Introduction aux systèmes et réseaux informatiques

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Introduction aux systèmes et réseaux

- **Objectifs du cours**
  - Présenter les principaux concepts des systèmes d'exploitation et des réseaux, plutôt du point de vue de l'utilisateur (le point de vue du concepteur étant présenté en M1). Illustrer concrètement ces concepts au moyen d'exemples pratiques (mini-projets)
  - Orientation pratique : 13h30 de cours, 7h30 de TD, 9h de TP + 3 "apnées"
- **Plan du cours**
  - Cours 1 à 6 : Introduction aux systèmes d'exploitation
    - Fonctions, organisation
    - Processus, mémoire et fichiers
    - Vie d'un programme
  - Cours 7 à 9 : Introduction aux réseaux
    - Organisation d'un réseau, protocoles et interfaces
    - Utilisation du niveau de transport
- **Plan des TD et TP**
  - 1-ère série : Introduction aux systèmes (5 TD et 4 TP de 1h30)
    - Processus et fichiers en Unix
    - Programmation d'un mini-*shell* (apnée)
  - 2-ème série : Introduction aux réseaux
    - Utilisation des *sockets* Unix (1 TP de 1h30 + apnée)
    - Programmation d'un serveur web (1 TP de 1h30 + apnée)

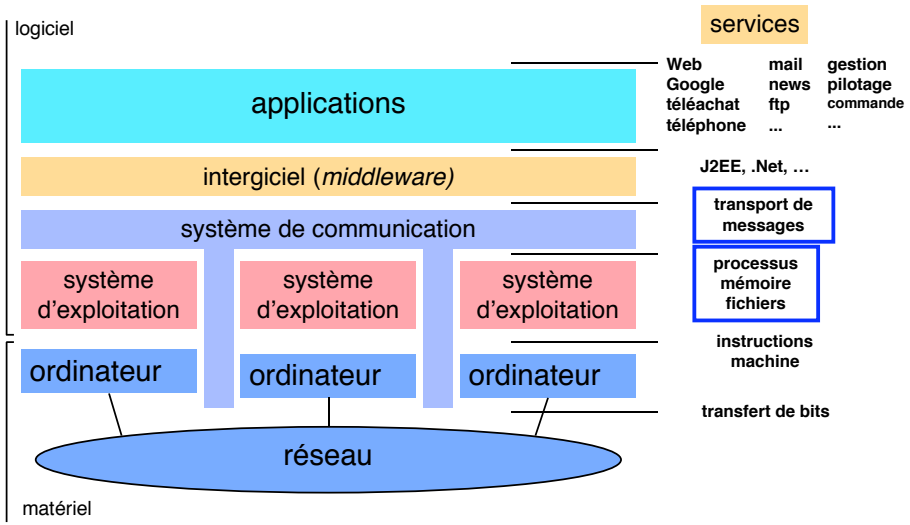
Planning du semestre (janvier-avril 2007)

Semaine	Cours	TD	TP	
1 (22 janvier)	C1 : Processus (intro.)	Processus		
2 (29 janvier)	C2 : Processus (suite)		Processus	
3 (5 février)		Signaux		
4 (12 février)	C3 : Synchronisation		Signaux	
19/2 vacances				
5 (26 février)	C4 : Interprétation, liaison	Mini-*shell*		
6 (5 mars)	C5 : Mémoire		Mini-*shell*	Apnée 1
7 (12 mars)	C6 : Fichiers, communication	Fichiers		
8 (19 mars)	C7 : Réseaux, protocoles		Fichiers	
9 (26 mars)	C8 : Communication par *sockets*			
10 (2 avril)	C9 : Services de l'Internet		*sockets*	Apnée 2
9/4 vacances				
11 (16 avril)		Mémoire, liaison		
12 (23 avril)			Serveur Web	Apnée 3

Page web du cours : http://sardes.inrialpes.fr/~krakowia/Enseignement/L3/SR-L3.html

Références : R. E. Bryant, D. O'Hallaron : *Computer Systems, A Programmer's Perspective*, Prentice Hall, 2003 (ch. 8, 11)
J.-M. Rifflet, J.-B. Yunès : *Unix : Programmation et Communication*, Dunod, 2003

Les composants d'un système informatique



Interfaces

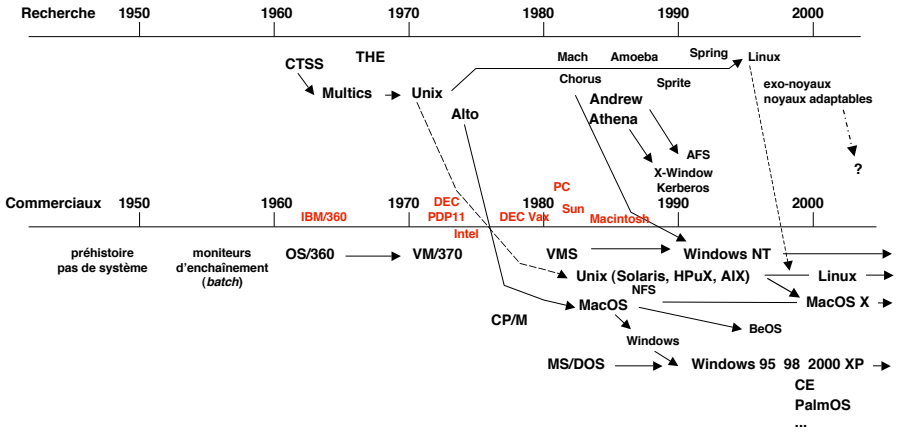
- **Un service est caractérisé par son interface**
  - L'interface est l'ensemble des fonctions accessibles aux utilisateurs du service
  - Chaque fonction est définie par
    - son format (la description de son mode d'utilisation) - ou encore sa syntaxe
    - sa spécification (la description de son effet) - ou encore sa sémantique
  - Ces descriptions doivent être
    - précises
    - complètes (y compris les cas d'erreur)
    - non ambiguës
- **Principe de base : séparation entre interface et réalisation**

  Les descriptions de l'interface d'un service doivent être totalement indépendantes du mode de réalisation du service. Les avantages sont les suivants :
  - Facilite la portabilité
    - transport d'une application utilisant le service sur une autre réalisation du service
    - passage d'un utilisateur sur une autre réalisation du système
  - Permet de remplacer une réalisation du service par une autre, à condition qu'elle réalise la même interface

Rôle d'un système d'exploitation

- **Place**
  - Le système d'exploitation est l'intermédiaire entre un ordinateur (ou en général un appareil muni d'un processeur) et les applications qui utilisent cet ordinateur ou cet appareil. Son rôle peut être vu sous deux aspects complémentaires.
- **Adaptation d'interface**
  - Le système fournit à ses utilisateurs une interface plus commode à utiliser que celle du matériel :
    - il dissimule les détails de mise en œuvre (plus haut niveau d'abstraction)
    - il dissimule les limitations physiques (taille de mémoire, nombre de processeurs) et le partage des ressources entre plusieurs utilisateurs
  - On dit parfois que le système réalise une "machine virtuelle"
- **Gestion de ressources**
  - Le système gère les ressources matérielles et logicielles : mémoire, processeurs, programmes, données, communications. Cette gestion comprend l'allocation, le partage et la protection.
- **Où trouve-t-on des systèmes d'exploitation ?**
  - sur les ordinateurs : Unix (Linux, Solaris, etc.), MacOS-X, Windows XP, Windows Server, etc.
  - sur des appareils divers : téléphone portable, assistant personnel, carte à puce.

Historique sommaire des systèmes d'exploitation



Fonctions d'un système d'exploitation

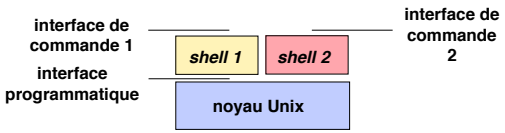
	organe physique	entité virtuelle
Gestion d'activités		
◆ déroulement de l'exécution		
◆ événements	processeur	processus
Gestion d'information		
◆ accès dynamique (exécution)		
◆ conservation permanente de l'information	mémoire principale	mémoire virtuelle
◆ partage de l'information	disque	fichier
Gestion des communications		
◆ interface avec l'utilisateur	écran clavier, souris	fenêtre
◆ impression	imprimante	
◆ réseau	réseau	flot d'entrée-sortie
◆ organes spécialisés	capteurs, scanner, DVD, ...	
Protection		
◆ de l'information	tous	domaine
◆ des ressources		

Interfaces d'un système d'exploitation

Un système d'exploitation présente en général deux interfaces

- Interface "programmatique", ou API (*Application Programming Interface*)
  - utilisable à partir des programmes s'exécutant sous le système
  - composée d'un ensemble d'appels systèmes (appels de procédures, avec paramètres)
- Interface de l'utilisateur, ou interface de commande
  - utilisable par un usager humain, sous forme textuelle ou graphique
  - composée d'un ensemble de commandes
    - textuelles (exemple en Unix : `rm *.o`)
    - graphiques (exemple : déplacer l'icone d'un fichier vers la corbeille)

Exemple (Unix)



Exemple d'usage des interfaces (Unix)

- Interface programmatique (en C)

  le morceau de programme ci-contre utilise les fonctions `read()` et `write()` pour recopier un fichier dans un autre

- Interface de commande

  `cp fich1 fich2`

  recopie `fich1` dans `fich2`

- Documentation

  Documentation en ligne par `man`

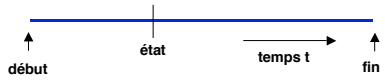
```
...
while (bytesread = read(from_fd, buf, BLKSIZE
)) {
    if ((bytesread == -1) && (errno != EINTR))
        break;

    else if (bytesread > 0) {
        bp = buf;
        while(byteswritten = write(to_fd, bp,
bytesread )) {
            if ((byteswritten == -1) && (errno !=
EINTR))
                break;
            else if (byteswritten == bytesread)
                break;
            else if (byteswritten > 0) {
                bp += byteswritten;
                bytesread -= byteswritten;
            }
        }
        if (byteswritten == -1)
            break;
    }
}
...
```

`man 1 <nom de la commande>` : documentation des commandes (option par défaut)
`man 2 <nom de la commande>` : documentation des appels système
`man 3 <nom de la commande>` : documentation de la bibliothèque C

Processus

- Définition
  - un processus (séquentiel) est l'entité dynamique représentant l'exécution d'un programme sur un processeur
    - différence entre processus et programme : le programme est une description statique ; le processus est une activité dynamique (il a un début, un déroulement et une fin, il a un état qui évolue au cours du temps)

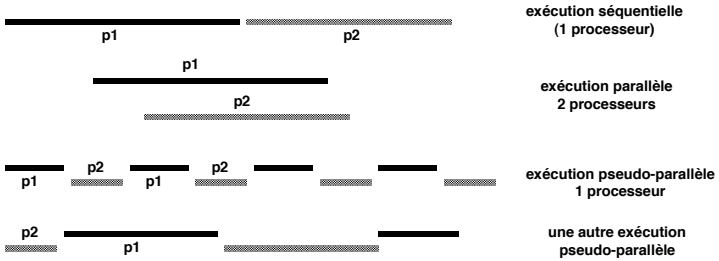


- Intérêt de la notion de processus
  - abstraction de la notion d'exécution séquentielle, qui la rend indépendante de la disponibilité effective d'un processeur physique
  - représentation des activités parallèles et de leurs interactions
- Exemple de processus
  - l'exécution d'un programme
  - la copie d'un fichier sur disque
  - la transmission d'une séquence de données sur un réseau

Parallélisme et pseudo-parallélisme

- Soit deux processus p1 et p2 (exécution de deux programmes séquentiels P1 et P2

- Mise en œuvre concrète de l'exécution de p1 et p2



Relations entre processus

- **Compétition**
  - Situation dans laquelle plusieurs processus doivent utiliser simultanément une ressource à accès exclusif (ressource ne pouvant être utilisée que par un seul processus à la fois)
  - Exemples
    - processeur (cas du pseudo-parallélisme)
    - imprimante
  - Une solution possible (mais non la seule) : faire attendre les processus demandeurs jusqu'à ce que l'occupant actuel ait fini (premier arrivé, premier servi)
- **Coopération**
  - Situation dans laquelle plusieurs processus collaborent à une tâche commune et doivent se synchroniser pour réaliser cette tâche.
  - Exemples
    - p1 produit un fichier, p2 imprime le fichier
    - p1 met à jour un fichier, p2 consulte le fichier
  - La synchronisation se ramène au cas suivant : un processus doit attendre qu'un autre processus ait franchi un certain point de son exécution

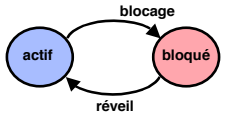
Faire attendre un processus

- **Dans les deux types de relations (compétition ou coopération), on doit faire attendre un processus. Comment réaliser cette attente ?**
- **Solution 1 : attente active**

```
p1
while (ressource occupée)
  { };
ressource occupée = true;
...
```

```
p2
ressource occupée = true;
utiliser ressource;
ressource occupée = false;
```

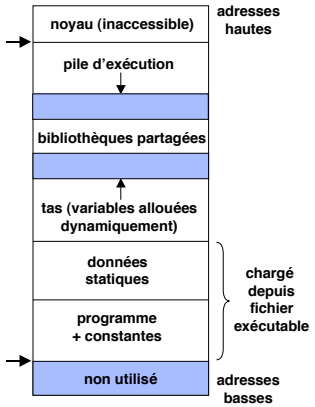
  - très peu économique si pseudo-parallélisme
  - difficulté d'une solution correcte (à voir plus tard)
- **Solution 2 : blocage du processus**
  - On définit un nouvel état pour les processus, l'état bloqué. L'exécution d'un processus bloqué est arrêtée, jusqu'à son réveil explicite par un autre processus ou par le système




```
...
sleep(5); /* se bloquer pendant 5 secondes) */
...
```


Processus dans Unix

- **Un processus réalise l'exécution d'un programme**
  - commande (du langage de commande)
  - programme d'application
- **Un processus comprend**
  - une mémoire qui lui est propre (mémoire virtuelle)
  - un contexte d'exécution (état instantané)
    - pile (en mémoire)
    - registres du processeur
- **Un processus et identifié par un numéro (`PID`)**
  - La commande `ps` donne la liste des processus en cours d'exécution (voir `man ps`)
  - La commande `top` montre l'activité du processeur (voir `man top`)
  - La primitive `getpid()` renvoie le numéro (`PID`) du processus qui l'exécute



mémoire d'un processus dans Unix

Environnement d'un processus

Dans Unix, un processus a accès à un certain nombre de variables qui constituent son **environnement**. Leur rôle est double

- faciliter la tâche de l'utilisateur en évitant d'avoir à redéfinir tout le contexte du processus (nom de l'utilisateur, de la machine, terminal par défaut, etc.)
- personnaliser différents éléments comme le chemin de recherche des fichiers, le répertoire de base (*home*), le *shell* utilisé, etc.

Certaines variables sont prédéfinies dans le système. L'utilisateur peut les modifier, et peut aussi créer ses propres variables d'environnement.

La commande `setenv` (sans paramètres) affiche l'environnement courant
La commande `setenv VAR <val>` attribue la valeur `<val>` à la variable `VAR`
La commande `echo $VAR` affiche la valeur courante de la variable `VAR`

Exemple :

`setenv DISPLAY argon.inrialpes.fr:0.0` définit le terminal utilisé

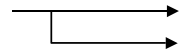
On peut aussi modifier les variables d'environnement par programme (voir TD)

Vie et mort des processus

- **Un processus a généralement un début et une fin**
  - **Début : création par un autre processus - par `fork()`**
    - il existe un processus "primitif" (PID = 1) créé à l'origine du système
  - **Fin**
    - auto-destruction (à la fin du programme) - par `exit()`
    - destruction par un autre processus - par `kill()`
    - certains processus ne se terminent pas ("démons", réalisant des fonctions du système)
- **Dans Unix**
  - **Dans le langage de commande**
    - un processus est créé pour l'exécution de chaque commande
    - on peut créér des processus pour exécuter des commandes en (pseudo)-parallèle :
      - `prog1 & prog2 &` /* crée deux processus pour exécuter `prog1` et `prog2` */
      - `prog1 & prog1 &` /* crée deux exécutions parallèles de `prog1` */
  - **Au niveau des appels système**
    - un processus est créé par une instruction `fork()` (voir plus loin)

Création des processus dans Unix (1)

- **L'appel système `pid_t fork()` permet de créer un processus**
- **Le processus créé (fils) est un clone (copie conforme) du processus créateur (père)**
- **Le père et le fils ne se distinguent que par le résultat rendu par `fork()`**
  - **pour le père : le numéro du fils (ou –1 si création impossible)**
  - **pour le fils : 0**



1 programme, 2 processus

```
if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    /* en général exec (exécution d'un nouveau programme) */
  }
   ...
}
```

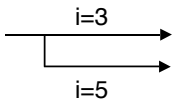

Création des processus dans Unix (2)

1 programme, 2 processus

donc **2 mémoires virtuelles, 2 jeux de données**

```
int i ;
if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    i = 3;
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    i = 5;
  }
printf("pour %d, i = %d\n", getpid(), i);
}
```

```
je suis le fils, mon PID est 10271
pour 10271, i = 5
je suis le père, mon PID est 10270
pour 10270, i = 3
```

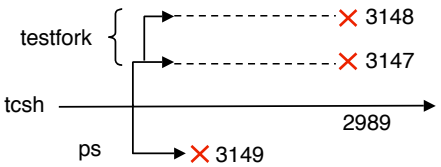


programme `testfork.c`

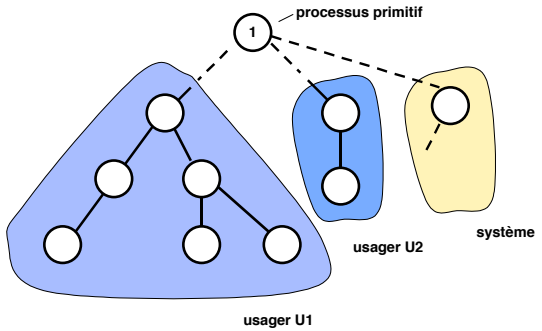
```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    sleep(10)  /* blocage pendant 10 secondes */
    exit(0);
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(10)  /* blocage pendant 10 secondes */
    exit(0);
  }
}
```

```
<unix> gcc -o testfork testfork.c
<unix> ./testfork & ps
je suis le fils, mon PID est 3148
je suis le père, mon PID est 3147
[2] 3147
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3147 pts/0    00:00:00 testfork
3148 pts/0    00:00:00 testfork
3149 pts/0    00:00:00 ps
<unix>
```

Exemple simple d'exécution



Hiérarchie de processus dans Unix



- **Fonctions utiles**
  - `getppid()` : obtenir le numéro du père
  - `getuid()` : obtenir le numéro d'usager (auquel appartient le processus)

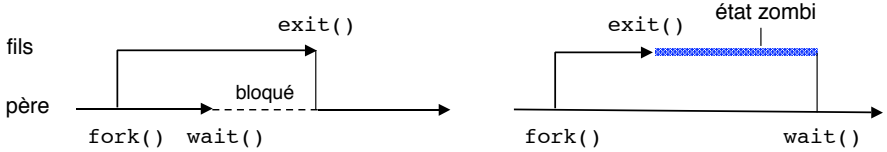
Quelques interactions entre processus dans Unix

- **Synchronisation entre un processus père et ses fils**
  - Le fils termine son exécution par `exit(statut)`, où `statut` est un code de fin (par convention : 0 si normal, sinon code indiquant une erreur)
  - Un processus père peut attendre la fin de l'exécution d'un fils par la primitive : `pid_t wait(int *ptrStatut)`. La variable (facultative) `ptrStatut` recueille le statut, `wait` renvoie le `PID` du fils.
  - On peut aussi utiliser `pid_t waitpid (pid_t pid, int *ptrStatut)` pour attendre la fin de l'exécution d'un fils spécifié `pid`
- **Envoyer un signal à un autre processus**
  - Sera vu plus tard en détail. Pour le moment, on peut utiliser `kill pid` au niveau du langage de commande, pour tuer un processus spécifié `pid`
- **Faire attendre un processus**
  - `sleep(n)` : se bloquer pendant `n` secondes
  - `pause` : se bloquer jusqu'à la réception d'un signal envoyé par un autre processus

Synchronisation entre père et fils

Quand un processus se termine, il délivre un code de retour (paramètre de la primitive `exit()`). Par exemple `exit(1)` renvoie le code de retour 1.

Un processus père peut attendre la fin d'un ou plusieurs fils en utilisant `wait()` ou `waitpid()`. Tant que son père n'a pas pris connaissance de sa terminaison par l'une de ces primitives, un processus terminé reste dans un état dit **zombi**. Un processus zombi ne peut plus s'exécuter, mais consomme encore des ressources (tables). Il faut éviter de conserver des processus dans cet état.



Les détails de `wait()` et `waitpid()` seront vus en TD

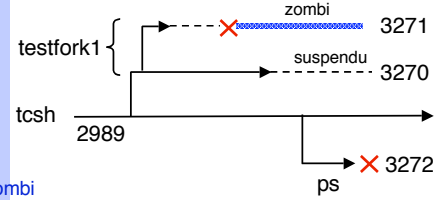
programme `testfork1.c`

```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    while (1) ; /* boucle sans fin sans attendre le fils */
  else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(0);
  }
}
```

```
<unix> gcc -o testfork1 testfork.c
<unix> ./testfork1
je suis le fils, mon PID est 3271
je suis le père, mon PID est 3270
fin du fils
==>frappe de <control-Z> (suspendre)
Suspended
<unix> ps
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3270 pts/0    00:00:03 testfork1
3271 pts/0    00:00:00 testfork1 <defunct>
3272 pts/0    00:00:00 ps
<unix>
```

zombi

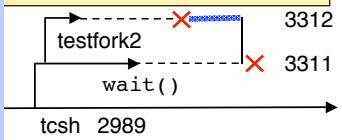
Exemple simple d'exécution



programme testfork2.c

```
#include <sys/types.h>
#include <sys.wait.h>
int main() {
  if (fork() != 0) {
    int statut; pid_t fils);
    printf("je suis le père %d, j'attends mon fils\n", getpid());
    fils = wait(&statut);
    if (WIFEXITED(statut)) {
       printf("%d : mon fils %d s'est terminé avec le code %d\n",
          getpid(), fils, WEXITSTATUS(statut); };
    exit(0);
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(1);
```

```
<unix> ./testfork2
je suis le fils, mon PID est 3312
je suis le père 3311, j'attends mon fils
fin du fils
3311: mon fils 3312 s'est terminé avec le code 1
<unix> ps
 PID TTY          TIME CMD
2989 pts/0   00:00:00 tcsh
3313 pts/0   00:00:00 ps
<unix>
```



Exemple simple d'exécution

Exécution d'un programme spécifié

La primitive `exec` sert à faire exécuter un nouveau programme par un processus. Elle est souvent utilisée immédiatement après la création d'un processus. Son effet est de "recouvrir" la mémoire virtuelle du processus par le nouveau programme et de lancer celui-ci en lui passant des paramètres spécifiés dans la commande.

Diverses variantes d'`exec` existent selon le mode de passage des paramètres (tableau, liste, passage de variables d'environnement). Exemple :

```
main() {
  if (fork() == 0) {
     execl("bin/ls", "ls", "-a", 0);
  }
  else {
     wait(NULL);}
  exit(0)
}
```

le fils exécute :

`/bin/ls -a ..`

le père attend la fin du fils

Résumé de la séance 1

- **Services et interfaces**
- **La place et les fonctions du système d'exploitation**
  - Les deux rôles d'un système d'exploitation
    - fournir une interface commode (machine virtuelle)
    - gérer les ressources
  - Les deux interfaces d'un système d'exploitation
    - appels systèmes (pour les programmes
    - commandes (pour les utilisateurs)
- **La notion de processus**
  - parallélisme et pseudo-parallélisme
  - compétition et coopération ; faire attendre un processus : blocage
  - processus dans Unix
    - mémoire virtuelle, environnement
    - création

Bibliographie sommaire

J.-M. Rifflet, J.-B. Yunès, *Unix, programmation et communication*, Dunod, 2003
R. E. Bryant, D. O'Hallaron, *Computer Systems, A Programmer's Perspective*, Prentice Hall, 2003 (chap. 8)