

Introduction aux systèmes et aux réseaux informatiques

Noel De Palma
Université Joseph Fourier

Janvier 2016

Ce cours est basé sur les transparents de Sacha Krakowiak / R. Lachaize

Introduction aux systèmes et aux 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 : 16h30 de cours (11 séances), 15h de TD, 15h de TP + 2 ou 3 "apnées"

■ Plan du cours

- ◆ Introduction aux systèmes d'exploitation
 - ❖ Fonctions, organisation
 - ❖ Processus, mémoire et fichiers
 - ❖ Vie d'un programme
- ◆ Introduction aux réseaux
 - ❖ Organisation d'un réseau, protocoles et interfaces
 - ❖ Utilisation du niveau « transport »

■ Plan des TD et TP

- ◆ 1ère série : Introduction aux systèmes
 - ❖ Processus et fichiers Unix
 - ❖ Utilisation et programmation d'un *shell*
- ◆ 2ème série : Introduction aux réseaux
 - ❖ Utilisation des *sockets* Unix
 - ❖ Programmation d'une application de type « serveur »

Introduction aux systèmes et aux 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 : 16h30 de cours (11 séances), 15h de TD, 15h de TP + 2 ou 3 "apnées"

■ Plan du cours

- ◆ Introduction aux systèmes d'exploitation
 - ❖ Fonctions, organisation
 - ❖ Processus, mémoire et fichiers
 - ❖ Vie d'un programme
- ◆ Introduction aux réseaux
 - ❖ Organisation d'un réseau, protocoles et interfaces
 - ❖ Utilisation du niveau « transport »

■ Plan des TD et TP

- ◆ 1ère série : Introduction aux systèmes
 - ❖ Processus et fichiers Unix
 - ❖ Interactions entre processus
- ◆ 2ème série : Introduction aux réseaux
 - ❖ Utilisation des *sockets* Unix
 - ❖ Programmation d'une application de type « serveur »

Planning prévisionnel du semestre (janvier-avril 2016)

Semaine	Cours	TD	TP	
3 (18 janvier)	C1	TD1	TP1	
4 (25 janvier)	C2	TD2	TP2	
5 (1 février)	C3	TD3	TP3	
6 (8 février)	C4	TD4	TP4	Apnée 1
7 (15 février)	C5	TD5	TP5	Apnée 1 - fin
8 (22 février)	vacances	vacances	vacances	
9 (29 février)	C6	TD6	TP6	
10 (7 mars)	C7	TD7	TP7	
11 (14 mars)	C8	TD8	TP8	
12 (21 mars)	C9	TD9	TP9	Apnée 2
13 (28 mars)	C10	TD10	TP10	
14 (4 avril)	C11	soutenances	soutenances	
15 (11 avril)				Apnée 3 ?
16 (18 avril)	vacances	vacances	vacances	
17 (25 avril)				
18 (2 mai)				

Documents et références

■ Moodle de l'UFR

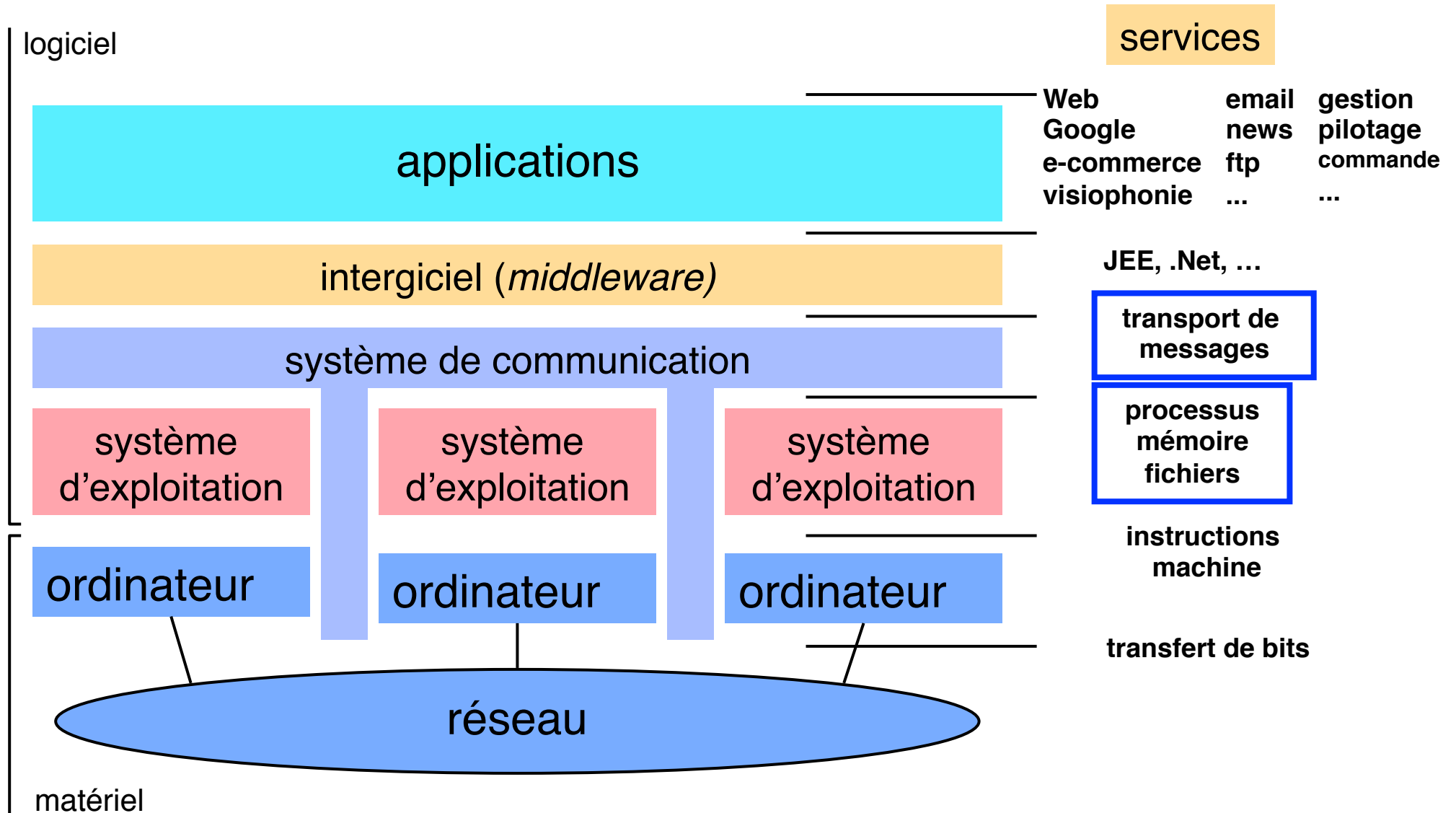
- ◆ Diapositives de cours, fichiers pour les TP, documentation techniques

■ Principaux ouvrages (disponibles à la Bibliothèque universitaire) :

- ◆ Référence principale : R. E. Bryant, D. O'Hallaron, *Computer Systems, A Programmer's Perspective*, Prentice Hall, 2003 (ch. 8, 11) (*)
- ◆ J. Kurose, K. Ross, *Computer Networking, a top-down approach*, Addison-Wesley, 2000 (les premiers chapitres, pour l'introduction aux réseaux) (*)
- ◆ Liste complète sur la page de l'UE

(*) : des éditions plus récentes existent et sont également valables pour le cours

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”

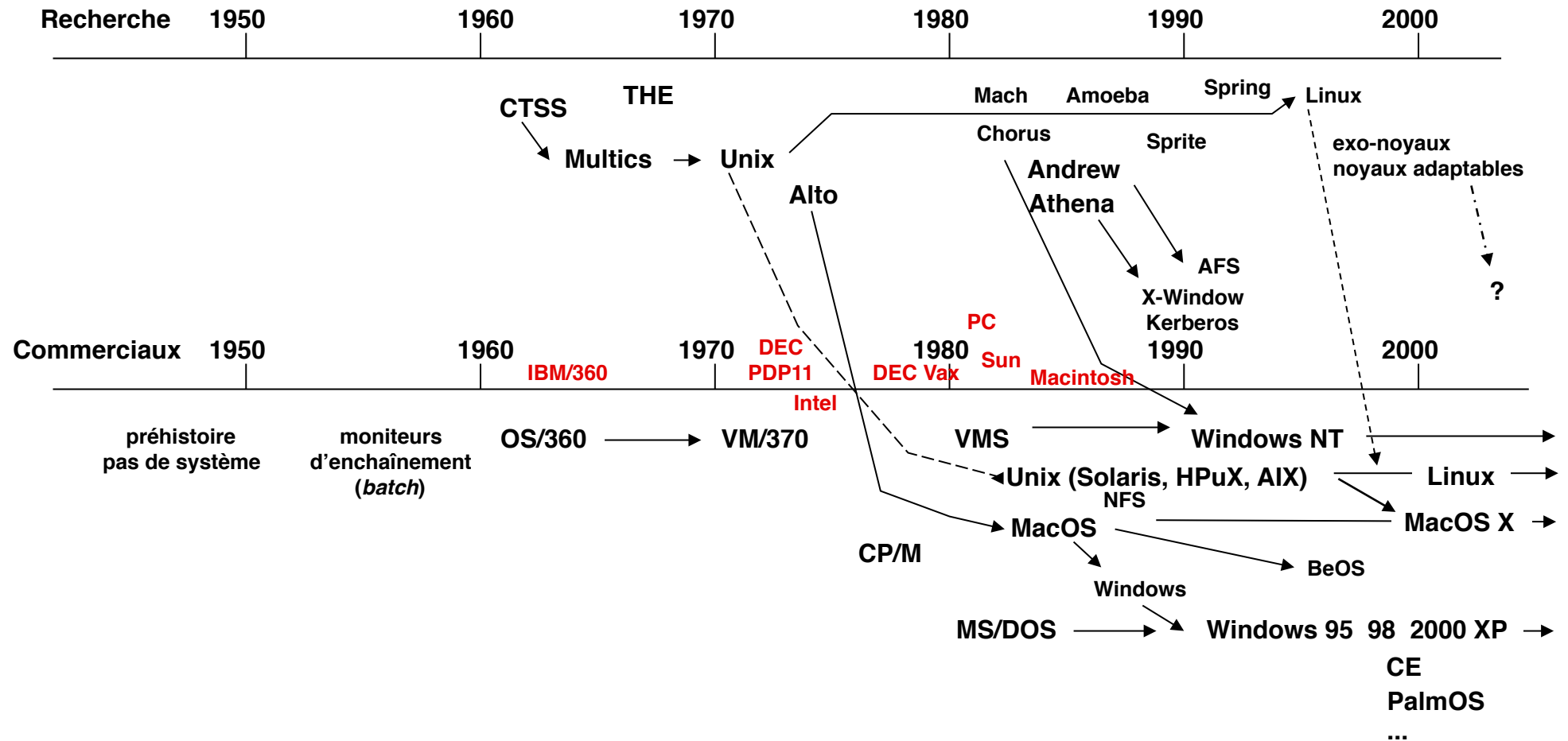
■ Gestion de ressources

- ◆ 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.), Mac OS X, Windows 7, 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

■ Gestion d'activités

- ◆ déroulement de l'exécution
- ◆ événements

organe physique

processeur

entité virtuelle

processus

■ Gestion d'information

- ◆ accès dynamique (exécution)
- ◆ conservation permanente de l'information
- ◆ partage de l'information

mémoire principale

mémoire virtuelle

disque

fichier

■ Gestion des communications

- ◆ interface avec l'utilisateur
- ◆ impression
- ◆ réseau
- ◆ organes spécialisés

écran
clavier, souris

fenêtre

imprimante
réseau

flot d'entrée-sortie

capteurs, scanner,
DVD, ...

■ Protection

- ◆ de l'information
- ◆ des ressources

tous

domaine

Interfaces d'un système d'exploitation

Un système d'exploitation présente en général **deux** interfaces

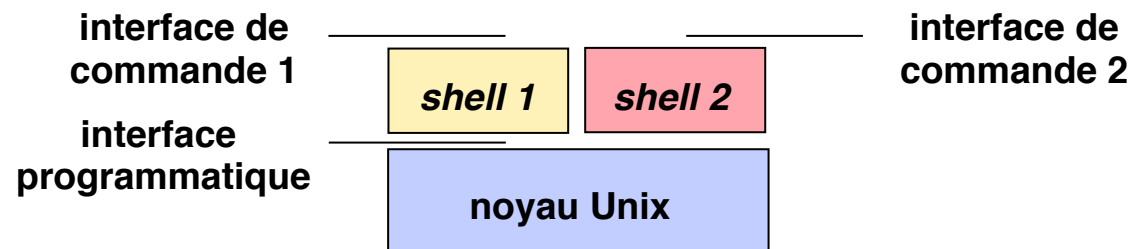
■ **Interface “programmatische”, ou API (*Application Programming Interface*)**

- ◆ utilisable à partir des programmes s'exécutant sur 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

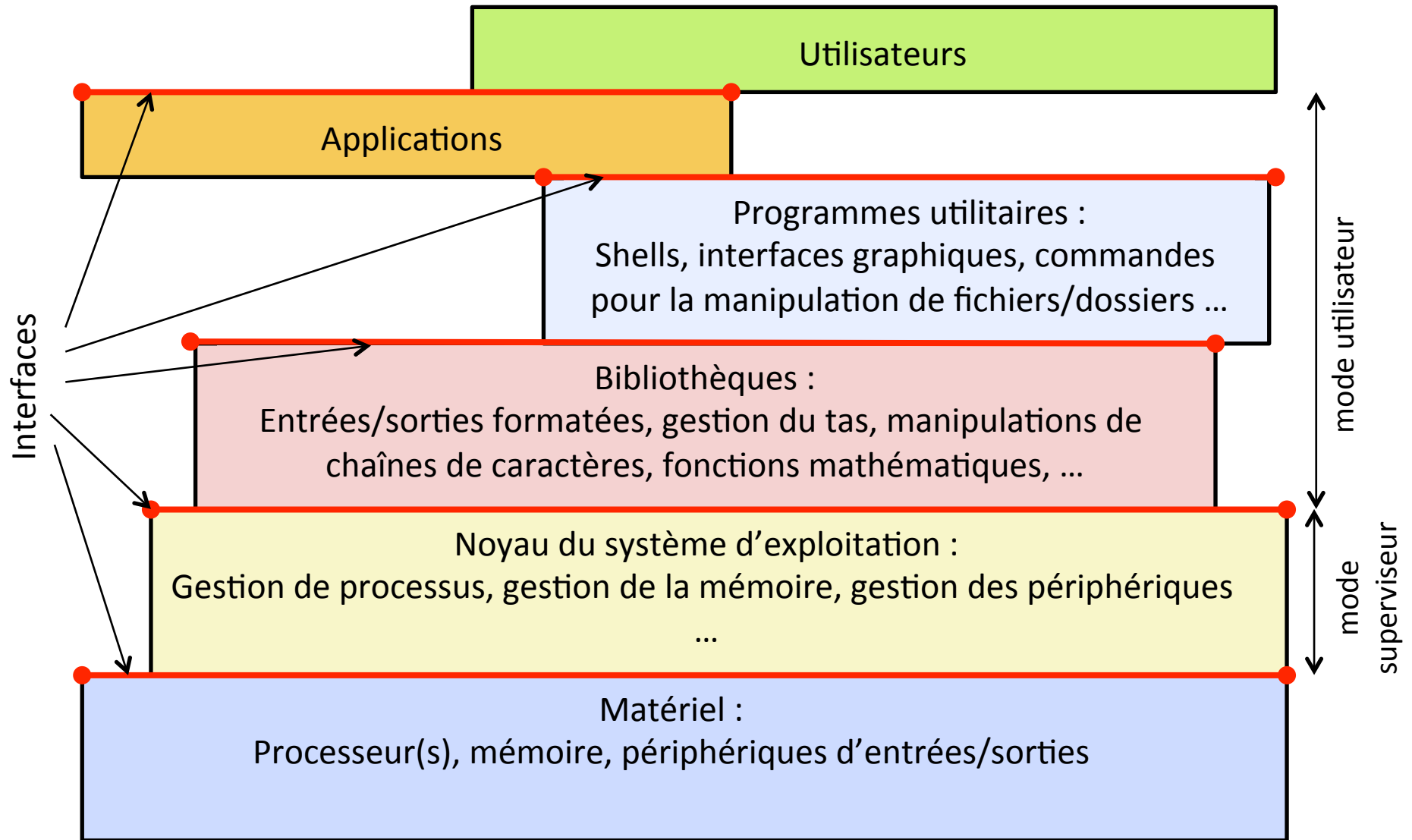
Documentation en ligne par `man`

`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

```
...
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;
    }
} ...
```

Structure classique d'un système informatique



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

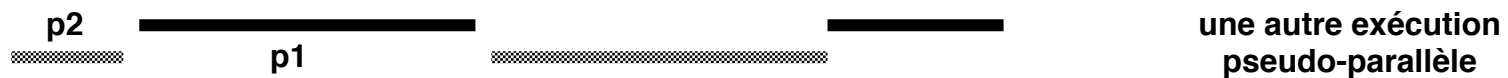
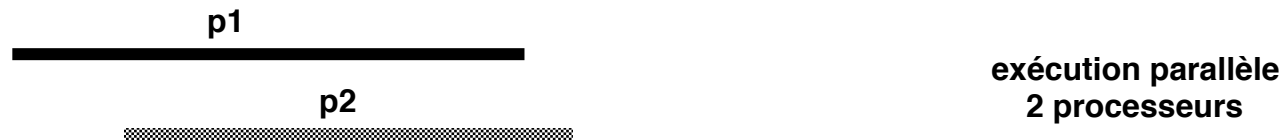
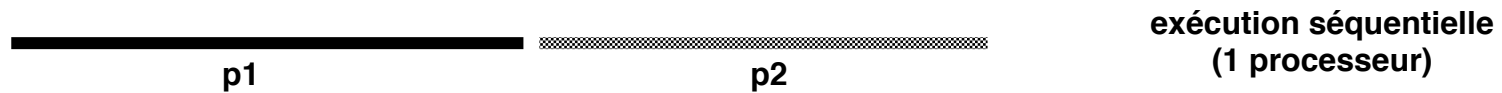
- ◆ l'exécution d'un programme
- ◆ la copie d'un fichier sur disque
- ◆ la transmission d'une séquence de données sur un réseau

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**

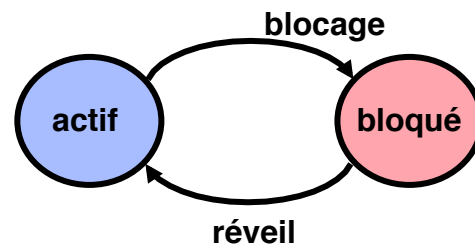
```
p1
while (ressource occupée)
{ };
ressource occupée = true;
...
```

```
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

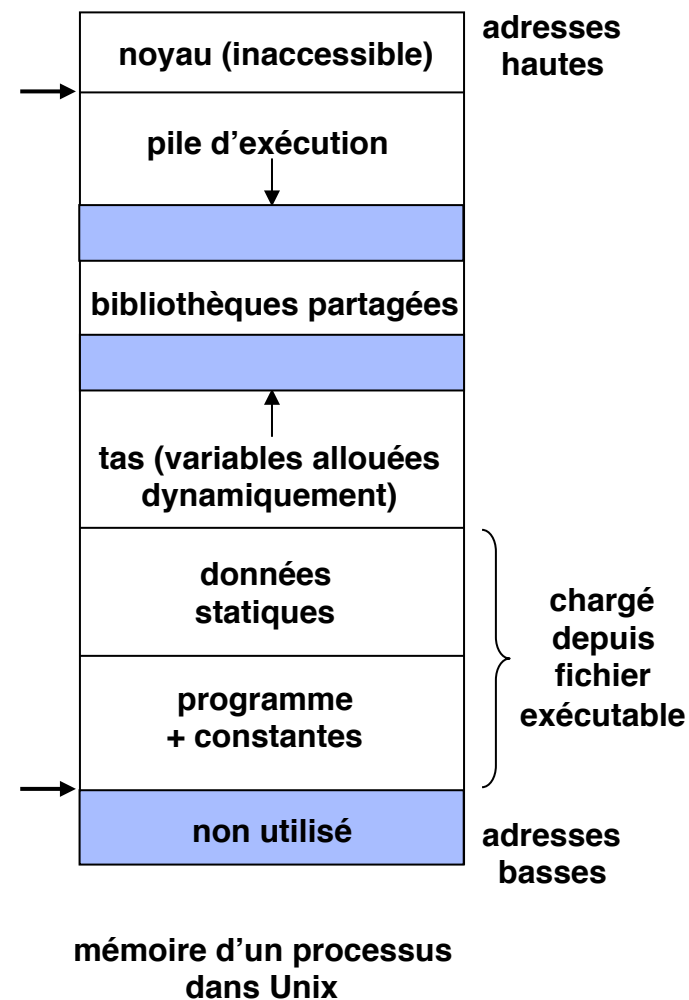
- ◆ commande (du langage de commande)
- ◆ programme d'application

■ 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 est 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



Environnement d'un processus (1)

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 **env** (sans paramètres) affiche l'environnement courant

Pour attribuer la valeur `<val>` à la variable `VAR`,

- Shell tcsh : **setenv VAR <val>**
- Shell bash : **export VAR=<val>**

La commande **echo \$VAR** affiche la valeur courante de la variable **VAR**

Environnement d'un processus (2)

On peut aussi consulter et modifier les variables d'environnement par programme

```
#include <stdlib.h>
char * getenv (const char *nom);
int putenv (const char *chaine);
int setenv (const char *nom, const char *valeur, int écraser);
int unsetenv (const char *nom);
```

Exemples :

```
char *var = getenv("USER");
if (var != NULL) printf("utilisateur: %s", var);
```

```
putenv("MAVARIABLE=toto");
setenv("MAVARIABLE", "toto", 1);
```

```
unsetenv("MAVARIABLE"); // équivalent à putenv("MAVARIABLE") !!!
```

Vie et mort des processus

■ Un processus a généralement un début et une fin

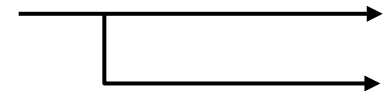
- ◆ Début : création par un autre processus - par `fork()`
 - ❖ il existe un processus “primitif” (PID = 1) créé à l’origine du système
- ◆ Fin
 - ❖ auto-destruction (à la fin du programme) - par `exit()`
 - ❖ destruction par un autre processus - par `kill()`
 - ❖ certains processus ne se terminent pas avant l’arrêt du système (“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 créer 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 suivi par exec (exécution d'un nouveau  
programme) */  
}  
...
```

```
je suis le fils, mon PID est 10271  
je suis le père, mon PID est 10270
```

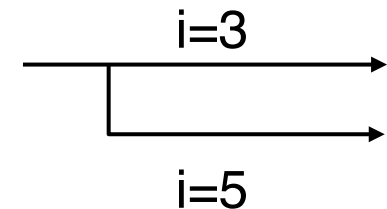
Création des processus dans Unix (2)

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
```

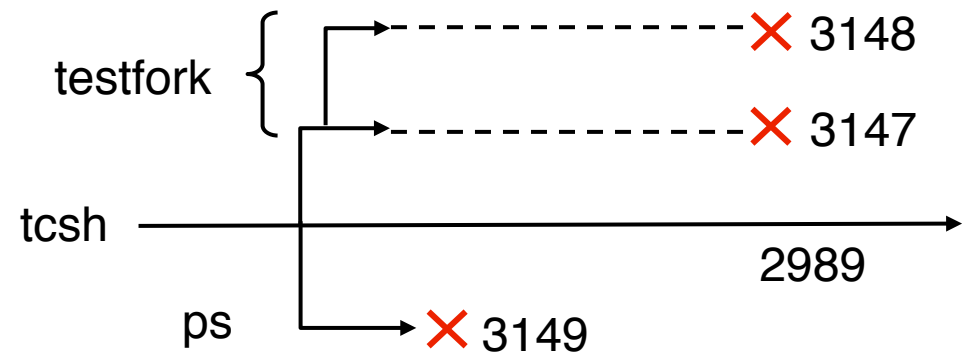


programme
testfork.c

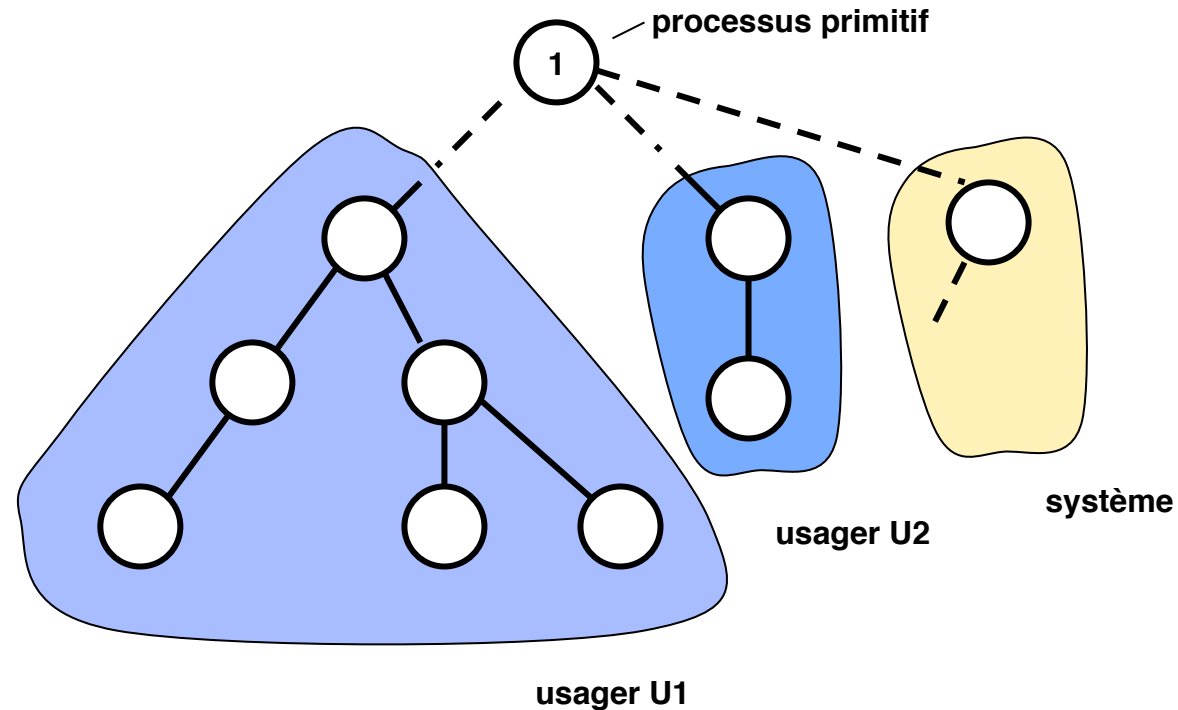
```
int main() {  
    if (fork() != 0) {  
        printf("je suis le père, mon PID est %d\n", getpid());  
        sleep(10) /* blocage pendant 10 secondes */  
        exit(0);  
    } else {  
        printf("je suis le fils, mon PID est %d\n", getpid());  
        sleep(10) /* blocage pendant 10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

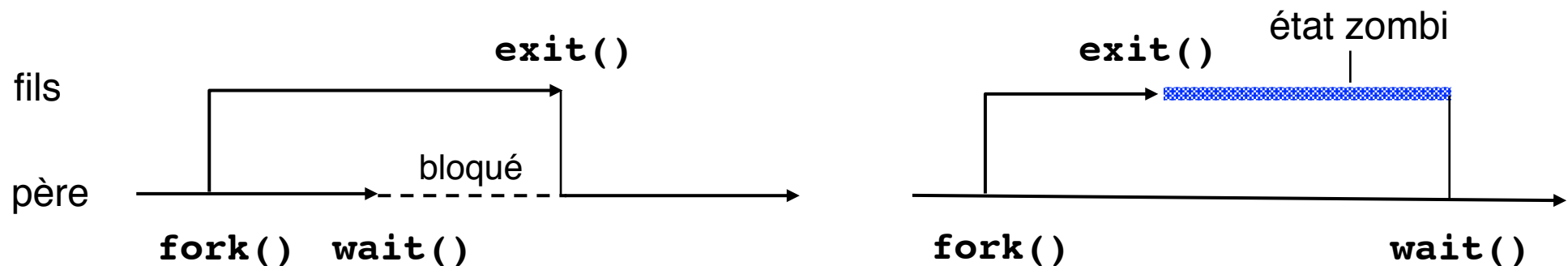
■ Faire attendre un processus

- ◆ `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

```

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);
    }
}

```

programme
testfork1.c

```
<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
```

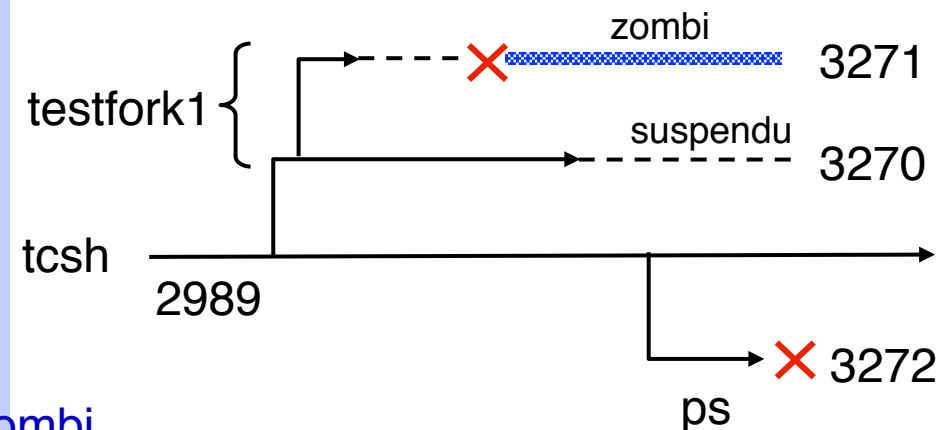
```
<unix> ps
```

PID	TTY	TIME	CMD
2989	pts/0	00:00:00	tcsch
3270	pts/0	00:00:03	testfork1
3271	pts/0	00:00:00	testfork1 <defunct>
3272	pts/0	00:00:00	ps

```
<unix>
```

zombi

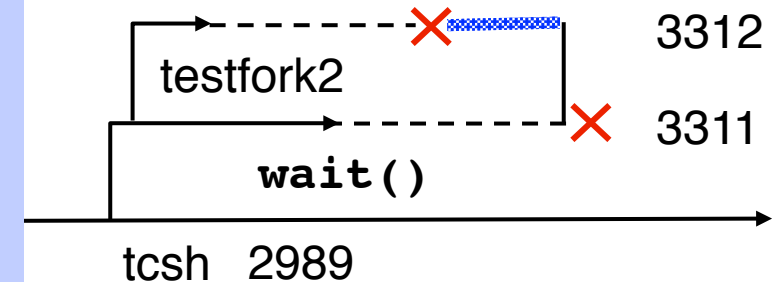
Exemple simple d'exécution



programme
testfork2.c

```
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

Les différentes variantes d' Exec

■ Il existe 6 versions différentes de la primitive Exec:

- ◆ `execv`, `execve`, `execvp`, `execl`, `execle`, `execlp`

■ Nuances

◆ Passage des paramètres

- ❖ Sous forme de tableau de pointeurs (`v`) : `execv`, `execve`, `execvp`

- ❖ Sous forme de liste (`l`) : `execl`, `execle`, `execlp`

◆ Passage de l'environnement

- ❖ Explicite : paramètre pointeur vers tableau de pointeurs (`e`) : `execve`, `execle`

- ❖ Implicite (héritage de l'environnement courant) : les autres primitives

◆ Chemin de l'exécutable

- ❖ Recherche dans le `PATH` (`p`) : `execlp`, `execvp`

- ❖ Nécessité de spécifier le chemin complet : les autres primitives

■ En général, une seule de ces primitives est fournie par le système d'exploitation (`execve`) et les autres sont construites au dessus dans des bibliothèques

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