

Gestion des processus et des ressources par un OS

22 septembre 2025

Introduction

Référence

Le livre

Operating Systems : Three Easy Pieces

- ▶ disponible sur www.ostep.org
- ▶ pour les NSI de dumezil, demander au prof

Le code

Dépôt github à cloner :

<https://github.com/remzi-arpacidusseau/ostep-code>

```
git clone https://github.com/remzi-arpacidusseau/ostep-code
```

```
cd ostep-code
```

```
cd intro
```

```
make
```

Les outils

Ligne de commande

- ▶ `top`
- ▶ `ps` : liste les processus `ps`, `ps aux`, `ps afx`
- ▶ `kill PID / kill -9 PID`
- ▶ `killall nomduprogramme`

Les systèmes de fichiers virtuels (VFS)

- ▶ `/proc` : système de fichiers processus : informations du noyau sur les processus
 - ▶ `/proc/cpuinfo`
 - ▶ `/proc/meminfo`
 - ▶ `/proc/PID/...`
- ▶ `/sys` : informations de l'arbre des périphériques

Exécution simultanée

Le programme

```
int main(int argc, char *argv[])
{
    char *str = argv[1];

    while (1) {
        printf("%s\n", str);
        Spin(1);
    }
    return 0;
}
```

Exécution simultanée

```
cat /proc/cpuinfo
```

```
./cpu 1 & ./cpu 2 & ./cpu 3 & ./cpu 4 &
```

```
# résultat de l'exécution...
```

```
# Les programmes s'exécutent "ensemble"
```

```
# 1 2 3 4 1 3 2 4 1 3 2 4 1 3 4 2 ...
```

```
./cpu 1 & ./cpu 2 & ./cpu 3 & ./cpu 4 &\
```

```
./cpu 5 & ./cpu 6 & ./cpu 7 & ./cpu 8 &\
```

```
./cpu 9 & ./cpu A & ./cpu B & ./cpu C &\
```

```
./cpu D &
```

```
# résultat de l'exécution...
```

```
# Les programmes s'exécutent "ensemble"
```

```
# On a des variations de l'ordre des sorties
```

```
D " " "
```

```
1 1 1 2 1 1 1 2 1 1 1 2 1 1 1 2
```

Killall (à installer)

```
sudo apt update
```

```
sudo apt install psmisc
```

```
killall cpu
```

Conclusion

`cat /proc/cpuinfo | grep processor | wc -l` donne le nombre de “processeurs” (coeurs) du système

On a été capables de faire tourner plus de programmes en même temps qu’il n’y a de processeurs. Le système a donc une méthode pour “partager” le processeur entre les différents programmes.

Espace mémoire

Le code

```
int main(int argc, char *argv[]) {  
    int *p;  
    p = malloc(sizeof(int));  
    assert(p != NULL);  
    printf("(%d) addr pointed to by p: %p\n", (int) getpid(), p);  
  
    *p = atoi(argv[1]); // assign value to addr stored in p  
    while (1) {  
        Spin(1);  
        *p = *p + 1;  
        printf("(%d) value of p: %d\n", getpid(), *p);  
    }  
    return 0;  
}
```

Exécution

```
$ ./mem 100 & ./mem 200
```

```
# (2710) addr pointed to by p: 0x555555756260
```

```
# (2711) addr pointed to by p: 0x555555756260
```

```
# (2710) value of p: 101
```

```
# (2711) value of p: 201
```

Conclusion

Dans chaque processus, le pointeur mémoire p a la même adresse¹ en mémoire, or les valeurs pointées par p dans chaque programme sont différentes.

Le système a donc une méthode pour “virtualiser” la mémoire pour les différents programmes.

1. En situation complètement réelle, les processus n'ont pas exactement la même adresse, pour éviter certains types d'attaques, les adresses mémoires sont aléatoires. On peut désactiver ce comportement avec `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`, ce qui a été fait pour cette démo.

Simultanéité

Le code I

```
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
```

Le code II

```
Pthread_create(&p1, NULL, worker, NULL);  
Pthread_create(&p2, NULL, worker, NULL);  
Pthread_join(p1, NULL);  
Pthread_join(p2, NULL);  
printf("Final value    : %d\n", counter);  
return 0;  
}
```

Exécution

```
./threads 100
```

```
# Initial value : 0
```

```
# Final value   : 200
```

```
./threads 100000
```

```
# Initial value : 0
```

```
# Final value   : 141148 (variable)
```

Conclusion

Les deux processus tournent simultanément en accédant à une ressource partagée (l'espace mémoire counter).

La valeur de counter est lue, incrémentée et réécrite de manière non *atomique*. Dans ce cas on peut avoir des *situations de compétition* (race conditions).

counter contient 20

P1: lecture (registre = 20)

P2: lecture (registre = 20)

P1: incrémentation (registre = 21)

P2: incrémentation (registre = 21)

P1: écriture (counter = 21)

P2: écriture (counter = 21)

La solution est d'utiliser des méthodes **d'exclusion mutuelle** (mutex) ou

Gestion des processus

Notion de processus

Définition

Un programme est un ensemble d'instructions machine et de données. Un **processus est un programme en cours d'exécution**

Lancement d'un processus

Deux appels systèmes :

- ▶ **fork** : appel système permettant la création d'un nouveau processus (fils) copie conforme du processus appelant (père). Valeur de retour : PID dans le processus père, 0 dans le processus fils, -1 si erreur.
- ▶ **exec** : charge le code et les données statiques d'un programme et remplace le segment de code courant par celui-ci.

On a donc un **arbre des processus**, avec des relations père/fils, des processus frères...

Le premier processus

- ▶ au lancement du système : création du processus 0, fils de rien
- ▶ le processus 0 lance un processus `init`, qui crée ensuite les processus nécessaires au bon fonctionnement du système

Voir l'arbre des processus :

```
ps afx
```

Ordonnancement

Stratégie

Lorsque plusieurs processus doivent tourner sur un processeur, comment organiser leur exécution ?

Sans interruption du processus :

- ▶ FIFO = la queue à la caisse du supermarché
- ▶ SJF (shorter job first) ~ caisse - de 10 articles

Avec interruption du processus : ordonnancement préemptif utilisant **context switch / timer interrupts**

- ▶ partage du temps
- ▶ prise en compte des entrées sorties

Les états d'un processus

- ▶ **Running** / élu : en cours d'exécution réelle sur un processeur
- ▶ **Ready** : prêt à être exécuté, mais l'OS a choisi de ne pas le faire tourner à ce moment précis
- ▶ **Blocked** : en attente d'une ressource (ex : lecture sur disque dur)

L'OS est responsable des changements d'état des processus (élection, blocage...)

Utiliser la commande top pour suivre les processus en cours

`top`

Priorité

La décision d'élection de l'**ordonnanceur des tâches** fait aussi intervenir un **niveau de priorité** : `nice`

- ▶ la plus élevée = -20 (la moins “sympa”)
- ▶ la plus basse = 19

Si plusieurs processus sont en concurrence pour l'élection, le processus le moins “nice” bénéficiera de la plus grande quantité de processeur.

```
man nice
```

```
man renice
```

Exemple I

```
import sys
import datetime
print("Waiting to start")
i=0
while i < 10000000:
    i+=1
i=0
start = datetime.datetime.now()
while i < 100000000:
    i+=1
    if i%1000000 == 0:
        print(datetime.datetime.now(),sys.argv[1], i)
print("Done ", sys.argv[1], "in", datetime.datetime.now() -
    ↪ start)
```

Exemple II

Exécution à deux priorités différentes

```
nice -n 0 python3 loop.py A & nice -n 20 python3 loop.py B &
```

Done A in 0:00:18.003129

Done B in 0:00:18.680570

Partage de ressources

Mécanisme de blocage

Un processus accédant à une ressource non partageable peut la bloquer
(acquisition d'un mutex)

Interblocage

<https://fr.wikipedia.org/wiki/Interblocage>

Deux mutex (M1 et M2), deux processus légers (P1 et P2) et la séquence d'opération suivante :

- ▶ P1 acquiert M1.
- ▶ P2 acquiert M2.
- ▶ P1 attend pour acquérir M2 (qui est détenu par P2).
- ▶ P2 attend pour acquérir M1 (qui est détenu par P1).

Dans cette situation, les deux processus sont définitivement bloqués.

Solution :

- ▶ acquérir les mutex dans le même ordre
- ▶ détecter les deadlocks (ce n'est pas vraiment une solution)