Systèmes d'exploitation

William SCHMITT & Othmane AJDOR

2018-2019

1 Virtualisation

1.1 Niveaux de virtualisation

1.1.1 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.

1.1.2 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).

1.1.3 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)		

2 Processus sous UNIX

2.1 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)

2.2 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.

2.3 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");
```

2.4 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

2.5 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

3 Threads

3.1 Création d'un thread

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

3.2 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("...")
}
```