

Tache et Process

Sur posix une tache:

- Process
- Thread

Un process c'est ce qu'on obtient quand on lance un programme. Lorsqu'un process a plusieurs threads alors tous les threads vont partager le meme espace d'adressage.

Une tache est une entite schedulable avec un ou plusieurs threads.

L'etat d'une tache:

- MetaData:
 - PID
 - TID (Thread ID)
 - EUID
 - GID
 - EGID
 - State (C'est en train de tourner ou pas)
- Context (Registers)
- Memoire (AS)
- File Descriptor Table
- Signal Handlers
- Directories
 - cwd (current working directory) → `chdir()`
 - root → `chroot()`
- rlimit (runtime limit)

Binaire SetUID

Comment creer une nouvelle tache

On peut utiliser `fork()`. On peut aussi utiliser `clone()` c'est comme `fork` mais avec des parametres.

Historiquement les process avait un maximum de 32 file descriptor. Aujourd'hui ils peuvent en avoir ~65 000

Redirection

Chevron

```
1 echo foo > toto
```

Pour faire ca on va fork:

```
1 fork()
2
3 // parent
4 wait();
5
6 // child
7 close(stdout);
8 open(toto);
9 execvp("echo", ...);
```

Execve

```
1 int execve(char *filename, char **argv, char **envp)
```

`envp` c'est l'ensemble des variables d'environnement. Ex: `["VAR=toto", ...]`

`environ` est une variable qui possède l'ensemble des variables d'environnement. `getenv()` permet d'obtenir une variable. `setenv()` permet de la set.

Quand on `execve` on va creer une stack on va ensuite empiler un tas de choses:

```
_____  
env  
args  
AUXV  
envp  
argv  
argc  
_____
```

Execvp

C'est un syscall qui va appeler `execve`. Le **p** est pour **path**.

```
1 execvp("ls", ["ls", "toto", NULL]);
```

Ce syscall est bien plus pratique. Si on veut modifier les variables d'environnement il suffit d'utiliser `setenv` dans le fork.

File Descriptor

Si on a une app en reseau, on est le serveur et on veut écouter des clients.

```
1 fd = socket();
2 bind(fd, addr);
3 listen(fd);
4 while (client = accept(fd))
5 {
6     fork();
7     read / write (client);
8     close(client);
9 }
```

Le probleme c'est que la lecture / écriture est bloquante du coup on peut écouter uniquement un client a la fois.

On peut utiliser `poll(struct pollfd*)`. Il prend plusieurs filedescriptor et nous permet de savoir quand un d'entre eux est pret. Il fait la meme chose que `select` mais de maniere plus sympa.

Ca reste chiant car on est lineaire en nombre de FD.

epoll / kqueue

`epoll` permet de creer une file d'event. Elle est mieux gerer qu'un simple tableau. `kqueue` est l'equivalent sur *FreeBSD*.

Childs

Lorsqu'un fils meurt le process parent:

- Etait en train de wait et donc reprend la main avec la valeur de retour.
- Se prend un SIGCHLD (qui est de base ignore).

Lorsqu'on lance un process enfant on doit donc `wait`. On peut ajouter un handler pour SIGCHLD.

```
1 void sighildhandler(int)
2 {
3     for(;;)
4     {
5         rc = wait(0, WNOHANG);
6         if (rc < 0 && errno == NOCHILD)
7             return;
8     }
9 }
```

Ca ne marche pas complètement il faut sauver et restaurer `errno` car sa valeur peut changer entre le `wait` et la reception asynchrone du signal.

Scheduler

On va se mettre dans un cas simple. Un CPU un coeur. On a cinq etats de base:

- Running
- Ready
- Dead (terminated)
- New
- Waiting (Blocked)

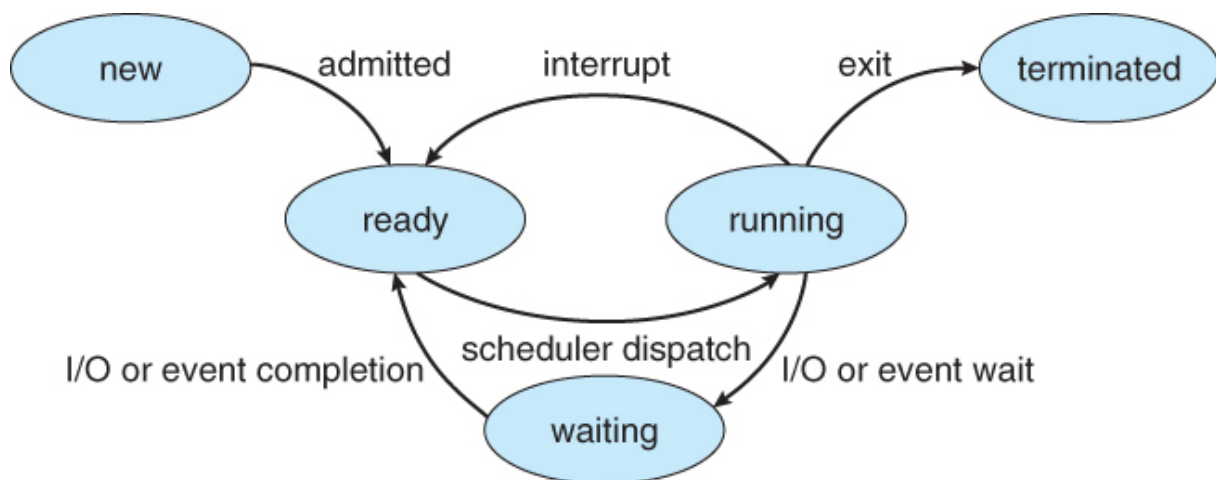


Figure 1: Schema des etats du CPU

On veut un algo pour gerer ca. (FCFS et SJF ne sont pas valide)

On va commencer par implementer un algo debile **FIFO** (On ignore l'**interrupt**). Ca ne peut pas marcher car ce n'est pas interactif.

On va faire un algo avec de la preemption **Round-Robin**. On va avoir un *quantum* de temps.

Il y a deux type de programmes ceux qui vont etre interactif, par exemple un shell qui va globalement passer son temps a dormir et attendre. Et les programmes qui vont faire utiliser du CPU, par exemple un compilateur.

Le *waiting time* c'est le temps passer en *ready* et pas le temps passe en *waiting*