

# Composantes d'un Ordinateur

---

## **Lecture2: Chapitre 3 (Silberchatz)**

**Objectif:**

# Concepts importants du Chapitre 2 - Processus

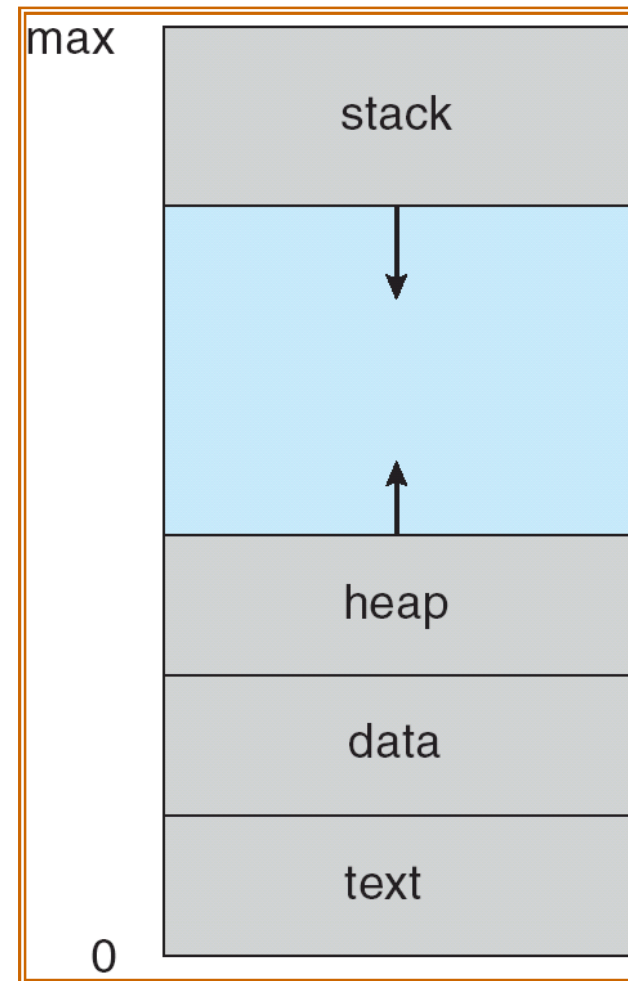
- **Le processus – pour exécuter un programme**
  - États et transitions d'état des processus
  - Bloc de contrôle de processus (Process Control Block)
- **Commutation/permutation de processus**
  - Sauvegarde, rechargement de PCB
  - Files d'attente de processus et PCB
  - Ordonnanceurs à court, moyen, long terme
- **Opérations des processus**
  - ◆ Création, terminaison, hiérarchie
- ◆ **La communication entre les processus**

# Processus et terminologie (aussi appelé **job**, **task**, **user program**)

- **Concept de processus: un programme en exécution**
  - Possède des ressources mémoires, périphériques, etc
- **Ordonnancement de processus**
- **Opérations des processus**
- **Exemple de la création et terminaison de processus**
- **Processus coopérants (communication entre processus)**

# Exécution de programme par un processus

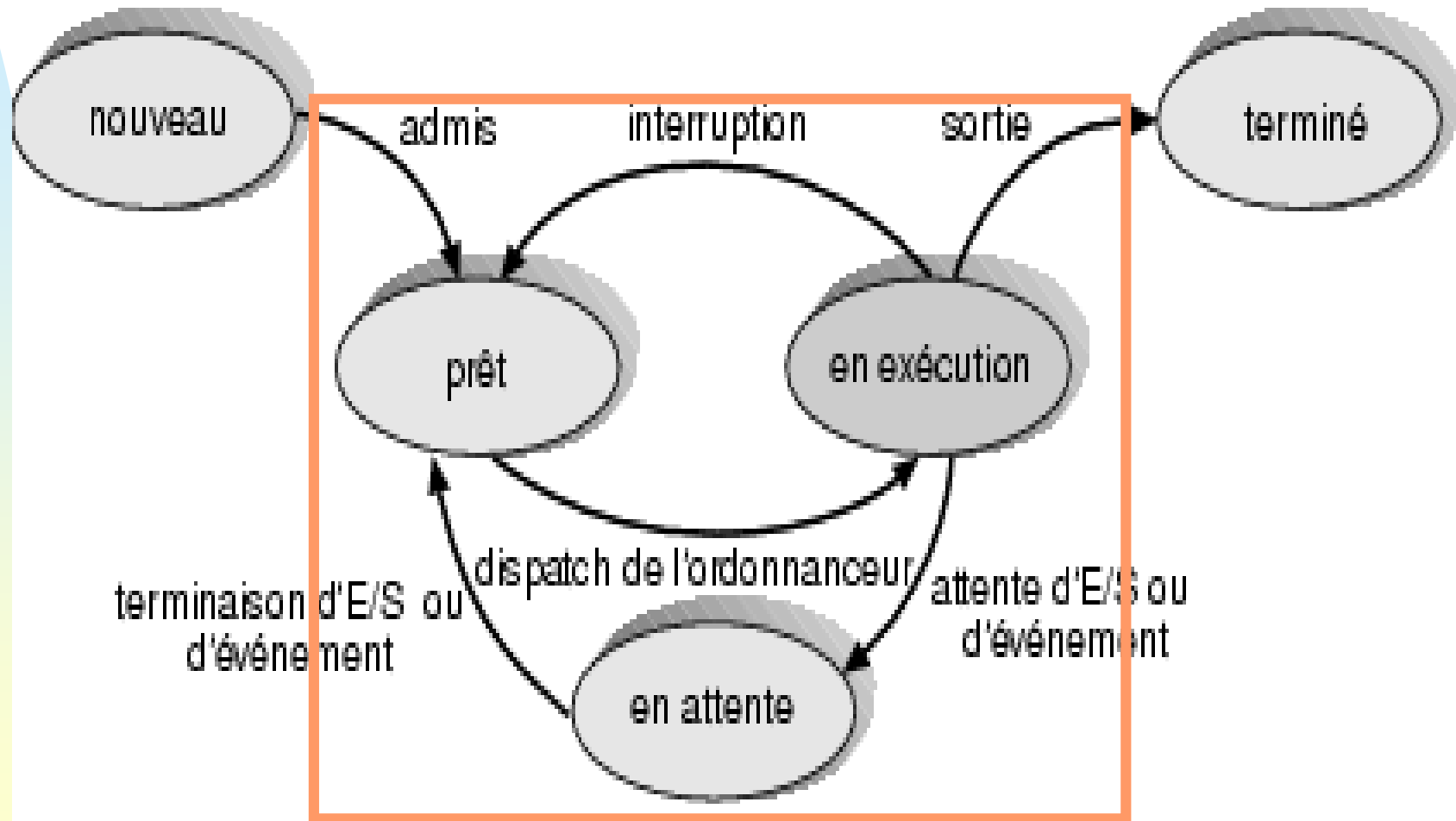
- Texte – code à exécuter
- Données
- Tas (heap)
- Pile (stack)



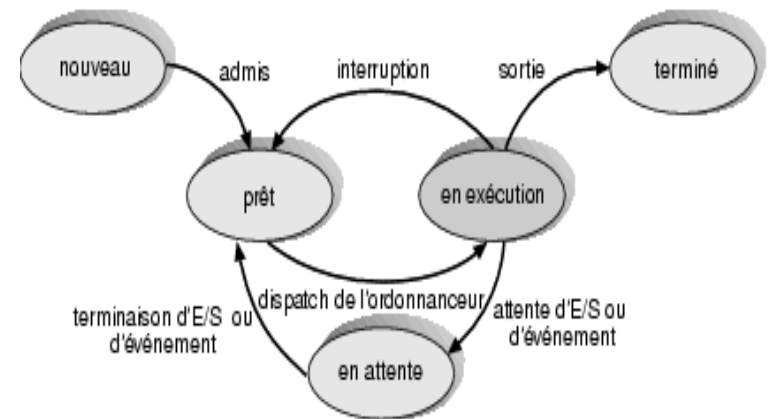
# État de processus **IMPORTANT**

- **Durant son exécution un processus change d'états:**
  - **Nouveau:** le processus vient d'être créé
  - **En exécution (running):** le processus est en train d'être exécuté par l'UCT
  - **En attente (waiting):** le processus est en train d'attendre un événement (ex. la fin d'une opération d'E/S)
  - **Prêt (ready):** le processus est en attente d'être exécuté par l'UCT
  - **Terminaison:** fin d'exécution

# Diagramme de transition d'états d'un processus



# Transitions entre processus



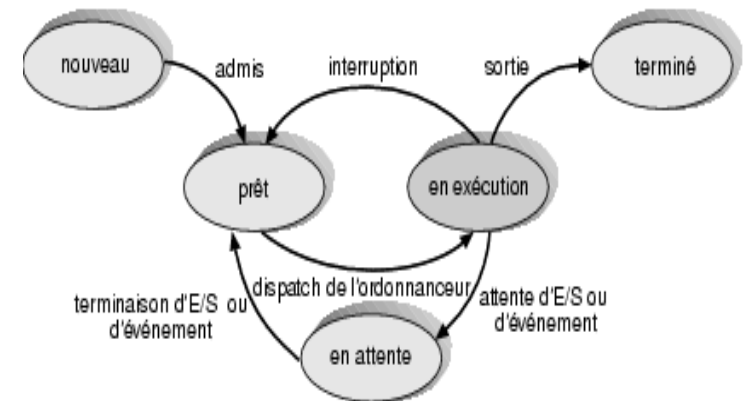
## ■ Prêt → Exécution

- Lorsque l'ordonnanceur UCT choisit un processus à exécuter

## ■ Exécution → Prêt

- Résultat d'une interruption causée par un événement indépendant du processus
  - Il faut traiter cette interruption, donc le processus courant perd le contrôle de l'UCT
    - Cas important: le processus a épuisé son intervalle de temps (minuterie)

# Transitions entre processus



## ■ Exécution → Attente

- Lorsqu'un processus fait une requête de service du SE et que le SE ne peut satisfaire immédiatement (interruption causée par le processus lui-même)
  - un accès à une ressource pas encore disponible
  - initie une E/S: doit attendre le résultat
  - a besoin de la réponse d'un autre processus

## ■ Attente → Prêt

- lorsque l'événement attendu se produit



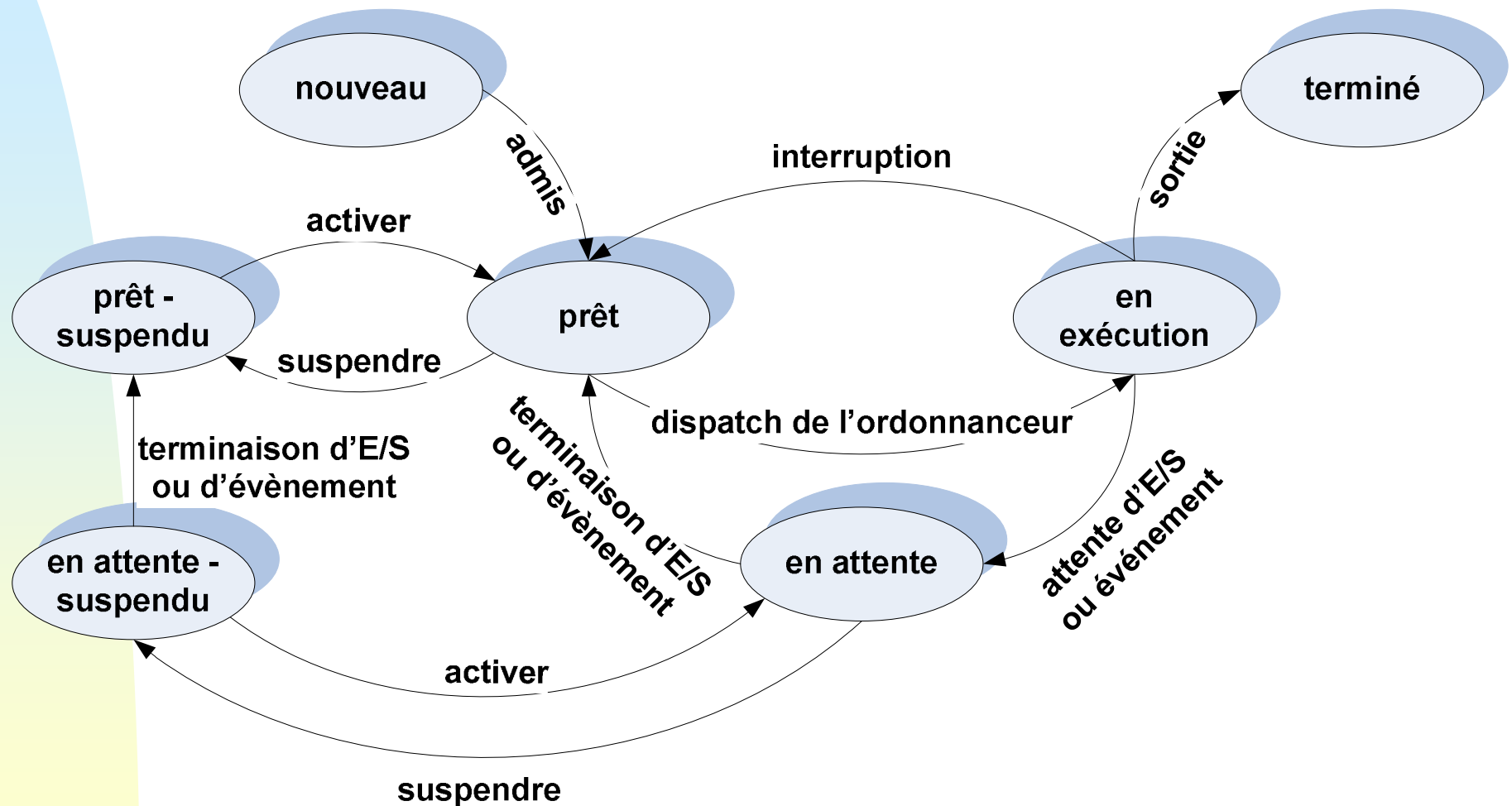
# La nécessité d'échanger (swapping)

- Jusqu'à maintenant, tous les processus étaient (au moins partiellement) dans la mémoire (RAM)
- Même avec la mémoire virtuelle, le SE ne peut pas maintenir trop de processus en mémoire sans causer de détérioration de la performance
- Le SE doit parfois **suspendre** certains processus, c.à.d: **les transférer au disque**. Donc nécessité de 2 autres états (ci-dessous):
- **En attente Suspendu**: processus bloqués transférés au disque
- **Prêt Suspendu**: processus prêts transférés au disque

# Transitions additionnelles

- **En attente --> En attente Suspendu**
  - Choix privilégié par le SE pour libérer de la mémoire
- **En attente Suspendu --> Prêt Suspendu**
  - Lorsque l'évènement attendu se produit (info d'état est disponible au SE)
- **Prêt Suspendu --> Prêt**
  - lorsqu'il n'y a plus de processus prêt (en mém.)
- **Prêt --> Prêt Suspendu (rare)**
  - On doit libérer de la mémoire mais il n'y a plus de processus bloqués en mémoire

# Un modèle de processus à 7 états





# Sauvegarde d'informations de processus

- En multiprogrammation, un processus s'exécute sur l'UCT de façon intermittente
- Chaque fois qu'un processus reprend le contrôle de l'UCT (transition prêt → exécution) il doit la reprendre dans la même situation où il l'a laissée (même contenu des registres UCT, etc.)
- Donc au moment où un processus sort de l'état d'exécution il est nécessaire de sauvegarder ses informations essentielles, qu'il faudra récupérer quand il retourne à cet état.

# PCB = Process Control Block:

*Représente la situation actuelle d'un processus, réutilisable plus tard*

pointeur 	état de processus 
numéro de processus	
compteur programme	
registres	
limites mémoire	
liste des fichiers ouverts	
⋮	

} Registres UCT

# Process Control Block (PCB)

## **IMPORTANT**

- **pointeur**: les PCBs sont rangés dans des listes enchaînées (à voir)
- **état de processus**: (ready, running, waiting ...)
- **compteur de programme**: le processus doit reprendre à l'instruction suivante
- **autres registres UCT**
- **bornes de mémoire**
- **Gestion de la mémoire**
- **fichiers qu'il a ouvert**
- **etc., v. manuel**

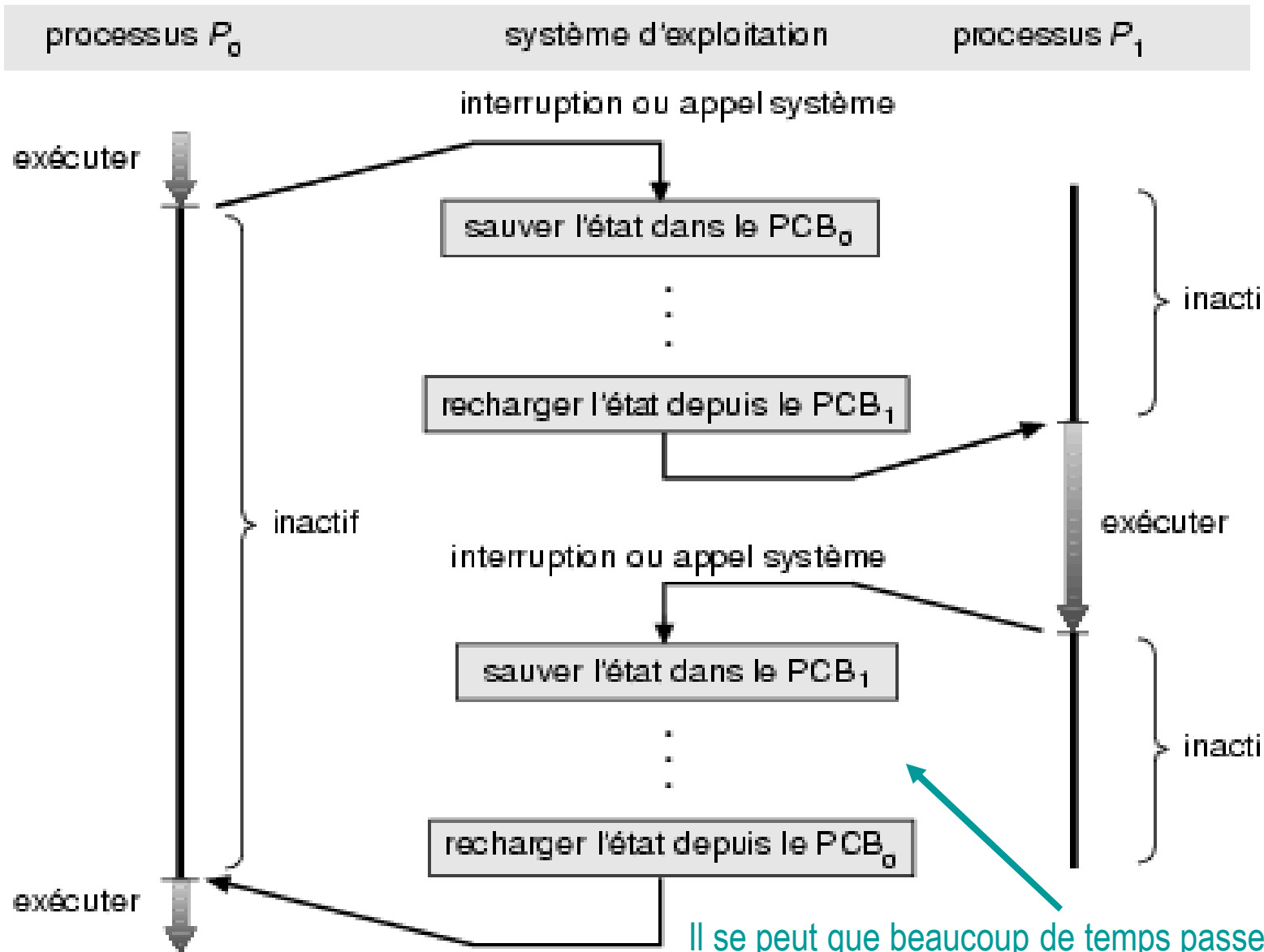
pointeur	état de processus
numéro de processus	
compteur programme	
registres	
limites mémoire	
liste des fichiers ouverts	
⋮	

# Permutation de processus

Aussi appelé commutation de contexte (context switching)

- Quand l'UCT passe de l'exécution d'un processus 0 à l'exécution d'un processus 1, il faut :
  - mettre à jour le PCB du processus 0
  - sauvegarder le PCB du processus 0
  - Retrouver et accéder le PCB du processus 1, qui avait été sauvegardé avant
  - Mettre à jour les registres d'UCT, compteur d'instructions etc. avec l'information décrite dans le PCB du processus 1

# Commutation de processeur (context switching)



Il se peut que beaucoup de temps passe avant le retour au processus 0, et que beaucoup d'autres proc soient exécutés entre temps



## ***Le PCB n'est pas la seule information à sauvegarder...*** (le manuel n'est pas clair ici)

- **Il faut aussi sauvegarder l'état des données du programme**
- **Ceci se fait normalement en gardant l'*image du programme* en mémoire primaire ou secondaire**
- **Le PCB pointera à cette image**

# Processus et terminologie (aussi appelé **job**, **task**, **user program**)

- **Concept de processus: un programme en exécution**
  - Possède des ressources de mémoire, périphériques, etc
- **Ordonnancement de processus**
- **Opérations sur les processus**
- **Exemple de la création et terminaison de processus**
- **Processus coopérants (communication entre processus)**

# L'Ordonnancement des processus

- **C'est quoi?**
  - La sélection du prochain processus à exécuter
- **Pourquoi?**
  - Multiprogrammation - pour optimiser l'utilisation du système
  - Partage de temps - Pour améliorer le temps de réponse
- **Comment?**
  - En détails au Module 4 (Chapitre 5 du texte)
  - Pour le moment, introduisons les notions fondamentales de l'ordonnancement
  - Déjà introduit : la permutation de processus

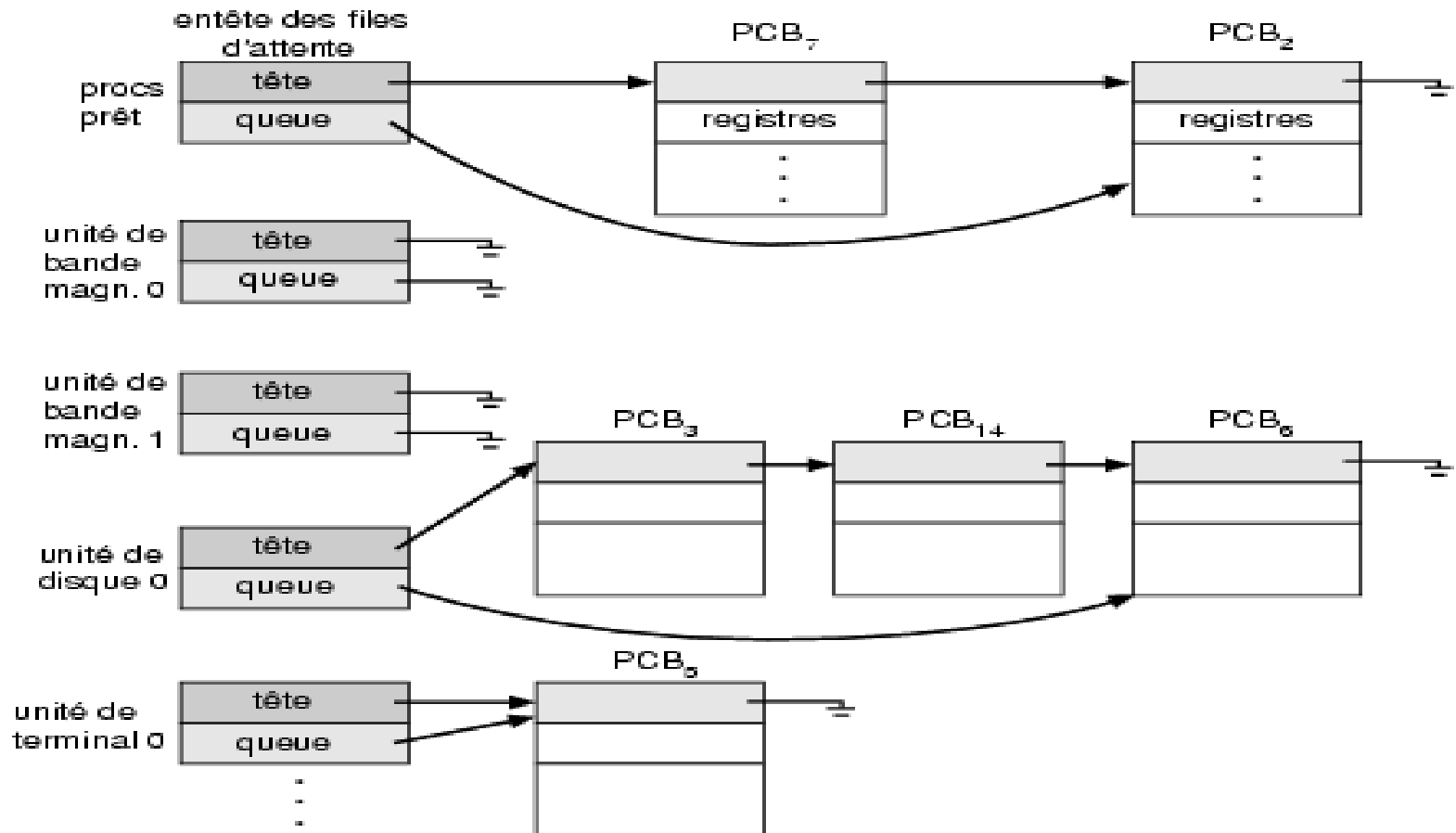
# Files d'attente

## ***IMPORTANT***

- Les ressources d'ordinateur sont souvent limitées par rapport à la demande requise des processus
- Chaque ressource a sa propre file d'attente de processus
- En changeant d'état, les processus passent d'une file à l'autre
  - File prêt: les processus en état prêt (ready)
  - Files associés à chaque unité E/S
  - etc.

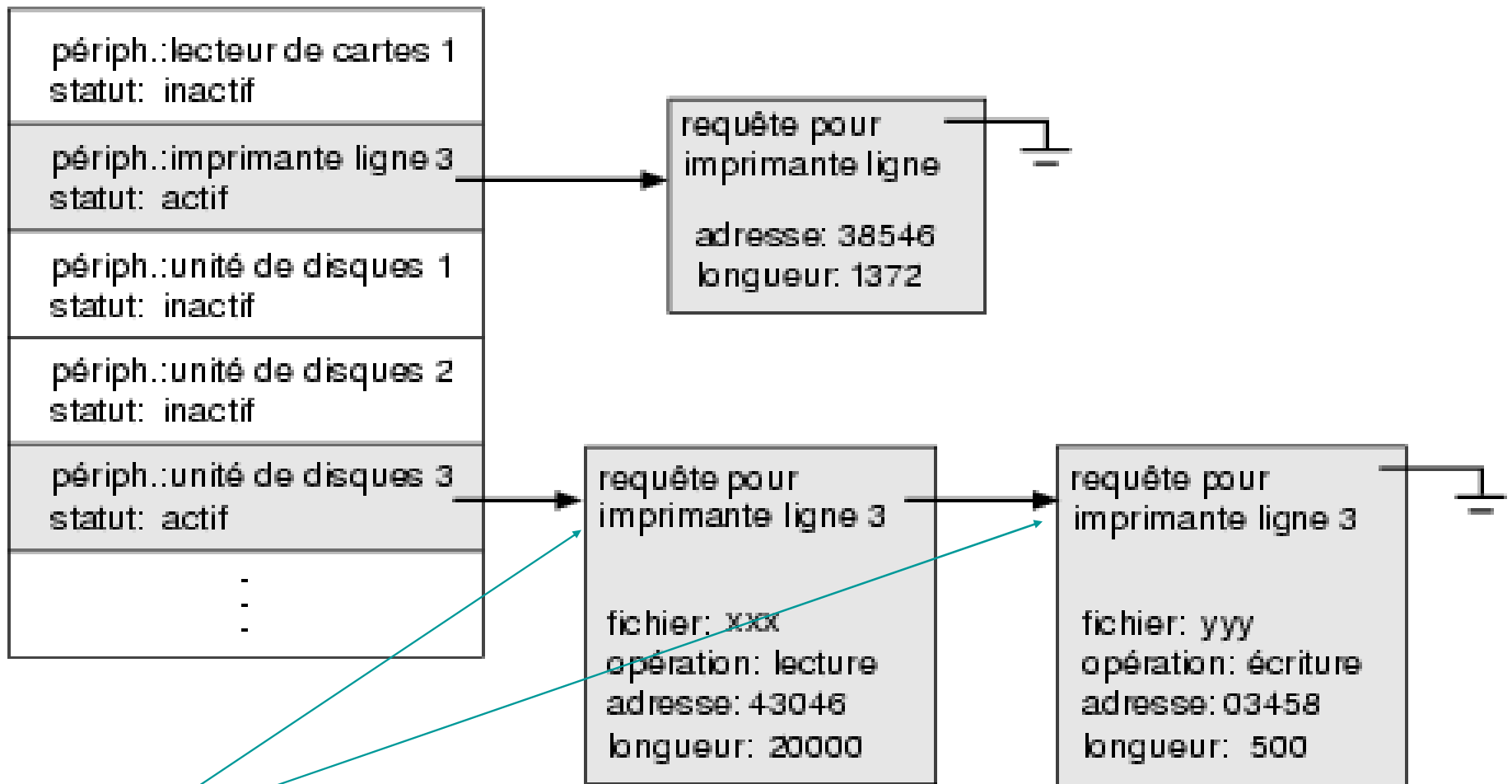
# Ce sont les PCBs qui sont dans les files d'attente (d'où le besoin d'un *pointeur* dans le PCB)

file prêt



Nous ferons l'hypothèse que le premier processus dans une file est celui qui utilise la ressource: ici, proc7 exécute, proc3 utilise disque 0, etc.

# Cet ensemble de files inclut donc la table de statut périphériques



# Une façon plus synthétique de décrire la même situation

prêt  $\rightarrow 7 \rightarrow 2$

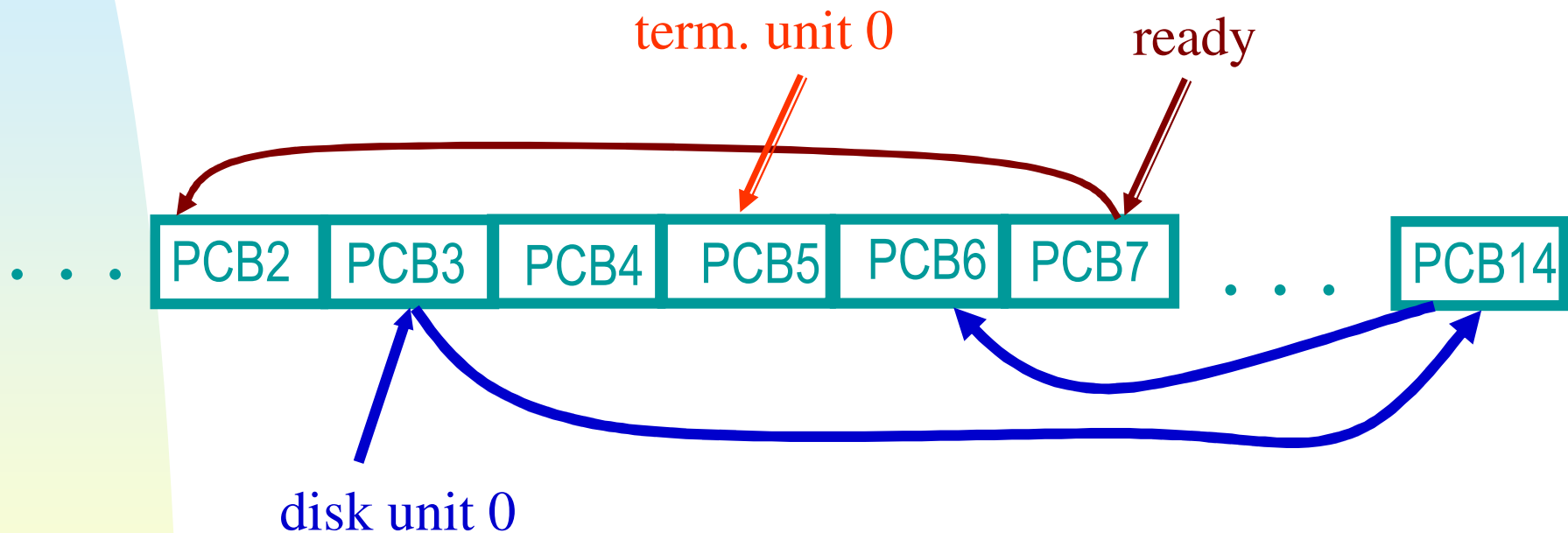
bandmag0  $\rightarrow$

bandmag1  $\rightarrow$

disq0  $\rightarrow 3 \rightarrow 14 \rightarrow 6$

term0  $\rightarrow 5$

**Les PCBs ne sont pas déplacés dans la mémoire lorsqu'ils sont mis dans différentes files: ce sont les pointeurs qui changent.**

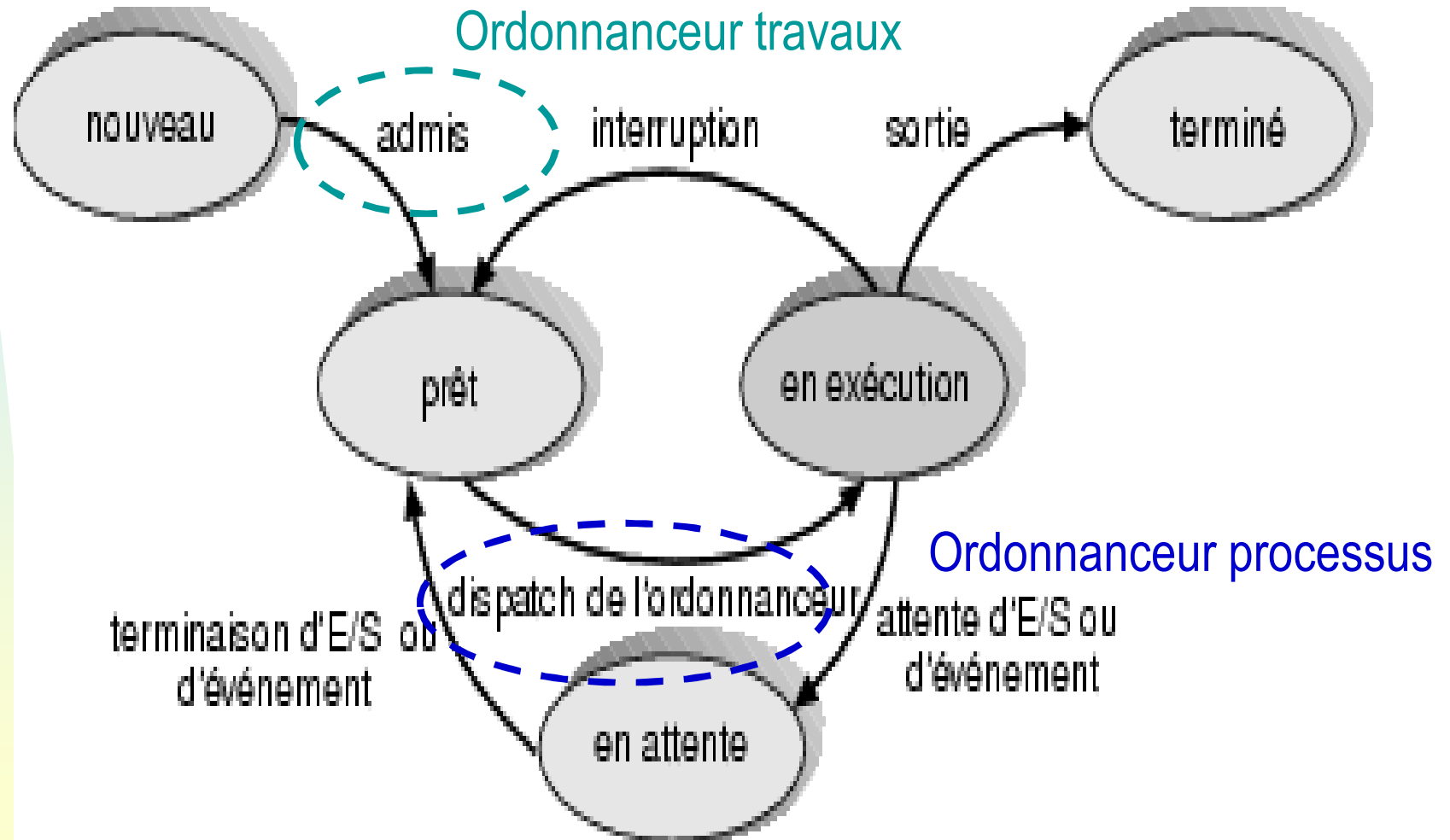




# Ordonnanceurs/Planificateurs (schedulers)

- Programmes qui gèrent l'utilisation de ressources de l'ordinateur
- Trois types d'ordonnanceurs :
  - À court terme = **ordonnanceur de processus**: sélectionne quel processus doit exécuter la transition **prêt** → **exécution**
  - À long terme = **ordonnanceur de travaux**: sélectionne quels processus peuvent exécuter la transition **nouveau** → **prêt** (événement *admis*) (de spoule travaux à file prêt) (pour système de traitement par lots, *batch*)
  - À moyen terme: nous verrons (systèmes à temps partagé)

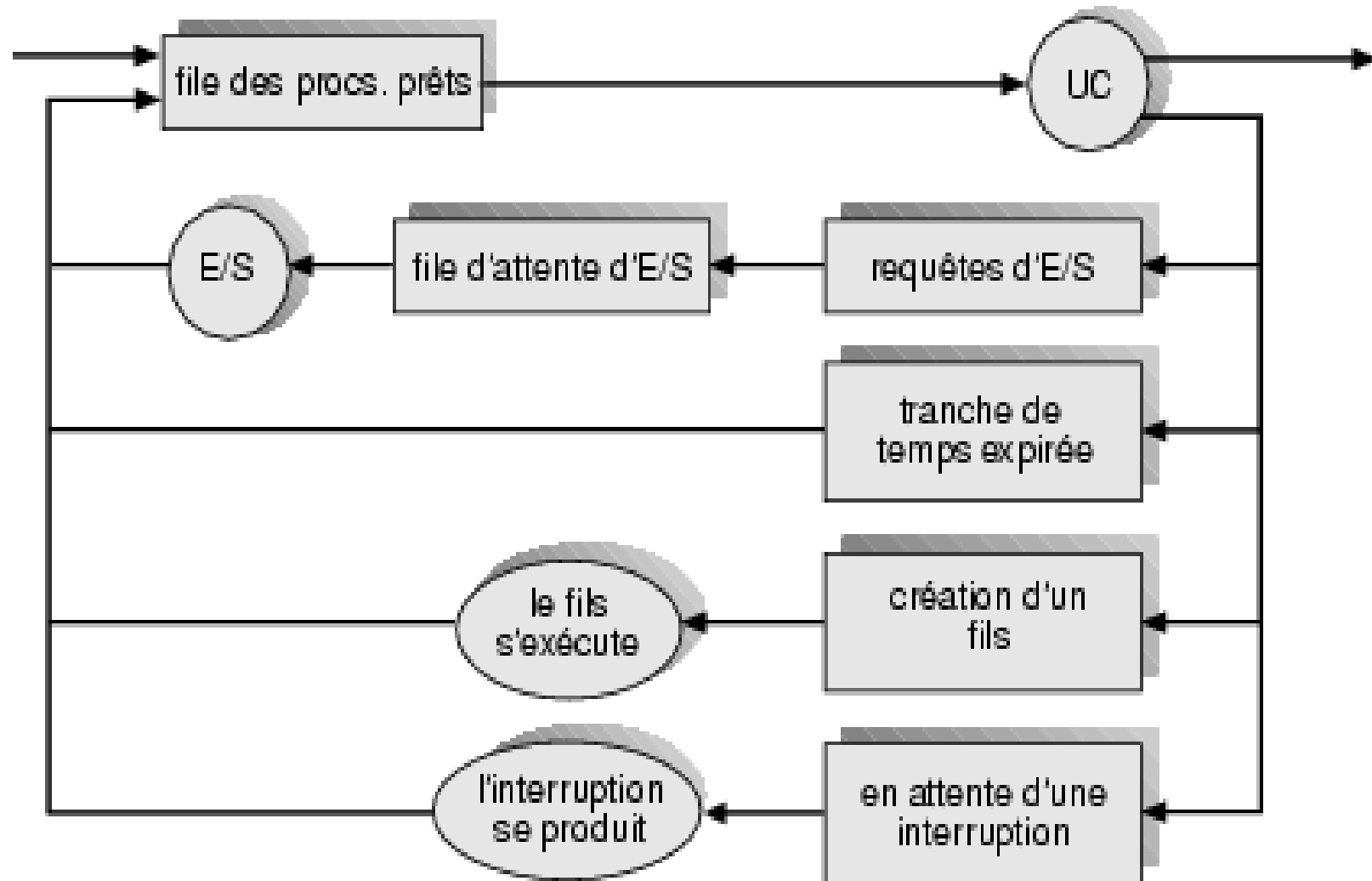
# Ordonnanceur travaux = long terme et ordonnanceur processus = court terme



# Ordonnanceurs

- **L'ordonnanceur à court terme est très souvent exécuté (millisecondes)**
  - **doit être très efficace**
- **L'ordonnanceur à long terme doit être exécuté beaucoup plus rarement: il contrôle le niveau de multiprogrammation**
  - **doit établir une balance entre travaux liés à l'UCT et ceux liés à l'E/S**
  - **de façon à ce que les ressources de l'ordinateur soient bien utilisées**

# Ordonnancement de processus (court terme)

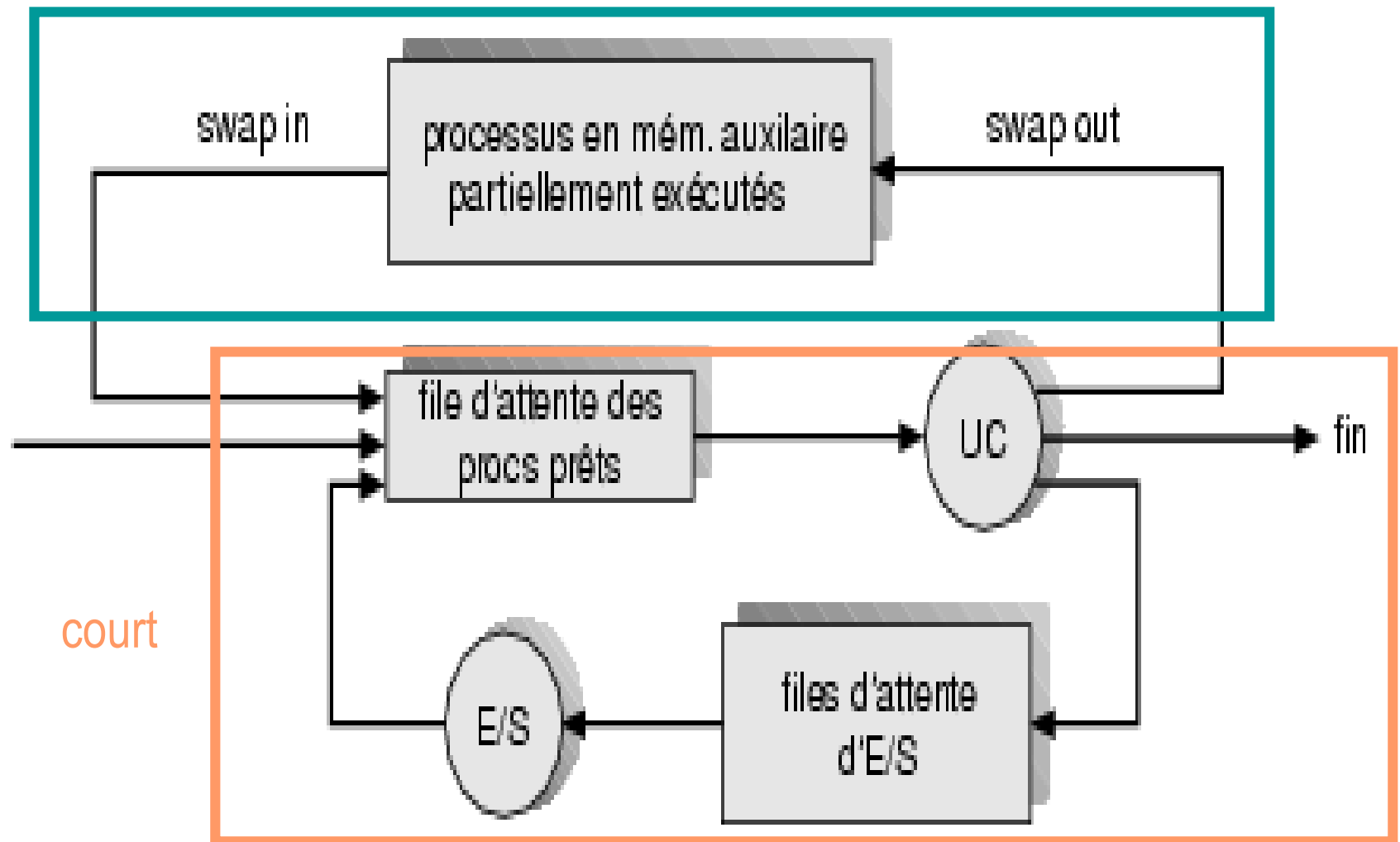


Disponibilité Ress.

# Ordonnanceur à moyen terme

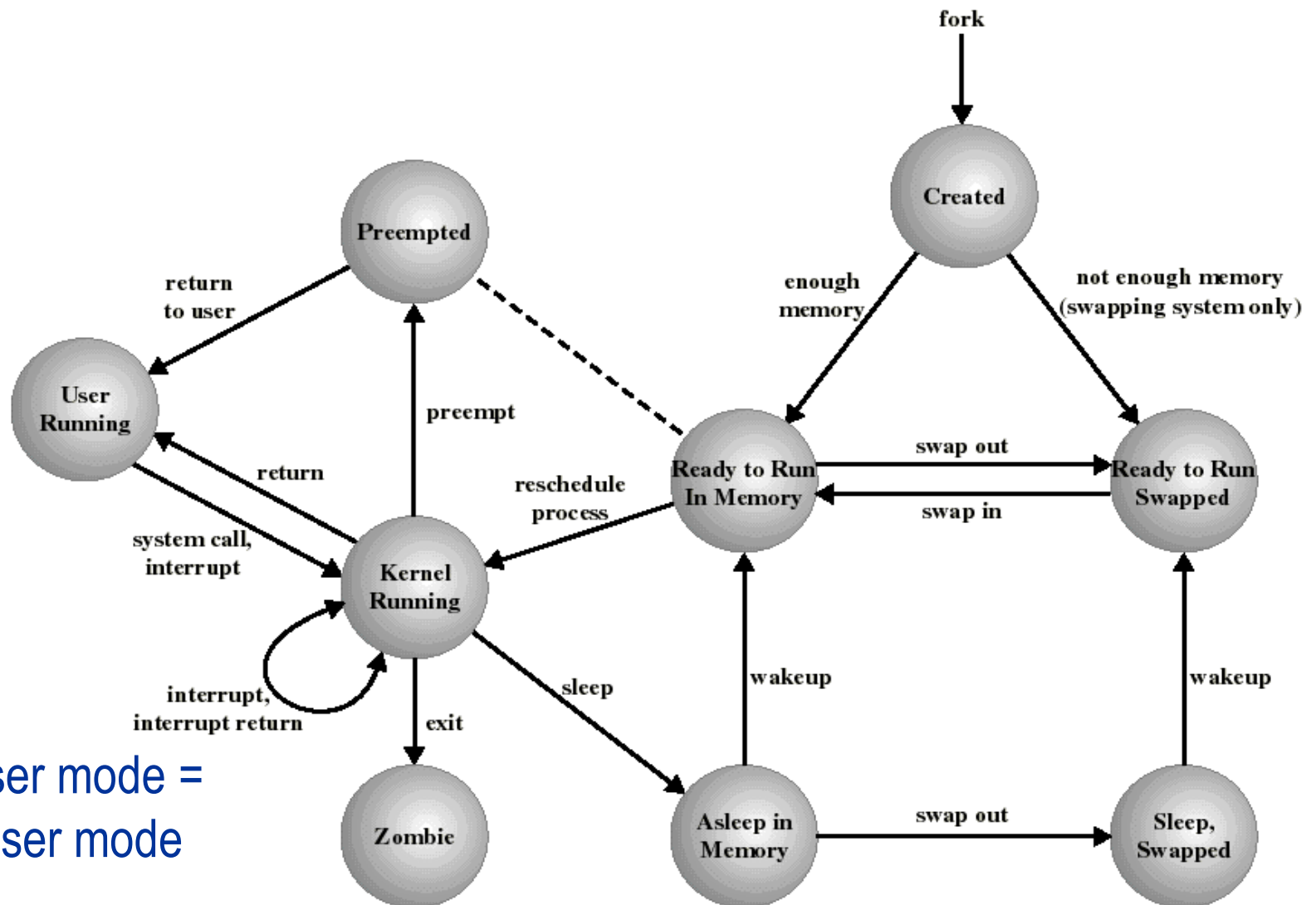
- Le manque de ressources peut parfois forcer le SE à *suspendre* des processus
  - ils seront plus en concurrence avec les autres pour des ressources
  - ils seront repris plus tard quand les ressources deviendront disponibles
- Ces processus sont enlevés de la mémoire centrale et mis en mémoire secondaire, pour être repris plus tard
  - `swap out`, `swap in`, va-et-vient

# Ordonnanceurs à court et moyen terme



# États de processus dans UNIX SVR4 (Stallings)

Un exemple de diagramme de transitions d'états pour un SE réel



Kernel, user mode =  
monitor, user mode

# Processus et terminologie (aussi appelé **job**, **task**, **user program**)

- **Concept de processus: un programme en exécution**
  - Possède des ressources de mémoire, périphériques, etc.
- **Ordonnancement de processus**
- **Opérations sur les processus**
- **Exemple de la création et terminaison de processus**
- **Processus coopératifs (communication entre processus)**



# La création des processus

D'où viennent les processus?

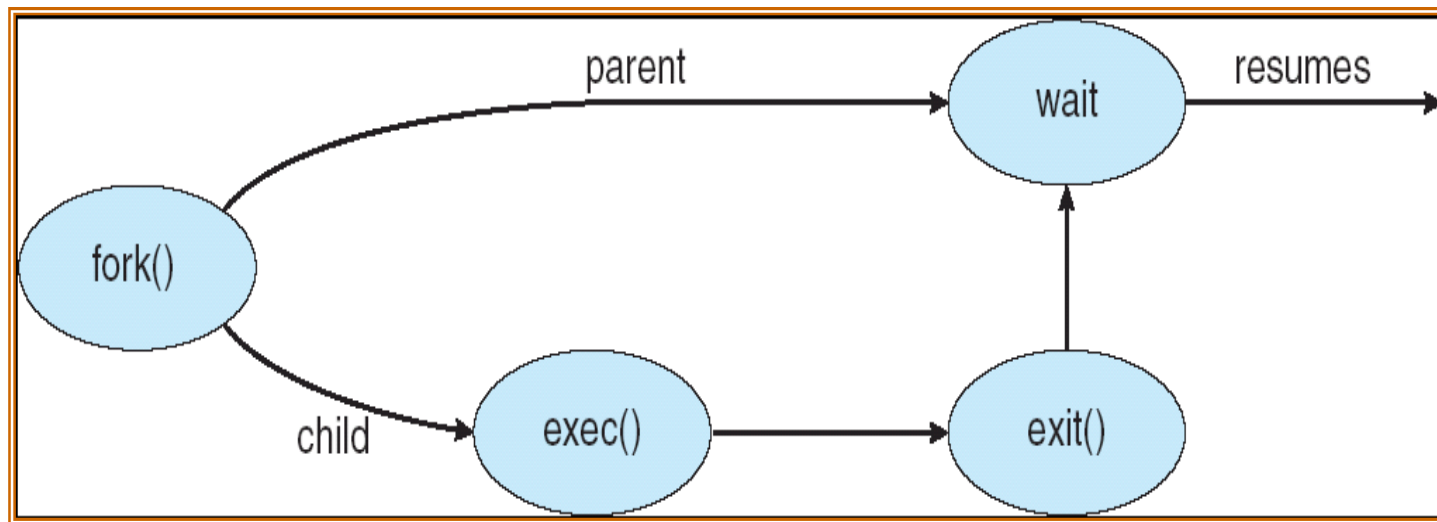
- **Un processus parent crée des processus enfants, qui à leur tour peuvent créer d'autres processus, ainsi formant un arbre de processus**
- **Plusieurs propriétés peuvent être spécifiées à la création d'un enfant:**
  - **Comment un parent et enfant partagent des ressources?**
    - Partage toutes les ressources, une partie, ou aucune
  - **Le parent continue peut opérer en même temps ou doit attendre que l'enfant termine.**
  - **Le contenu du processus de l'enfant**
    - Peut être une image du parent
    - Peut contenir un nouveau programme

# La création de processus (suite)

## Exemple UNIX:

- L'appel de système `fork()` crée un nouveau processus contenant l'image du parent
  - Pas de mémoire partagé, simplement une copie
  - Retourne le PID de l'enfant au parent, et 0 au nouveau processus enfant.
  - Le parent peut utiliser l'appel `wait()` pour attendre que l'enfant termine
- L'appel système `exec(...)` est utilisé dans l'enfant après le `fork()` pour remplacer la mémoire du processus avec un nouveau programme (c.à.d charger/installer un nouveau logiciel)

# UNIX: fork(), exec(), exit() & wait()



# Programme C qui “fork” un nouveau processus

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    int pid;
    pid = fork();  /* fork un autre processus */
    if (pid < 0)
    { /* error occurrence d'erreur */
        fprintf(stderr, "Echec du Fork");
        exit(-1);
    }
    else if (pid == 0) /* processus de l'enfant */
    { execlp("/bin/ls", "ls", NULL); }
    else
    { /* le processus du parent attendra que celui de l'enfant finisse */
        wait(NULL);
        printf("Processus de l'enfant Terminé");
        exit(0);
    }
}
```

# Exemple 1 - Fork

```
int pid, a = 2, b=4;
pid = fork();  /* fork un autre processus */
if (pid < 0) exit(-1); /* Echec du fork */
else if (pid == 0) /* processus de l'enfant */
    { a = 3; printf("%d\n", a+b); }
else
{
    wait();
    b = 1;
    printf("%d\n", a+b);
}
```

Qu'est ce qui sera imprimé au terminal?

## Exemple 2 - Fork

```
int pid, a = 2, b=4;
pid = fork();  /* fork un autre processus */
if (pid < 0) exit(-1); /* Echec du fork */
else if (pid == 0) /* processus de l'enfant */
{ a = 3; printf("%d\n", a+b); }
else
{
    b = 1;
    wait();
    printf("%d\n", a+b);
}
```

Qu'est ce qui sera imprimé au terminal?

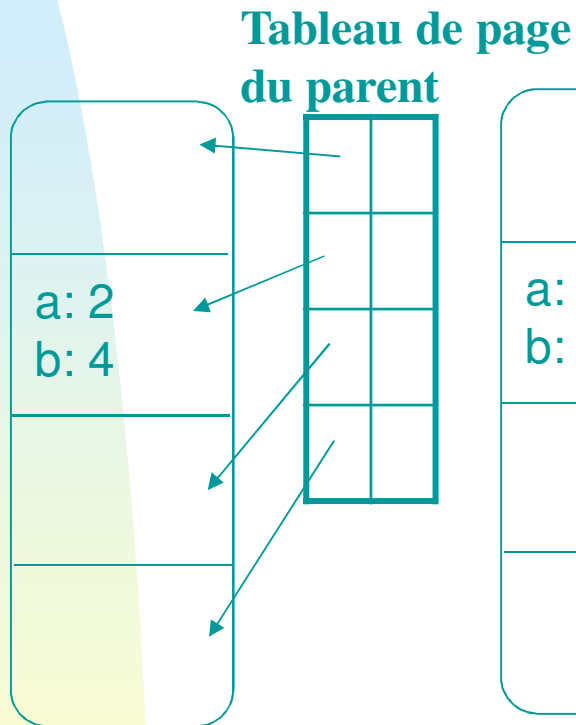
## Exemple 3 - Fork

```
int pid, a = 2, b=4;
pid = fork();  /* fork un autre processus */
if (pid < 0) exit(-1); /* Echec du fork */
else if (pid == 0) /* processus de l'enfant */
{ a = 3; printf("%d\n", a+b); }
else
{
    b = 1;
    printf("%d\n", a+b);
    wait();
}
```

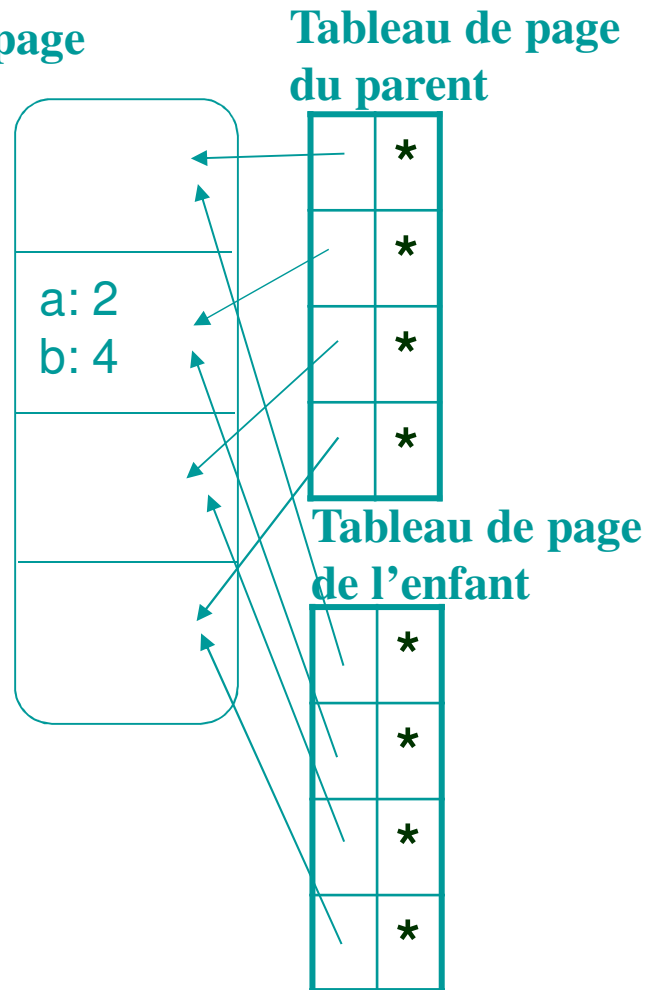
Qu'est ce qui sera imprimé au terminal?

# Comprendre fork()

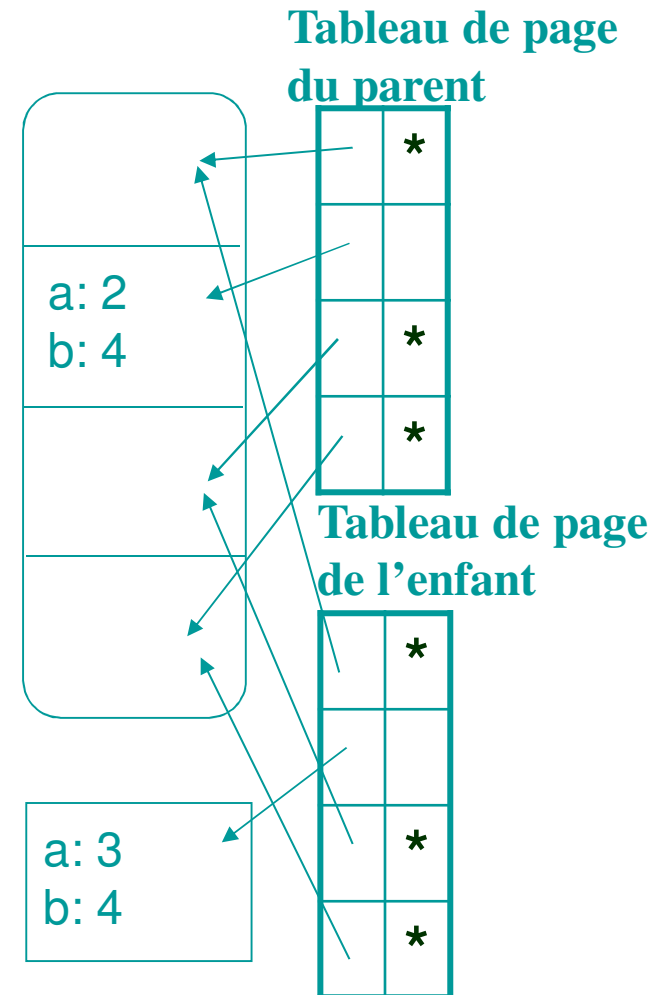
**Avant fork**



**Après fork**



**Après a=3 dans enfant**





# La création de processus (suite)

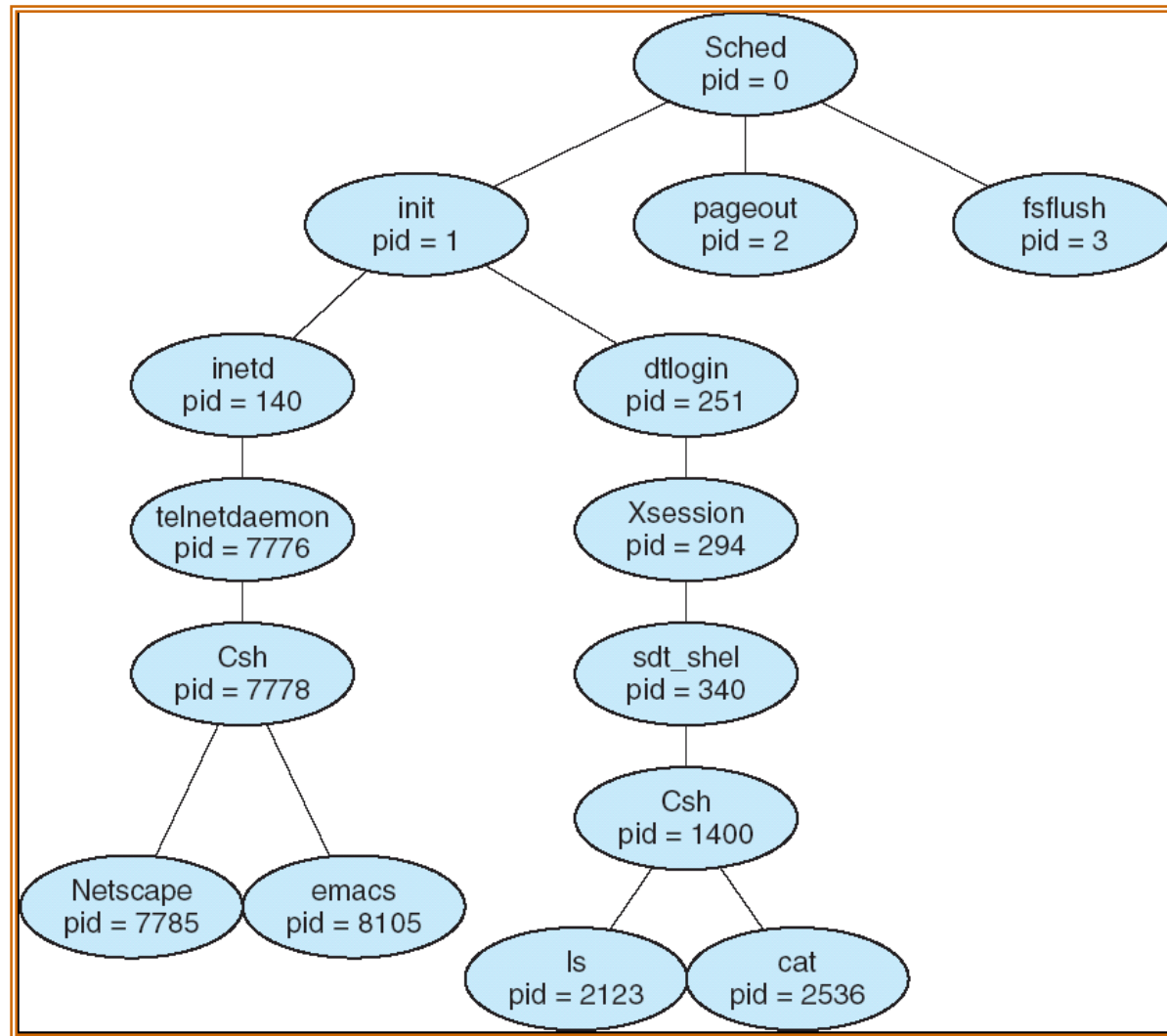
## Windows:

- `CreateProcess(...)`
- **Semblable à `fork()` suivie du `exec()` (un seul appel de système. (Voir Silberchatz, section 3.3.1)**

## Discussion

- `Fork()` est très flexible pour envoyer des données et paramètres du parent à l'enfant
- Très utile pour la configuration (par exemple établir une communication entre processus).
- `CreateProcess()` est plus efficace lorsqu'on veut simplement faire démarrer un nouveau programme

# Exemple d'un arbre de processus (Solaris)



# La terminaison d'un processus

Comment les processus terminent?

- **Un processus finit avec une dernière instruction, l'appel système, `exit()` qui demande au SE de l'effacer.**
- **Terminaison anormale**
  - **Avec erreur: division par zéro, accès invalide de mémoire**
- **Un autre processus peut demander au SE sa terminaison**
  - **Normalement un parent demande la terminaison de ses enfants**
    - SE protège les processus d'autres usagés.
  - **Windows: `TerminateProcess(...)`**
  - **UNIX: `kill(PID, signal)`**

# La terminaison de processus

Que doit faire le SE?

- **Relâche les ressources tenues par le processus**
  - Quand un processus termine et que tous ses ressources ne sont pas encore relâchées, il est dans l'état « *terminated* » (un zombie)
- **Un code de retour (exit state) est envoyé au parent**
  - Le parent vient chercher le code avec l'appel système `wait()`

Que se passe-t-il lorsqu'un processus ayant des enfants termine?

- **Certains SEs (VMS) ne permet pas aux enfants de poursuivre – tous les enfants sont terminés (cascading termination)**
- **D'autres SEs (UNIX) trouve un autre parent pour les orphelins**

# Quelques questions!

- **Les états de processus**
  - Un processus peut-il être déplacé directement de l'état attente à l'état exécutant?
  - De l'état exécutant à l'état prêt?
- **PCB**
  - Est-ce que le PCB contient les variables globales du programme?
- **L'ordonnancement de l'UCT**
  - Quel est la différence entre l'Ordonnanceur à long terme et l'Ordonnanceur à moyen terme?
  - Que se passe-t-il lorsqu'il n'y a aucun processus dans la file prêt?

# Autres questions!

- **La création de processus**

- Quel est la différence entre `fork()`, `exec()` et `wait()`?
- Combien de processus sont créés avec le code suivant

```
for(i=0 ; i<3 ; i++)  
{  
    fork();  
}
```

# Processus et terminologie (aussi appelé **job**, **task**, **user program**)

- **Concept de processus: un programme en exécution**
  - Possède des ressources de mémoire, périphériques, etc
- **Ordonnancement de processus**
- **Opérations sur les processus**
- **Exemple de la création et terminaison de processus**
- **Processus coopérants (communication entre processus)**

# Processus coopérants

- **Les processus indépendants n'affectent pas l'exécution d'autres processus**
- **Les processus coopérants peuvent affecter mutuellement leur exécution**
- **Avantages de la coopération entre processus:**
  - **partage de l'information**
  - **vitesse en faisant des tâches en parallèle**
  - **modularité**
  - **la nature du problème pourrait le demander**



# La communication entre processus (IPC – InterProcess Communication)

Mécanismes qui permettent au processus de  
communiquer et de se synchroniser entre eux


Comment peuvent-ils communiquer?

- En partageant de la mémoire
- En échangeant des messages
- Autres mécanismes
  - Signaux
  - Tuyaux et points de communication (sockets)
  - Sémaphores

# Mémoire partagée

- La mémoire d'un processus est protégé de l'interférence des autres processus
- Un SE doit permettre l'accès à une partie de la mémoire d'un processus (partie spécifique) à un autre processus
  - Un processus fait un appel système pour créer une région de la mémoire partagée
  - L'autre processus fait un appel système pour lier cette région de la mémoire partagée dans son espace mémoire
- Une attention particulière est nécessaire pour accéder les données de la mémoire
  - Synchronisation est important
  - Sera étudiée plus tard (Silberchatz Chapitre 6)

# Échange de messages

- **Deux opérations principales**
  - Send (**destination, message**)
  - Receive (**source, message**)
- **Pour communiquer, on doit d'abord établir une *liaison de communication* entre processus**
  - **Déterminer la destination**
- **Plusieurs variances existent dans le comportement de send() et receive()**
  - **Communication directe ou indirecte** 
  - **Communication synchrone ou asynchrone**
  - **Tampons automatiques ou explicites**

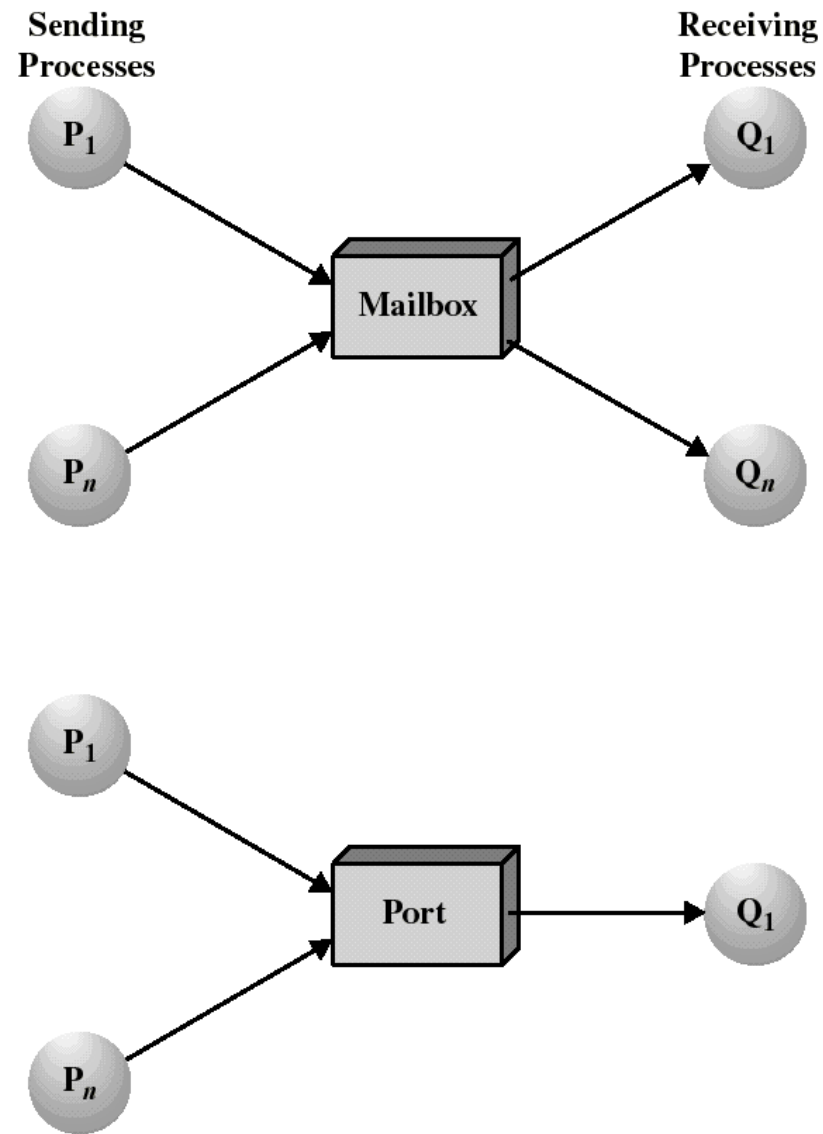
# Communication directe

- **Le Processus nomme explicitement l'autre:**
  - `send(P, message)` – envoie un message au processus *P*
  - `receive(Q, message)` – reçoit un message du processus *Q*
- **Propriétés de la liaison de communication**
  - Les liaisons sont établies automatiquement, plus exactement une liaison entre chaque paire de processus
  - La liaison peut être unidirectionnelle, mais elle est plus souvent bidirectionnelle

# Communication indirecte

- Messages sont échangés avec des boîtes aux lettres (ou ports)
  - Chaque boîte à un identificateur unique
  - Les processus communiquent en partageant une boîte aux lettres
- Primitives de base:
  - send(*A*, *message*) – envoie message à la boîte A
  - receive(*A*, *message*) – reçoit message de la boîte A
- Opérations
  - Créer une nouvelle boîte aux lettres
  - Envoyer et recevoir des messages via la boîte aux lettres
  - Destruction de la boîte au lettres

# Communication indirecte



# Communication indirecte

- **Partage d'une boîte aux lettres**
  - $P_1$ ,  $P_2$ , et  $P_3$  partagent la boîte A
  - $P_1$  envoie;  $P_2$  et  $P_3$  reçoivent
  - Qui reçoit les messages?
- **Solutions**
  - Liaison réservée entre deux processus seulement
  - Permettre un seul processus l'exécution de l'opération *receive()*.
  - Laisser le SE arbitrairement déterminer le récepteur. L'émetteur est avisé de l'identité du récepteur.

# Échange synchrone de messages

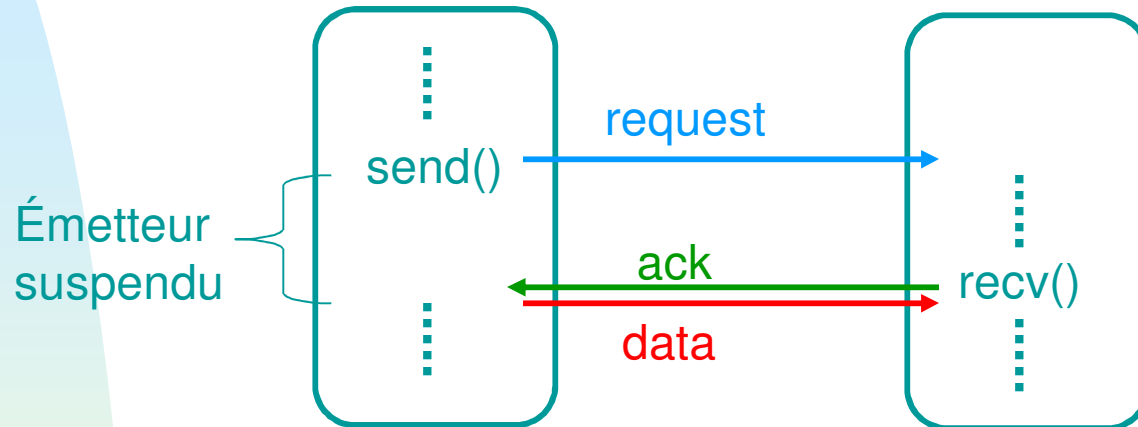
**Communication synchrone entre processus  
(blocked message passing)**

- **L'émetteur attend que le récepteur reçoit le message**
- **Le récepteur attend que l'émetteur envoie un message.**
- **Avantages:**
  - **Synchronise le/les émetteurs avec le/les récepteurs**
  - **Une copie simple est suffisant**
- **Désavantages**
  - **Impasse (deadlock) possible**

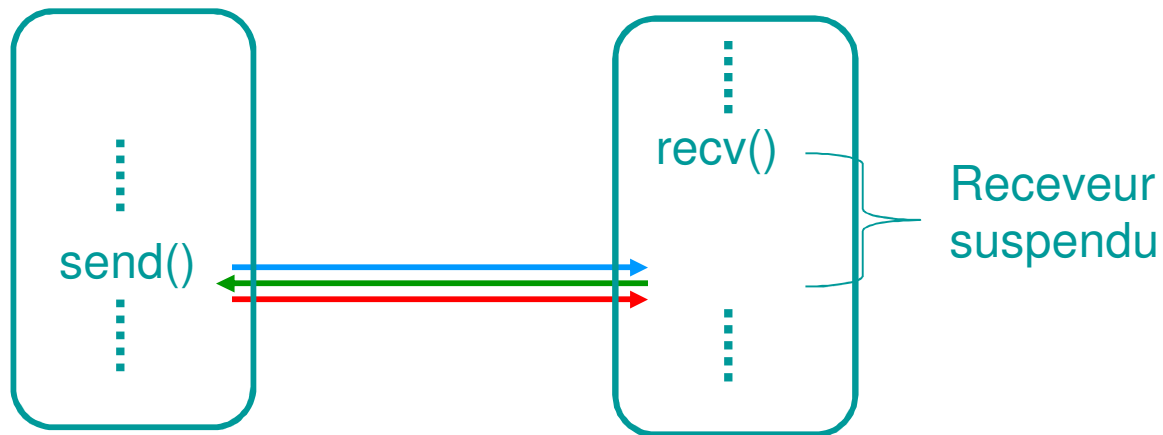


# Échange synchrone de messages

**send()** avant le **receive()** correspondant



**receive()** avant le **send()** correspondant



# Échange asynchrone de message (Non-Blocking Message Passing)

- Émission sans blocage (Non-blocking send):  
**l'émetteur continue d'envoyer sans attendre qu'un message arrive à destination.**
- Réception sans blocage (Non-blocking receive):  
**vérifie si un message est disponible, sinon, retourne immédiatement.**

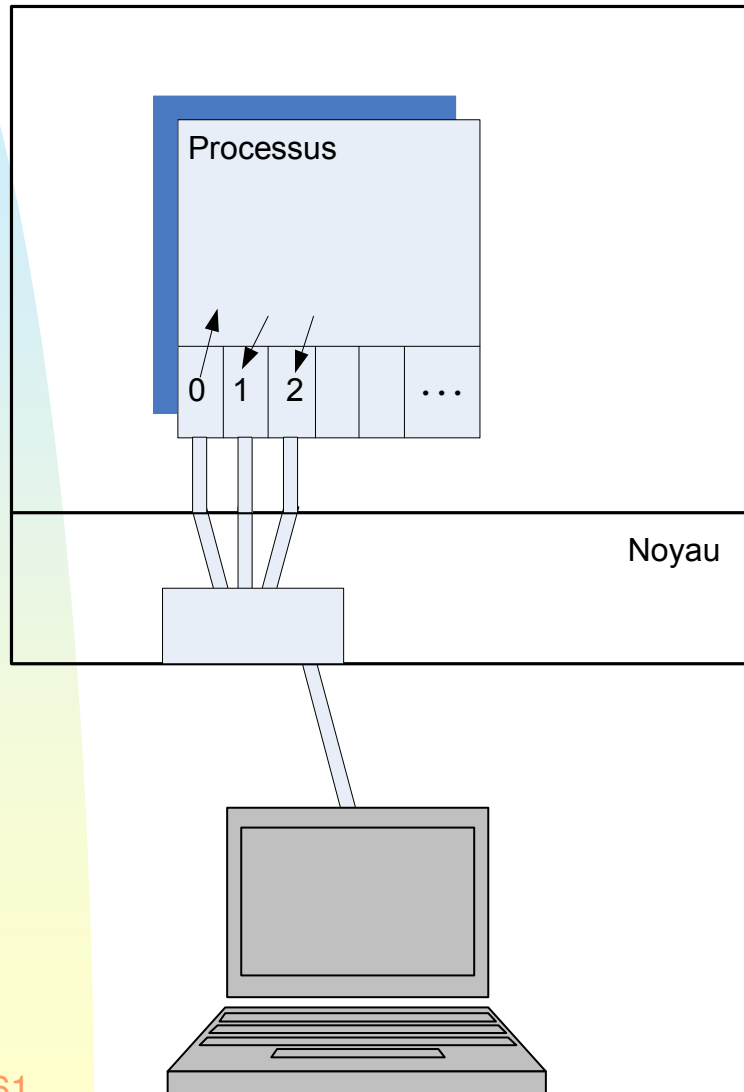
# Les tampons (buffering)

- Avec la communication directe bloquante, aucun tampon n'est requis – le message demeure dans le tampon de l'émetteur jusqu'à ce qu'il soit copié dans le tampon du receveur.
- Avec la communication sans blocage, l'émetteur peut réutiliser le tampon
  - Donc, le message est recopié dans un tampon du SE
  - Et, par la suite, le message est recopié dans le tampon du receveur
  - Si la mémoire tampon du SE est pleine, l'émetteur bloquera toute émission jusqu'à la libération de mémoire.

# Communication client-serveur

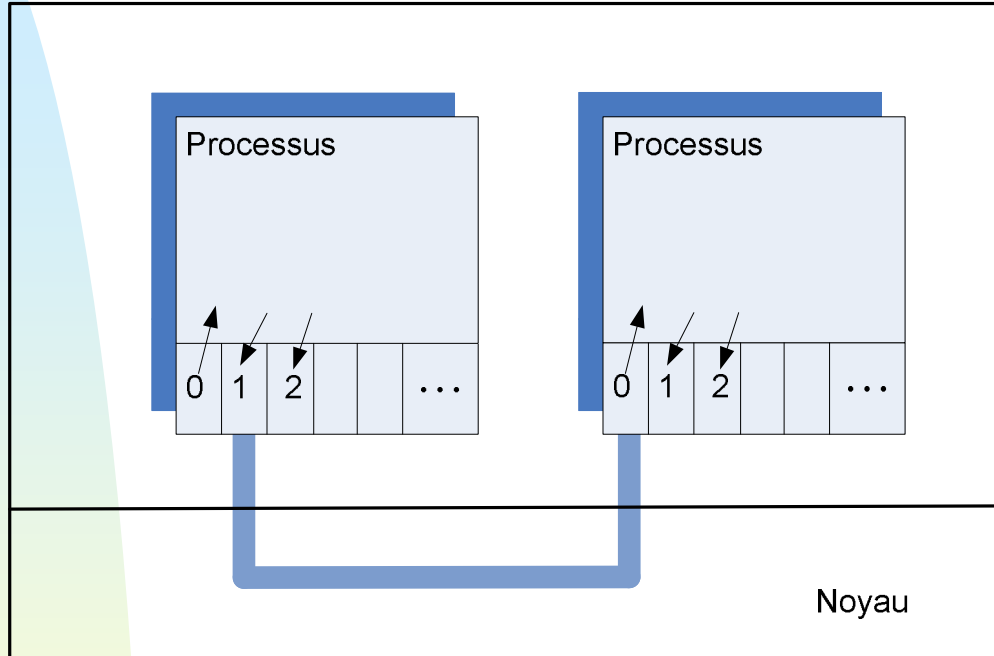
- Tuyaux (pipes)
- Point de connexion (Sockets)
- Appel de procédure à distance (Remote Procedure Calls)
- Appel RMI (Remote Method Invocation - Java)

# Les tuyaux – quelques faits



- Chaque processus **UNIX/Linux** est créé avec trois fichiers ouverts:
  - 0: entrée standard (standard input)
  - 1: sortie standard (standard output)
  - 2: erreur standard (standard error)
- Souvent attaché à un terminal
- Peut rediriger vers un fichier
  - `cmd >fichier`

# Le tuyaux - Communication

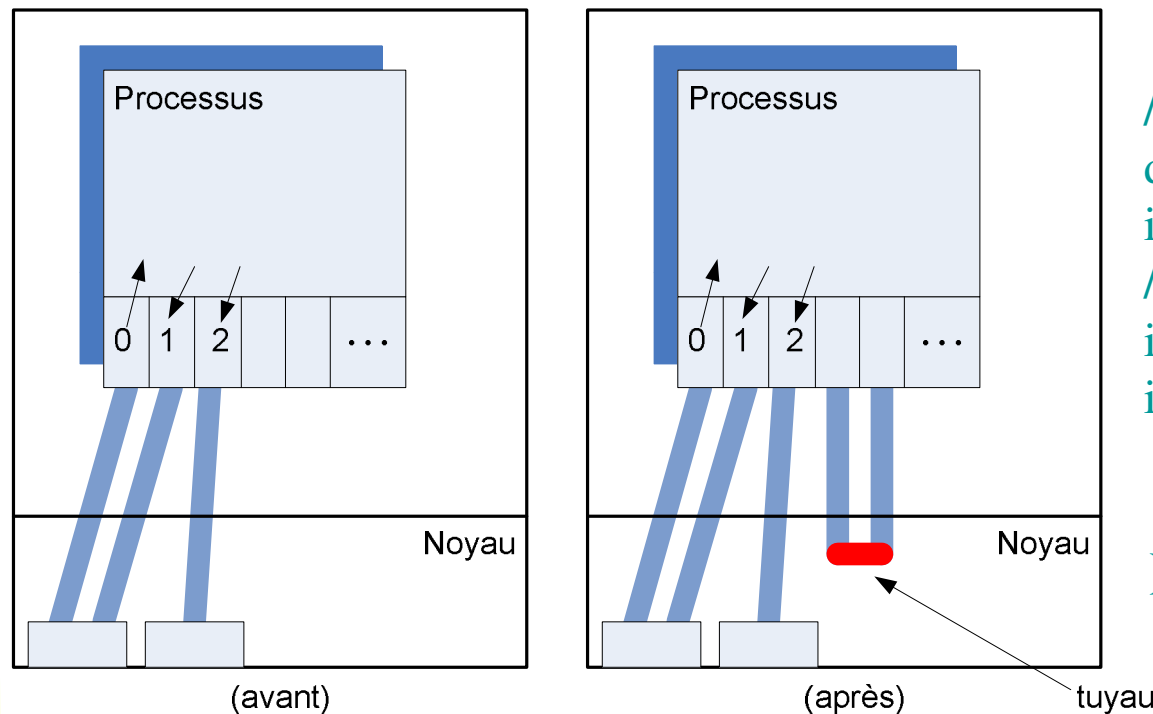


- **C'est quoi?**
  - Canal unidirectionnel
  - Un bout d'écriture
  - Un bout de lecture

- **Pourquoi?**
  - Les tuyaux Unix/Linux permettent de relier des processus pour réaliser des fonctions complexes à partir de commandes simples
  - `who | sort`

# Les tuyaux – comment?

- **Appel système: pipe(int df[2])**
  - Crée un tuyau avec deux descripteurs de fichiers
  - df[0] – référence au bout de lecture
  - df[1] – référence au bout d'écriture



```
/* Tableau pour stocker 2  
descripteurs de fichier */  
int pipes[2];  
/* création du tuyau */  
int ret = pipe(pipes);  
if (ret == -1) { /* erreur */  
    perror("pipe");  
    exit(1);  
}
```

# Programme 1 - Tuyaux

```
int pipes[2], pid, ret;
ret = pipe(pipes);
if (ret == -1) return PIPE_FAILED;
pid = fork();
if (pid == -1) return FORK_FAILED;
if (pid == 0) { /* l'enfant */
    close(pipes[1]);
    while(...) {
        read(pipes[0], ...);
        ...
    }
} else { /* parent */
    close(pipes[0]);
    while(...) {
        write(pipes[1], ...);
        ...
    }
}
```

Créer un vecteur, variable, etc. On crée le pipe, vérifie le résultat. On crée un enfant (fork) et vérifie le résultat. Si c'est 0, c'est l'enfant, sinon c'est le parent. On ne peut faire que la lecture ou l'écriture, donc dans l'exemple de l'enfant on ferme le [0] extrémité d'écriture et ensuite on lit. C'est unidirectionnel.



# Les tuyaux

Comment communiquer dans les 2 sens

- **Utiliser 2 tuyaux**

Défis:

- Chaque tuyau est réalisé avec un tampon de grandeur fixe.
- Si le tampon est plein, l'écriture est bloquée jusqu'à ce que le lecteur lise et ainsi crée de l'espace.
- Si le tampon est vide, la lecture est bloquée en attendant des données.
- Que se passe-t-il si deux processus commencent par lire le tuyau?
  - Une impasse
  - Même chose pour l'écriture
  - Avec les tuyaux on doit faire particulièrement attention aux impasses

# Les tuyaux nommés (named pipes)

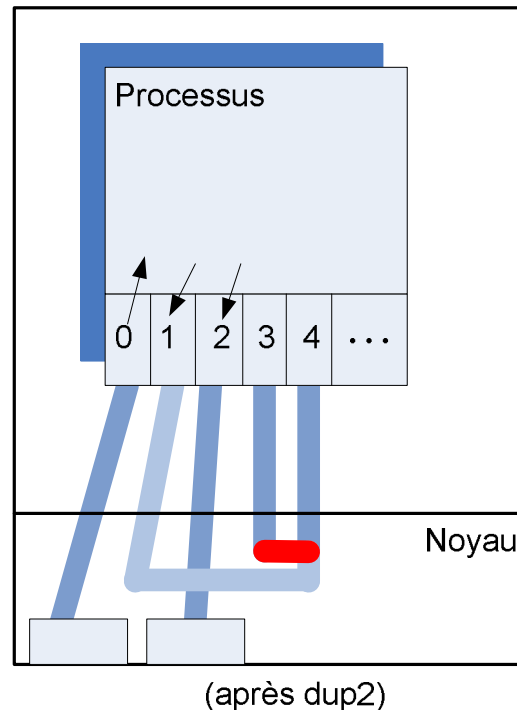
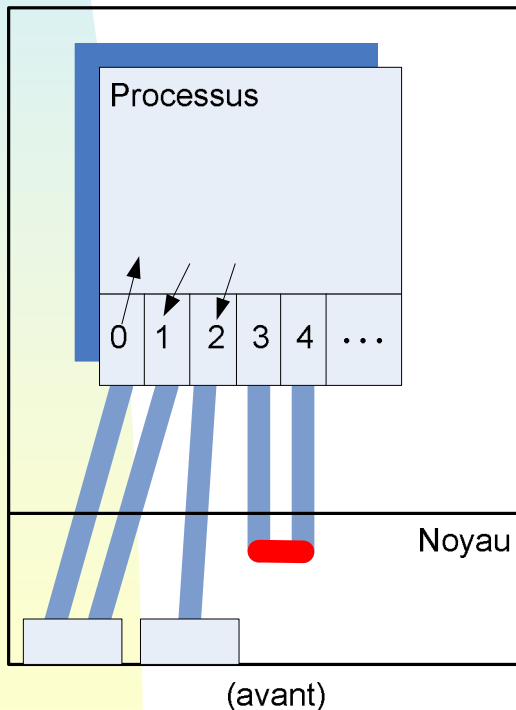
**La communication se fait entre processus reliés (apparentés)**

**Solution: tuyaux nommés**

- **La création d'un tuyau nommé, avec l'appel de système *mkfifo()*, génère un fichier de même nom dans le système de fichiers de tuyaux.**
- **L'ouverture de ce fichier pour la lecture donne accès au bout de la lecture du tuyau**
- **L'ouverture de ce fichier pour l'écriture donne accès au bout d'écriture du tuyau**

# Les tuyaux – attacher à 0 et 1

- **Appel système: `dup2(dfSrc,dfDst)`**
  - Créer une copie de `dfSrc` à `dfDst`
  - Si `dfDst` est ouvert, il est d'abord fermé
  - Pour attacher un tuyau à la sortie standard, `dup2(df[1],1)`

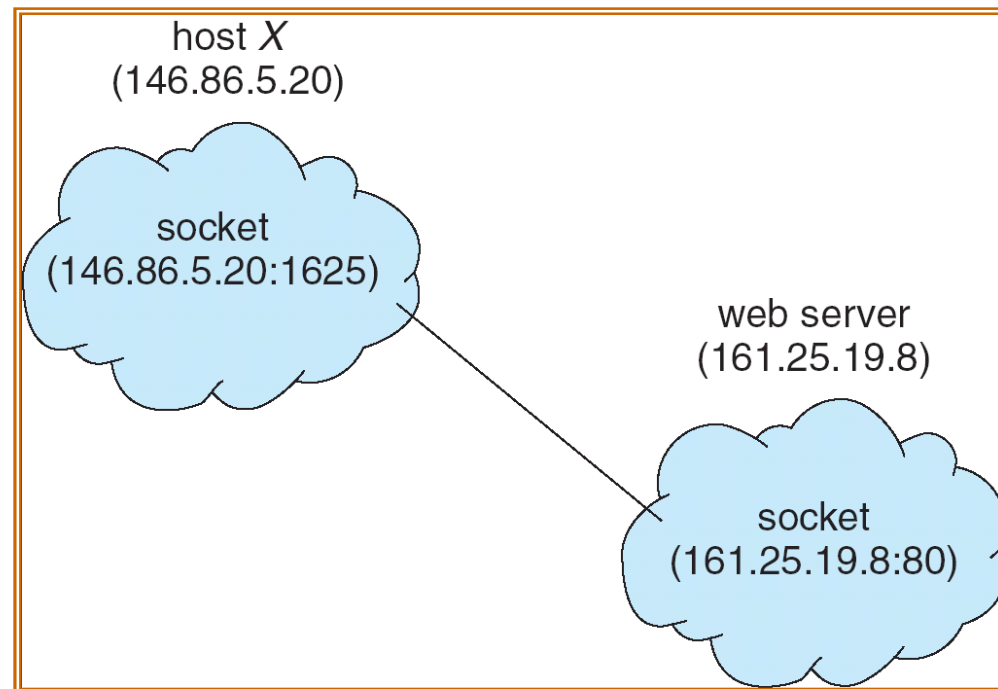


- **À noter**
  - Df 4 se réfère toujours au tuyau
  - Il est possible d'avoir plusieurs références aux bouts des tuyaux, y compris de ceux d'autres processus


# Point de communication (socket)

Concatenation de l'adresse avec le numero de port. Adresse specify un systeme specifique et le port est le port auquel on va arriver.

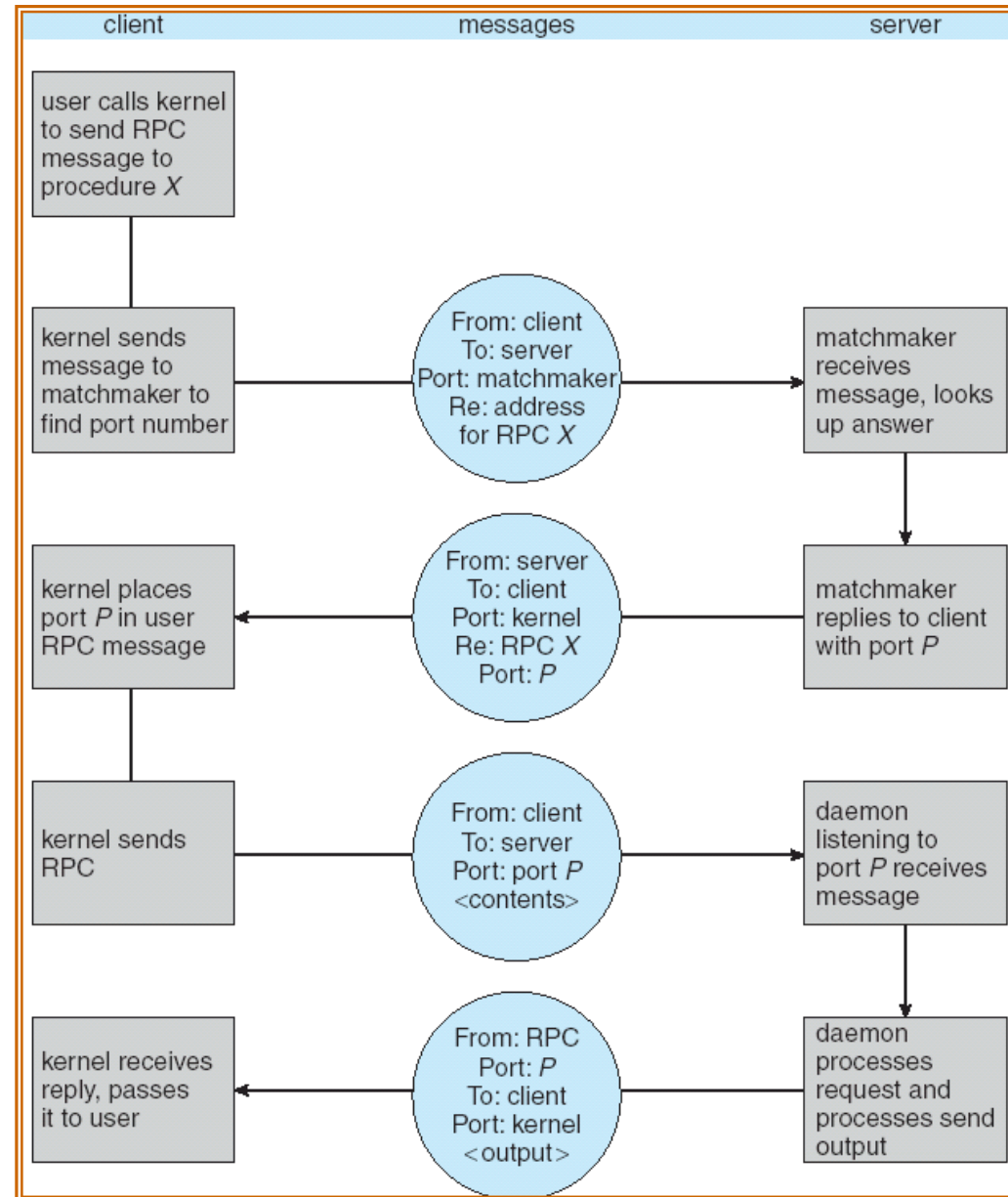
- **(endpoint for communication)**
- **Définie par la concaténation de l'adresse IP et le port TCP**
- **Le “socket” 161.25.19.8:1625 réfère au port 1625 sur l’hôte à l’adresse 161.25.19.8**
- **Une liaison de communication est définie par les deux points de communications de chaque bout.**



# Appel de procédure à distance (Remote Procedure Calls – RPC)

- Est-il possible de faire des appels vers d'autres ordinateurs?
- Doit donc échanger des paramètres et des valeurs de retour (possiblement plusieurs données) entre ordinateurs
  - Problème – des ordinateurs différents souvent ont des formats de représentation de données différents (big endian versus little endian).
  - Solution – représentation externe de données
- Support pour localiser le serveur/procédure désiré
- Élément de remplacement (Stub): cache les détails de communication entre client et serveur
- Chez le client le « stub »:
  - Localise le serveur
  - Convertit les paramètres (marshalling)  parsing
- Chez le serveur, le « stub »:
  - Reçoit le message et extrait les paramètres convertis
  - Exécute la procédure sur le serveur
- Utilise des ports pour identifier les serveurs qui peuvent répondre aux divers appels – ex. système de fichier (NFS de SUN)

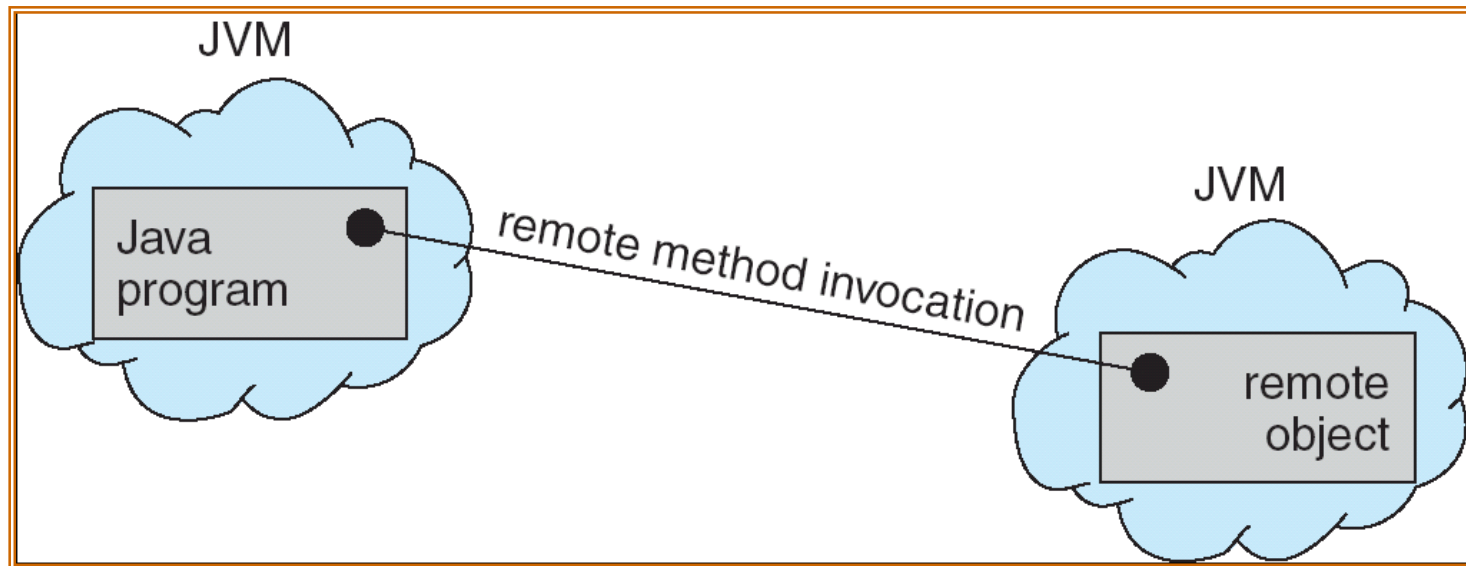
# Exécution du RPC (démon rendez-vous)



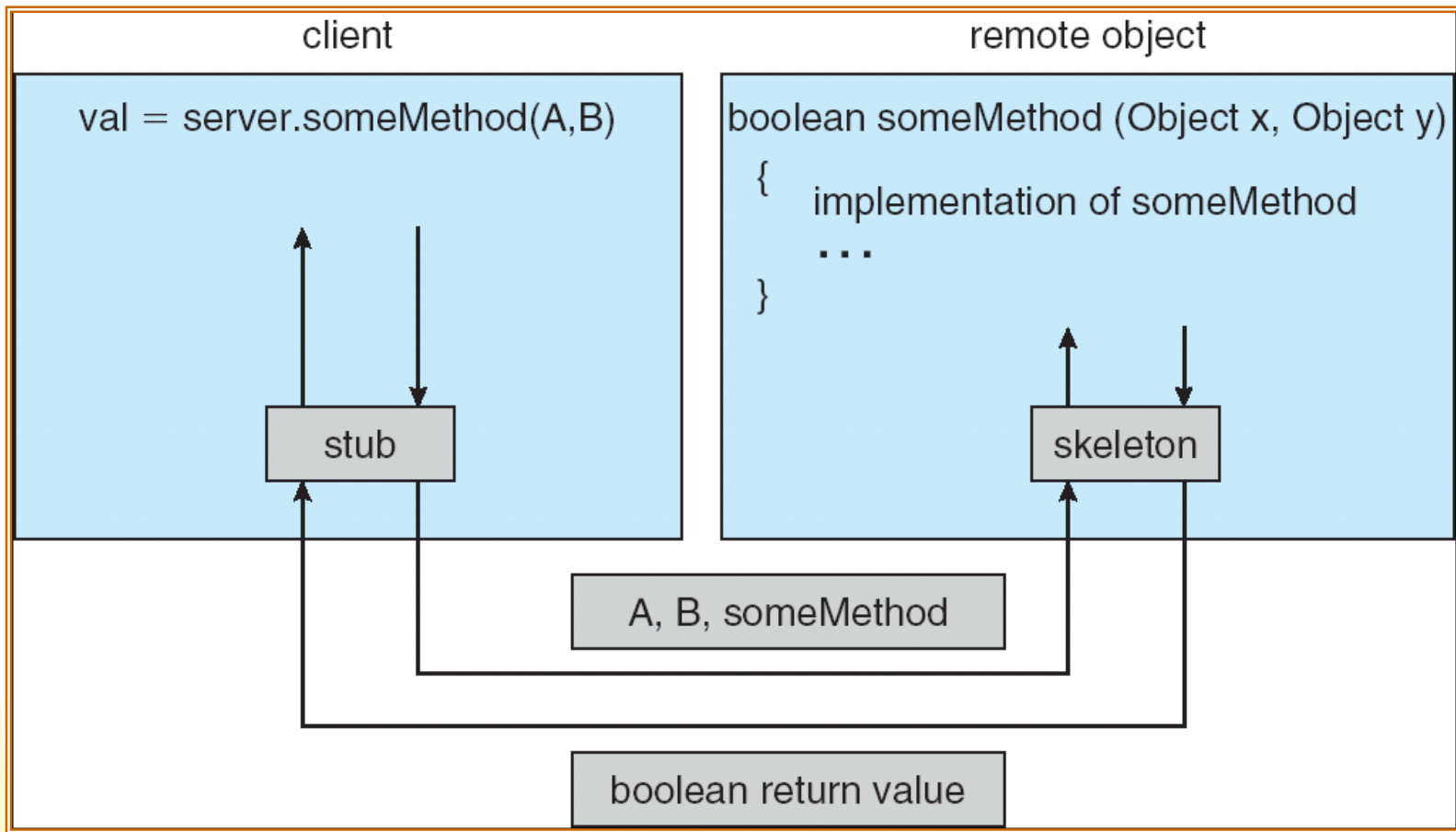
# Appel RMI de Java (Remote Method Invocation)

Library qui contient déjà des méthodes pour tout faire.

- **Le RMI de Java est similaire aux RPCs avec deux différences:**
  - RPC – programmation procédurale, RMI – est objet orienté
  - Donc le RMI, exécute des méthodes d'objets à distance et envoie des objets comme paramètres



# Conversion de paramètres



- **Paramètres locales:** copie à l'aide de la technique de la sérialisation d'objets (object serialization)
- **Paramètres distants:** utilise la référence



# Concepts importants du Chapitre 2

- **Processus**
  - Création, terminaison, hiérarchie
- **États et transitions d'état des processus**
- **Bloc de contrôle de processus: PCB**
- **Commutation/permutation de processus**
  - Sauvegarde, rechargement de PCB
- **Files d'attente de processus et PCB**
- **Ordonnanceurs à court, moyen et long terme**
- **Processus communicants**
  - Communication IPC
  - Tuyaux, Sockets, RPC, RMI