Les Processus

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Menu

- Introduction aux processus
- Ordonnancement des processus
- Opérations des processus
- Communications entre les processus

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Menu

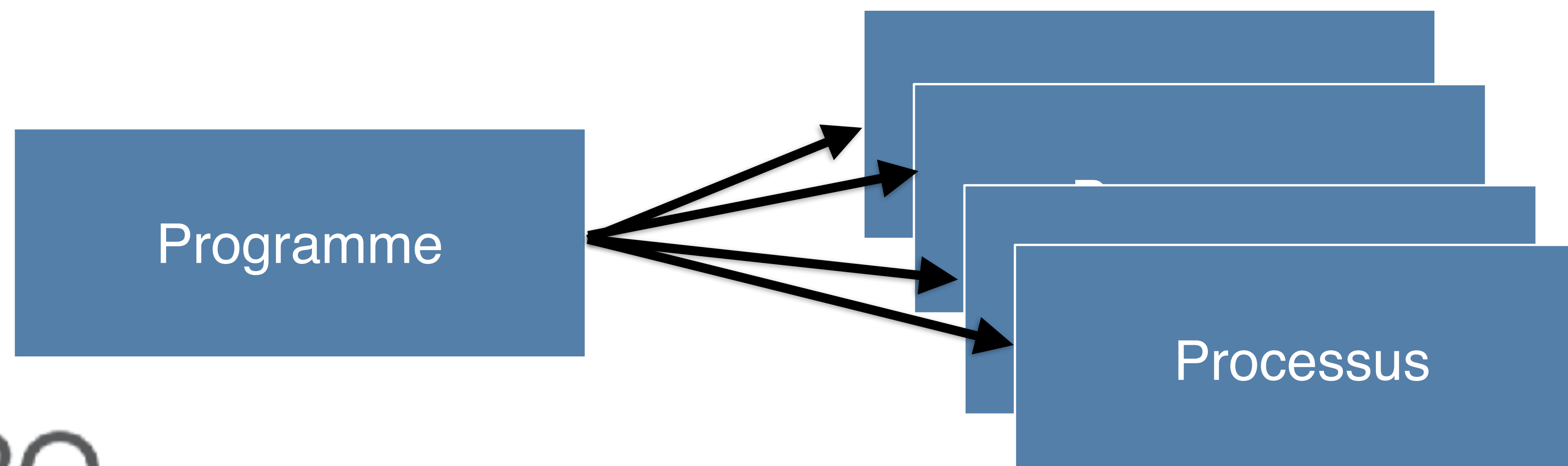
- **Introduction aux processus**
- Ordonnancement des processus
- Opérations des processus
- Communications entre les processus

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Les Processus

- Un SE exécutes divers **programmes**
- Tâche = job = processus
- Processus - un programme en cours **d'exécution**; l'exécution du processus doit progresser de manière **séquentielle**
- Le programme est une entité passive stockée sur le disque (fichier exécutable), le processus est actif
  - Le programme devient un processus lorsque le fichier exécutable est chargé dans la mémoire



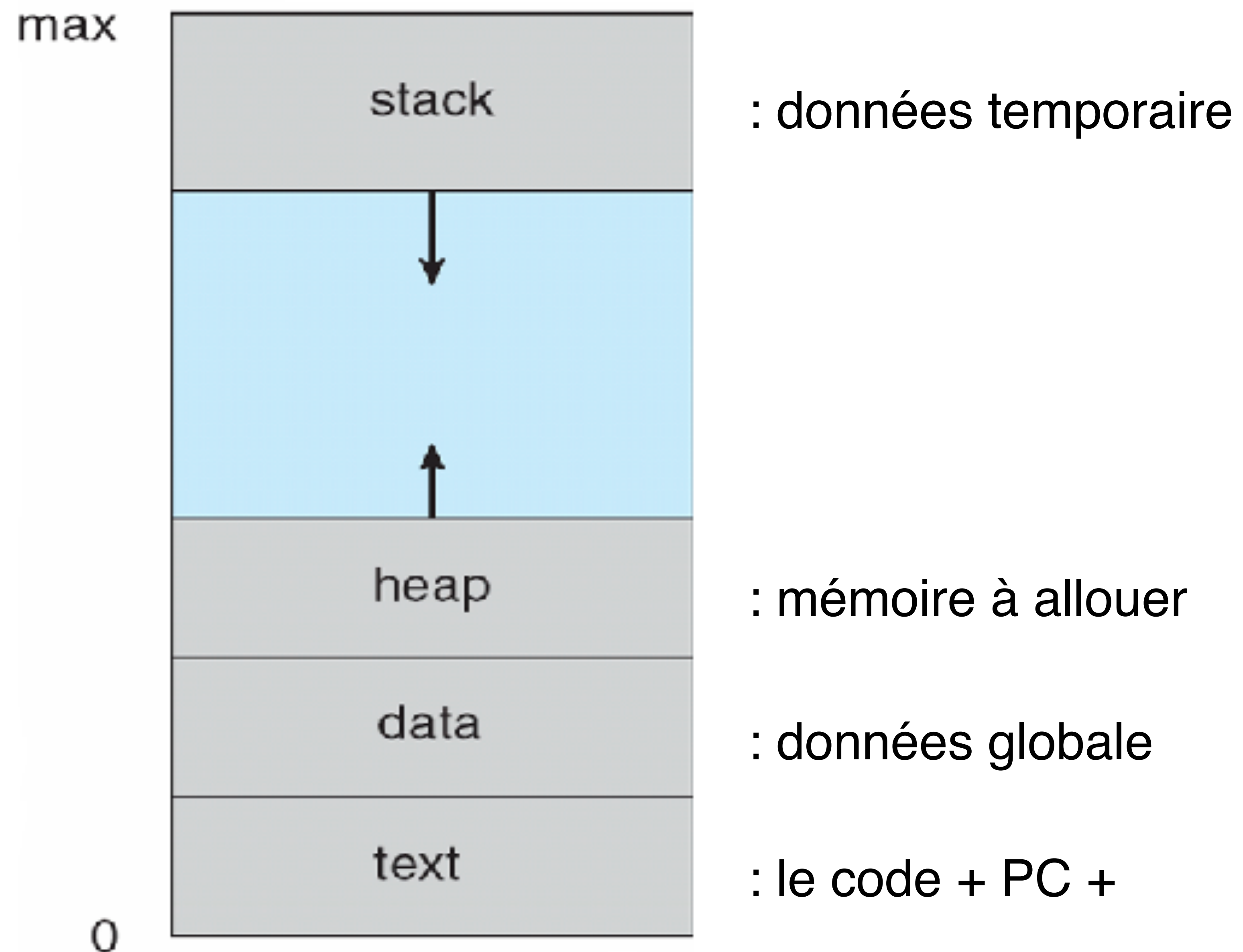
Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Les Parties d’un Processus

- Le code du programme, également appelé **section de texte**
- Activité en cours, y compris le **compteur de programmes**, les registres de processeurs
- **Pile** contenant des données temporaires
  - Paramètres de fonction, adresses de retour, variables locales
- Section de données contenant des **variables globales**
- “**Heap**” contenant de la mémoire allouée dynamiquement pendant l’exécution

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Un Processus dans Mémoire



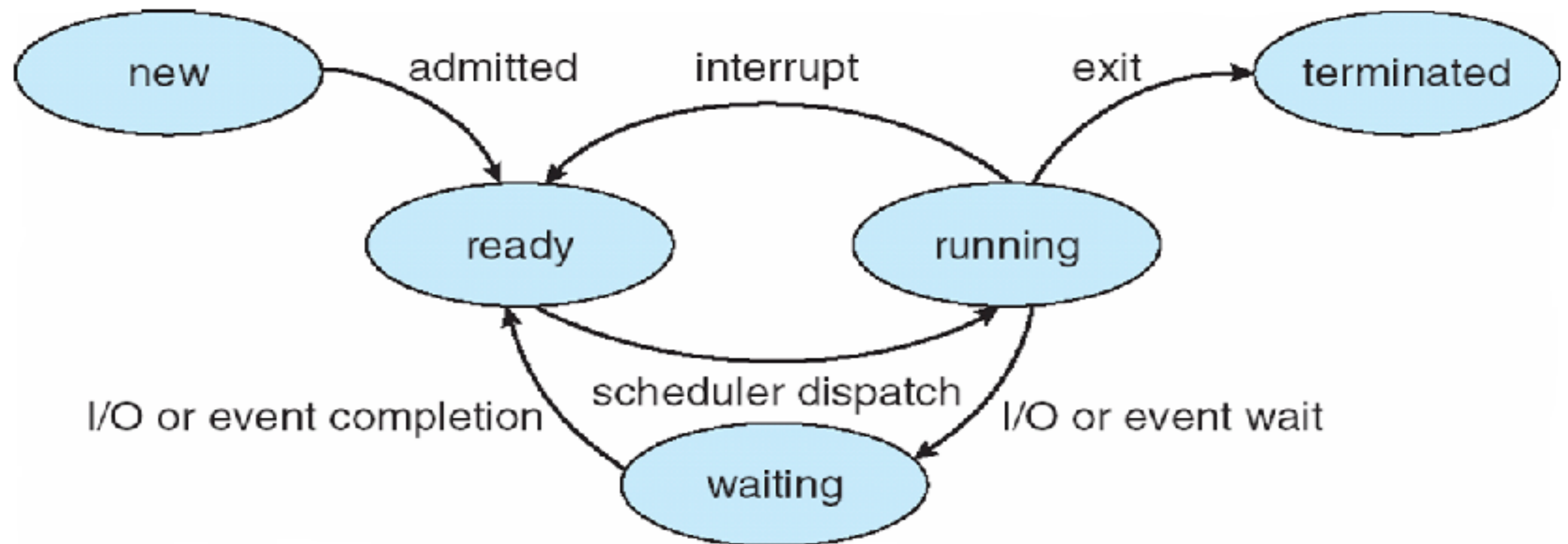
Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

l’État d’un Processus

- **Nouveau**: Le processus est en cours de création
- **En Marche**: Les instructions sont en cours d'exécution
- **Attendre**: Le processus attend qu'un événement se produise
- **Prêt**: Le processus attend d'être affecté à un processeur
- **Fini**: Le processus a terminé l’exécution

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

“FSM” Pour l’Execution des Processus



Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Bloc de Contrôle de Processus (PCB)

- **Etat du processus** - fonctionnement, attente, etc.
- **Compteur de programme** - emplacement de l'instruction à exécuter ensuite
- **Registres du processeur** - contenu de tous les registres centrés sur le processus
- **Informations sur la planification du processeur** - priorités, planification des pointeurs de file d'attente
- **Informations de gestion de la mémoire** - mémoire allouée au processus
- **Informations comptables** - CPU utilisé, temps écoulé depuis le démarrage, limites de temps
- **Informations d'état d'E/S** - Périphériques d'E/S affectés au processus, liste des fichiers ouverts

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Threads

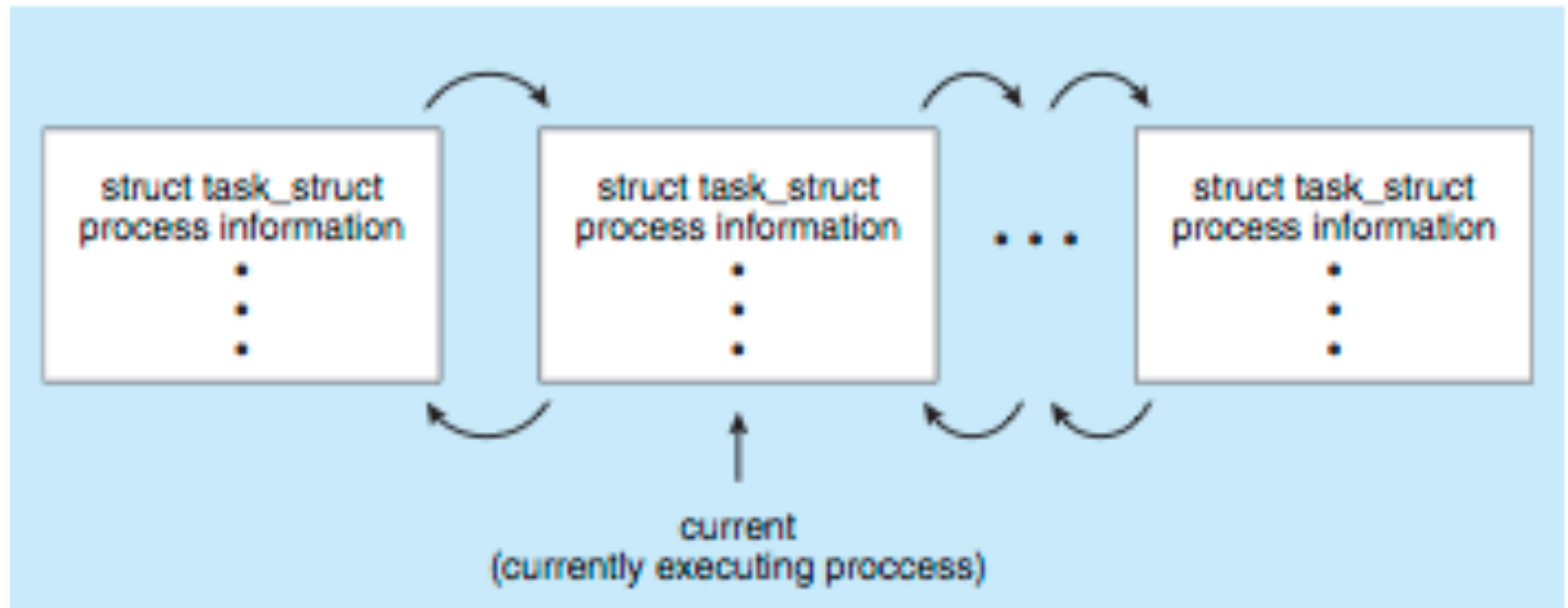
- Nous devons également prendre en compte plusieurs threads dans un processus
- Plusieurs threads = plusieurs compteurs de programme

- À venir …

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Représentation du PCB en Linux

- http://elixir.free-electrons.com/linux/latest/source/include/linux/sched.h



Menu

- Introduction aux processus
- **Ordonnancement des processus**
- Opérations des processus
- Communications entre les processus

DIRO

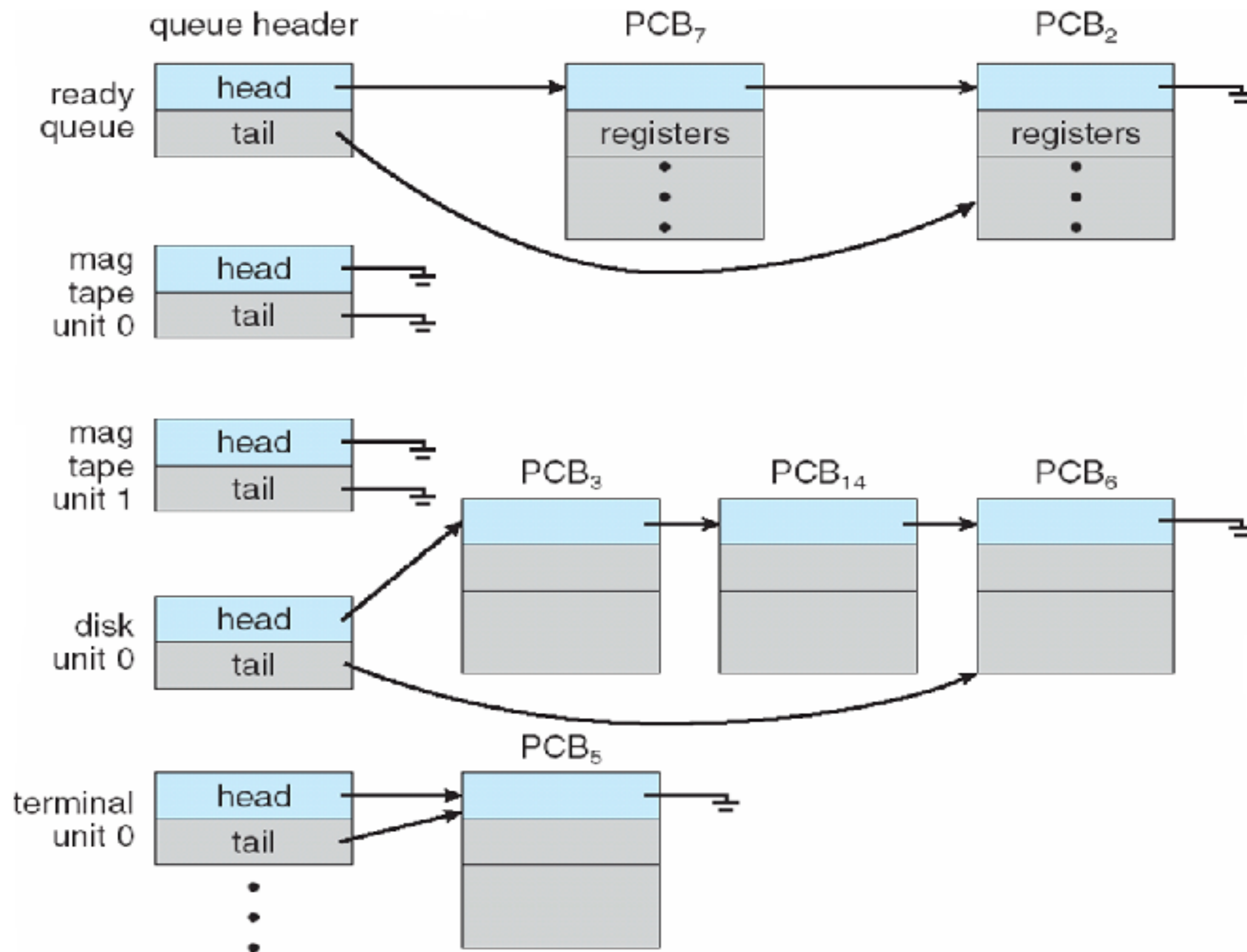
Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Ordonnancement

- Multiprogrammation pour maximiser l'usage du CPU
- Time-sharing veux rapidement donner un CPU à un processus prêt
  - i.e. pour minimiser le **temps de réponse** ou maximiser le **throughput**
- L'ordonnanceur choisi quand exécuter quel processus sur quel CPU
- Utilises des queues:
  - **Job queue** – tous les processus
  - **Ready queue** – les processus qui sont prêts (dans la mémoire centrale et attend l'execution)
  - **Device queues** – les processus qui attend un périphérique

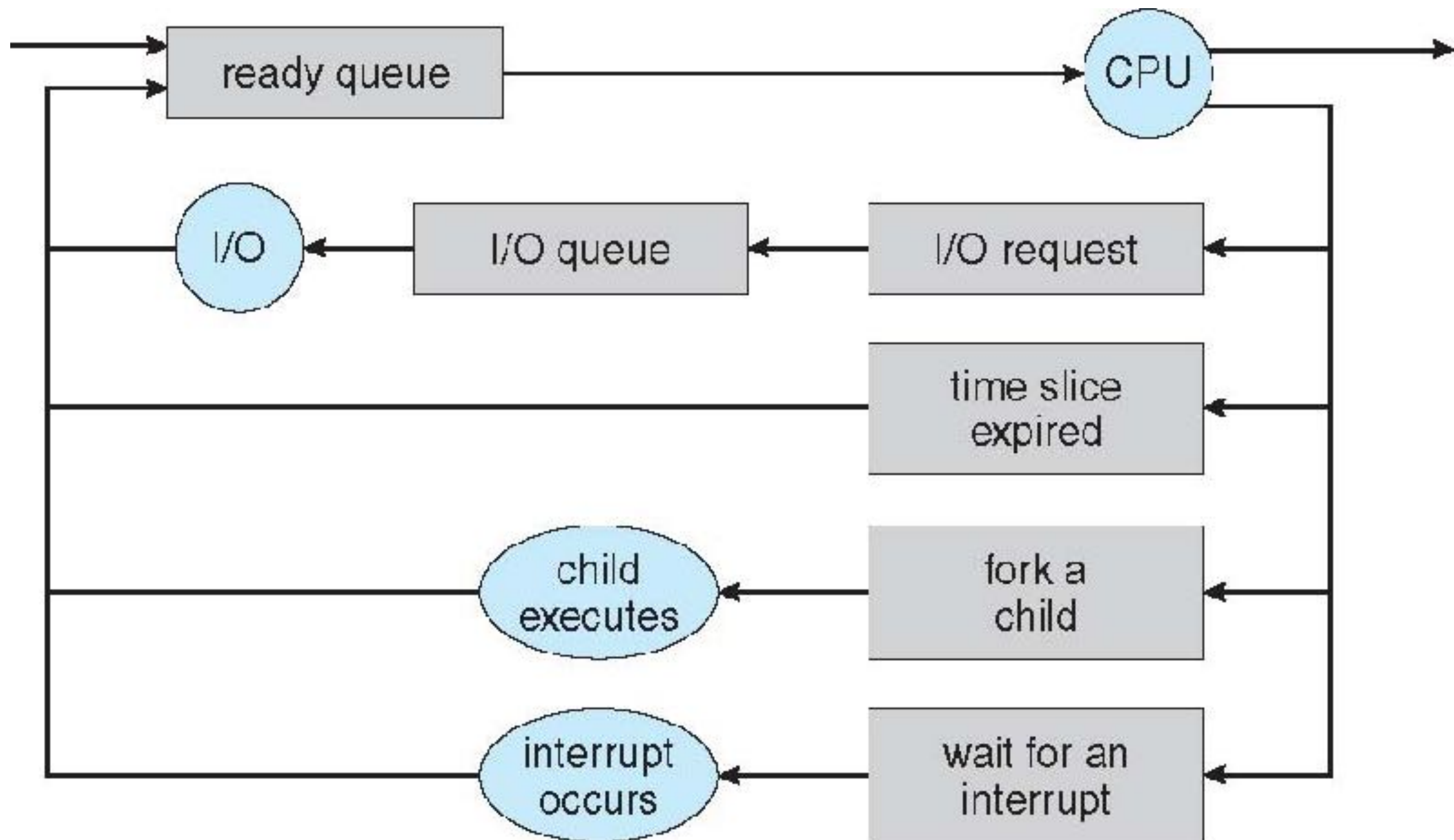
Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Queues de Processus



Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Ordonnanceur des Processus



Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Types d'Ordonnanceurs

- Ordonnanceur à long-terme:
  - Sélectionne quels processus doivent être placés dans la file d'attente prête
  - Est invoqué très rarement (secondes, minutes) ⇒ (peut être lent)
  - Contrôle le degré de multiprogrammation (nombre de processus en mémoire)
- Ordonnanceur à court-terme:
  - Sélectionne le processus qui doit être exécuté ensuite et alloue le processeur
  - Parfois, le seul planificateur dans un système
  - Est invoqué très fréquemment (en millisecondes) ⇒ (doit être rapide)

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Types de Processus

- Un Processus “I/O-bound”: passe plus de temps à faire des E / S qu'à des calculs, beaucoup de courtes rafales du CPU
- Un Processus “CPU-bound”: passe plus de temps à faire des calculs; quelques rafales du CPU très longues

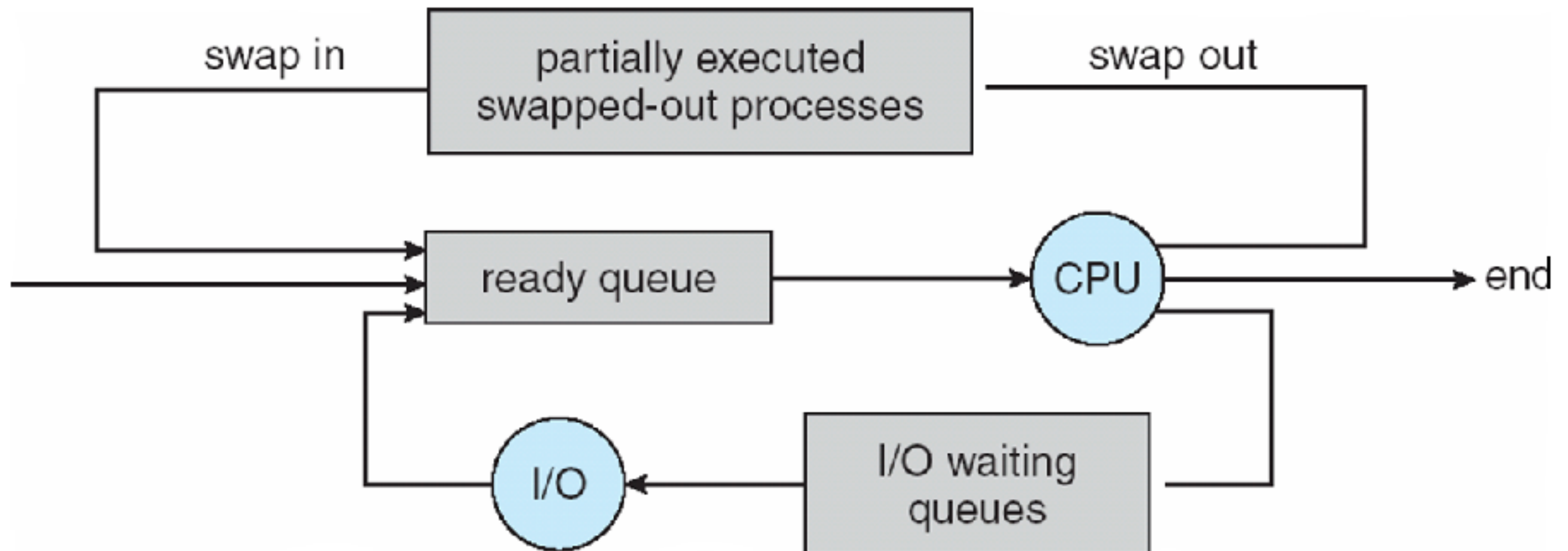
- Ordonnanceur à long-terme essaye d’avoir un balance entre les deux

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Un troisième type d'Ordonnanceur

- Ordonnanceur à terme medium: Contrôle la degree de multiprogrammation par “swapping”
  - Détails plus tard



Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Multitâche dans les Systèmes Mobiles

- En raison de l'écran de l'immobilier, l'interface utilisateur limite iOS pour fournir:
  - Un seul processus “foreground” – contrôler par le GUI (typiquement affiché à l'écran)
  - Beaucoup de processus “background” - dans mémoire, en cours, mais ont des limites
    - ✓ Les limites incluent une tâche unique et courte, (e.g. la réception des notifications d’événements), et des tâches spécifiques de longue durée (e.g. la lecture audio)
- Android s’exécute des process *foreground* et *background* mais avec moins de limites:
  - Le processus en arrière-plan utilise un service pour effectuer des tâches
  - Le service peut continuer à fonctionner même si le processus d'arrière-plan est suspendu
  - Le service n'a pas d'interface utilisateur, une petite utilisation de la mémoire

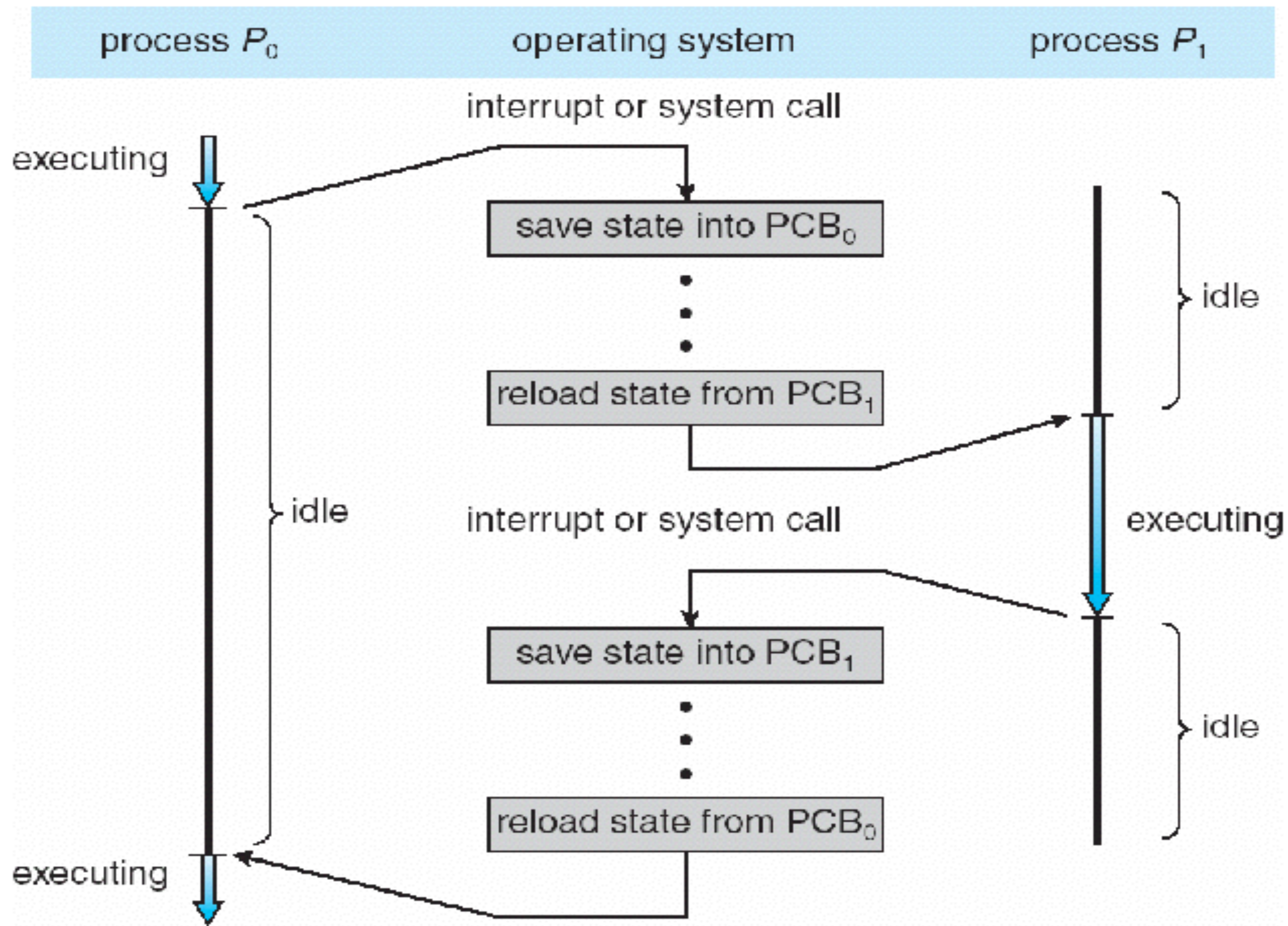
Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Changment de Contexte

- Lorsque la CPU passe à un autre processus, le système doit enregistrer l'état de l'ancien processus et charger l'état enregistré pour le nouveau processus via un commutateur de contexte.
  - Le contexte est représenter par le PCB
- Le temps de changement de contexte est le “overhead”; le système ne fait aucun travail utile lors de la commutation
  - Doit réduire autant que possible

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Changement de Contexte



Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Menu

- Introduction aux processus
- Ordonnancement des processus
- **Opérations des processus**
- Communications entre les processus

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

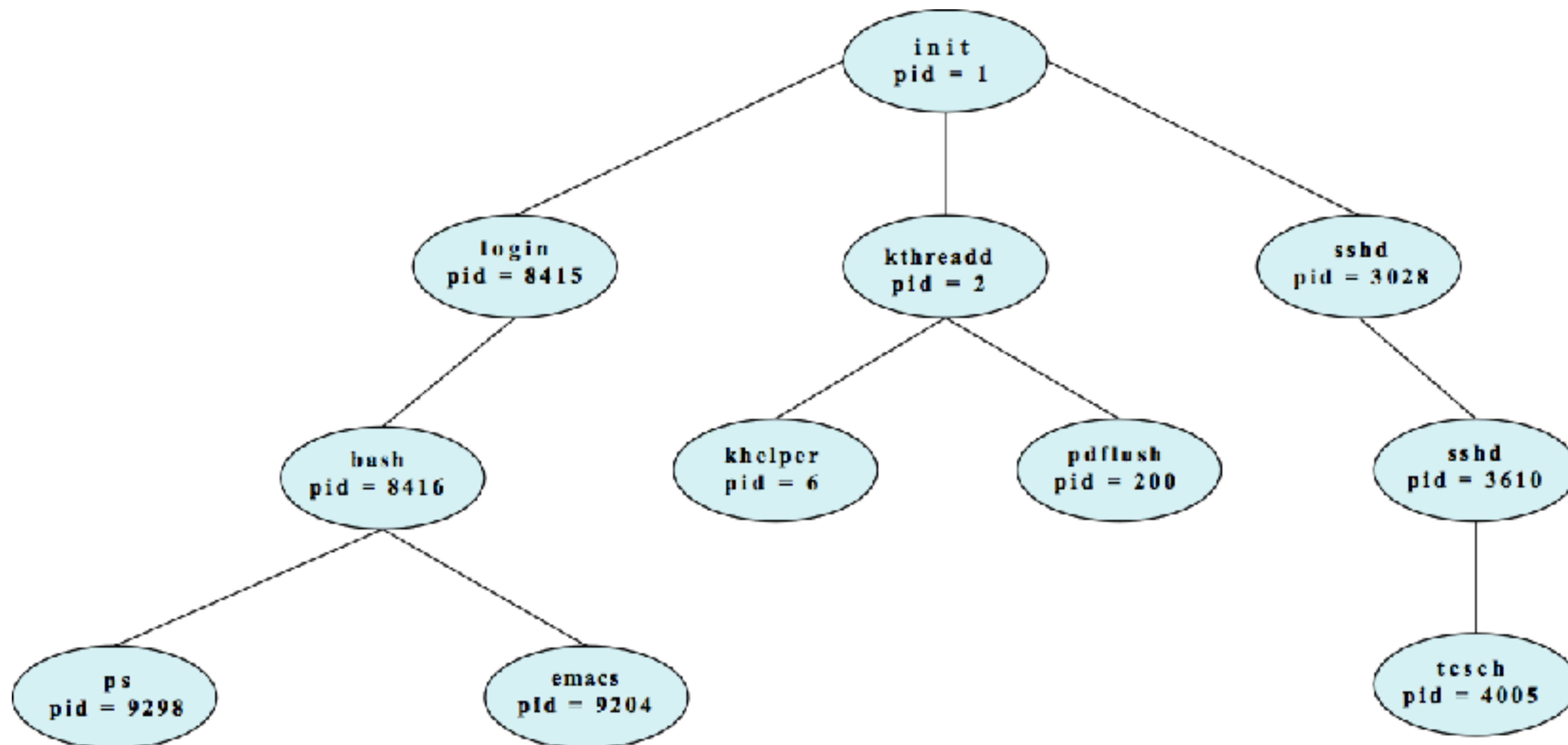
Création des Processus

- Le processus parent crée des processus enfants qui, à leur tour, créent d'autres processus, formant un arbre de processus
- Généralement, processus identifié et géré via un identifiant de processus (pid)
- Options de partage de ressources
  - Parent et enfants partagent toutes les ressources
  - Les enfants partagent le sous-ensemble des ressources parentales
  - Parent et enfant ne partagent aucune ressource
- Options d'exécution
  - Le parent et les enfants s'exécutent simultanément
  - Parent attend jusqu'à ce que les enfants se terminent

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

A Tree of Processes in Linux

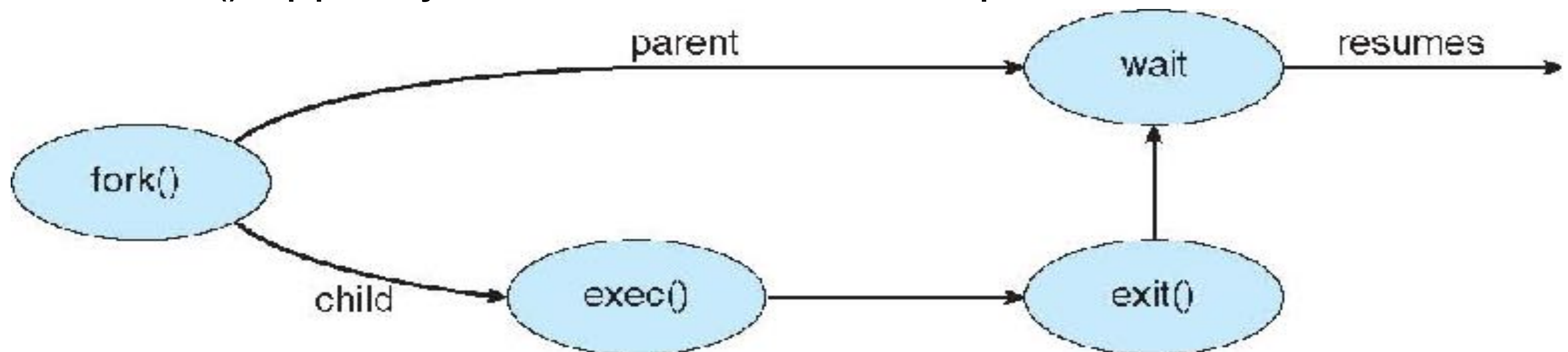


Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Création des Processus

- Possibilités d'espace d'adressage:
  - l'Enfant est un copie du parent
  - l'Enfant est un nouveau processus

- Exemples UNIX
  - fork() appel système crée un nouveau processus



Si l'enfant n'appelle pas (via exec()) un autre programme, une copie du parent continue son exécution dans son propre espace (variables, etc.)

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Unix fork()

```
/**
 * This program forks a separate process
 * using the fork()/exec() system calls.
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
  pid_t pid;

  /* fork a child process */
  pid = fork();

  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed\n");
    return 1;
  }
  else if (pid == 0) { /* child process */
    printf("I am the child %d\n",pid);
    execlp("ls","ls",NULL);
  }
  else { /* parent process */
    /* parent will wait for the child to complete */
    printf("I am the parent %d\n",pid);
    wait(NULL);

    printf("Child Complete\n");
  }

  return 0;
```

exécuter par l'enfant et le parent

- Le fonction fork() est utiliser pour créer des nouveau processus

- enfant fait un copie du mémoire du parent
- parent and child continue a executer dans leur espace

- fork() retournes des valeurs différents dans l'enfant et dans le parent
  - enfant -> retourne 0
  - parent -> retourne le pid de l'enfant
    - on utilise ça pour brancher l'execution

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Unix exec*()

```
else if (pid == 0) { /* child process */
   printf("I am the child %d\n",pid);
   execlp("ls","ls",NULL);
}
```

- Remplace l'image d'un processus avec un nouveau image dans mémoire

- Dans C - ca cherche un ligne come:

```
int main (int argc, char *argv[]);
```

- `execl*` - pass les arguments dans une liste:

```
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
```

- `execv*` - pass les arguments dans une "array":

```
int execv(const char *path, const char *arg0, ... /*, (char *)0 */);
```

- `exec*e` - une autre array de variables vont être inclus dans l'environment

- `exec*p` - utilise le variable environ $PATH pour chercher l'executable

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Unix wait()

```
 else { /* parent process */
    /* parent will wait for the child to complete */
    printf("I am the parent %d\n",pid);
    wait(NULL);
```

- pid = wait(&Status): attend que **un** de ses enfant termines

- Le **status** est le statut du processus quand il a fini (0 vs pas 0).

- Le pid est le pid du processus qui a terminer

- Si tu veut attendre que tous vos enfants se termine, faire une boucle

- Si tu veut attendre un processus spécifique, utilise `waitpid`

```
pid_t waitpid(pid_t pid, int *status, int options);
```

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

beaucoup plus de paramètres pour **CreateProcess()** que **fork()**!

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Termination des Processus

- Le processus exécute la dernière instruction et demande au système d'exploitation de la supprimer (**exit ()**)
  - Sortie des données de l'enfant vers le parent (via **wait ()**)
  - Les ressources du processus sont désallouées par le système d'exploitation
- Parent peut terminer l'exécution des processus fils (**abort ()**)
  - L'enfant a dépassé les ressources allouées
  - Tâche assignée à l'enfant n'est plus nécessaire
  - Si le parent est en train de quitter
    - Certains systèmes d'exploitation n'autorisent pas l'enfant à continuer si son parent se termine
    - Tous les enfants terminés - “**cascading termination**”
- Attendre la fin, en retournant le statut PID:

```
pid t_pid; statut int;
pid = wait (& statut);
```

- Si aucun parent n'attend (c'est-à-dire n'a pas atteint wait()), le processus terminé est un **zombie**
- Si le parent a terminé, les processus sont **orphelins**

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Ex: Chrome Browser

- De nombreux navigateurs Web ont fonctionné comme un seul processus (certains le font encore)
  - Si un site Web provoque des problèmes, l'ensemble du navigateur peut se bloquer ou se bloquer
- Le navigateur Google Chrome est multiprocess avec 3 catégories
  - Le processus de **Browser** gère l'interface utilisateur, le disque et les E/S réseau
  - Le processus **Renderer** rend les pages Web, traite HTML, Javascript, etc., et il y en a un nouveau pour chaque “tab”
    - Fonctionne dans un **sandbox** limitant le disque et les E/S réseau, minimisant l'effet des exploits de sécurité
  - Processus de **plug-in** pour chaque type de plug-in (par exemple, Flash, QuickTime)

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Menu

- Introduction aux processus
- Ordonnancement des processus
- Opérations des processus
- **Communications entre les processus**

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

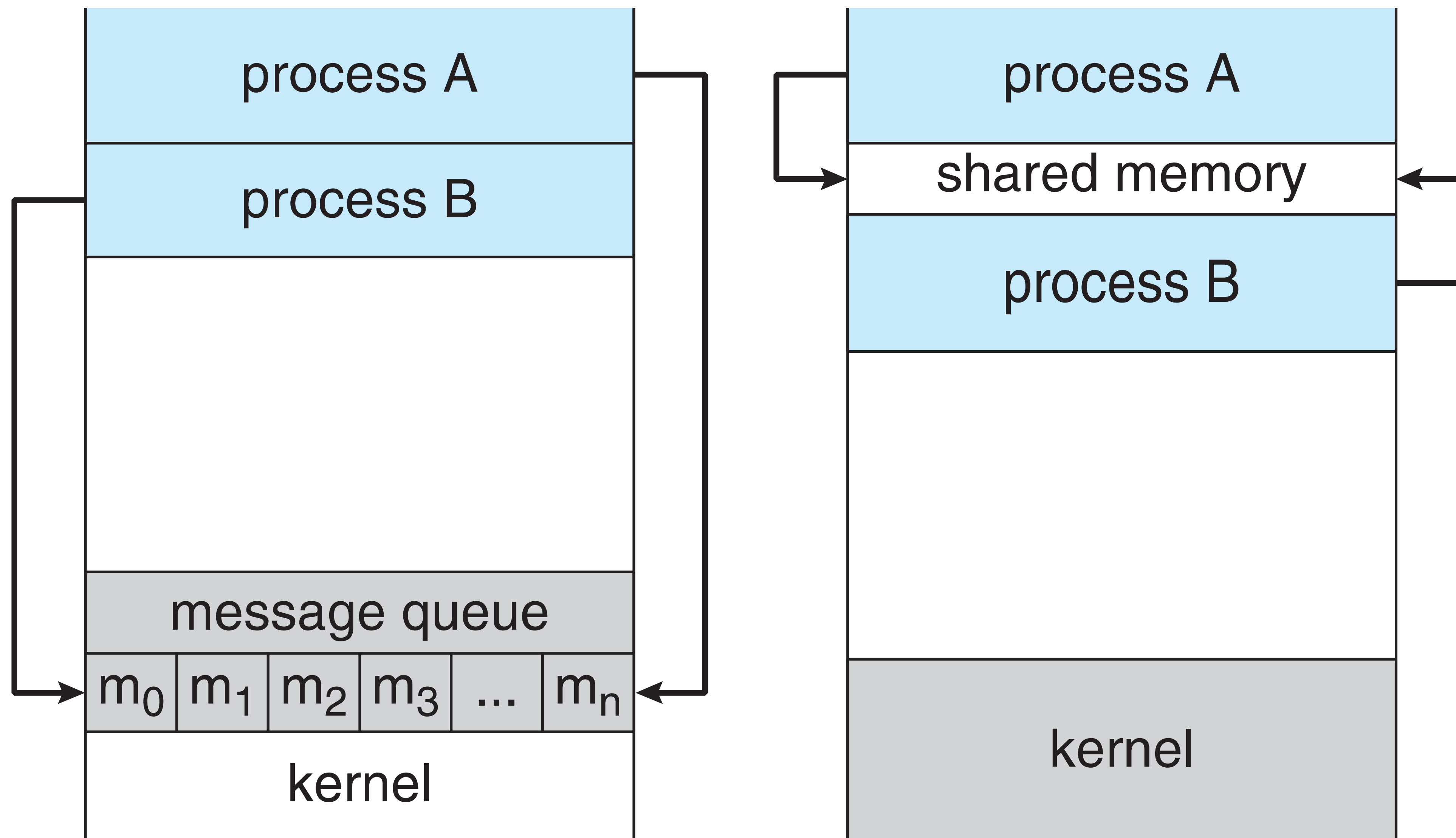
Communication Interprocessus (IPC)

- Les processus au sein d'un système peuvent être **indépendants** ou **coopérer**
- Processus **indépendant** ne peut pas affecter ou être affecté par l'exécution d'un autre processus
- Le processus de **coopération** peut affecter ou être affecté par d'autres processus, y compris le partage de données
- Raisons des processus de coopération:
  - Partage d'information
  - Accélération du calcul
  - Modularité
  - Commodité
- Les processus de coopération nécessitent une **communication interprocessus** (IPC)
- Deux modèles d'IPC
  - **La memoire partagée** (shared memory)
  - **Passage de message** (message passing)

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Modèles de communication



Passage de Message

La Mémoire Partagée

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Problème producteur-consommateur

- Paradigme pour les processus de coopération, le processus producteur produit des informations qui sont consommées par un processus consommateur
- Deux modèles:
  - "unbounded-buffer" n'impose aucune limite pratique à la taille de la mémoire buffer (le consommateur peut devoir attendre de nouveaux éléments, mais le producteur peut toujours générer de nouveaux éléments)
  - "bounded-buffer" suppose qu'il y a une taille de buffer fixe (le consommateur doit attendre si le buffer est vide, le producteur doit attendre si le buffer est plein)

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Bounded-Buffer avec Mémoire Partagée

- Données partagées (sous forme de tableau circulaire)

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0; /* next free */
int out = 0;/* first full */
```


Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Bounded-Buffer avec Mémoire Partagée

```
item next_produced;
while (true) {
      /* produce an item in next_produced */
      while (((in + 1) % BUFFER_SIZE) == out)
            ; /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
      while (in == out)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

      /* consume the item in next consumed */
}
```

- Q: Combien de “items” peut être dans le “buffer” ?   A: BUFFER_SIZE - 1
- N'aborde pas ici l'accès simultané - beaucoup d’issue de **synchronisation** - a plus tard

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

IPC avec Passage de Message

- Mécanisme permettant aux processus de communiquer et de synchroniser leurs actions
- Système de messagerie - les processus communiquent entre eux sans recourir à des variables partagées
- L'installation IPC fournit deux opérations:
  - **send**(message) - taille du message fixe ou variable
  - **receive**(message)
- Si P et Q souhaitent communiquer, ils doivent:
  - établir un **lien de communication** entre eux
  - échanger des messages via envoyer / recevoir
- Implémentation du lien de communication
  - physique (par exemple, mémoire partagée, bus matériel)
  - logique (par exemple, direct ou indirect, synchrone ou asynchrone, mise en mémoire tampon automatique ou explicite)

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Questions d'implémentation

- Comment les liens sont-ils établis?
- Un lien peut-il être associé à plus de deux processus?
- Combien de liens peut-il y avoir entre chaque paire de processus de communication?
- Quelle est la capacité d'un lien?
- La taille d'un message que le lien peut contenir est-elle fixe ou variable?
- Un lien est-il unidirectionnel ou bidirectionnel?

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Communication Directe

- Les processus doivent se nommer explicitement:
  - `send(P, message)` - envoyer un message à processus P
  - `receive(Q, message)` - recevoir un message du processus Q

- Propriétés du lien de communication
  - Les liens sont établis automatiquement
  - Un lien est associé à exactement une paire de processus communicants
  - Entre chaque paire, il existe exactement un lien
  - Le lien peut être unidirectionnel (asymétrie), mais est généralement bidirectionnel (symétrie)
    - `send(P, message)`
    - `receive(id, message)`

  - Si un identifiant de processus (pid) doit être modifié, le système doit scanner tous les processus pour mettre à jour un lien vers le nouveau pid (coûteux et peu souhaitable)

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Communication Indirecte

- Les messages sont dirigés et reçus des mailbox (également appelés ports)
  - Chaque mailbox a un identifiant unique
  - Les processus ne peuvent communiquer que s'ils partagent une mailbox

- Propriétés du lien de communication
  - Lien établi uniquement si les processus partagent une mailbox commune
  - Un lien peut être associé à de nombreux processus
  - Chaque paire de processus peut partager plusieurs liens de communication
  - Le lien peut être unidirectionnel ou bidirectionnel

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Communication Indirecte

- Opérations
  - créer une nouvelle mailbox
  - envoyer et recevoir des messages via une mailbox
  - détruire une mailbox
- Les primitives sont définies comme:
  - `send(A, message)` - envoyer un message à la mailbox A
  - `receive(A, message)` - recevoir un message de la mailbox A

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Communication Indirecte

- Partage de mailbox
  - P1, P2 et P3 partagent la mailbox A
  - P1 envoie; P2 et P3 reçoivent
  - Qui reçoit le message?
- Solutions
  - Autoriser un lien à être associé à au plus deux processus
  - Autoriser un seul processus à la fois pour exécuter une opération de réception
  - Laissez le système sélectionner arbitrairement le récepteur. L'expéditeur est informé du destinataire.

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Synchronisation

- Le passage de message peut être soit bloquant ou non bloquant

- Le **blocage** est considéré comme **synchrone**
  - **Blocking send** a le bloc expéditeur jusqu'à la réception du message
  - **Blocking receive** a le bloc récepteur jusqu'à ce qu'un message soit disponible

- Non bloquant est considéré comme asynchrone
  - **Non-blocking send** a l'expéditeur envoie le message et continue
  - **Non-blocking receive** a le récepteur recevoir un message valide ou null

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Synchronisation

- Différentes combinaisons possibles
  - Si à la fois envoyer et recevoir bloquent, nous avons un rendez-vous
- Le producteur-consommateur devient trivial

```
message next_produced;
while (true) {
      /* produce an item in next_produced */
      send(next_produced);
}
```

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next_consumed */
}
```

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Buffering des Messages

- File d'attente des messages attachés au lien; implémentation de l'une des trois façons:
  1. Capacité nulle - 0 message (c'est-à-dire, pas de buffer)
     L'expéditeur doit attendre (bloquer) pour le destinataire (rendez-vous)
  2. Capacité bornée - longueur finie de *n* messages
     Sender doit attendre (bloquer) si le lien est plein
  3. Capacité illimitée - longueur infinie
     L'expéditeur n'attend jamais (ne bloque jamais)

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Examples - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

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

  - Also used to open an existing segment to share it
  - Set the size of the object in bytes

```
ftruncate(shm_fd, 4096);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory ptr, "Writing to shared memory");
```


Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

IPC POSIX Producteur

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```


Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

IPC POSIX Consommateur

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```


Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Exemples - Mach

- La communication de Mach est basée sur les messages
  - Même les appels système sont effectués par des messages
  - Chaque tâche reçoit deux mailbox à la création - Kernel et Notify
  - Seulement trois appels système nécessaires pour le transfert de messages

    `msg_send(), msg_receive(), msg_rpc()`
  - Mailbox nécessaires à la communication, créées via

    `port_allocate ()`
  - Envoyer et recevoir sont flexibles, par exemple quatre options si la mailbox est pleine:
    - Attendez indéfiniment
    - Attendez au maximum n millisecondes
    - Retour immédiat
    - Mettre temporairement en cache un message

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

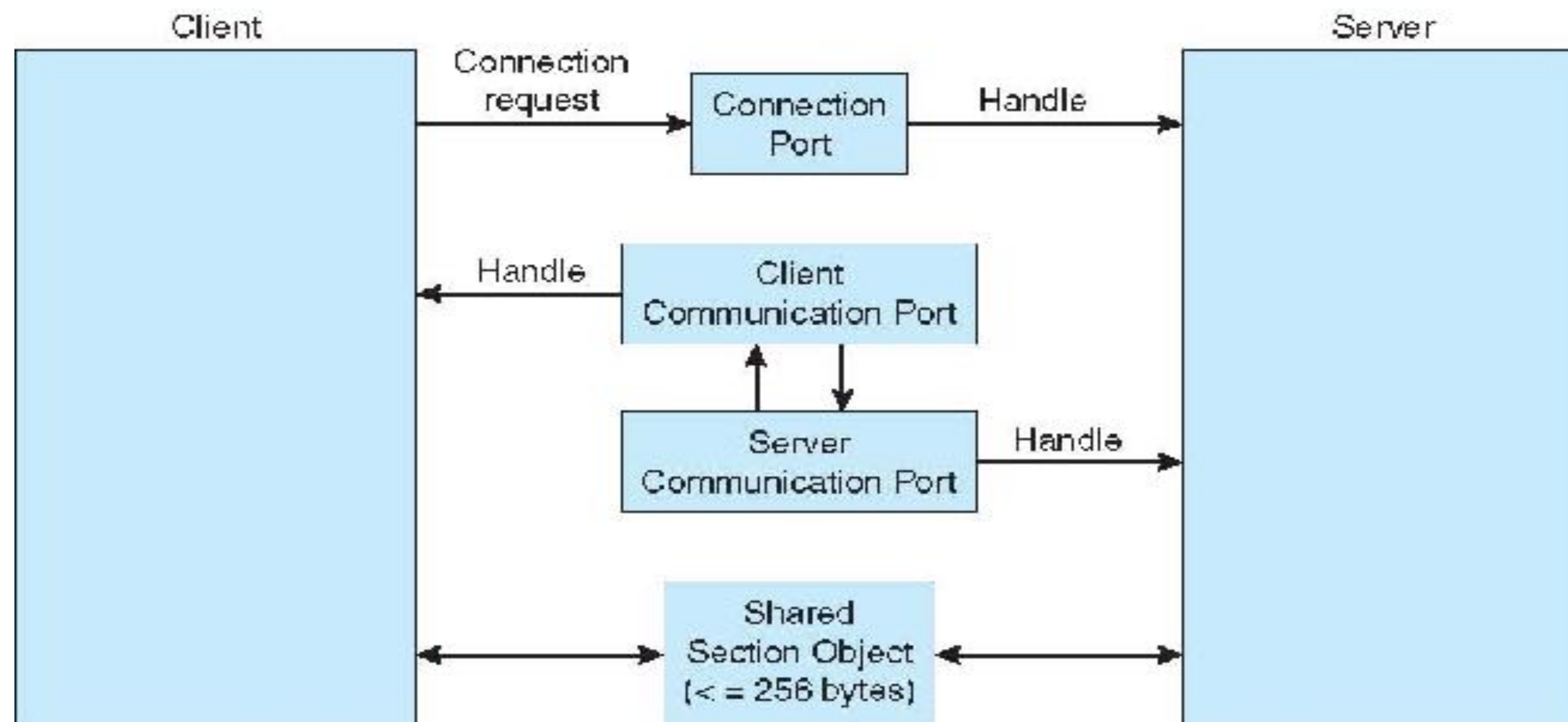
Exemples – Windows

- Centrage de messages centré via une fonction d'appel de procédure locale avancée (LPC)
  - Ne fonctionne que entre les processus sur le même système
  - Utilise les ports (comme les mailbox) pour établir et maintenir des canaux de communication
  - La communication fonctionne comme suit:
    - Le client ouvre un handle vers l'objet du **port de connexion** du sous-système
    - Le client envoie une demande de connexion
    - Le serveur crée deux **ports de communication** privés et renvoie le handle à l'un d'entre eux au client
    - Le client et le serveur utilisent le descripteur de port correspondant pour envoyer des messages ou des rappels et pour écouter les réponses

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Local Procedure Calls in Windows XP



Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Communications dans les systèmes client-serveur

- Sockets
- Remote Procedure Calls (RPC)
- Pipes

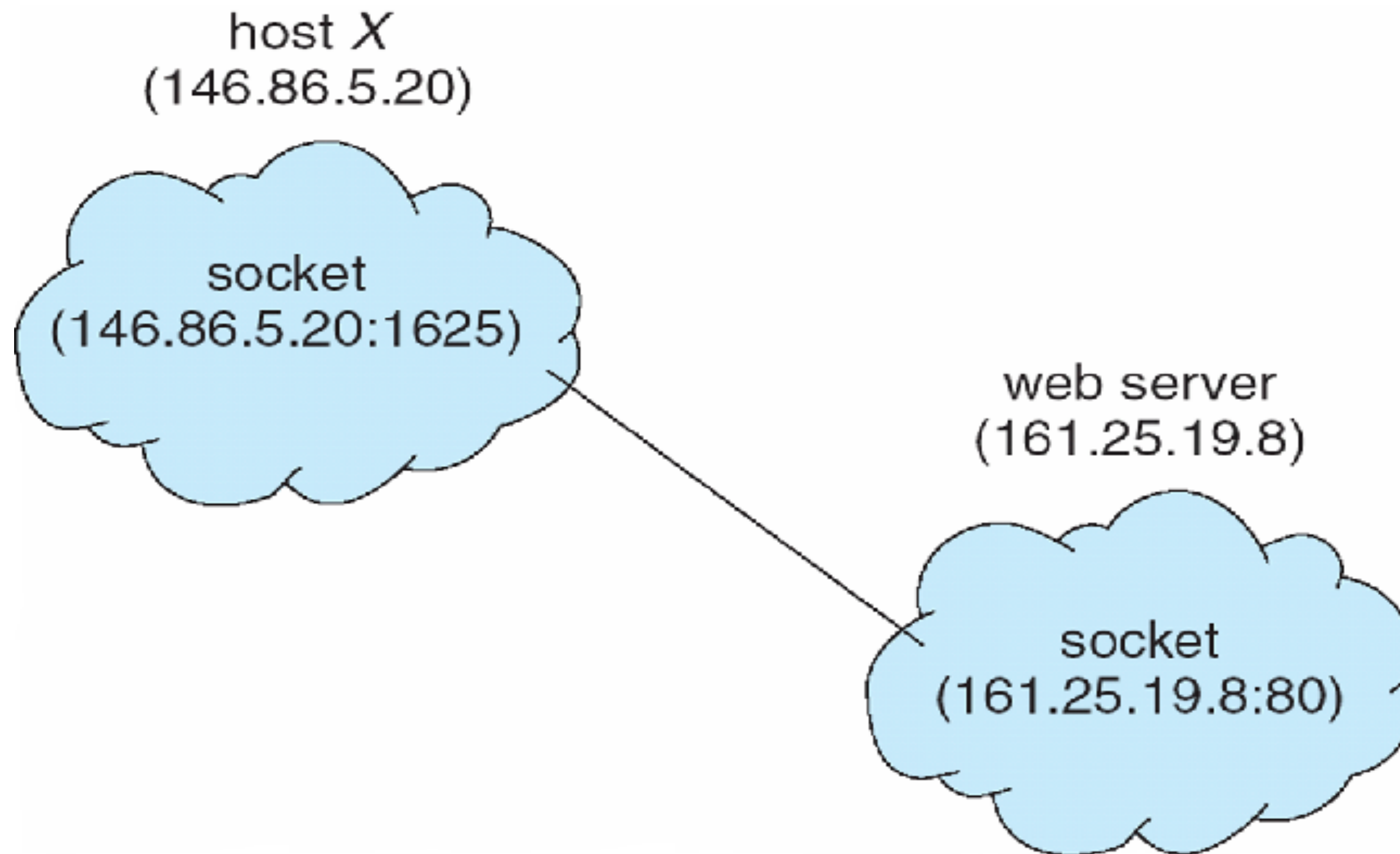
Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Sockets

- Un **socket** est défini comme un point de terminaison pour la communication
- Concaténation de **l'adresse IP** et du **port** - un nombre inclus au début du paquet de message pour différencier les services réseau sur un hôte
- Le socket **161.25.19.8:1625** fait référence au port **1625** sur l'hôte **161.25.19.8**
- La communication consiste en une paire de prises
- Tous les ports inférieurs à 1024 sont bien connus, utilisés pour les services standard
- Commun et efficace, mais de bas niveau car les données ne sont pas structurées. La structure doit être gérée par l'application client-serveur
- Adresse IP spéciale 127.0.0.1 (loopback) pour désigner le système sur lequel le processus est en cours d'exécution (c'est-à-dire sur son propre ordinateur)

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Socket Communication



Sockets en Java

- Trois types de prises
  - Orienté connexion (TCP)
  - Sans connexion (UDP)
  - Classe MulticastSocket - les données peuvent être envoyées à plusieurs destinataires
- Considérez ce serveur "Date":

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```


Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Remote Procedure Calls

- Les appels de procédures d'appel de procédure distante (RPC) entre des processus sur des systèmes en réseau
  - Utilise à nouveau les ports pour différencier les services
- **Stubs** - proxy côté client pour la procédure réelle sur le serveur
- Le talon côté client localise le serveur et **coordonne** les paramètres
- Le talon côté serveur reçoit ce message, décompresse les paramètres gérés et exécute la procédure sur le serveur
- Sous Windows, le code de remplacement est compilé à partir d'une spécification écrite en langage MIDL (**Microsoft Interface Definition Language**)
- Représentation des données gérée via le format XDL (**External Data Representation**) pour prendre en compte différentes architectures
  - **Big-endian** et **little-endian** (byte le plus significatif en premier / byte le moins significatif en premier)
- Exemple: système de fichiers distribués

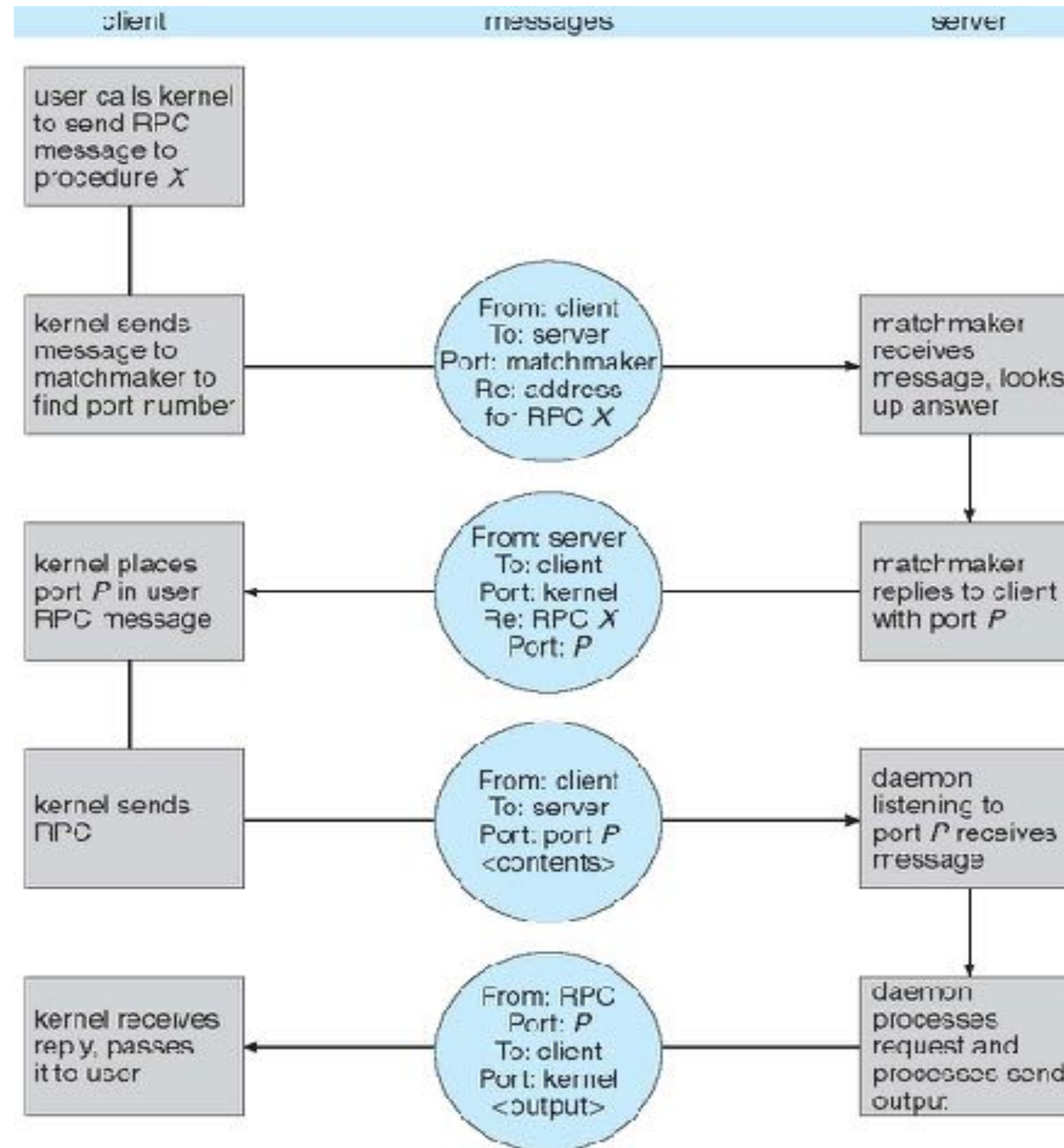
Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Remote Procedure Calls

- Problème 1: La communication à distance a plus de scénarios d'échec que locale (a cause du networking)
  - Les messages doivent être livrés **exactement une fois**
- Solution:
  - Le serveur utilise le “timestamping” pour être sur qu’il execute pas plus qu’un fois
  - Le client attend un accusé de réception
- Problème 2: Comment le client peut savoir les numéros des ports
- Solution: Le système d'exploitation fournit généralement un service de rendez-vous (ou de **matchmaker**) pour connecter le client et le serveur

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Execution of RPC



Les Tuyaux (Pipes)

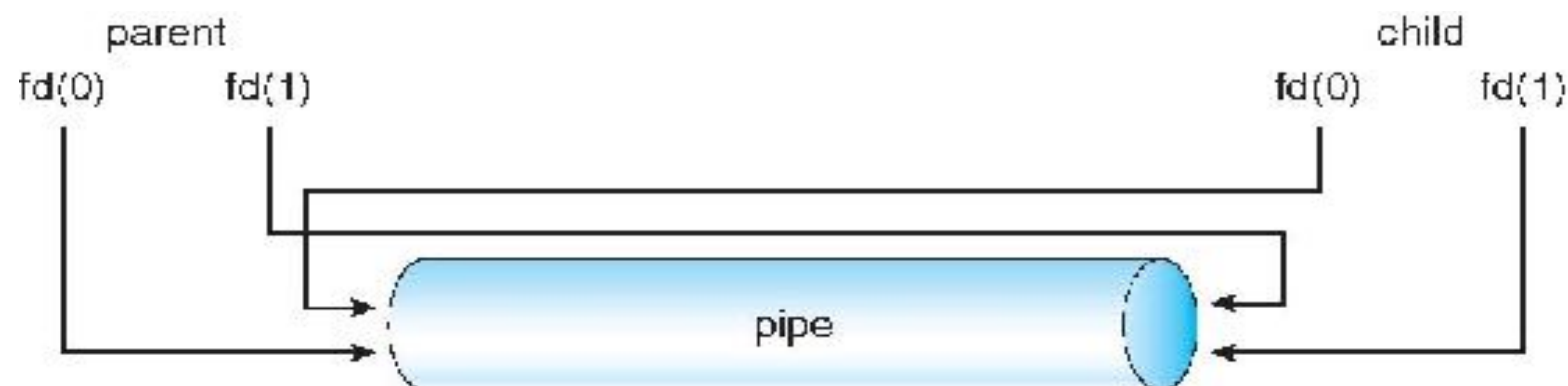
- Un **tuyau** agit comme un conduit permettant à deux processus de communiquer

- **Problèmes**
  - La communication est-elle **unidirectionnelle** ou **bidirectionnelle**?
  - Dans le cas d'une communication bidirectionnelle, s'agit-il d'un **half-duplex** ou d'un **full-duplex**?
  - Doit-il exister une relation (c'est-à-dire **parent-enfant**) entre les processus de communication?
  - Les tuyaux peuvent-ils être utilisés sur un réseau?

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Tuyaux Ordinaires

- Les **tuyaux ordinaires** permettent la communication dans le style standard producteur-consommateur
- Le producteur écrit à une extrémité (“**write-end**” du tuyau)
- Le consommateur lit à l'autre extrémité (“**read-end**” du tuyau)
- Les tuyaux ordinaires sont donc unidirectionnels (besoin de deux tuyaux bidirectionnels)
- Exiger une relation parent-enfant entre les processus communicants (valide uniquement sur la même machine)



- Windows appelle ces **tuyaux anonymes**

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Tuyaux Nommés

- Les tuyaux nommés sont plus puissants que les tuyaux ordinaires
- La communication est bidirectionnelle
- Aucune relation parent-enfant n'est nécessaire entre les processus de communication
- Plusieurs processus peuvent utiliser le tube nommé pour la communication
- Fourni sur les systèmes UNIX et Windows

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

Sommaire

- Un processus est une programme **en exécution**
- Chaque processus a un **état**
- Chaque processus est représenter par son bloc de contrôle (**PCB**)
- Le système d'exploitation choisi quel processus a mettre dans le "ready queue" (**ordonnancement longe terme**) et quel processus dans le ready queue vont être exécuter par le CPU (**ordonnancement cours terme**)
- Les fonctions POSIX qui contrôle la création est sont `fork()`, `wait()` et `exec*()`
- Les processus peuvent communiquer avec **la mémoire partagée** or la **passage des messages**
- Dans les système client-serveur, la communication peut être fait par: **les sockets**, **les RPC**, ou **les tuyaux**

DIRO

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018

End of Chapter 3

Silberschatz, Galvin and Gagne ©2013, Monnier ©2016, Paull ©2018