Systèmes d'exploitation

William SCHMITT & Othmane AJDOR

2018-2019

1 Introduction

1.1 Bibliographie

Tanenbaum : Modern Operating Systems
Silberschatz : Operating Systems Concepts

1.2 Plan

Cours/TD
- Moniteurs
- Sémaphores
- futex

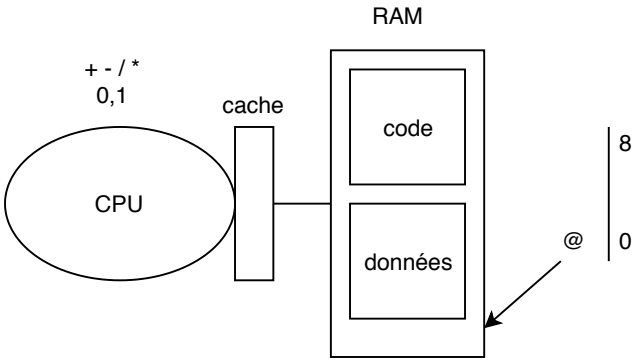
TP (pas à rendre)
- 1ère période : rappels de C
- 2ème période : mmap

TP (à rendre)
- Allocateur mémoire virtuel (malloc) : révisions sur les pointeurs
- Shell (fork, exec, redirection IO)
- Thread (lecteur vidéo multithread)

Présentation scientifique : article de usenix.org (de 2019, 2018 ou dans les *best papers* des 3 dernières années) à présenter devant la classe.

1.3 Examen

- QCM : 1h (20 à 25 questions)



— TP :
  — 2h (75%)
  — Présentation scientifique (25%)

Notation : 50/50.

1.4 Fibonacci

Comparaison avec/sans printf, ratio de performance : 397

Avec redirection de stdout dans /dev/null, le ratio plonge à 31.

Lorsque les tampons sont pleins (dans le terminal), on bloque les producteurs de données le temps de vider les tampons (et donc d'afficher les résultats).

Facteurs décorrélés de l'algorithme permettant des gains :
— IO
— Gestion de la mémoire

2 Hardware

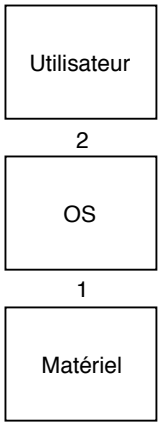
On se base sur un modèle simpliste.

Depuis les années 1990, la vitesse du CPU a dépassé celle de la mémoire. En un cycle CPU (pour un CPU à 4GHz), un photon peut parcourir 7,5cm. On rajoute à cette époque du cache dans les CPU pour compenser la lenteur de la mémoire.

De plus, la taille de la mémoire a explosé.

3 Système d'exploitation

Dans les premier ordinateurs, on réglait l'état mémoire à la main avec des interrupteurs.



Un OS est à la base composé de différentes routines qui ont deux buts :

1. Gestion du matériel (partage, arbitrage, partition, droits/utilisateurs)
   — Processeurs (dont la complexité augmente)
   — Disques/SSD/Burst buffer/Bandes magnétiques
   — RAM
   — Interfaces réseau
2. Abstractions
   — Fichiers, répertoires, systèmes de fichiers
   — processus
   — utilisateurs
   — socket réseau

3.1 Composition d'un OS

Le noyau Linux est constitué d'environ 18 millions de lignes en C. Ingérable par une seule personne, il est décomposé en modules exposant des API puis compilé en un programme comme les autres, à quelques exceptions près.

Mode d'exécution un OS s'exécute dans un mode spécial et a accès à des instructions spéciales.

Interruptions et exceptions permettent de passer du mode user au mode kernel. La différence entre les deux : les interruptions sont masquables.

Bibliothèque système côté utilisateur permettant (notamment) d'éviter de coder les interruptions à la main.

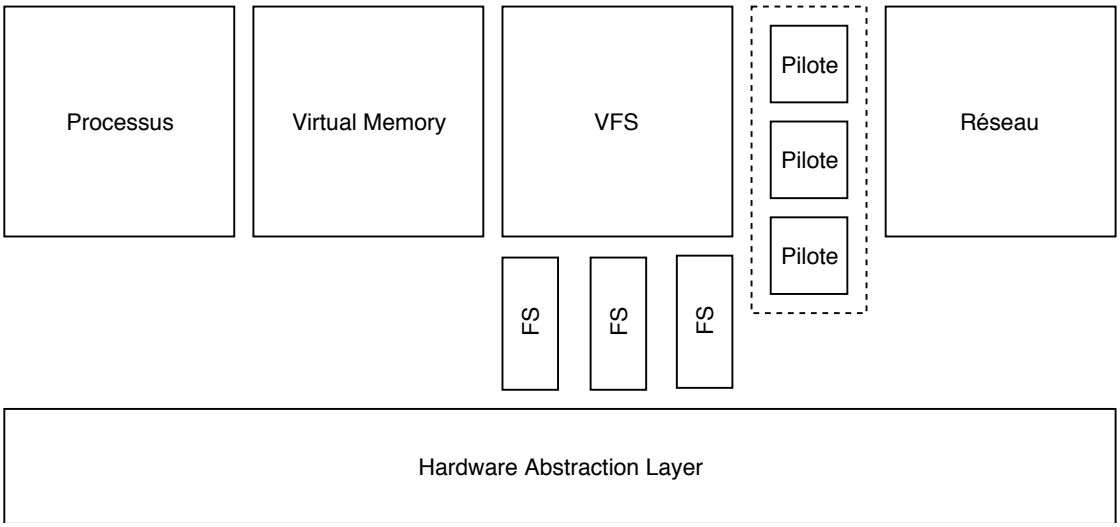
S'il s'agit d'un programme comme les autres : où et quand est-il en RAM ? Où et quand s'exécute-t-il ?

Il est toujours en RAM (notamment pour pouvoir répondre aux interruptions), avec certaines nuances : tout l'OS n'est pas en mémoire ! Pour Linux, seuls certains modules sont chargés, notamment pour des drivers. Sous Windows, les parties non utilisées sont sur disque (swap).

En termes d'exécution, il n'est exécuté que lors d'interruptions/exceptions.

Déroulement du boot
— Intel ME (management engine), codé sur le processeur et peut faire tourner Minix.
— BIOS/UEFI : exécuté directement par le CPU au boot, accessible comme de la mémoire normale, pour chercher le chargeur et le mettre en mémoire. Fourni par le fabricant de mobo, il y a déjà des interactions avec le matériel avant le chargement de l'OS. L'accroissement de complexité de l'UEFI (possibilité de surfer) présente des problématiques de sécurité.
— Chargeur : mise en RAM des données sur disque
  — Windows
  — GRUB
  — Autre...
— Initialisation
— Vecteurs d'interruption/exception
La maîtrise de la machine physique devient de plus en plus lointaine, même au niveau des OS.

3.2 Modularité



3.3 Plan du cours

— Processus / Threads
— Synchronisation / prog concurrente
— Ordonnancement (Théorie+pratique)
— Mémoire -> coté user
— Pagination, mémoire virtuelle
— FileSystem (période 6)

— Sysd (intro)
— Multi Proc (programmation OpenMP)

Les **systèmes de fichier virtuels** sont une API haut niveau pour l'ouverture, la fermeture, la lecture, l'écriture de fichiers. Les filesystems sont par exemple ext4, btrfs, fat, fat32, ntfs.

4 Histoire

"L'histoire de l'informatique, ça remonte aux machines automatiques pour faire des trucs."

4.1 Préhistoire

— Babylone
— Machine Héron d'Alexandrie

4.2 Histoire

Epoque mécanique
— Pascal
— Babbage : machine universelle programmable
— Ada Lovelace (if, boucle)

Antiquité
— Turing
— ENIAC

Moyen-Âge
— IO : cartes perforées
— Tampon : copie des IO en RAM

Monde moderne (1960)
— MULTICS : OS moderned (processus, droits, fichiers, multi-utilisateurs)
— 1969 : Mini-ordinateur : PDP11
— UNIX : (asm pour la première version, C pour la deuxième)
— 1974 : Micro-ordinateurs
  — Grand public : pas assez puissants pour faire tourner UNIX, donc utilisation de CP/M (8 bits, 16ko RAM) puis DOS en 1981
  — Station de travail : permettent de faire tourner UNIX
— 1987 : Minix, reste petit pour permettre d'être un outil pédagogique (bouquin avec src + explication)
— 1991 : Linux

— internet
— GPL : possibilité de voir et modifier le code, obligation de distribuer le code modifié

4.3 1990-

— Linux (1991)
— Internet
— GPL libre (source, modification, reverser)

5 Processus

5.1 Activités parallèles

— materiel
— abstraite : logiciel, utilisateurs
— competition (tout ce qui n'est pas partageable ei : entrée RAM)
— cooperation (coexistence de deux programmes qui doivent tourner en meme temps)

Pour gérer cela, il faut des programme**S**
-> séparation des activités
-> cohabitation des données et programmes
Utilisation de mémoire virtuelles au lieu de celles physiques.
Indirection dans les adresses manipulées par les programmes.

```
int a; // juste la somme de la taille
const in b = 10; // read only
int c = 20;
int main(integer c, char **argv){
    int *d = malloc(atoi(argv[1])); // allocation dynamique de 20 oct
    free(d);
    return 0;
}
```

Pagination : espace virtuel est coupé en page de 4ko pour n'avoir que les morceaux intéressants de la mémoire

5.2 Processus

5.2.1 Quantum

— Partage de plusieurs processus ($\tilde{1}000$) (peu de coeurs/processeurs)
— Partage du temps CPU en tranche (quantum)
— Horloge -> interruption -> traitant -> context_switch -> fin traitant restauration -> retour d'interruption restauration minimal
— I/O
— terminaison
— Synchronisation
— sleep

5.2.2 Contexte d'un processus

— code -> RAM
— données -> RAM + registres du processeur | sauvegarde restauration
— instructions i1 | i2

5.2.3 Processus

Processus Espace de mémoire (virtuelle/privée) + 1 programme (données)

Threads connus par l'OS partie executive d'un processus, execute *1* fonction

Threads user connus par le programme (fiber, task, co-routine, go-routine...)

5.2.4 Scheduler / Ordonnanceur

Fonction qui choisit le prochain élu à partie d'une liste (structure de données centrale par l'OS) de processus prêts.
Le scheduler est exécuté à chaque changement de processus.

6 Virtualisation

6.1 Niveaux de virtualisation

6.1.1 Hardware

— mainframe IBM : fait tourner des OS où chacun croit être le seul sur la machine
— Intel (peut faire pareil, marche +/- bien) -> Xen
4 mode de protection du CPU (3 user à 0 kernel), mode -1 permet de changer la position de l'indice PC, qui normalement ne peut pas être changée.

6.1.2 OS

VM VirtualBox, Qemu ...où l'OS hôte fait tourner des OS invités KVM permet de mettre des instructions de l'OS invité sur le matériel de l'OS hôte.

Container Container Docker, on ne simule pas un OS complet au dessus d'un autre, mais on va par exemple séparer le Filesystem, le réseau, allouer la RAM différemment. Il existe un seul OS dans ce cas mais certaines bibliothèques peuvent être rendues privées pour un conteneur mais pas pour l'autre.

JAVA -> JVM Java compile le code en instruction assembleur qui seront exécutées sur une JVM qui tourne sur l'OS. Python et Perl font à peu près la même chose (JiT).

6.1.3 Sondage Partage Processus/Thread

	Processus	Threads
code	1	1 fonction du code
pile (variables locales)	autant que de threads	1 par thread
fichiers (open, close, read, write)	descripteurs de fichiers ouverts	_____
réseaux (comme les fichiers)		
exit (terminer un processus)		
données (variables globales/comme le code)		

7 Processus sous UNIX

7.1 Outils

- ps, htop, top, px, pstree
- taille en mémoire, taille complète (mais la plupart des données sont inutiles)
- taille en RAM (trompeur comme plusieurs portions de la mémoire sont partagées, e.i : libc, printf, malloc...). Smem pour donner des info
- OOM_Killer -> tue le + gros (mais pas X11 quand meme)

7.2 Fork

```
int r = fork(); // cree une copie du processus appelant
switch(r){
        case -1:
                perror("mon fork:"); //mon fork:not enough memory ou autre
                break;
        case 0:
                lefils
                break;
        default:
                le pere de "r" (PID de mon fils)
}
```

Après l'appel de fork, le processus fils continue ce qu'on père était en train de faire.

7.3 exec

exec [p|up|lp|le|e]

```
        execvp("file", tab);
        // arg1: nom programme apres lancement, arg2: nom commande,
        // arg3: parametres de la commande lancee
        execlp("ls", "ls", "-l");
```

7.4 Wait

wait(), waitpid() (exclusivité du père), attend la mort du processus

Lors du lancement de ls -R /, le shell attend la fin du ls avant de refaire un prompt. Si on utilise ls -r / &, le shell n'attend pas

7.5 Signal

kill(pid, SIGNumber)
kill -9 -1, tuer tous les processus brutalement, meme le shell kill -15 -1, tue plus gentillement, les applis ont le temps

8 Threads

8.1 Création d'un thread

```
int main(){
        auto th = new thread([](){cout << "hello" << endl;});
        th->join();
}
```

8.2 Gestion de la concurrence

Variables d'états partagées Stratégie : modification 1 à la fois
— instructions atomiques (couteuses, marchent 1 mot à la fois soit <- très localisé)

Exemple de base : 2 threads executant A() et B() : le but est d'afficher 20

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

Ne marche pas

```
int a = 10;
bool fini = false;
A(){
        a=20;
        fini=true;
}
B(){
        while(!fini);
        printf("%d\n", a);
}
```

Ne marche toujours pas

Tony Hoare, le meme gars qui a inventé le qsort, a proposé l'utilisation de monitors, des fonctions en exclusion mutuelle.
On utilise aussi des variables de condition, elles servent à attendre jusqu'à ce qu'on nous reveille (wait, signal).

```
mtx_t m; // mutex
cnd_c c;
int a = 10;
bool fini = false;
A(){
        mtx_lock(&m);
        a=20;
        fini=true;
        mtx_unlock(&m);
}
B(){
        mtx_lock(&m);
        while(!fini); // attente active (c'est mal !) (energie)
                // auto[unlock au wait | lock au reveil] sur m
                cnd_wait(&c, &m)
        printf("...")
}
```

9 TD1 - Dekker

Avec deux processeurs :

```
debut_SCC();
        load $i, %r0
        inc %r0
        store %r0, $i
fin_SCC();
```

Après l'execution des instructions, i=1|2.
Ce qu'on veut avoir c'est obtenir i=2 et pas i=1

9.1 Zero

En utilisant cette fonction :

```
bool occ=false; //occupe
debut_SCC(){
        alpha:
                if (occ)
                        goto alpha;
                occ = true;
}

finSCC(){
        occ = false;;
}
```

On n'obtient pas d'exclusion mutuelle.

9.2 One

```
bool occ = false;
\\ecrire avant le test

debut_SCC(){
        int old = occ; // local var
        occ = true;
        alpha : if(old)
                                old = occ;
                        goto alpha;
}

finSCC(){
        occ = false;
}
```

Pas d'exclusion mutuelle.

9.3 Two

```
int tour = 0;
p; // num de processeur {0;1}
q = 1 - p; // num de l'autre processeur

debut_SCC(){
        alpha:
                if (tour != p){
                        goto alpha;
                }
}

finSCC(){
        tour = 1 - p;
}
```

Exclusion mutuelle OK, par contre, le bout de code cause un problème de famine.
L'autre processeur ne pourra jamais executer la section critique.

9.4 Three

```
bool access[2] = {false};
//Accede a acces[] avec p/q <-annoncer ma demande avant la boucle

debut_SCC(){
        acces[p] = true;
        alpha:
                if (acces[q])
                        goto alpha;
}

finSCC(){
        acces[p] = false;
}
```

Exclusion mutuelle OK.
Si les deux processeurs executent le code en même temps, ils rentrent dans une boucle infinie.
Probleme de **dead_lock**, les deux processeurs ne font plus rien.

9.5 Four

```
bool access[2] = {false, false};
// Three et renoncer a sa demande
// On recommence tout

debut_SCC(){
        alpha:
                acces[p] = true;
                if (acces[q])
                        acces[p] = false;
                        // en reseau recursive doubling
                        sleep(random);
                        goto alpha;
}

finSCC(){
        acces[p] = false;
}
```

Probleme de **dead_lock** si synchrone Il faut qu'ils arrivent en même temps et qu'ils restent synchrones pour que ça marche.

9.6 Five

```
bool access[2] = {false, false};
int tour;
// Comme la Four
// Celui dont ce n'est pas le tour renonce

debut_SCC(){
        alpha:
                acces[p] = true;
                if (p != tour && acces [q])
                        acces[p] = false;
                        sleep(random);
                        goto alpha;
}

finSCC(){
        acces[p] = false;
        tour = q;
}
```

Pas d'exclusion mutuelle

9.7 Six

```
bool access[2] = {false, false};
int tour;
// Comme la Four
// Celui dont ce n'est pas le tour renonce

debut_SCC(){
        acces[p] = true;

        alpha:
                if(acces[q]){
                        if(tour==p){
                                beta:
                                        if(access[q])
                                                goto beta;
                        }
                        else {
                                acces[p] = false
                                sigma: if(tour==q):
                                        goto sigma;
                                goto alpha;
                        }
                }

        }

finSCC(){
        acces[p] = false;
        tour = q;
}
```

Avant de commencer, je verifie que je suis dans le bon etat.
Meme si c'est mon tour, je vais attendre que l'autre renonce à acces.
L'autre cas c'est si je suis en conflit, je renonce à mon tour.
Une fois qu'il a fini, le tour devriendra le miens

9.8 Seven - La solution correcte de Peterson

La solution correcte :

```
bool access[2] = {false, false};
int dernier;

debut_SCC(){
        acces[p] = true;
        dernier = p;
        alpha:
                if (acces[q]){
                        if(dernier ==p)
                                goto alpha;
                }

finSCC(){
        acces[p] = false;
}
```