# Processus

Le concept de processus

Ordonnancement de processus

Opérations sur les processus

Communication entre processus

Communication client-serveur

# Le concept de processus

Un SE exécutes divers programmes

Un *processus* est un programme en cours d'exécution

S'appelle aussi *tâche* ou *job*

Un *programme* est une entité passive (e.g. sur le disque)

Un *processus* est une entité active, avec un *état*

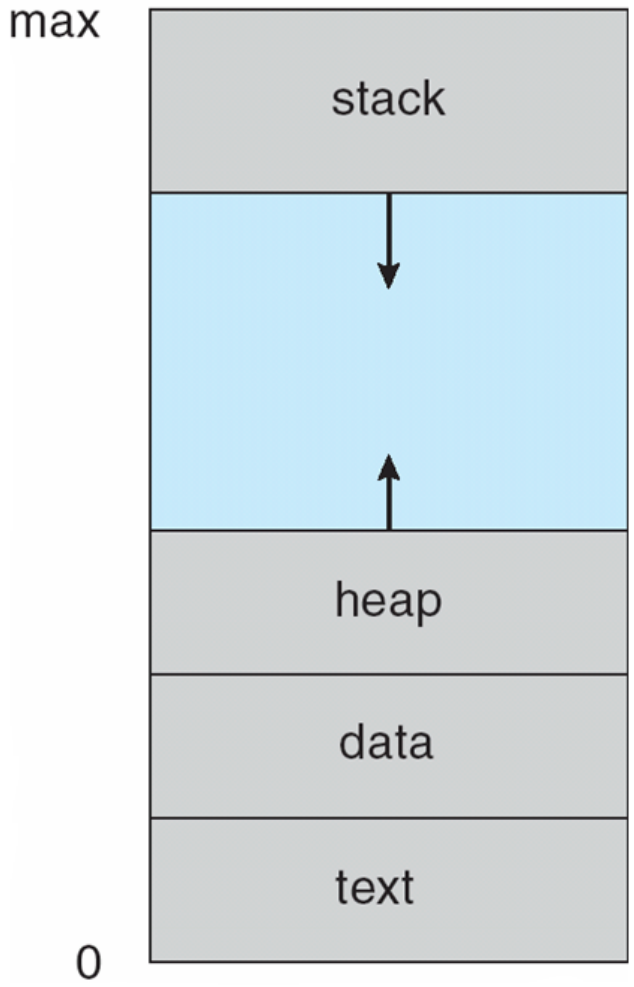
Un programme peut avoir plusieurs processus

# Éléments de processus

Un processus a plusieurs parties

- Le code du programme à exécuter (*text section*)
- Les *registres* (y compris le *program counter*)
- La *pile* (*stack*), qui contient des données temporaires
- La *data section* qui contient les variables globales
- Le *tas* (*heap*) qui contient les données allouée dynamiquement

# La mémoire d'un processus

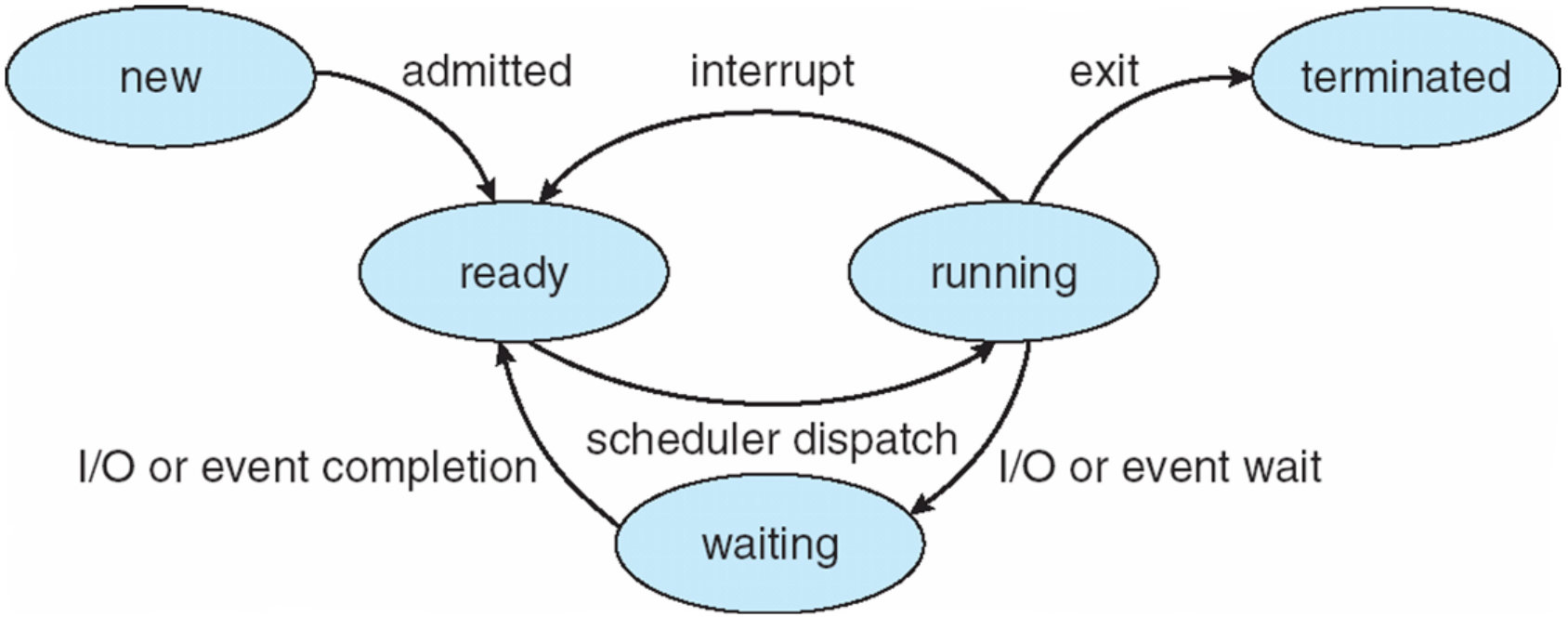


# États d'un processus

Au cours de sa vie le processus passe par plusieurs états

- *New*: processus en cours de création
- *Running*: processus en cours d'exécution
- *Waiting*: En attente d'un événement
- *Ready*: Prêt à l'exécution, en attente d'un processeur
- *Terminated*: Le processus a fini son exécution

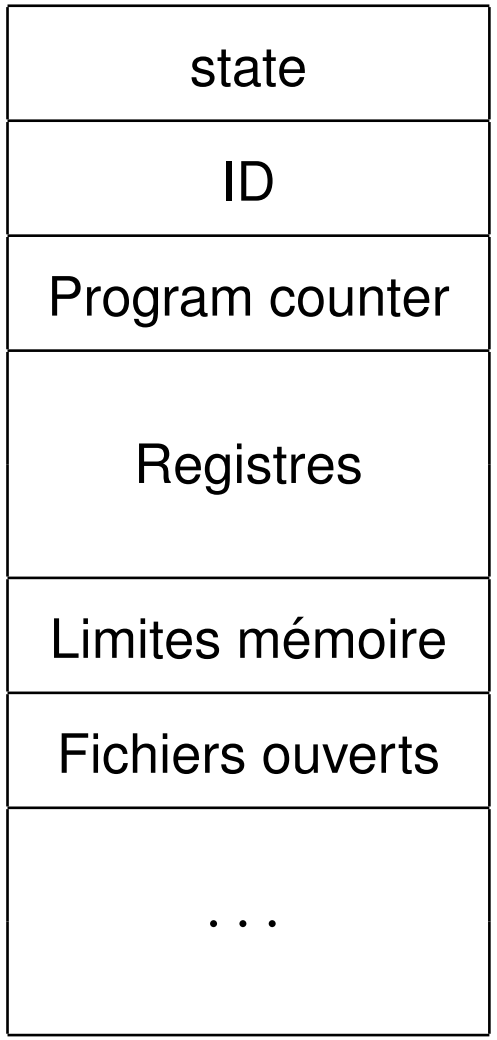
# Diagramme des états



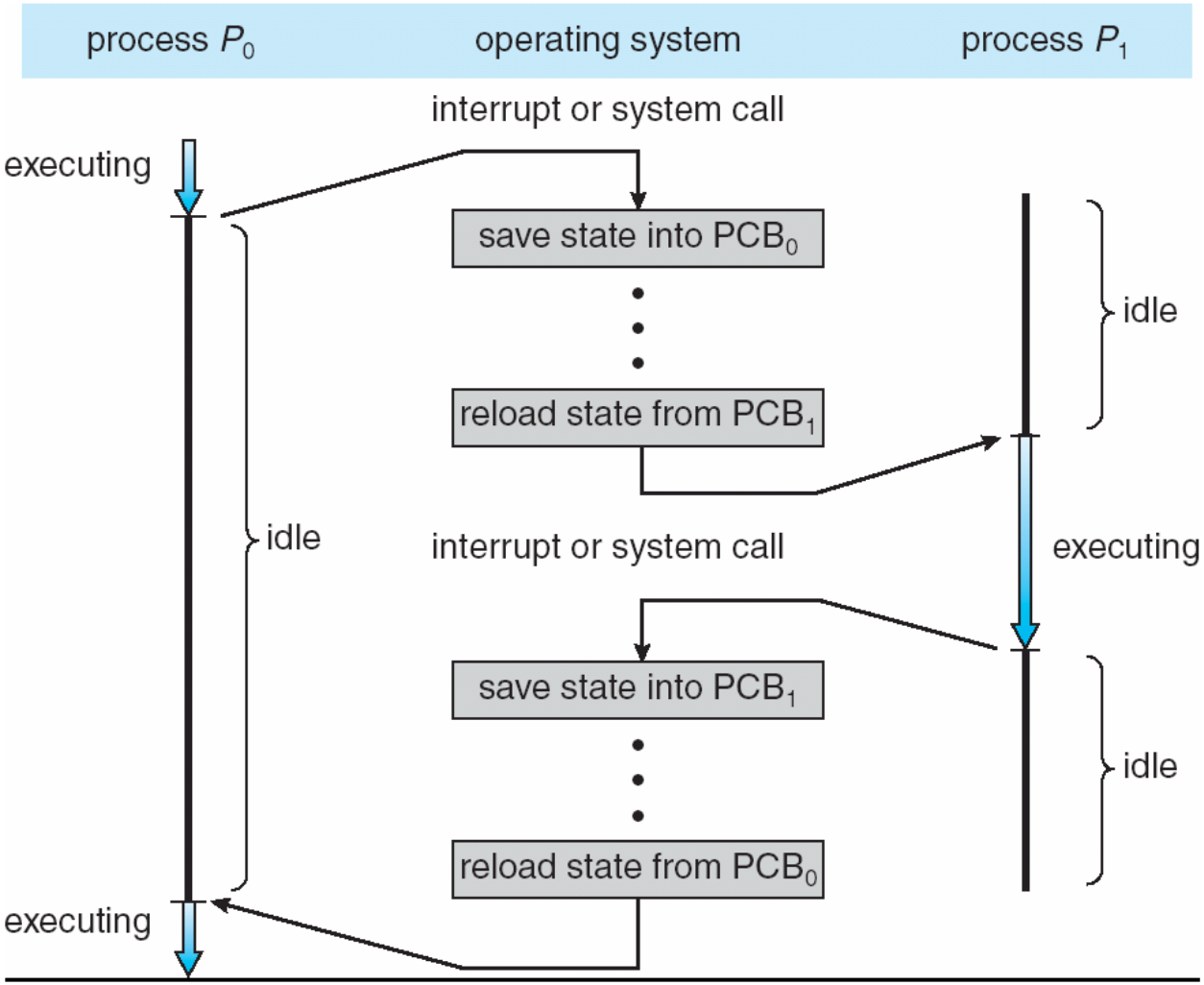
# Bloc de contrôle de processus (PCB)

Descripteur de l'état détaillé du processus

- *state*: *running*, *waiting*, ...
- *ID*: Identifiant, typiquement un nombre
- Filiation: parent, enfants
- État du processeur: registres, PC
- ordonnancement: priorité, queue d'attente
- Resources mémoires utilisées
- *Accounting*: Resources déjà utilisées
- E/S: fichiers ouverts, périphériques associés



# Changement de contexte



# Threads

Un *thread* décrit l'exécution d'une séquence d'instructions

Autres noms: *fil d'exécution* ou encore *processus léger*

Descripteur d'un thread: ID, PC, registres, et pile

Certains SE permettent plusieurs *threads* par *processus*

Chaque thread d'un processus peut s'ordonnancer indépendemment

Exécuter "fonctions" en même temps dans le même espace mémoire

Le PCB contient alors une liste de descripteurs de threads

# Ordonnancement

Multiprogrammation pour maximiser l'usage du CPU

Time-sharing veux rapidement donner un CPU à un processus prêt

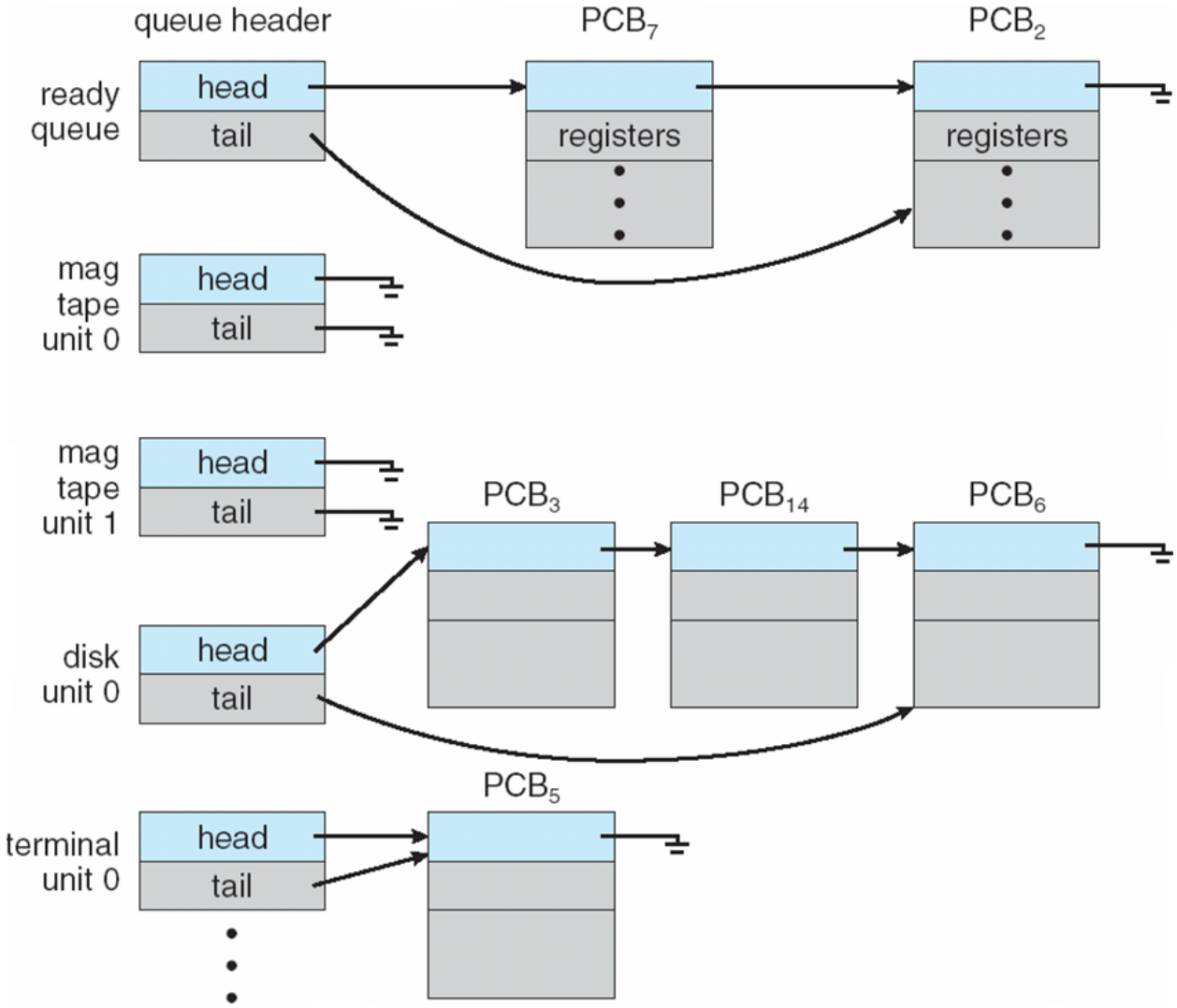
I.e.: minimiser le *temps de réponse*, maximiser le *throughput*

*Ordonnanceur* choisi quand exécuter quel thread sur quel CPU

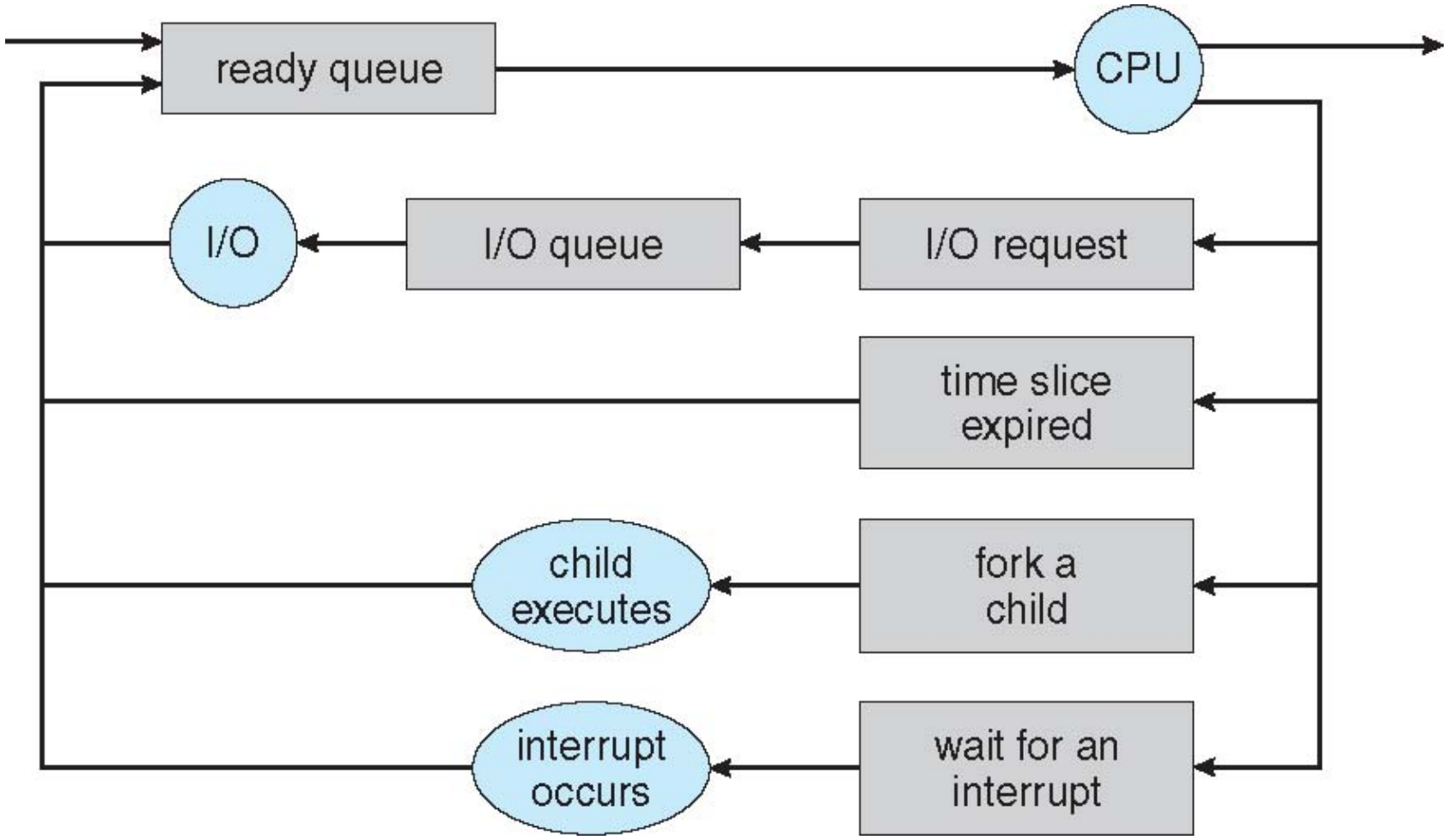
Utilises des *queues*:

- Queue de tous les threads
- Queue des threads qui sont prêts
- Queue des threads en attente

# Queues de processus



# Ordonnanceur



# Ordonnanceurs

Souvent l'ordonnanceur se divise en deux parties

*Ordonnanceur à long-terme* choisi quels threads ont le droit d'avancer

*Ordonnanceur à court-terme* les répartis sur les CPUs

Court-terme: invoqué très fréquemment, doit être très rapide

Processus généralement classés dans 2 catégories:

- *CPU-bound*: beaucoup de calculs, peut d'E/S
- *I/O-bound*: peu de calculs, beaucoup d'E/S

# Multitâche et GUI

L'ordonnanceur ne voit pas l'écran

Le GUI peut dire au noyau quelle(s) tâche(s) est *en avant* (*foreground*)

De même il peut ralentir/stopper les *tâches de fond* (*background*)

Pas forcément nécessaire

On ne vout pas stopper *toutes* les tâches de fonds

# Changement de contexte (le retour)

Lors d'un changement de contexte, il faut

- sauvegarder l'état du processus sortant
- restorer l'état du processus entrant
- ...plus les caches

Temps perdu et source d'inefficacité

L'ordonnanceur doit aussi minimiser les changements de contexte

# Création de processus

Un *parent* crée des *enfants*: un *arbre* de processus

Le nouveau processus obtient un nouvel *ID*

Choix d'exécution, selon que le parent attend ou pas

Choix de quelles resources partager entre l'enfant et son parent

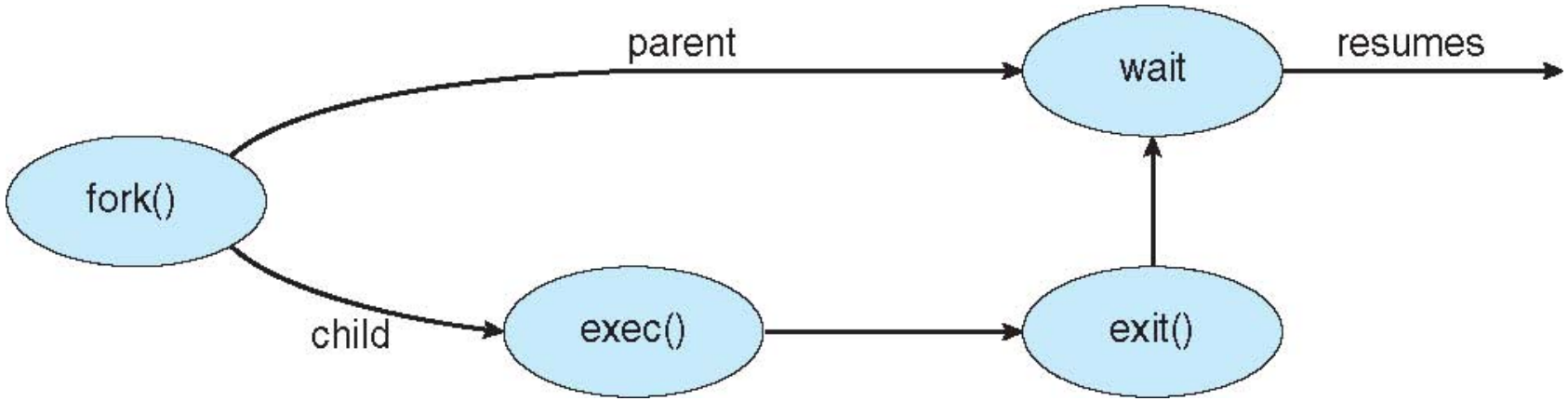
- Fichiers ouverts
- Mémoire allouée

# Création en POSIX

*fork* crée un nouveau processus

- Partage les fichiers ouverts, reçoit une *copie* de la mémoire

*exec* modifie le processus en changeant le programme



# Création en POSIX: Exemple

```
pid_t pid = fork();
if (pid < 0) {
    printf (stderr, "Help!!\n");
} else if (pid == 0) {
    execlp ("/bin/ls", "ls", NULL);
} else {
    waitpid (pid);
    printf ("Done!\n");
}
```

# Fin d'un processus

Le processus peut se terminer en suicide (appel système *exit*)

Ou on peut le tuer (infanticide, ou entre amis) avec *kill*

Ou il peut être terminé par le système en cas d'erreur

Le parent est averti pour constater le décès

Entre temps là, le processus terminé est appelé *zombie*

# Communication entre processus

Utilisée lorsque le travail est divisé entre plusieurs processus

- Pour profiter du parallélisme
- Pour des raisons de modularité
- Pour des raisons de sécurité
- Efficacité du partage d'information

Deux grandes catégories

- *Passage de messages*: synchronisation implicite
- *Mémoire partagée*: communication implicite

# Problème producteur-consommateur

Exemple classique: un processus génère séquentiellement des données utilisée par un autre

Le consommateur doit attendre que le *buffer* se remplisse

Deux modèles:

- *unbounded-buffer*: le producteur n'a jamais besoin d'attendre
- *bounded-buffer*: le producteur doit attendre si le buffer est plein

# Bounded-buffer, mémoire partagée naïve

Le *buffer* est un tableau en mémoire partagée, avec deux compteur:

- *in*: index où insérer le prochain élément
- *out*: index où trouver le prochain élément

```
while (((in + 1) % BUFFER_SIZE) == out)
   /*Wait*/;
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
```

```
while (in == out) /*Wait*/;
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

# Message passing

Permet aux processus de communiquer et se synchroniser

Pas de partage de variables

Une fois établie la connection:

- *send* pour envoyer un message
- *receive* pour lire les messages reçu

Flot peut être une séquence de bytes ou sequence de paquets

Diverses techniques d'implantation (souvent, par mémoire partagée)

# Messages (a)synchrones et (non-)bloquant

L'échange de message est *synchrone* quand le message n'est pas considéré comme envoyé tant qu'il n'a pas été reçu

Une primitive est *bloquante* si elle oblige le processus à attendre

- Un *receive* non-bloquant renvoie NULL s'il n'y rien
- Un *send* asynchrone peut bloquer si le buffer est plein

# POSIX shared memory

Système basé sur la partage de fichier

Ouverture, comme celle d'un fichier:

```
shm_fd = shm_open (name, O_CREAT|O_RDWR, 0666);
```

La partage vient de l'ouverture simultanée dans plusieurs processus

Taille fixée: `ftruncate (shm_fd, size);`

Accès en mémoire:

```
base = mmap (NULL, size, PROT_READ|PROT_WRITE,
             MAP_SHARED, shm_fd, 0);
```

# Pipes

Pipes ordinaires: communication style producteur-consommateur

Producteur écrit d'un côté

Consommateur lit de l'autre

Communication unidirectionnelle

Ne fonctionnent que localement

Anonymes: utilisables seulement entre processus liés

# IPC par messages dans Mach

Mach est un (gros) micro-noyau. Tout se fait par messages

Chaque tâche a 2 *mailboxes* prédéfinies: *Kernel* et *Notify*

Trois primitives: `msg_send`, `msg_receive`, `msg_rcp`

Nouvelles mailboxes: `port_allocate`

Diverses options si la mailbox est pleine

# Sockets

Système client-serveur

Un *socket* est le “bout” (*endpoint*) d’une communication

Chaque socket a une *addresse IP* et un *port*: `bind`

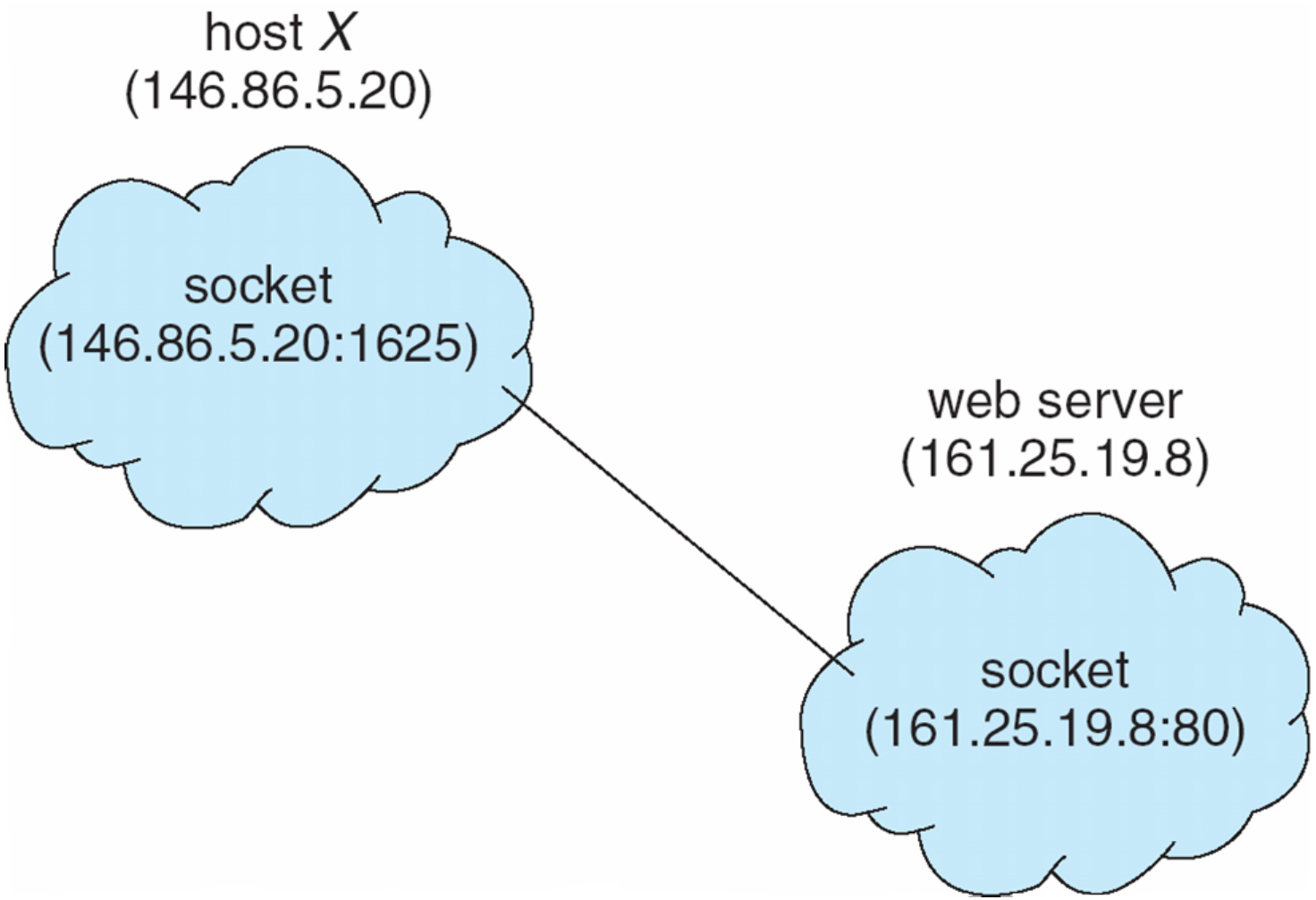
Certains numéros de ports dédiés, standards

La communication a lieu entre deux sockets

Communication établie d’abord: `listen`, `bind`.

Échange bidirectionnel de séquences de bytes

# Communication par sockets



# Appels de procédure distants - RPC

Système client-serveur

*RPC* abstrait la communication derrière une interface de procédure

*Stubs* des deux côtés sont des procédures “proxy”

Le stub client *marshall* les arguments et les envoie

Le stub du serveur les reçoit, les *unmarshall*, et fait l’appel local

La résultat fait le chemin inverse

RPC n’a pas la même fiabilité qu’un appel de fonction

Appels *bloquants* ou non

Stubs générés semi-automatiquement

# Communication par sockets