Systèmes d'exploitation : processus

William SCHMITT

2018-2019

1 Activités parallèles

Exemple d'activités parallèles :
— Interactions avec le matériel
— Abstraite : logiciel, utilisateurs

1.1 Mémoire

Les activités parallèles sont en **compétition et en coopération** pour faire le travail demandé par l'utilisateur. Il s'agit d'écrire des programmes permettant de réaliser ces activités, en séparant les programmes par activité. Intervient alors la notion de cohabitation entre les données et programmes différents, sachant qu'on ne dispose que d'un seul espace mémoire et un seul CPU.

La notion d'adresse virtuelle est introduite pour isoler les programmes, à l'aide d'indirections dans les adresses manipulées par les programmes. Chaque programme est isolé, permettant de garantir de la sécurité.

L'espace virtuel est coupé en pages (**pagination**).

Il manque un bout ici.

1.2 Processus et partage du temps

Un millier de processus sur un système Linux, selon les paquets installés. On a donc beaucoup de processus, peu de processeurs/coeurs. Le temps CPU est coupé en tranche de temps, les **quanta**. Initialement, le temps était géré par les processus eux-mêmes, mais en cas de boucle infinie par exemple, ils n'étaient pas capables de rendre la main.

Modalités de changement de processus
— Tick horloge

1. Tock horloge déclenche interruption à la fin du quantum
2. Sauvegarde minimale (hardware, sauvegarde PC)
3. Traitant d'interruption : sauvegarde complète (contexte du processus)
4. Commutation de contexte : **context_switch**
5. Fin du traitant (du processus qui a pris la main entre temps) : restauration
6. Retour d'interruption : restauration minimale (PC)

— I/O
— Terminaison de processus
— Synchronisation de processus
— sleep

Contexte d'un processus Un processus *comporte* :
— du code : en RAM
— des données
  — en RAM
  — dans les registres CPU : c'est ces données qui sont sauvegardées & restaurées.

En particulier, la sauvegarde du registre PC (instruction courante) et effectuée par le matériel.

Composition d'un processus
— Processus
  — Espace de mémoire : virtuel et privé.
  — 1 programme : données.
— Thread (connu par l'OS) : partie "exécutive" d'un processus : exécute 1 fonction.
— *Thread users* (connu par le programme) (= fibre, task, co-routine, go-routine)

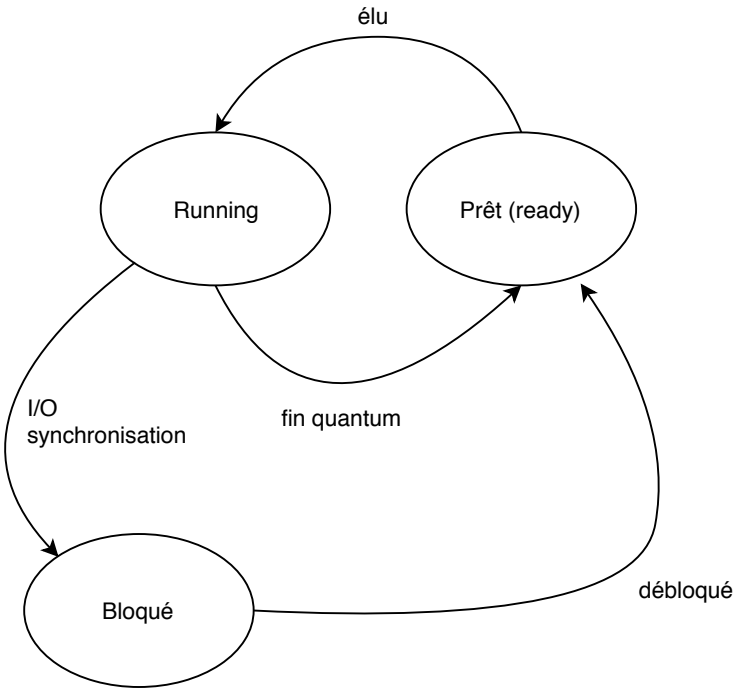
Les deux premiers points sont souvent confondus (ex. la commande ps donne le numéro de processus, mais le thread a le même numéro).

Ordonnanceur Cette fonction est appelée à chaque changement de processus et choisit le prochain processus élu parmi les processus **prêts**, on essaie de garantir la localité des informations dans le cache, selon certains critères.

Il s'agit d'optimiser le coût des interruptions (20k cycles) et de leurs traitants.

Etat des processus Une des structures de données fondamentales d'un système d'exploitation est une **liste** de processus dans l'état *prêt*.

Il manque le morceau avec les explications sur la localité



1.3 Partage

	Processus	Thread
Code	1	1 fonction du code
Pile (variables locales)	Autant que de thread	1 par thread
Fichiers (open, close, read, write)	Descripteurs de fichiers ouverts	-
Réseaux (cf. fichiers)		
Exit (terminer un **processus**)		
Données = variables globales (cf. code)		

2 Processus (et threads) sous UNIX

2.1 Outils

`ps`, `htop`, `top`, `px`, `pstree`

2.2 Taille (en mémoire) d'un processus

La taille peut faire référence à plusieurs choses :
— Sa taille **complète** : si on mettait tout le processus en mémoire. La plupart des données sont inutiles.
— Taille en RAM : indicateur trompeur car certaines portions (comme les libs) sont partagées

Des outils comme `smem` partagent équitablement l'espace partagé par le nombre de processus utilisant des ressources partagées.

Lorsque le système manque de mémoire, le noyau linux appelle `OOM_killer`, une fonction qui tue le plus gros processus (à certaines exceptions près).

2.3 fork

Exemple de `fork()` :

```
int r = fork(); // crée une copie du processus appelant
switch(r) {
    case -1:
        perror("mon fork:");
        break;
    case 0:
        // je suis le fils
        break;
    default:
        // je suis le père de "r" (PID de mon fils)
}
```

Le souci de `fork()` est qu'on obtient le même processus.

2.4 exec(p/vp/lp/e/le)

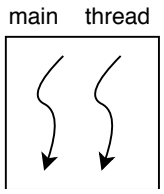
La famille des `exec` fait du recouvrement : change le programme. L'exécution démarre donc au début du programme, mais cela ne concerne que l'exécution : les fichiers ouverts dans l'appelant sont accessibles dans l'appelé.

2.5 wait, waitpid

Utilisable uniquement par le père : donc un processus orphelin ne peut pas être attendu.

2.6 Signaux

Utilisable avec `kill(pid, signal_nr)`. Si CTRL+C tue les processus, il laisse le temps aux processus de fermer ses fichiers, contrairement à `kill -9`



3 Threads

```
auto th = new thread([]( { cout <<"hello"<<endl; }));
th->join(); // Attend la fin du thread
```

La communication entre threads est facile (variable globale), mais l'écriture n'est pas forcément atomique.

Exemple : multithread en C11 qui incrémente variable partagée par tous les threads : le résultat est bien inférieur à celui souhaité.

3.1 Gestion de la concurrence

Il s'agit donc de les modifier les variables d'état globales un à la fois, via (par exemple) :
— Instructions atomiques : mais elles sont coûteuses et ne peuvent modifier qu'un mot à la fois.

3.1.1 Exemple

Enoncé Exemple de base : 2 threads exécutant `A()` et `B()` :

```
int a = 10;
A() {
    a = 20;
}
B() {
    printf("%d\n", a);
}
```

But : afficher 20 (càd que `A()` soit finie avant que `B()` ne soit lancée)

Solution 1 : booléen

```
bool fini = false;
int a = 10;
```

```
A() {
    a = 20;
    fini = true;
}
B() {
    while (!fini); // attente active (c'est mal !)
    printf("%d\n", a);
}
```

Cette stratégie ne fonctionne pas forcément : sur une architecture ARM l'ordre d'affectation des variables dans `A()` n'est pas garanti. L'attente active utilise du temps CPU, ce qui est mauvais notamment pour la batterie (fréquence au maximum pour tester une variable qui ne change pas).

Solution 2 : moniteurs Inventés par Tony Hoare, le principe des moniteurs est de faire tourner des fonctions en exclusion mutuelle. Pour gérer l'attente, Hoare introduit des *variables de condition* : des files d'attente de threads avec lesquelles on peut interagir grâce à wait et signal.

```
mtx_t m; // un mutex
cnd_c c;
bool fini = false;
int a = 10;
A() {
    mtx_lock(&m);
    a = 20;
    fini = true;
    cnd_signal(&c);
    mtx_unlock(&m);
}
B() {
    mtx_lock(&m);
    while (!fini);
        cnd_wait(&c, &m); // lock automatique au réveil et unlock automatique au wait
    printf("%d\n", a);
    mtx_unlock(&m);
}
```

L'utilisation de `cnd_wait` permet d'éviter les cas où `B()` à la main avant `A()` et garde l'exclusion mutuelle pour exécuter son `wait()`.