Systèmes d’exploitation II

Chapitre 2 : Gestion des processus

www.achrafothman.net

Processus

- Concept de Processus
- Ordonnancement de Processus
- Opérations sur les Processus
- Processus Coopératifs
- Communication Interprocessus
- Communication dans les systèmes Client-Serveur

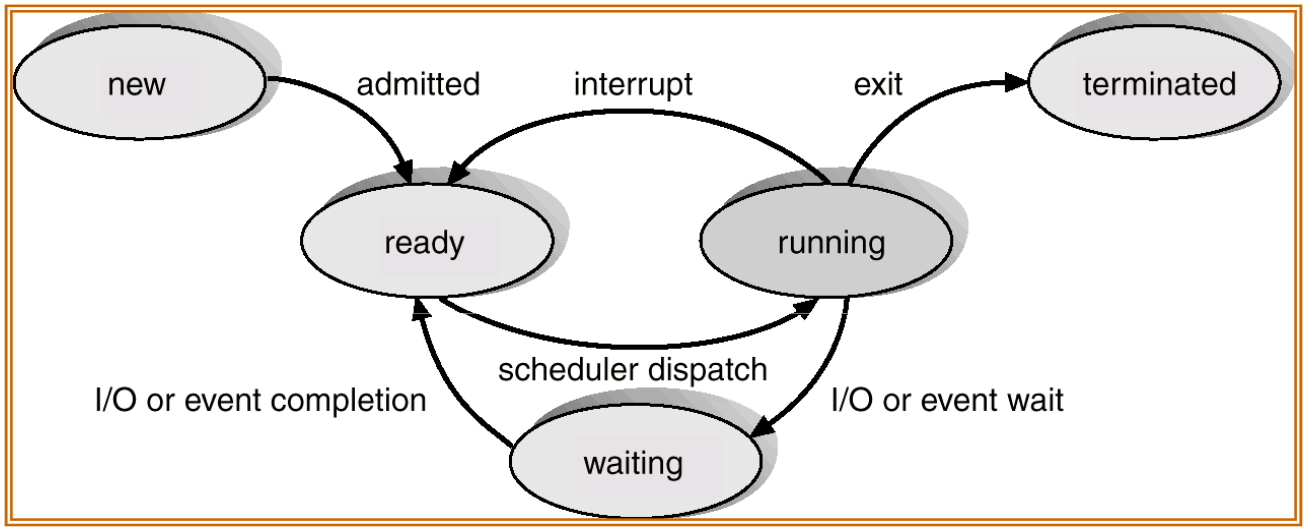
Concept de Processus

- Un OS exécute une variété de programmes:
  - Système Batch – tâches
  - Systèmes à temps partagé – Programmes utilisateurs ou tâches
- Les livres utilisent les termes *tâche* et *processus* indifféremment
- Processus – un programme en exécution
- Un processus inclut:
  - Compteur de programme (PC)
  - Pile (stack)
  - Section données

Etats d'un Processus

- En exécution, un processus change d'état
  - **new**: processus en train d'être créé
  - **running**: processus en exécution
  - **waiting**: processus en attente d'un évènement
  - **ready**: processus en attente du processeur
  - **terminated**: processus exécuté et terminé

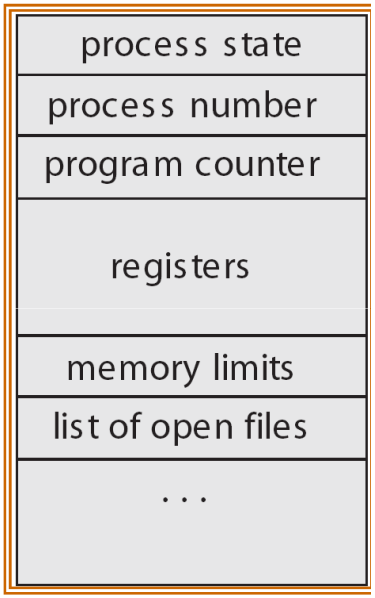
Diagramme des Etats des Processus



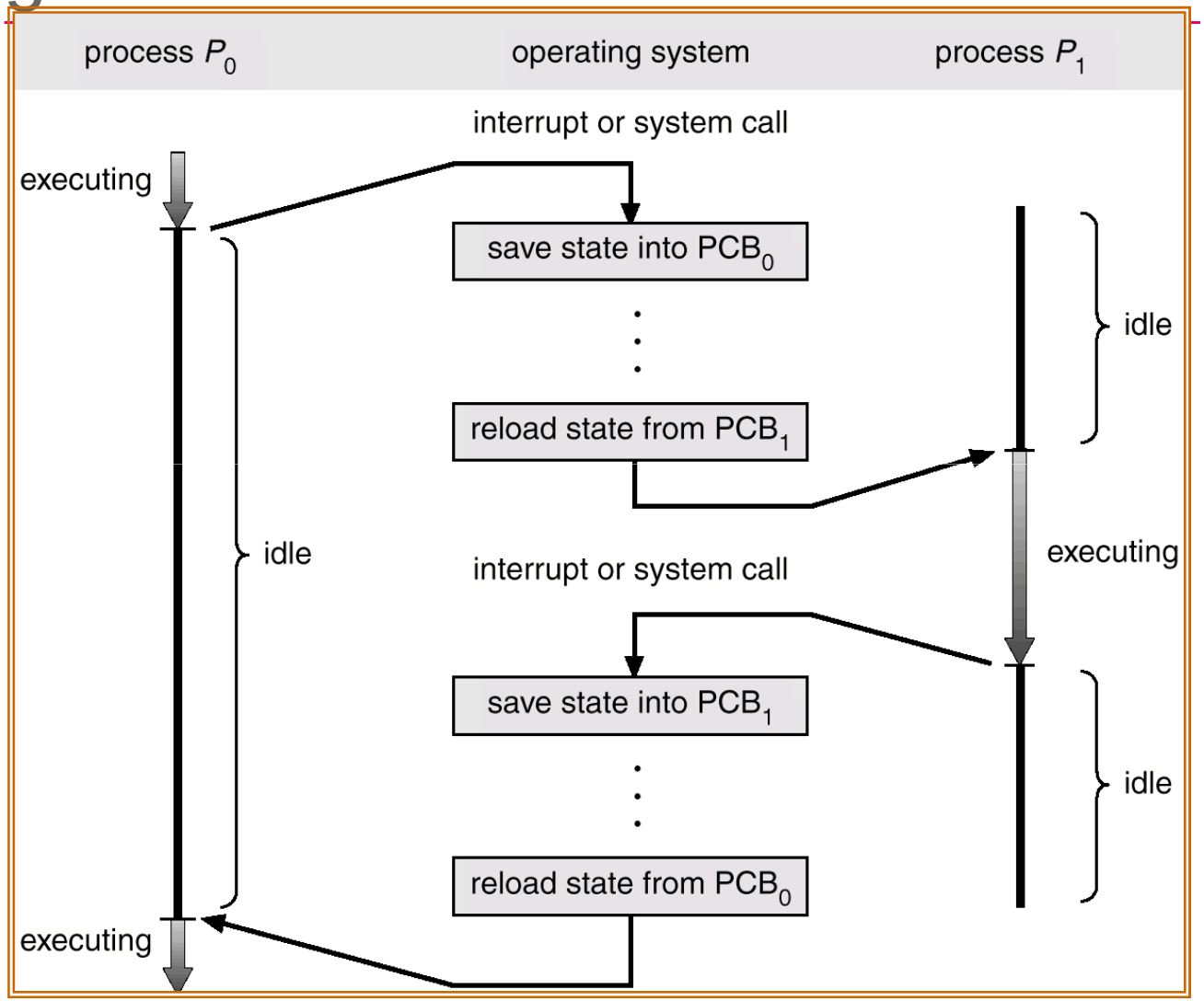
Process Control Block (PCB)

Information associée à chaque processus

- Etat du processus
- Compteur de programme (PC)
- Registres CPU
- Information sur l'ordonnancement CPU
- Information sur la gestion mémoire
- Information de comptabilité
- Information sur les E/S



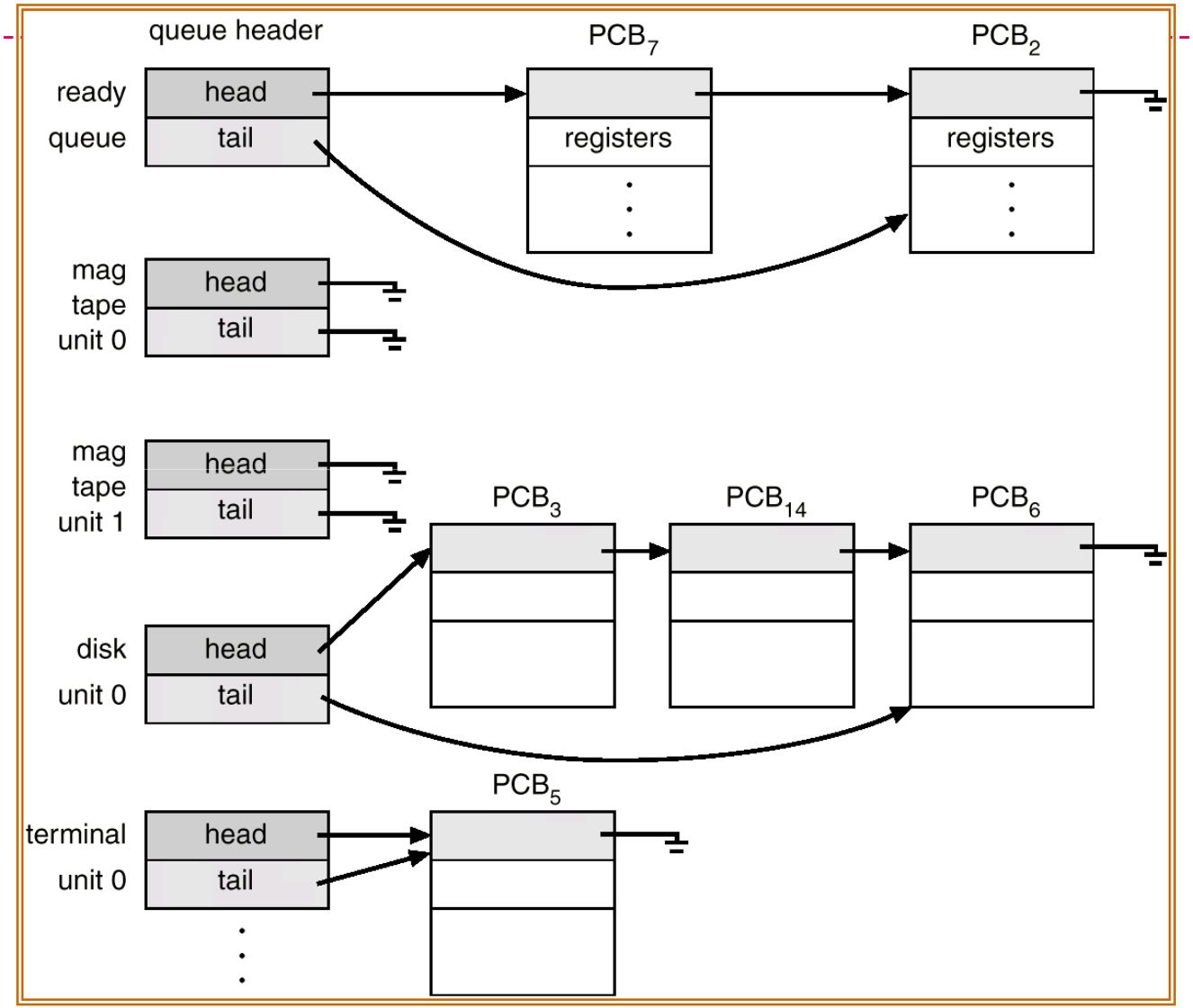
Changement de Contexte



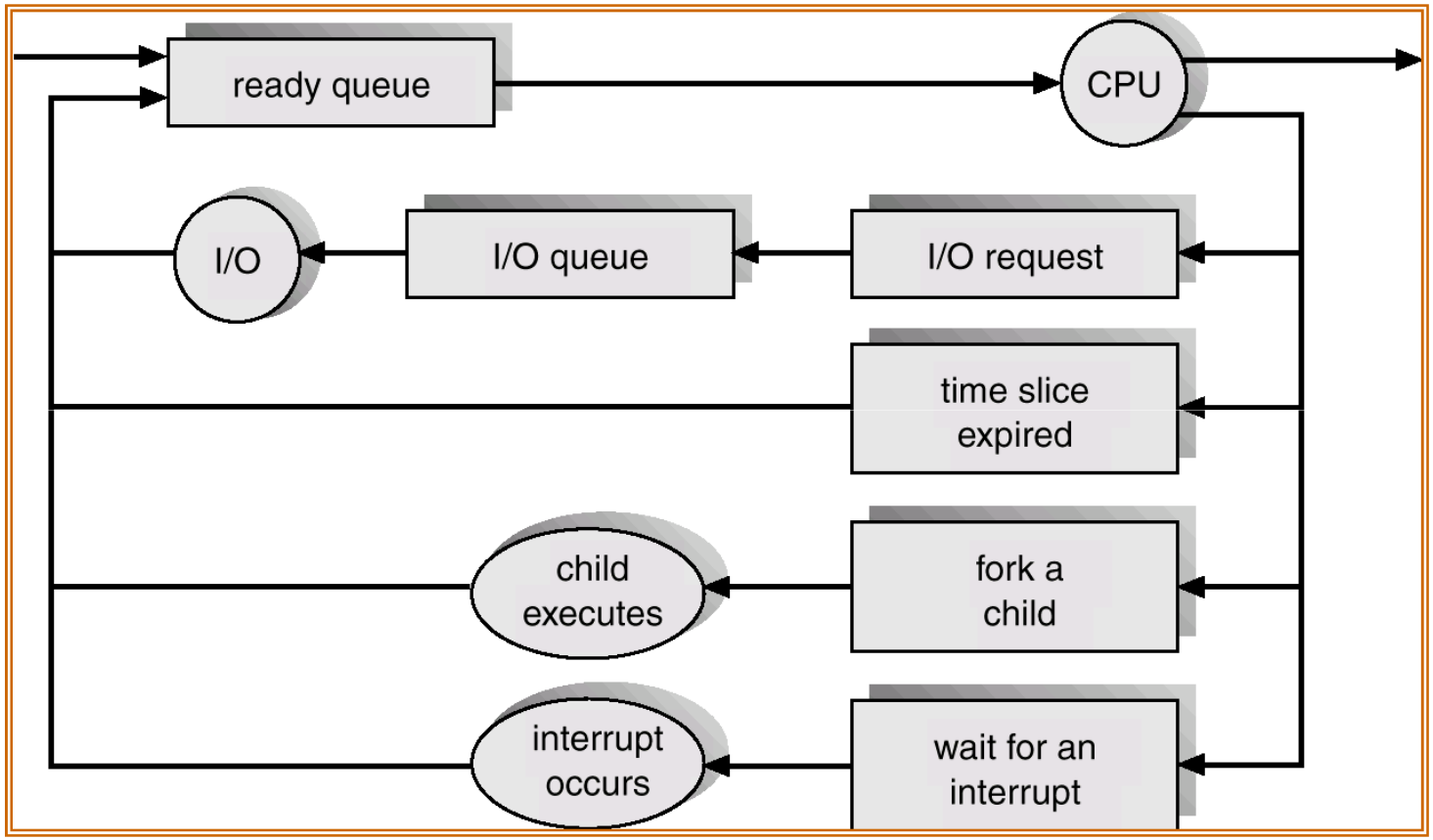
Queues d'Ordonnancement des Processus

- *Process queue* – ensemble de tous les processus du système
- *Ready queue* – ensemble de tous les processus en mémoire, prêts et en attente d'exécution
- *Device queues* – ensemble des processus en attente d'une E/S
- Migration de processus entre les différentes files

Ready Queue Et Différents Device Queues d'E/S



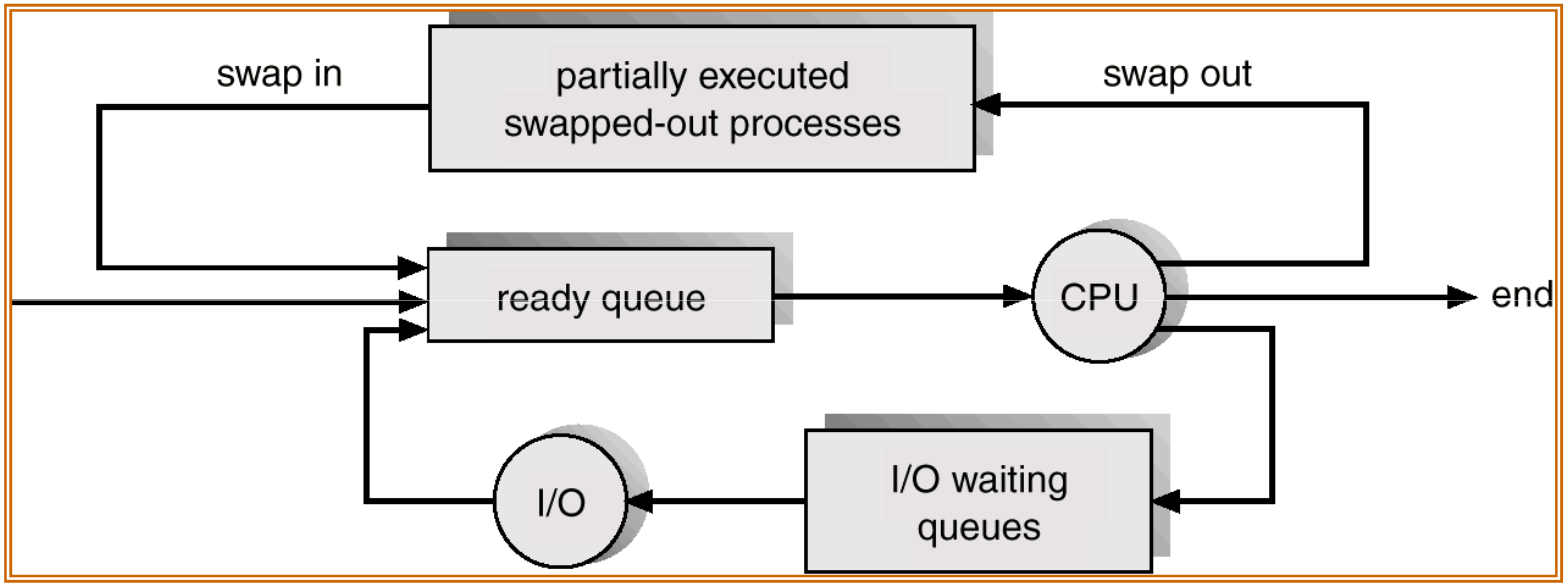
Représentation de l'Ordonnancement des Processus



Ordonnanceurs

- *Ordonnanceur à long terme* (ou ordonnanceur de tâches) – choisit quel processus doit être mis dans la file d'attente des processus prêts (Ready Queue)
- *Ordonnanceur à court terme* (o ordonnanceur CPU) – choisit quel processus doit être exécuté et lui alloue le CPU

Addition d'Ordonnanceur à Moyen Terme



Ordonnanceurs (Cont.)

- Ordonnanceur à court terme s'exécute très fréquemment (milliseconds) ⇒ (doit être rapide)
- Ordonnanceur à long terme intervient peu (secondes, minutes) ⇒ (peut être relativement plus lourd)
- L'ordonnanceur à long terme contrôle le *degré de multiprogrammation*
- Les processus peuvent être décrits comme étant:
  - *Processus E/S* – met plus de temps à faire des E/S que des calculs sur la CPU; beaucoup de demandes de la CPU à temps réduit
  - *Processus CPU* – met plus de temps à faire des calculs CPU; few very long CPU bursts; très peu de demande du CPU à temps prolongé

Changement de Contexte

- Quand un processus prend le contrôle de la CPU, l'OS doit sauvegarder l'état de l'ancien processus et charger l'état sauvegardé du nouveau processus
- Le temps pour le changement de contexte est l'*overhead*; l'OS ne fait pas de travail *utile* lors du changement
- Temps dépendant du support matériel

Création de Processus

- Un processus parent crée des processus fils, qui, à leur tour, peuvent créer d'autres processus, formant ainsi un arbre de processus
- Partage de Ressources
  - Les parents et les fils partagent toutes les ressources
  - Les fils partagent un sous-ensemble des ressources du parent
  - Le parent et les fils ne partagent aucune ressource
- Exécution
  - Le parent et les fils s'exécutent simultanément
  - Le parent attend la terminaison des fils

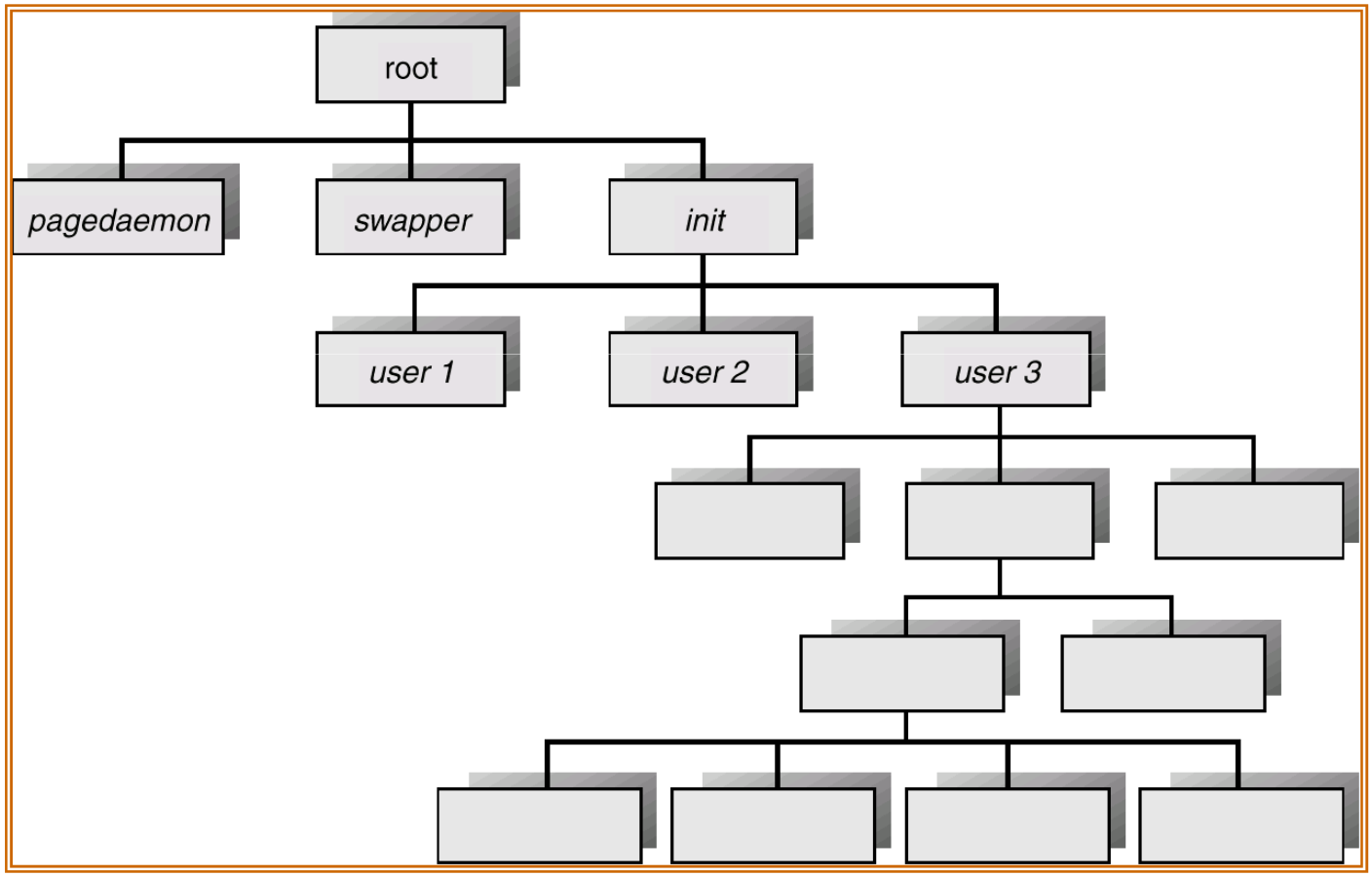
Création de Processus (Cont.)

- Espace d'Adressage
  - Le fils duplique le parent
  - Le fils a un programme différent du parent
- Exemples UNIX
  - Appel système **fork** crée de nouveaux processus
  - Appel système **exec** utilisé après un **fork** pour remplacer la mémoire du processus parent par un nouveau programme

Programme C Créant Plusieurs Processus

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete
    */
            wait(NULL);
            printf("Child Complete");
            exit(0);
    }
```

Arbre de Processus sur un Système UNIX



Terminaison de Processus

- Le processus exécute la dernière expression et demande à l'OS de décider (**exit**)
  - Données de terminaison du fils renvoyées au parent intéréssé (via **wait**)
  - Ressources systèmes libérées par l'OS
- Le parent peut terminer l'exécution des processus fils (signal **abort**)
  - Le fils a dépassé les ressources allouées
  - La tâche du fils n'est plus utile
  - Si le parent se termine
    - Certains OSs ne permettent pas aux fils de continuer
      - Tous les fils terminés – terminaison en cascade

Coopération Inter-Processus

- *Les processus indépendants* ne peuvent pas s'affecter
- *Les processus coopérants* peuvent s'affecter mutuellement
- Avantages des processus coopérants
  - Partage d'information
  - Accélération du calcul
  - Modularité
  - Commodité

Problème du Producteur-Consommateur

- Paradigme pour les processus coopérants; le processus *producteur* produit des informations qui sont utilisées par un processus *consommateur*
  - *Tampon infini* ne place aucune limite sur la taille du tampon
  - *Tampon borné* assume l'existence d'un tampon à taille limitée

Communication Inter-Processus (IPC)

- Mécanisme pour la communication inter-processus et la synchronisation de leurs actions
- Système de messages – les processus communiquent entre eux sans utiliser des variables partagées
- L'interface IPC fournit deux opérations:
  - **send**(*message*) – message de taille fixe ou variable
  - **receive**(*message*)
- Si *P* et *Q* désirent communiquer, ils ont besoin:
  - D'établir un lien de communication
  - D'échanger des messages via send/receive
- Implémentation d'un lien de communication
  - physique (e.g., mémoire partagéé, bus matériel)
  - logique (e.g., propriétés logiques)

Questions d'Implémentation

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

Communication Directe

- Les processus doivent se nommer explicitement:
  - **send** (*P, message*) – envoyer un message au processus P
  - **receive**(*Q, message*) – recevoir un message du processus Q
- Propriétés d'un lien de communication
  - Liens établis automatiquement
  - Un lien est associé avec exactement une paire de processus communiquants
  - Entre chaque paire, il existe exactement un lien
  - Le lien peut être unidirectionnel, mais il est habituellement bidirectionnel

Communication Indirecte

- Les messages sont dirigés et reçus dans des boîtes aux lettres (mailbox, appelées aussi des ports)
  - Chaque mailbox a un id unique
  - Les processus peuvent communiquer seulement s'ils partagent une mailbox
  - Un lien peut être associé à plusieurs processus
  - Chaque paire de processus peut partager plusieurs liens communiquants
  - Un lien peut-être unidirectionnel ou bidirectionnel

Communication Indirecte

- Opérations
  - Créer une boîte aux lettres
  - Envoyer et recevoir des messages via la boîte aux lettres
  - Détruire une boîte aux lettres
- Les primitives sont définies comme:

  send(*A*, *message*) – envoyer un message à la boîte aux lettres A

  receive(*A*, *message*) – recevoir un message de la boîte aux lettre A

Communication Indirecte

- Partage de boîte aux lettres (bal)
  - P_1, P_2, and P_3 partagent la “bal” A
  - P_1 envoie; P_2 and P_3 reçoivent
  - Qui reçoit le message?
- Solutions
  - Permettre à un lien d’être associé avec au plus deux processus
  - Permettre à un processus à la fois d’exécuter une opération receive
  - Permettre au système de choisir arbitrairement le receveur. Le processus émetteur est notifié de l’identité des receveurs.

Synchronisation

- L'échange de messages peut être bloquant ou non bloquant
- **Bloquant** est consdéré **synchrone**
  - **L'envoi bloquant** bloque l'émetteur jusqu'à ce que le message soit reçu
  - **La réception bloquante** bloque le récepteur jusqu'à la disponibilité d'un message
- **Non bloquant** est considéré **asynchrone**
  - **L'envoi non bloquant** fait que l'émetteur envoie le message et continue son exécution sans rien attendre
  - **La réception non bloquante** fait que le récepteur reçoit un message ou null suivant la disponibilité des messages à l'instant de l'appel

Tampons

- File de messages attachée au lien; implémenté de trois façons
  1. Capacité zéro – 0 messages
     L'émetteur doit attendre le récepteur (rendezvous)
  2. Capacité bornée – longueur finie de *n* messages
     L'emetteur doit attendre si le lien est plein
  3. Capacité non bornée – longueur infinie
     L'émetteur n'attend jamais

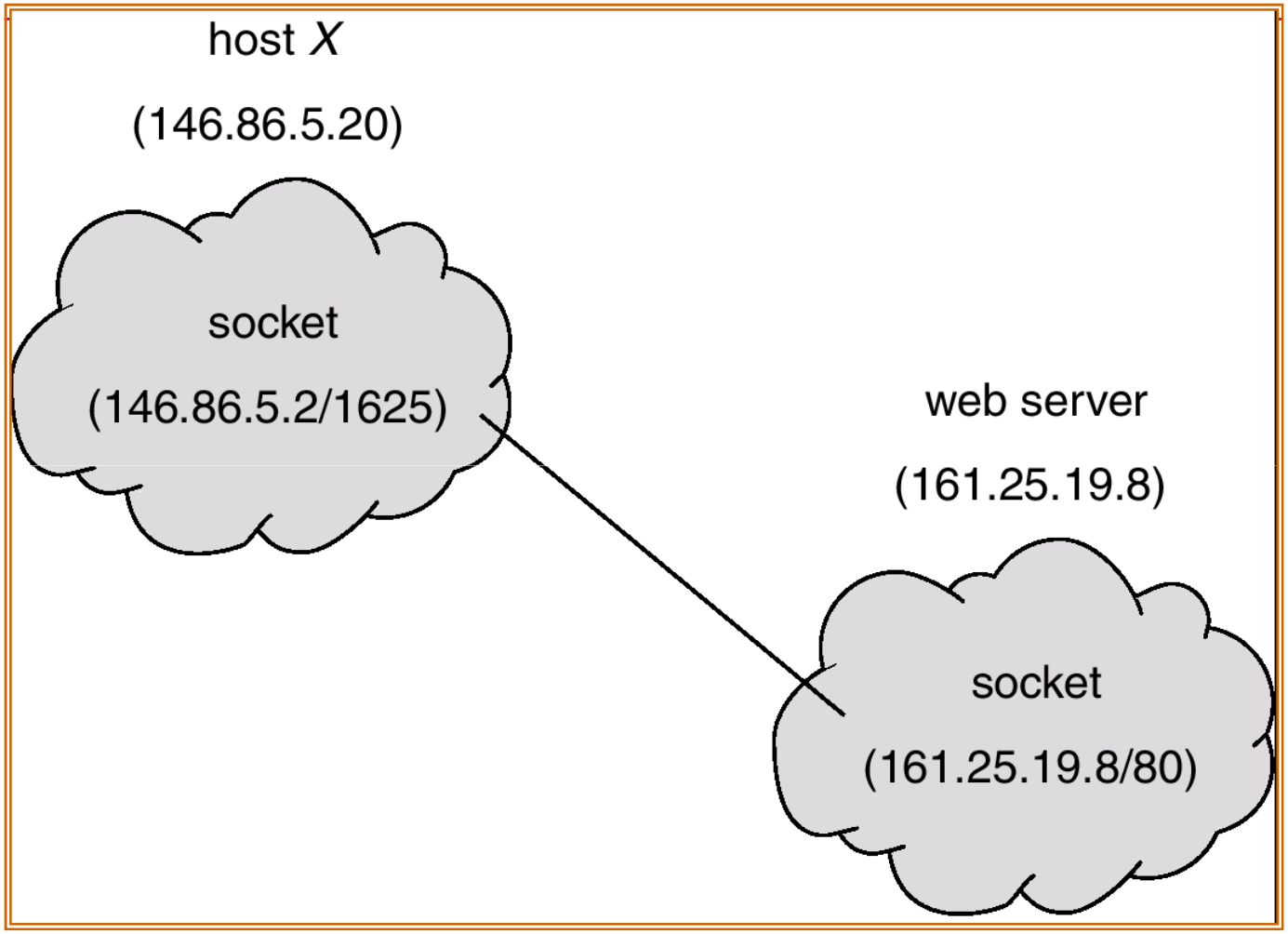
Communication Client-Serveur

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- Une socket est définie comme un *endpoint de communication*
- Concatenation d'une adresse IP et d'un numéro de port
- La socket **161.25.19.8:1625** désigne le port **1625** sur l'hôte **161.25.19.8**
- Une communication se fait entre une paire de sockets

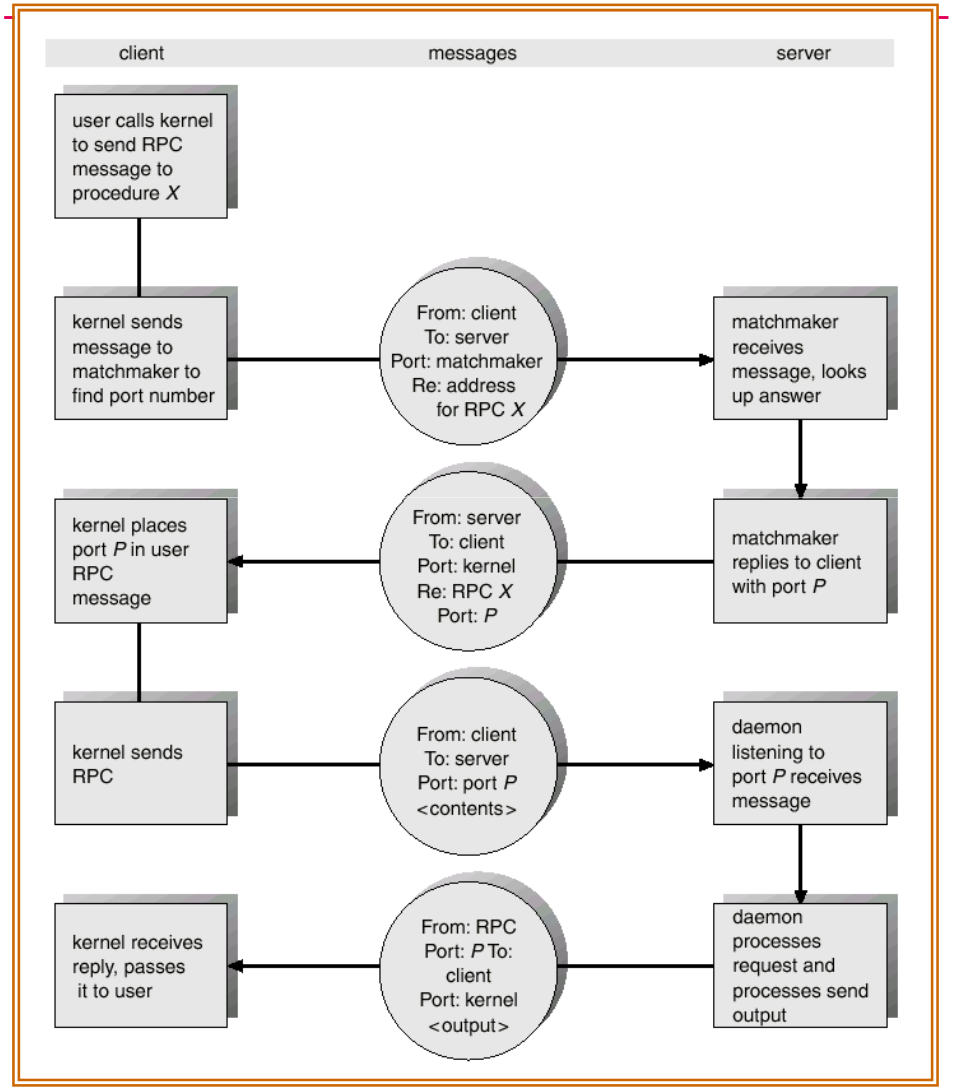
Communication Socket



Remote Procedure Calls

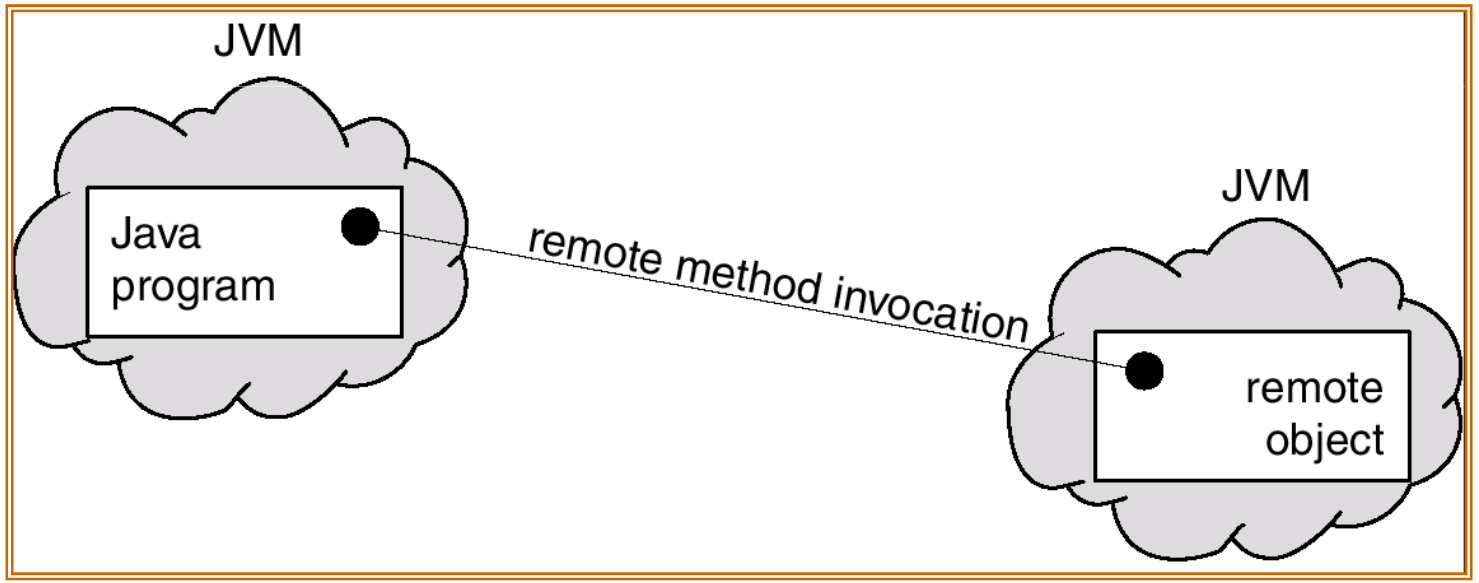
- Remote procedure call (RPC) émule un appel de procédure entre des processus distants
- **Stubs** – proxy côté client pour la procédure côté serveur
- Le stub côté client localise le serveur et lui transfère (sérialize) les paramètres
- Le stub côté serveur reçoit le message, lit les parmètres (dé-sérialise) les paramètres, exécute la procédure sur le serveur

Execution du RPC

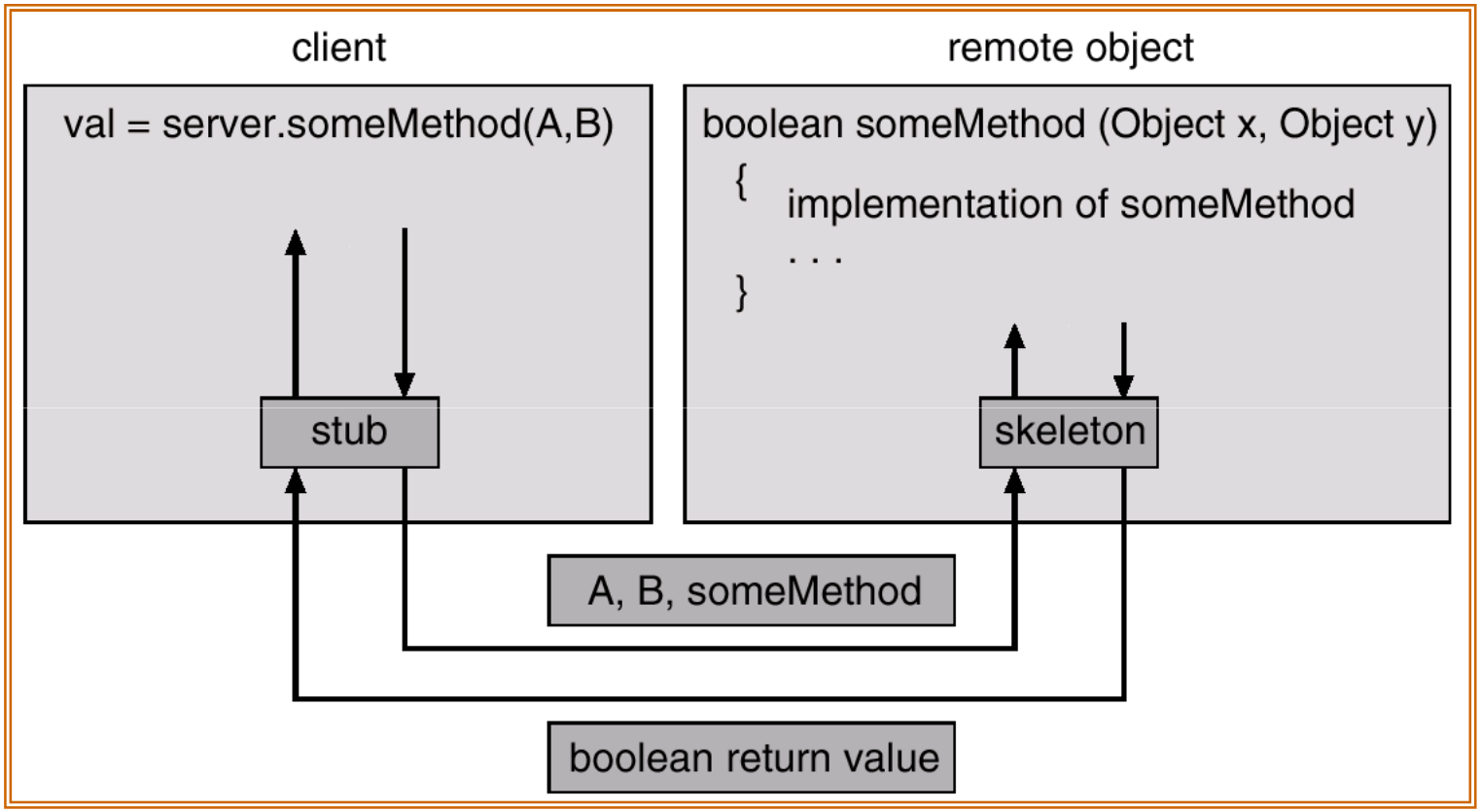


Remote Method Invocation

- Remote Method Invocation (RMI) est un mécanisme Java similaire aux RPCs.
- RMI permet à un programme Java sur une machine pour invoquer une méthode sur un objet distant.



Sérialisation de Paramètres



Threads

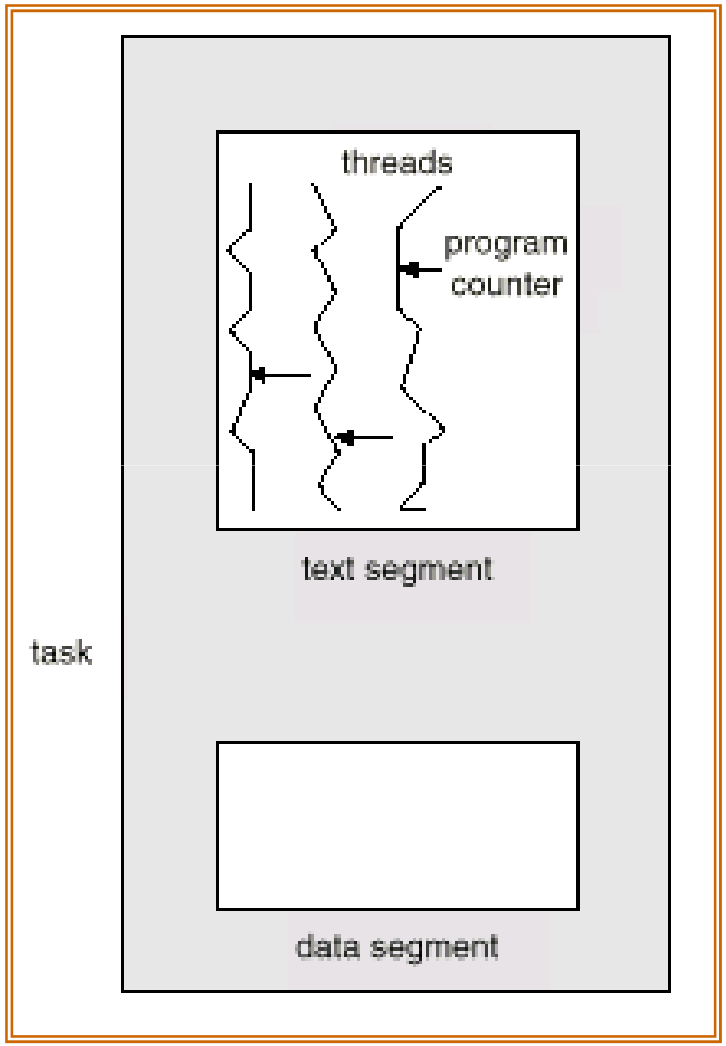
- Un *thread* (ou *lightweight process*) est une unité de base de l'utilisation de la CPU; il consiste en:
  - Compteur de programme
  - Ensemble de registres
  - pile
- Un thread partage avec ses threads frères:
  - Section code
  - Section données
  - Ressources OS

  Connu collectivement comme une tâche
- Un processus traditionnel (heavyweight) est l'équivalent d'une tâche avec un seul thread

Threads (Cont.)

- Dans une tâche à plusieurs thread, alors qu'un thread du serveur est bloqué et en attente, un autre thread dans la même tâche peut s'exécuter.
  - Coopération de plusieurs threads permet un *throughput* plus élevé et une performance améliorée.
  - Les applications qui requièrent le partage d'un tampon commun (i.e., producteur-consommateur) beneficient de l'utilisation des threads.
- Les threads fournissent un mécanisme qui permet à des processus séquentiels de faire des appels système bloquants tout en continuant à s'exécuter en parallèle.
- Threads supportés par le noyau (Mach et OS/2).
- Threads utilisateurs; supportés par dessus le noyau, via un ensemble d'appels de bibliothèque au niveau utilisateur (Project Andrew from CMU).
- Approche hybride implémentant des threads niveau utilisateur et niveau noyau (Solaris 2).

Plusieurs Tâches dans un Thread



Threads Supportés dans Solaris 2

- Solaris 2 est une version d'UNIX avec un support des threads au niveau noyau et au niveau utilisateur, les multiprocesseurs symétriques, et l'ordonnancement temps réel.
- LWP – niveau intermédiaire entre les threads au niveau utilisateur et les threads au niveau noyau.
- Besoins en ressources des différents type de thread:
  - Thread noyau: une petite structure de données et une pile; un changement de thread ne requiert pas de changement des données relatives à l'accès aux informations – relativement rapide.
  - LWP: PCB avec un registre de données, comptabilité et de l'information mémoire; le changement entre LWPs est relativement lent.
  - Thread utilisateur: seulement besoin d'une pile et d'un compteur de programme; le noyau n'intervient pas et ainsi le changement de threads est rapide. Le noyau ne voit que les LWPs qui supportent des threads utilisateur.

Threads Solaris 2