est ignoré ou il est traité par un gestionnaire prédéfini. Pour changer ce comportement, il faut installer un gestionnaire de signal.

Commandes shell utiles :

- `kill -s signal pid` : pour envoyer un signal,
- `kill -l` : pour lister les signaux existants.

En C/C++ POSIX :

- `int kill(pid_t pid, int sig)` : pour envoyer un signal,
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` : mettre en place un gestionnaire de signal.

La structure `sigaction` est « intimidante » :

```
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int      sa_flags;
  void     (*sa_restorer)(void);
};
```

Mais il en faut pas se laisser impressionner : on repart d'une structure existante, et on modifie juste le pointeur de fonction `sa_handler`.

Un exemple :

```
// un compteur global
int cpt = 0;

// gestionnaire de signal
void fct(int s) {
    cpt++;
    std::cout << "M'enfin ! (" << cpt << ")"<< std::endl;
    if (cpt == 10) {
        std::cout << "Aie !!!" << std::endl;
        exit(EXIT_SUCCESS);
    }
}

int main(void) {
    // installation du gestionnaire pour SIGINT (Ctrl+c)
    struct sigaction s;
    if( sigaction(SIGINT, NULL, &s) == -1 ) exit(EXIT_FAILURE);
    s.sa_handler = fct;
    if( sigaction(SIGINT, &s, NULL) == -1 ) exit(EXIT_FAILURE);

    while(1) {
        std::cout << "RRRRR..." << std::endl;
        sleep(1);
    }

    return EXIT_SUCCESS;
}
```

Tubes (anonymes)

Les fichiers spéciaux les plus simples pour la communication inter-processus sont les tubes anonymes, appelés tubes, pipes ou tuyaux.

Les tubes fournissent un canal de communication unidirectionnel entre deux processus :

- ils ont une extrémité d'écriture et une de lecture ;
- ils ont une taille limitée et peuvent être remplis ;
- ils n'ont pas de nom et doivent donc être partagés après leur création.

man pipe

- `int pipe(int pipefd[2]);`
- `pipe()` creates a pair of file descriptors and places them in the array pointed to by `pipefd` :
  - `pipefd[0]` is for reading,
  - `pipefd[1]` is for writing.
- On success, zero is returned. On error, -1 is returned.

Pour échanger via un tube, deux processus doivent avoir un lien familial et se partager le tube : un processus sera écrivain, l'autre sera lecteur.

Tant qu'un processus du système possède un descripteur de fichier en écriture sur le tube, tout appel à `read()` sur le tube est bloquant.

Si un processus tente d'écrire sur un pipe avec `write()` alors qu'il n'y a aucun lecteur sur le pipe, cet écrivain reçoit le signal SIGPIPE.

Chaque processus doit fermer le descripteur qu'il n'utilise pas :

- le lecteur ferme le descripteur en écriture du pipe (`pipefd[1]`),
- le rédacteur ferme le descripteur en lecture du pipe (`pipefd[0]`).

Lorsque le rédacteur a fini d'écrire, il ferme aussi le descripteur en écriture, pour que le lecteur sache qu'il n'aura plus rien à lire, et se termine.

Toujours libérer les ressources non utilisées ! On a verra en TD/TP que ne pas fermer les descripteurs non-utilisés peut conduire à un interblocage.

Tubes nommés (fifo)

Pour que deux processus sans lien familial puissent échanger via un tube, il faut qu'il ait un nom dans le système de fichier : notion de tube nommé.

Un tube nommé ou fifo :

- est un tube qui a un nom dans le système de fichiers ;
- il est géré comme tout autre fichier :
  - emplacement dans le système de fichiers, droits,
  - ouverture avec `open()`, fermeture avec `close()`,
  - lecture avec `read()`, écriture avec `write()` ;
- c'est un tube, et une fois ouvert, il s'utilise comme tel.

Appel système : `int mkfifo(const char *name, mode_t mode);`
`name` est le nom du fichier, `mode` les droits d'accès.

Commande : `mkfifo [-m mode] path`
`path` et `mode` jouent les mêmes rôles que pour l'appel système.

Conclusion

Appels systèmes / primitives :

- pour la gestion des processus : `fork()`, `exec()`, `waitpid()`.
- pour l'utilisation des signaux : `kill()`, `sigaction()`.
- pour l'utilisation des pipes : `pipe()` (plus `read()` et `write()`).

En ligne de commande :

- gestion des processus : `ps`.
- pour les variables d'environnement : `env`, `export`.
- pour l'envoi de signaux : `kill`.

Nous verrons encore un autre moyen de communication entre processus : les sockets, pour la transmission de données en réseau.