L2 Informatique

Systèmes d'exploitation I

Processus

Jessica BECHET

Processus

Notions

Processus

- Chargement et exécution noyau → fourniture de services au système et aux utilisateurs
  - Services fournis en dehors du noyau par des programmes système chargés au démarrage pour devenir des démons système qui seront exécutés tant que le noyau sera en fonctionnement
- Premier programme système lancé sous Linux : `systemd (init)`

- Aller plus loin : `pid_namespaces(7)`

Processus

- Processus dans un espace mémoire dédié : espace d'adressage du processus
  - Chaque processus dispose d'une zone mémoire indépendante et protégée des autres

- Le noyau régule l'exécution des processus
  - Ordonnancement : mécanisme de régulation des tâches

➢Garantir à l'utilisateur un comportement multitâche performant

Processus

- Instance de programme en train de s'exécuter
- Processus : représentation d'une tâche en cours d'exécution
  - Numéro d'identification : PID (*Process IDentifier*)
- Processus = données + code
  - Données : variables d'environnement
    - Exécutable, données du programme et sa pile, liste de fichiers ouverts, alarmes, etc.
    - Dépendent de l'OS et de sa version
- Unité élémentaire dans l'activité du système
- Processus dans des espaces mémoire distincts

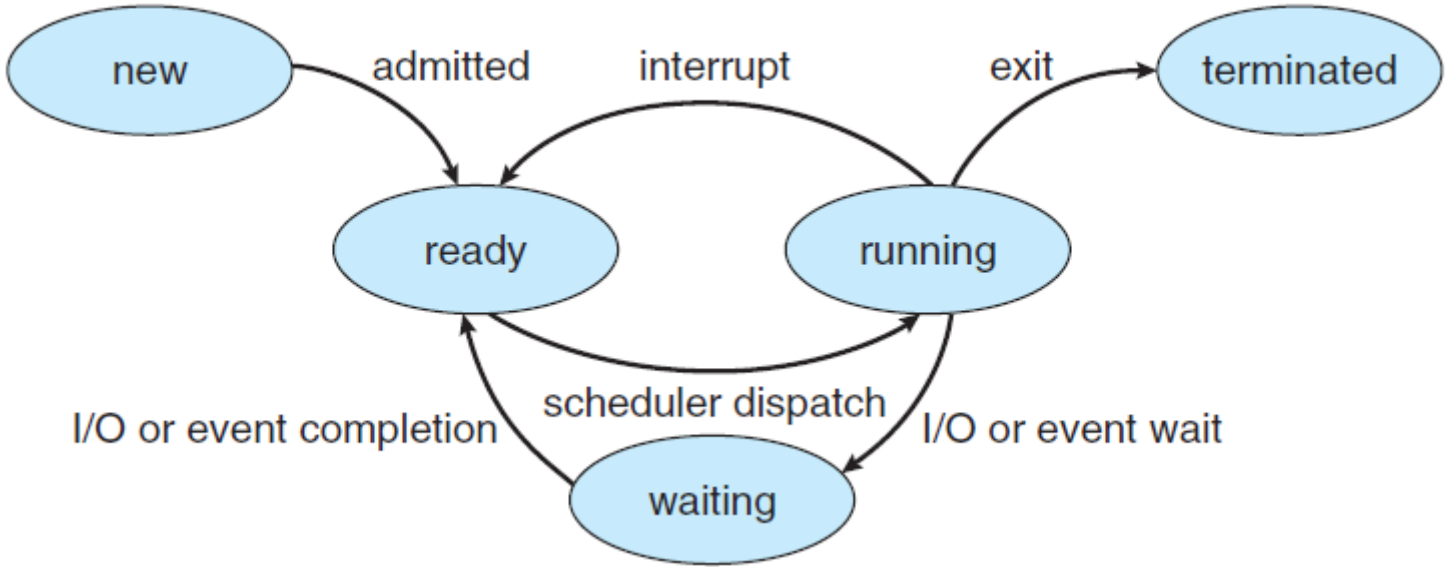
Processus

- Table des processus
  - Une saisie par processus
    - Etat du processus
    - Compteur de programme
    - Pointeur sur la pile
    - Allocation mémoire
    - Statut des fichiers ouverts
    - Informations d'ordonnancement
    - Etc.

Processus

Vie

Processus



Cycle d'un processus

Operating system concepts ; Abraham Silberschatz, Peter Galvin, Greg Gagne ; 10e edition.

Processus

- Fin d’un processus causée par
  - Terminaison normale (volontaire)
    - Fin du travail du processus
  - Terminaison avec erreur (volontaire)
  - Terminaison par erreur fatale (involontaire)
  - Tué par un autre processus (involontaire)

Processus

Primitives algorithmiques

- Créer processus

```
id = create_process()
```

- Terminaison processus

```
exit_process(state)
```

- Attendre processus

```
state = wait_process([id])
```

- Identifiant processus

```
id = id_process()
```

- Identifiant processus parent

```
idp = id_parent_process()
```

Processus

- Liste des processus en cours d'exécution

```
$ps x
```

- Liste des processus du système

```
$ps aux
```

- Liste des attributs des processus

```
$ps -f x
```

Processus

- Certains processus sont créés directement par le noyau
  - Par exemple **`init`** le premier processus

- Seule méthode de création d'un processus : `fork(2)`
  - Duplique le processus appelant
  - Deux processus identiques qui continuent d'exécuter le code
  - Deux processus avec PID différents
  - Processus initial : processus père
  - Copie : processus fils
  - Possibilité d'exécuter deux codes différents au retour de **`fork(2)`**

Processus

- Seule méthode de création d'un processus : **`fork(2)`**

```
#include <unistd.h>

pid_t fork(void);
```

  - Retour de **`fork(2)`**
    - **`0`** : on se trouve dans le processus fils
    - Valeur strictement positive : on se trouve dans le processus père
      - Valeur renvoyée : PID du fils
    - **`-1`** : échec
      - Code d'erreur (défini dans **`<errno.h>`**) inscrit dans la variable (globale) **`errno`**

Processus

- PID
  - Numéro d'identification
  - Type : `pid_t`
  - **`pid_t`** : entier signé
  - Lecture : type **`pid_t`** en **`long int`** (`%ld`) dans **`printf()`**

Exemple :

```
fprintf(stdout, "Mon PID est : %ld\n", (long) pid);
```

Processus

- Processus créé par **`fork(2)`** dispose d'une copie
  - Des données du processus père
  - De l'environnement du processus père
  - Table des descripteurs de fichiers, …

  ❖Héritage

- Appel-système **`fork(2)`** : économe
  - Méthode de *copie sur écriture*
    - Les données à dupliquer pour chaque processus ne sont pas directement copiées
    - Un seul exemplaire de ces données dans le système tant qu'aucun des processus ne modifie des informations dans ces pages en mémoire
    - *Coût faible en termes de ressources système*

Processus

- Connaître son propre identifiant PID : **`getpid(2)`**
  - Renvoie le PID du processus appelant
    - Correspond à celui affiché dans la colonne PID du résultat de la commande **`ps`**

```
#include <unistd.h>

pid_t getpid(void);
```

Processus

- PID du shell
  - Variable spéciale $$

```
$echo $$
```

Processus

- Connaître l'identifiant PID du processus père : **`getppid(2)`**
  - **`PPID`** : Parent PID
  - Renvoie le PID du processus père du processus appelant

```
#include <unistd.h>

pid_t getppid(void);
```

Processus

- Aparté Windows
  - Processus tous égaux, pas d'arbre
  - « Jeton » remis au processus parent à la création du fils mais ce jeton peut être passé à un autre processus

Terminaison d’un processus

Terminaison d'un processus

- Le processus fils termine avant le processus père
  - Il ne disparaît pas complètement
    - Devient un *zombie* en attendant que le processus père lise son code de retour
    - Si le processus père ne lit jamais son code de retour il est adopté par **`init`**
    - Lecture de retour d'un processus fils : **`wait(2)`**, **`waitpid(2)`**, etc.

Terminaison d'un processus

- Appel système **`wait(2)`**

```
#include <sys/wait.h>

pid_t wait(int *status);
```

- *Bloque le processus appelant jusqu'à ce qu'un processus fils se termine*
- Renvoie le PID du fils terminé
- Renvoie **`-1`** si le processus n'a pas de fils

➢On ne peut pas attendre la fin d'un fils en particulier

Terminaison d'un processus

- Appel système **`wait(2)`**

```
#include <sys/wait.h>

pid_t wait(int *status);
```

- Pointeur **`status`**
  - Non **`NULL`** : renseigné par une valeur informant sur les circonstances de la fin du fils
  - **`NULL`**

Terminaison d'un processus

- Appel système **`waitpid(2)`**

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

- Argument **`pid`** : déterminer le processus fils dont il faut attendre la terminaison
  - Prend entre autres les valeurs :
    - **`pid`**>0 : l'appel système attend la fin du processus de pid correspondant
    - **`pid`**=-1 : l'appel système attend la fin de n'importe quel fils
      - **`waitpid(-1, &status, 0)`** équivalent à **`wait(&status)`**

Terminaison d'un processus

- Appel système **`waitpid(2)`**

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

- Argument **`options`** : préciser le comportement de **`waitpid(2)`**
  - Peut prendre les valeurs (pouvant être associées avec un OU binaire) :
    - **`0`** : bloquer le processus père jusqu'à la terminaison du fils
    - **`WNOHANG`** : ne pas rester bloqué si aucun processus correspondant aux spécifications n'est terminé
      - **`waitpid(2)`** renvoie 0
    - **`WUNTRACED`** : accéder aux informations concernant les processus fils temporairement stoppés

Exemple

Pseudo-code

```
/*
 * Ce programme est un test de création de processus
 * Le processus fils affiche un message et le processus père
 * un message différent
 */
Programme principal
Début
            /* Creation du processus */
            id = create_process()
            Si id == -1 alors /*Echec */
                        Ecrire "Erreur"
            Sinon si id == 0 alors /*Processus fils */
                        Ecrire "Pid du fils :", id_process(), "et son père
:", id_parent_process()
                        exit_process(END_OK)
            Sinon /*Processus père */
                         Ecrire  "Pid  du  père  :",  id_process(),  "et  du
grand-père :", id_parent_process()
            /* Attente du fils */
            wait_process(id)

            Ecrire "Message entre la fin de l'attente du fils et la fin du
père"
            /* Terminaison du processus père */
            exit_process(END_OK)
            Finsi

            Ecrire "PID fils après tout : ", id
Fin
```

Langage C

```
/*
 * Ce programme est un test de l'appel-système fork
 * Le processus fils affiche un message et le processus père
 * un message différent
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(void) {
        pid_t pid_fils;

        pid_fils = fork();

        if (pid_fils == -1) {/*Echec */
                perror("fork");
                return EXIT_FAILURE;
        } else if (pid_fils == 0) {/*Processus fils */
                fprintf(stdout, "Pid du fils : %d et son père : %d.\n", getpid(), getppid());
                exit(EXIT_SUCCESS);
        } else {/* Processus père */
                fprintf(stdout,  "Pid  du  père  :  %d  et  du  grand-père  :  %d.\n",  getpid(),
getppid());
                /* Attente du fils */
                wait(NULL);

                fprintf(stdout,  "Message  entre  la  fin  de  l'attente  du  fils  et  la  fin  du
père.\n");
                /* Terminaison du processus père */
                return EXIT_SUCCESS;
        }

        fprintf(stdout, "PID fils après tout : %d.\n", pid_fils);
}
```

Exemple

```
etudiant@LinuxVM:~/ProgSys/Exemples$ ./exemple_fork
Pid du père : 5085 et du grand-père : 4092.
Pid du fils : 5086 et son père : 5085.
Message entre la fin de l'attente du fils et la fin du père.
etudiant@LinuxVM:~/ProgSys/Exemples$ echo $$
4092
etudiant@LinuxVM:~/ProgSys/Exemples$
```

- On remarque bien une « arborescence »
  - Le PID du père est celui du parent du fils
  - Le PID du grand-père correspond à celui du shell !

Terminaison d'un processus

- Par défaut, le processus parent reprend son exécution avant son enfant (sans `wait(2)`)

- Configurable dans `sched_child_runs_first` situé dans `/proc/sys/kernel`
  - Valeur par défaut `0`, mettre `1` pour un comportement inverse

Identification de l’utilisateur

Système multi-utilisateurs

- Exécution de façon pseudo-concurrente et indépendante des applications appartenant à plusieurs utilisateurs
  - Concurrente : applications actives au même moment et se disputent l'accès aux ressources
  - Indépendante : chaque application peut réaliser son travail sans se préoccuper de ce que font les autres applications des autres utilisateurs
- Emulé par attribution de laps de temps aux utilisateurs

Système multi-utilisateurs

- Authentification
  - Vérification de l'identité de l'utilisateur
- Protection
  - Protection contre les programmes erronés, mal intentionnés
- Comptabilité
  - Limitation du volume des ressources allouées à chaque utilisateur

Système multi-utilisateurs

- Utilisateur
  - Chaque utilisateur a un espace privé
  - `UID` : User IDentifier, numéro unique d'identification
- Groupe d'utilisateurs
  - Permet le partage de façon sélective du matériel entre utilisateurs
  - `GID` : Group IDentifier, numéro unique d'identification
- Super-utilisateur
  - Super-utilisateur / superviseur / root
  - Gestion des comptes des utilisateurs, tâches de maintenance, accès à tous les fichiers du système, etc.

Identification de l'utilisateur

- Trois types d'identifiants d'utilisateur par processus
  - UID réel
    - Celui de l'utilisateur ayant lancé le programme
  - UID effectif
    - Celui correspondant aux privilèges accordés au processus
  - UID sauvé
    - Copie de l'ancien UID effectif lorsqu'il est modifié par le processus

- UID root = 0

- En général UID effectif = UID réel
  - Différence suivant les applications

- UID : User IDentifier

Identification de l’utilisateur

- Essentiel des ressources sous UNIX : nœuds du système de fichiers
- Tentative d’accès à un fichier → vérifications d’autorisation par le noyau par rapport à l’UID effectif du processus appelant

Identification de l'utilisateur

- Commande **`shell`** pour connaître les identifiants et les groupes auxquels appartient l'utilisateur

```
$id
```

Identification de l'utilisateur

- Obtenir les UID

```c
#include <unistd.h>     /* Pour les appels système */
#include <sys/types.h>  /* Pour le type uid_t */

uid_t getuid (void);    /* UID réel */
uid_t geteuid (void);   /* UID effectif */
```

- `uid_t` :
  - Entier non signé sur 32 bits
  - Utiliser la conversion `%u` pour la conversion dans `printf(3)`

Identification de l'utilisateur

- Fixer les droits d'accès dans le shell

```
$chmod
```

- Permission spéciale pour les exécutables binaires (dans le shell) : **set_UID**
  - Permet à un utilisateur ayant les droits en exécution sur le fichier d'exécuter le fichier avec les privilèges du propriétaire du fichier

```
$chmod +s nom_fichier
```

Identification de l'utilisateur

- Il existe d'autres appels système :
  - Identification du groupe
  - `setuid(2)`
  - …

Exécution de programme et processus

Exécution d'un programme

- Utilisation des fonctions **`exec`** sans quitter le programme principal
  - Duplication du processus appelant **`fork(2)`**
  - Appel de **`exec()`** dans le processus fils
  - Adopté dans ce cours

Exécution d'un programme

- Fonction `popen(3)`/`pclose(3)`
  - Combinaison de `fork(2)`, exécution (shell) programme et `pipe`
    - Tube unidirectionnel → pas vu dans ce cours
  - Permet d'envoyer d'écrire sur le flux d'entrée ou récupérer le flux de sortie

  - Failles de sécurité pour les applications set-UID

Exécution d'un programme

- Fonction : `system(3)`
  - Exécute la commande shell donnée en paramètre
    - Comportement peut dépendre des interpréteurs shell
  - Failles de sécurité pour les applications set-UID
  - Attend la terminaison du fils avant de continuer l'exécution
    - Contrairement à un `fork(2)`/`exec()` où la terminaison du processus fils peut être attendue à un endroit choisi

Méthodes de terminaison d’un processus

Modes de terminaison d'un programme

- `return` du `main`
- Appel de la fonction `exit(3)`
- Terminaison par signaux (`kill(2)`)

Terminaison normale d'un processus

- Terminaison normale
  - Appel de la fonction **`exit(3)`**
  - return du main (équivalent à **`exit(3)`** dans ce cas)

```
#include <stdlib.h>

void exit (int status);
```

- **`status`** : statut de sortie du programme
- **`exit(3)`** ne retourne pas (!)

Terminaison normale d'un processus

- Fonctions enregistrées avec **`atexit(3)`** et **`on_exit(3)`** exécutées dans l'ordre inverse d'enregistrement
  - Permet de définir des fonctions spécifiques de « nettoyage »
- Fermeture des flux, mémoire allouée libérée
- **`_exit(2)`** est appelé
  - Fermeture des descripteurs (attention pas le cas des flux)
  - Envoi d'un statut de terminaison
  - Tout processus fils est affecté à un autre processus
  - …

Merci !