Unix et Programmation Web

Cours 2

kn@lri.fr
http://www.lri.fr/~kn

Plan

Définitions

Programme : séquences d'instructions effectuant une tâche sur un ordinateur

Exécutable : fichier binaire contenant des instructions machines interprétables par le microprocesseur

***Thread* :** plus petite unité de traitement (≡ séquence d'instructions) pouvant être ordonnancée par l'OS

Processus : instance d'un programme (≡ « un programme en cours d'exécution »). Un processus est constitué de un ou plusieurs *threads*.

Exemple: programme

Dans un fichier « `counter.c` » (attention c'est du pseudo C)

```
int count = 0;
int exit = 0;
void display() {
   while (exit == 0) {
     sleep (3);
     printf("%i\n", count);
   }
}
void listen() {
   while (exit == 0) {
     wait_connect(80);
     count++;
 }
}
```

```
void main () {
   run_function(display);
   run_function(listen);
   while (getc () != '\n') { };
   exit = 1;
   return;
}
```

Exemple: programme

Compilation

```
gcc -o counter.exe counter.c
```

Le fichier « `counter.exe` » est un exécutable (fichier binaire contenant du code machine)

```
./counter.exe      ← il faut la permission +x sur le fichier
```

Le contenu de l'exécutable est copié en mémoire et le processeur commence à exécuter la première instruction du programme.

Exemple: *threads*

1. **main**
2. **attente d'un évènement clavier →←changement de *thread***
3. **listen**
4. **attente de connexion →←changement de *thread***
5. **display (affiche 0 à l'écran)**
6. **attente pendant 3s →** (les 3 *threads* attendent un évènement externe)
   nouvelle connexion sur le port 80 **← réveil du *thread***
   `listen`
7. **listen (incrémente** `count`)
   attente de connexion →
   … fin des 3s
   ← réveil du *thread* `display`
8. **display (affiche 1 à l'écran)**

Les *threads* partagent leur mémoire (variables communes)

Exemple: processus

(différence: les processus ne partagent pas leur espace mémoire)

1. Exécution de
   `counter.exe`
   pendant 50μs
2. Exécution de
   `firefox.exe`
   pendant 50μs
3. Exécution du processus qui dessine le bureau pendant 50 μs

   ...

C'est le gestionnaire de processus qui décide quel programme a la main et pour combien de temps (priorité aux tâches critiques par exemple)
Le système d'exploitation stocke pour chaque processus un ensemble d'informations, le PCB (*Process Control Block*).

Proccess Control Block

Le PCB contient:

- l'identificateur du processus (pid)
- l'l'état du processus (en attente, en exécution, bloqué, ...)
- le compteur d'instructions (*i.e.* où on en est dans le programme)
- le contexte courant(état des registres, ...)
- position dans la file d'attente de priorité globale
- informations mémoire (zones allouées, zones accessibles, zones partagées)
- listes des fichiers ouverts (en lecture, en écriture), liste des connexions ouvertes, ...

...

Opérations sur les processus

- création et destruction de processus
- suspension et reprise
- duplication (*fork*)
- modification de la priorité
- modification des permissions

États d'un processus

Un processus change d'état au cours de son exécution

Nouveau :	le processus est en cours de création
Exécution :	le processus s'exécute
En attente :	le processus attend un évènement particulier (saisie au clavier, écriture sur le disque, ...)
Prêt :	le processus est prêt à reprendre son exécution et attend que l'OS lui rende la main
terminé :	le processus a fini son exécution

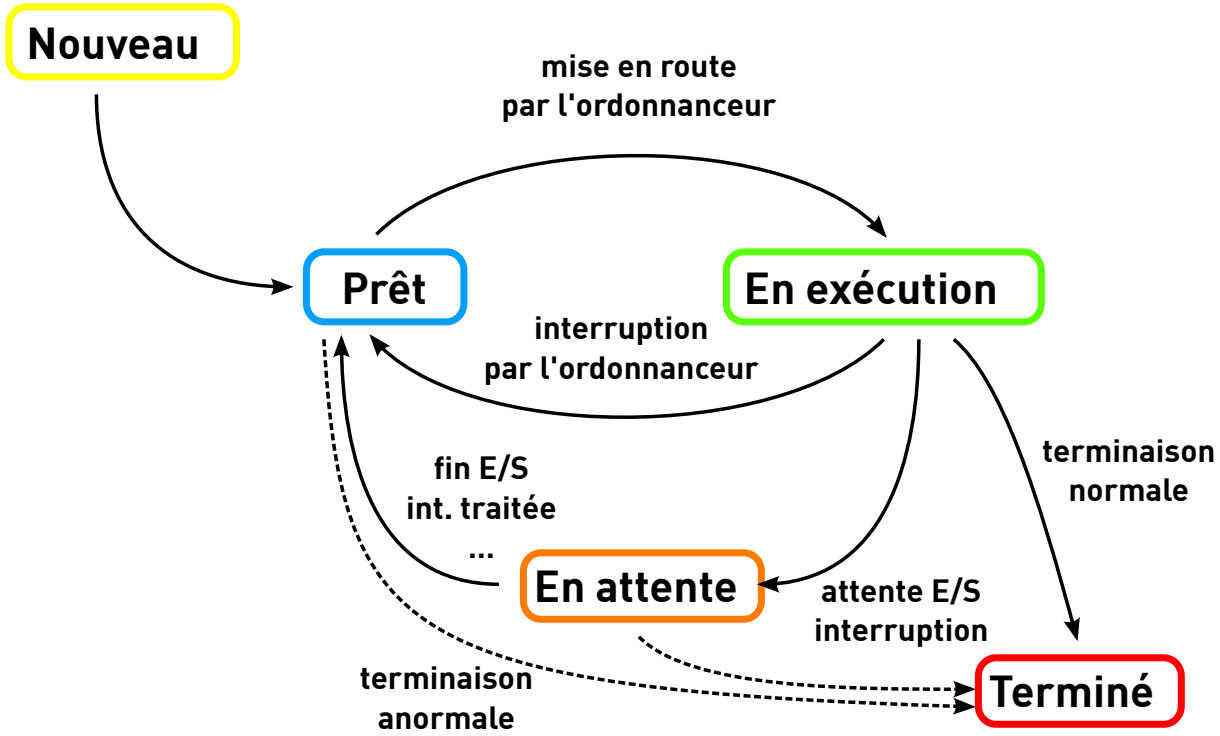
États d'un processus

L'OS détermine et modifie l'état d'un processus:

- En fonction d'évènements internes au processus:
  - lecture d'un fichier (si le contenu n'est pas disponible, le processus passe de « prêt » à « en attente »)
  - le processus attends volontairement pendant x secondes

  ...
- En fonction d'évènements externes au processus:
  - un fichier devient disponible
  - un *timer* arrive à 0
  - le matériel déclenche une interruption

États d'un processus



La commande
ps

Permet d'avoir des informations sur les processus en cours d'exécution (voir « `man ps` » pour les options):

```
$ ps -o user,pid,state,cmd x
          USER     PID   S CMD
          …
          kim      27030 Z [chrome] <defunct>
          kim      27072 S /opt/google/chrome/chrome --type=renderer
          kim      29146 S bash
          kim      29834 S evince
          kim      29858 S emacs cours.xhtml
          kim      29869 R ps -o user,pid,state,cmd x
```

États des processus (sous Linux)

- **R :** *Running* (en cours d'exécution)
- **S :** *Interruptible sleep* (en attente, interruptible)
- **D :** *Uninterruptible sleep* (en attente, non-interruptible)
- **T :** *Stopped* (interrompu)
- **Z :** *Zombie* (terminé mais toujours listé par le système)

Signaux

L'OS peut envoyer des signaux à un processus. Sur réception d'un signal, un processus peut interrompre son comportement normal et exécuter son gestionnaire de signal. Quelques signaux:

Nom	Code	Description
HUP	1	demande au processus de s'interrompre
INT	2	demande au processus de se terminer
ABRT	2	interrompt le processus et produit un *dump*
KILL	9	interrompt le processus immédiatement
SEGV	11	signale au processus une erreur mémoire
STOP	24	suspend l'exécution du processus
CONT	28	reprend l'exécution d'un processus suspendu

Processus et terminal

Un processus est lié au terminal dans lequel il est lancé. Si on exécute un programme dans un terminal et que le processus ne rend pas la main, le terminal est bloqué

```
$ gedit
```

On peut envoyer au processus le signal STOP en tapant
`ctrl-Z`
dans le terminal:

```
$ gedit
^Z
[1]+  Stopped                 gedit
```

Le processus est suspendu, la fenêtre est gelée (ne répond plus).

Processus et terminal

On peut reprendre l'exécution du programme de deux manières:

```
$ fg
```

Reprend l'exécution du processus et le remet en avant plan (terminal bloqué)

```
$ bg
```

Reprend l'exécution du processus et le remet en arrière plan (terminal libre)

On peut lancer un programme directement en arrière plan en faisant:

```
$ gedit &
```

On peut envoyer un signal à un processus avec la commande « `kill [-signal] pid` **»**

```
$ kill -9 2345
```

Processus et entrées/sorties

Le terminal et le processus sont liés par trois fichiers spéciaux:

1. L'entrée standard (`stdin`), reliée au clavier
2. La sortie standard (`stdout`), reliée à l'affichage
3. La sortie d'erreur (`stderr`), reliée à l'affichage

Dans le *shell*, on peut utiliser les opérateurs `<`, `>` et `2>` pour récupérer le contenu de `stdin`, `stdout` et `stderr`:

```
$ sort < toto.txt
$ ls -l  > liste_fichiers.txt
$ ls -l * 2> erreurs.txt
```

Processus et entrées/sorties

Dans le *shell*, l'opérateur | permet d'enchaîner la sortie d'un programme avec l'entrée d'un autre:

```
$ ls -l *.txt | sort -n -r -k 5 | head -n 1
```

1. affiche la liste détaillée des fichiers textes
2. trie (et affiche) l'entrée standard par ordre numérique décroissant selon le 5ème champ
3. affiche la première ligne de l'entrée standard

```
-rw-rw-r--  1 kim kim 1048576 Sep 24 09:20 large.txt
```

Plan

Script shell

Mentalité Unix beaucoup de petits programmes spécifiques, que l'on combine au moyen de scripts pour réaliser des actions complexes. Exemple de fichier script:

```
#!/bin/bash

for i in  img_*.jpg
do
  base=$(echo "$i" | cut -f 2- -d '_')
  nouveau=photo_"$base"
  if test -f "$nouveau"
  then
     echo "Attention, le fichier $nouveau existe déjà"
     continue
  else
     echo "Renommage de $i en $nouveau"
     mv "$i" "$nouveau"
  fi
done
```

Rendre un script exécutable

Si un fichier texte (quel que soit son extension), commence par les caractères `#!/chemin/vers/un/programme`**, on peut rendre ce fichier exécutable (**`chmod +x`**). Si on l'exécute, le contenu du fichier est passé comme argument à** `programme` **(qui est généralement un interpréteur)**

`#!/bin/bash` **signifie que le corps du fichier est passé au programme** `bash` **qui est l'interprète de commande (le *shell*).**

Que mettre dans un script

- **des commandes (comme si on les entrait dans le terminal)**
- **des structures de contrôle (boucles `for`, `if then else`)**
- **des définitions de variables**

Définitions de variables

On peut définir des variables au moyen de la notation `VARIABLE=`*`contenu`*
et on peut utiliser la variable avec la notation `$VARIABLE`

- **Attention, pas d'espace autour du** `=`
- **nom de variable en majuscule ou minuscule**
- **contenu est une chaine de caractères. Si elle contient des espaces, utiliser** `"..."`

exemple de définition :

```
i=123
j="Ma super chaine"
TOTO=titi
echo $TOTO
```

exemple d'utilisation: `echo $j $i $TOTO`
affiche « `Ma super chaine 123 titi`

Boucles `for`

Les boucles `for` ont la syntaxe:

```
for VARIABLE in elem_1 ... elem_n
do
    ....
done
```

chaque $elem_i$ est expansé (comme une ligne de commande) avant l'évaluation de la boucle:

```
for i in *.txt
do
   echo $i est un fichier texte
done
```

On peut quitter une boucle prématurément en utilisant `break` et passer directement au tour suivant avec `continue`

Conditionnelle

La syntaxe est :

```
if commande
then
      ...
else
      ...
fi
```

commande **est évaluée. Si elle se termine avec succès, la branche** `then` **est prise. Si elle se termine avec un code d'erreur, la branche** `else` **est prise. On peut utiliser la commande** `test` **qui permet de tester plusieurs conditions (existance d'un fichier, égalités de deux nombres, ...) et se termine par un succès si le teste est vrai et par un code d'erreur dans le cas contraire**

Conditionnelle (exemple)

On regarde tour à tour si fichier1.txt, fichier2.txt, ... existent :

```
for i in 1 2 3 4 5 6
do
  if test -f "fichier$i".txt
 then
    echo le fichier "fichier$i".txt existe
 fi
done
```

Sous-commandes et chaines

Il est pratique de pouvoir mettre l'affichage d'une commande dans une variable. On utilise `$( commande ... )`:

```
MESFICHIER=$(ls *.txt)
for i in $MESFICHIER
do
     echo Fichier: $i
done
```

Attention à la présence de guillemets autour des variables. S'il y a f1.txt et f2.txt dans le répertoire courant:

```
MESFICHIER=$(ls *.txt)
for i in $MESFICHIER
do
     echo Fichier: $i
done
affiche:
  Fichier: f1.txt
  Fichier: f2.txt
```

```
MESFICHIER=$(ls *.txt)
for i in "$MESFICHIER"
do
     echo Fichier: $i
done
affiche:
  Fichier: f1.txt f2.txt
```

Commandes utiles

- `seq` *`m n`* **: affiche la liste de tous les nombres entre *m n***
- `echo ...` **affiche ses arguments sur la sortie standard**
- `printf "chaine" ...` **affiche ses arguments au moyen d'une chaine de format (comme le printf de C)**
- `date` **: affiche la date courante**
- `cut` **: découpe une chaine selon des caractères de séparations ou des positions**

Plan

Processus de type *daemon*

Un *daemon* (prononcé démon) est un processus qui non-interactif qui tourne en tâche de fond (pas d'entrée/sortie sur le terminal, pas d'interface graphique, ...). On communique avec ce processus via des signaux ou en lisant ou écrivant dans des fichiers ou connexions réseau. Le plus souvent, leur but est de fournir un service

Exemple de scénario: « *Les utilisateurs doivent interagir avec le matériel. L'accès au matériel demande des droits administrateur.* »

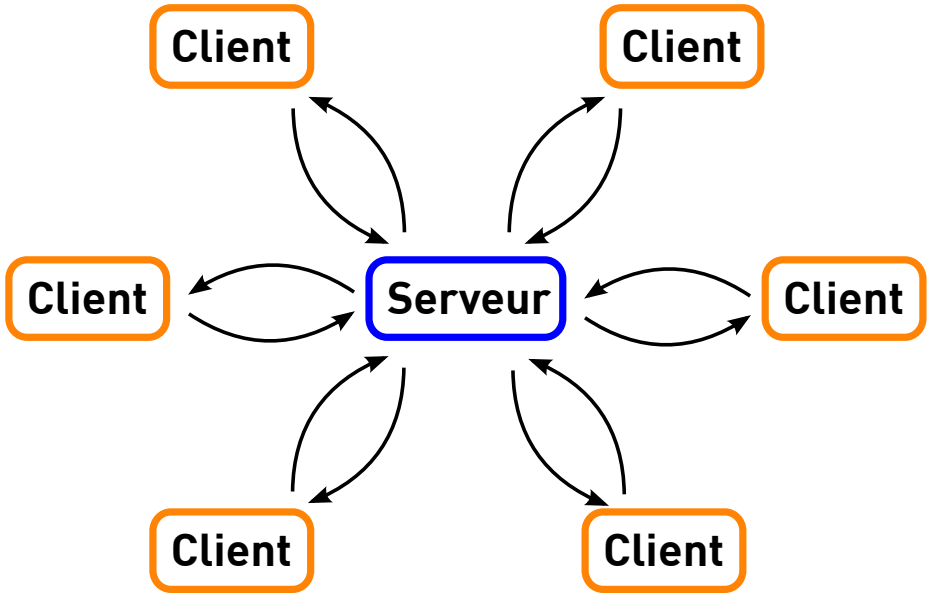
Solution 1 : tout le monde est administrateur (DOS, Win XP, ...)

Solution 2 : on crée un programme particulier qui a les privilèges suffisants pour la tâche en question. Les utilisateurs communiquent avec ce programme

Quelques *daemons* sous Linux

Nom	Description
sshd	*shell* distant sécurisé
crond	exécution périodique de programmes
cupsd	serveur d'impressions
pulseaudio	serveur de son (mixe les sons des différentes applications)
udevd	détection de matériel *hotplug*
nfsd	serveur de fichier réseau
smtpd	livraison des e-mail
httpd	serveur de pages Web

Architecture client-serveur



Des processus **clients** communiquent avec le **serveur** à travers le réseau. Les clients sont indépendant et ne communiquent pas entre eux. **Attention** plusieurs clients peuvent se trouver su la même machine physique!

Architecture client-serveur

- Le serveur attend des connexions entrantes
- Les clients peuvent se connecter à tout moment
- L'application client est généralement légère, envoie une requête au serveur et attend un résultat
- Le serveur est une application plus lourde qui:
  - effectue des calculs trop coûteux pour le client
  - gère l'accès à une ressource distante partagée
  - ...

Exemples: serveur de bases de données, serveur mail, serveur Web, terminal de carte bancaire, ...