Introduction aux systèmes d'exploitation

Partie 2 : Les processus

La structure des processus

Jalil BOUKHOBZA

UBO / Lab-STICC

Email : boukhobza@univ-brest.fr

Partie 2 : Les processus
La structure des processus

1. La notion de processus
2. Le pseudo-parallélisme
3. La hiérarchie des processus
4. Le contexte d'un processus

1. La notion de processus
1.1 Définition d'un processus

- **Un processus est un programme qui s'exécute.**

- Un processus est dynamique par opposition à un programme qui lui est statique.

1.2 Différence entre un programme et un processus

- Imaginez un informaticien qui prépare un gâteau d'anniversaire pour sa fille.
  - Recette = Programme
  - Informaticien = Processeur
  - Ingrédients = Données
  - Activité cuisine = Processus

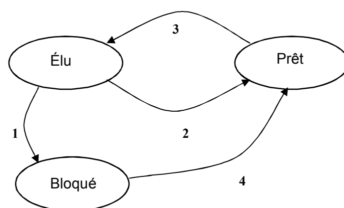
- Supposez maintenant que le fils de l'informaticien arrive en pleurant parce qu'il a été piqué par une abeille.
  - Le **processeur** " informaticien " basculera du **processus** " Activité cuisine " à un **processus** plus prioritaire " 1er soins ".

1.2 Différence entre un programme et un processus (2)

- L'idée clé est qu'un **processus** est <u>une activité</u> d'un certain type qui possède :
  - un programme (recette, livre de premier soin),
  - des données (ingrédients, trousse de soin),
  - ainsi qu'un état courant (un point précis dans la recette).
- Un processeur :
  - peut être partagé entre plusieurs processus,
  - en se servant d'un **algorithme d'ordonnancement** qui détermine
    - quand il faut suspendre un processus,
    - …pour en servir un autre (lequel?).

1.3 Diagramme simplifié des états et des transitions d'un processus

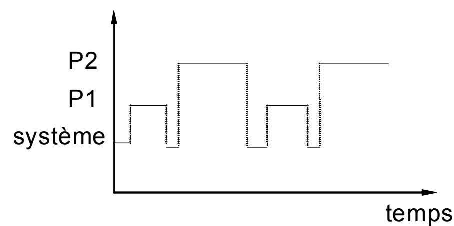
- A un instant donné un seul processus peut être actif (ÉLU) → monoprocesseur.
- Les autres processus pourront être :
  - soit en attente d'exécution (PRET),
  - soit bloqué en attente de ressources (BLOQUÉ)



1. *Le processus se bloque en attente de données/ressources.*
2. *L'ordonnanceur choisit un autre processus.*
3. *L'ordonnanceur choisit le processus.*
4. *Les données deviennent disponibles.*

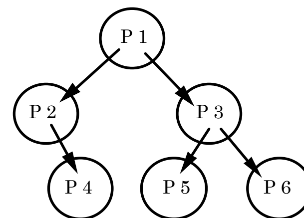
2. Le pseudo-parallélisme

- Le processeur à un instant donné ne peut exécuter réellement qu'un seul programme (sauf multicoeurs/multiprocesseurs !).
- Un système d'exploitation doit en général traiter plusieurs tâches en même temps (multi tâches).
- Il résout ce problème en passant d'un programme à un autre en exécutant chaque programme pendant quelques dizaines ou centaines de millisecondes (quantum de temps)



3. La hiérarchie des processus sous UNIX

- Au lancement du système, il n'existe qu'un seul processus, qui est l'ancêtre de tous les autres (processus 0: swapper/scheduler).
- Chaque processus peut lancer lui-même d'autres processus.
- Le processus créateur est le père, et les processus créés, les fils.
- Les processus se structurent sous la forme d'une arborescence.



4. Le contexte d'un processus

4.1. Introduction au contexte d'un processus
4.2. Composition du contexte d'un processus
4.3. Structure d'un processus
4.4. Composition standard d'un BCP (Bloc de Contrôle du Processus)
4.5. Commande 'ps'
4.6. Création d'un processus sous UNIX

4.1 Introduction au contexte d'un processus (1)

- Soient les deux commandes suivantes :
  - ls -t ordre chronologique
  - ls -l liste détaillée
- L'exécution de chacune de ces commandes va engendrer la création d'un processus :
  - un identifiant différent (**pid**),
  - des données différentes,
  - un état différent,

  malgré le même et unique programme (exécutable *ls*).

4.1 Introduction au contexte d'un processus (2)

- Le **contexte (image) du processus** contient tous les éléments nécessaires au système d'exploitation pour l'exécution d'un processus à un instant donné :
  - l'état des registres,
  - l'état des fichiers ouverts,
  - le contenu de la mémoire principale où l'on distingue :
    - le segment de code (accès en lecture uniquement)
    - le segment de donnée (accès en lecture/écriture)
    - le segment de pile
    - …
  - l'état du processus dans le système,
  - …
- Cet ensemble de données doit permettre au système de reprendre l'exécution d'un processus qui a été interrompu.

4.2 Composition du contexte d'un processus

- Le contexte d'un processus se décompose en deux parties :

1. **La structure interne du processus** qui sera contenu dans la mémoire principale sous la forme de plusieurs « segments » : **la pile**, **les données** et **le code**.
2. **Le bloc de contrôle du processus (BCP)** qui se décompose lui-même en :
   - *état du processus,*
   - *compteur ordinal,*
   - *pointeur de pile,*
   - *allocation mémoire,*
   - *Descripteurs de fichiers*
   - *Périphériques ouverts*
   - …

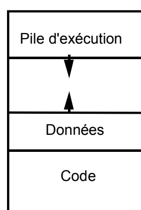
4.3 La structure interne d'un processus (1)

- Les compilateurs permettent de créer des fichiers exécutables dont le format peut ensuite être transformé en processus par le noyau du système.

- Schématiquement le format d'un exécutable comprend (format **ELF**) :
  - Une entête qui décrit l'ensemble du fichier, ses attributs et la description des sections qui suivent.
  - La taille à allouer pour les variables non initialisées (symboles).
  - Une section TEXT qui contient le code en langage machine.
  - Une section DATA qui contient les données initialisées.
  - d'autres sections : segments des données (initialisées), symboles, …

4.3 La structure interne d'un processus (2)

- Le chargement d'un exécutable en mémoire consiste à ***allouer un segment de mémoire pour chaque section*** :
  - Le segment de code pour la section TEXT.
  - Le segment de données pour la section DATA…



- *Le segment de code correspond aux instructions, en langage machine, du programme à exécuter.*
- *Le segment de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire (**tas**).*
- *Le segment de pile permet de stocker les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile. Lors d'un retour de fonction les paramètres et variables sont dépilées.*

4.4 Composition standard d'un BCP (1)

- Quelques champs typiques d'un Bloc de Contrôle d'un Processus (BCP) → gestion de processus, de mémoire et de fichiers

Processus	*Mémoire*	*Fichiers*
Registres Compteur ordinal État du programme Pointeur de pile Date de création Temps CPU utilisé Temps CPU des fils Date de la proch. alarme Pointeurs sur messages Bits signaux en attente PID	Pointeur sur code Pointeur sur données Pointeur sur pile Statut de fin d'exécution N° de signal du proc. tué PID Processus père Groupe de processus UID réel UID effectif GID réel GID effectif Bits des signaux	Masque `umask` Répertoire racine Répertoire de travail Descripteurs de fichiers UID effectif GID effectif

4.4 Composition standard d'un BCP (2)

- Les caractéristiques principales que l'on trouve dans un bloc de processus concernent :
  - L'identité du processus ainsi que celle de son père.
  - Le répertoire de travail du processus.
  - Le masque de création des fichiers.
  - La table des descripteurs de fichiers.
  - Les liens entre le processus et les utilisateurs.
  - …
- A la création d'un processus un très grand nombre de ses attributs sont hérités du processus père.

4.4.1 Identité du processus et celle de son père

- Un processus a accès respectivement à son PID et à celui de son père par l'intermédiaire respectivement de la fonction *getpid* et *getppid*.

```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
```

4.4.2 Répertoire de travail du processus

- Le répertoire de travail du processus est important car tous les chemins relatifs utilisés sont exprimés par rapport à celui-ci.
- La fonction *chdir* permet de changer le répertoire de travail du processus.
- La fonction *getcwd* permet de récupérer le chemin absolu du répertoire de travail courant.

```
#include <unistd.h>
int chdir (const char *chemin);
char * getcwd (char * buf, size-t taille);
```

4.4.3 La table des descripteurs de fichier

- Un processus hérite à sa création de tous les **descripteurs** de son père. Il a donc accès aux mêmes entrées dans la **table des fichiers ouverts** que son père.
- Il peut ensuite :
  - acquérir de nouveaux descripteurs par l'intermédiaire des primitives standards (open, creat, dup, ..)
  - ou en libérer par la primitive *close*.

4.5 La commande ps

- La command *ps* permet à un utilisateur quelconque d'obtenir la liste de processus :
  - appartenant à un ensemble particulier,
  - ainsi que certaines de leurs caractéristiques.
- Sans entrer dans le détail des options, regardons quelques cas d'utilisation :
  - ps
  - ps –l
  - ps -e

4.5.1 La commande *ps* sans option

- Fournit la liste des processus qui appartiennent à la même session que le processus shell depuis lequel la commande est lancée, avec pour chaque processus :
  - Le numéro par lequel le système identifie le processus (PID).
  - Le numéro du terminal de contrôle du processus (TTY).
  - Le temps cumulé d'exécution du processus (TIME).
  - Le nom du fichier correspondant au programme exécuté (CMD).

```
jboukh@Jalil:~$ ps
  PID TTY          TIME CMD
 3339 pts/5    00:00:00 bash
 3370 pts/5    00:00:00 ps
```


4.5.2 La commande *ps* avec l'option *-l*

- Permet d'obtenir pour chaque processus une liste d'informations plus complète :
  - L'identité du propriétaire réel du processus (UID).
  - L'identifiant du processus père (PPID).
  - L'indicateur de l'état du processus en mémoire (F).
  - L'état du processus (S) : R(actif), S (en sommeil), Z (zombi), ...
  - Des informations relatives à la priorité du processus (C/PRI/NI).
  - L'adresse du processus en mémoire ou sur disque (ADDR).
  - Taille du processus exprimée en nombre de blocs (SZ).
  - Adresse de l'événement attendu par le processus s'il est en sommeil (WCHAN).

```
jboukh@Jalil:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3339 18239  0  80   0 -  1606 -      pts/5    00:00:00 bash
0 R  1000  3371  3339  0  80   0 -   651 -      pts/5    00:00:00 ps
```

4.5.3 La commande *ps* avec l'option *-e*

- La commande ps avec l'option -e permet de visualiser tous les processus

```
jboukh@Jalil:~$ ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -   711 -      ?        00:00:01 init
1 S     0     2     0  0  75  -5 -     0 -      ?        00:00:00 kthreadd
1 S     0     3     2  0 -40   - -     0 -      ?        00:00:00 migration/0
1 S     0     4     2  0  75  -5 -     0 -      ?        00:00:00 ksoftirqd/0
5 S     0     5     2  0 -40   - -     0 -      ?        00:00:00 watchdog/0
…
```

- Les premiers processus sont des processus systèmes.
- Le processus *init* (1) correspondant à l'ancêtre de tous les processus :
  - Son père est le processus « idle » dont le pid est 0

4.6 Création d'un processus sous UNIX

- La création d'un processus peut se faire en une ou deux étapes :

  1. Duplication du processus à l 'aide de la fonction *fork()*.
  2. *[Recouvrement du code à l'aide de la fonction exec.]*

4.6.1 Duplication d'un processus (1)

- Un processus peut se dupliquer – et donc ajouter un nouveau processus – par la fonction :
  - **pid_t fork (void)**
- Le **code retour** pid_t est primordial :
  - -1 en cas d 'échec,
  - 0 on est dans le processus fils,
  - > 0 le n° du processus fils (PID) dans le père.
- Cette fonction transmet une partie du contexte du père au fils.
- À l'issue d'un *fork()* les deux processus s'exécutent **simultanément**.

4.6.1 Duplication d'un processus (2)

- Structure de code standard suite à un fork

```
int pid ;
pid = fork ();
switch (pid) {
      case –1 :
            printf (“Pbm lors de la creation du processus\n”);
      case 0    :
            printf (“Je suis le processus fils\n”);
      default   :
            printf (“Je suis le processus père\n”);
}
```

4.6.2 Recouvrement du code du processus inital (1)

- La création d'un nouveau processus peut se faire par le " **recouvrement** " du code d'un processus existant grâce à la fonction :
  - **int execl(char *ref, char *arg0, char *argn, NULL)**
- Les arguments de la fonction ***execl*** sont :
  - *ref* correspond au chemin (path) complet de l'exécutable,
  - *arg0* reprend en fait le nom du programme,
  - *arg1, …, argn* sont les arguments du programme.
- Il existe d'autres fonctions analogues dont les arguments sont légèrement différents : ***execle (), execlp (), execv (), execve (), execvp ()***.

4.6.2 Recouvrement du code du processus inital (2)

- Exemple de recouvrement :

```
main () {
        execl("/bin/ls", "ls", "-l", NULL);
        printf("Erreur lors de l'appel à ls \n");
}
```

- Remarque :
  - La partie de code qui suit l'appel d'une primitive de la famille exec **ne s'exécute pas**, car le processus où elle se trouve est remplacé par un autre.
  - Ce code ne sert donc que dans le cas où la primitive exec n'a pas pu lancer le processus de remplacement (le nom du programme était incorrect par exemple).

4.6.3 Principes de base de la genèse des processus

- La duplication d'un processus existant à l'aide du ***fork***. Ce dernier sera le père du nouveau processus.
- Le recouvrement du code du processus fils à l'aide des fonctions ***exec***.
- La terminaison d'un processus à l'aide de la fonction :
  - ***void exit(int statut)***
- L'élimination d'un processus terminé ne peut se faire que par son père, grâce à la fonction :
  - ***int wait(int * code_de_sortie).***
- Si le fils se termine sans que son père (vivant) l'attende, le fils passe à l'état ***defunct*** ou ***zombie*** .

Références bibliographiques

- Andrew Tanenbaum « *Systèmes d'exploitation* » 2ème édition Pearson Education.
- Jean-Marie Rifflet « *La programmation sous Unix* » 3ème édition EdiScience.
- Beauquier Joffroy « *Systemes d'exploitation - concepts et algorithmes* » Ediscience
- Daniel Bovet, Marco Cesati « *Le noyau Linux* » 3ème édition O'Reilly.