Programmation Concurrente

Partie 1: processus et threads

Polytech/INFO 4, 2020-2021

Fabienne Boyer
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

L'exécution concurrente, omniprésente..

- **Tous les jours, vous utilisez votre téléphone mobile sans vous questionner:**
  - Vous pouvez téléphoner et recevoir des sms en parallèle
  - Votre alarme réveil peut se déclencher pendant que vous êtes en train d'écouter de la musique ou regarder une vidéo
  - …
- **Toutes ces applications agissent de manière collaborative**
- **Cette collaboration repose sur des concepts et outils dédiés fournis par le système d'exploitation**

L'exécution concurrente, omniprésente..

- **Tous les jours, vous utilisez votre browser pour glaner des informations sur le web:**
  - Vous êtes potentiellement des dizaines / centaines / milliers à accéder à certains sites simultanément
  - Mais vous n'avez pas le sentiment d'attendre...
- **Les serveurs Web sont capables de traiter un grand nombres de requêtes en parallèle**
- **Cette parallélisation repose aussi sur des concepts et outils dédiés fournis par le système d'exploitation**

L'exécution concurrente, omniprésente..

- **Dans les exemples pré-cités, plusieurs *flôts d'exécution* sont en cours sur une machine**
  - Les flôts d'exécution sont aussi appelés tâches, lots, ..
  - Ils mettent en oeuvre une fonction applicative
  - Ils peuvent potentiellement partager des données ou se coordonner

- **Pour que tout se passe bien, certaines contraintes doivent être respectées**
  - Les données partagées doivent rester cohérentes
  - Les flôts doivent avoir un temps d'exécution correct
  - Les flôts doivent avoir un temps de réponse correct
  - …

Objectifs du cours

- **Maîtriser les bases de la programmation concurrente**
  - Connaître les concepts et outils fournis par le système
  - Savoir les utiliser avec rigueur et efficacité
- **Contexte technique**
  - C / Java
- **Portée du cours**
  - Concurrence locale (flôts d'exécution co-localisés sur une machine)

Plan du cours

1. **Modèles d'exécution concurrente**
   - Systèmes mono-programmés
   - Systèmes multi-programmés
   - Systèmes à temps partagé
2. **Concepts fournis par le système**
   - Processus
   - Threads
3. **Outils de synchronisation**
   - Outils de bas niveau
   - Outils de haut niveau
4. **Gestion des interblocages**
   - Prévention
   - Guérison

Organisation de l'enseignement

- **Equipe pédagogique**
  - Fabienne Boyer (Fabienne.Boyer@imag.fr, Equipe Erods / LIG)
  - Noël de Palma (Noel.dePalma@imag.fr, Equipe Erods / LIG)
  - Olivier Gruber (Olivier.Gruber@imag.fr, Equipe Erods / LIG)
- **Volume**
  - 8 Cours d'1H30
  - 10 TD d'1H30
  - 1 TP en C (coordination de processus / shell)
  - 1 TP en java (programmation concurrente multi-threadée)
  - 1 examen

Bibliographie

- **Systèmes d'exploitation, fournisseur des concepts et outils de bas niveau pour la programmation concurrente :**
  - Silberschatz, Galvin and Gagne, Operating System Concepts, Addison-Wesley, 2nd edition, 2014
  - A. Tanenbaum, Modern Operating Systems, dec 2015
- **Programmation multi-threadée (Java context)**
  - Java Concurrency in Practice

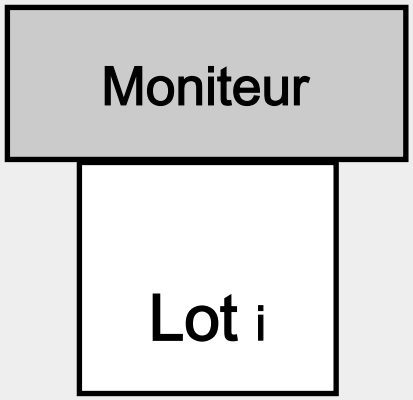
1- Modèles d'exécution concurrente

- **3 modèles de base**
  - Mono-programmation
  - Multi-programmation
  - Temps-partagé

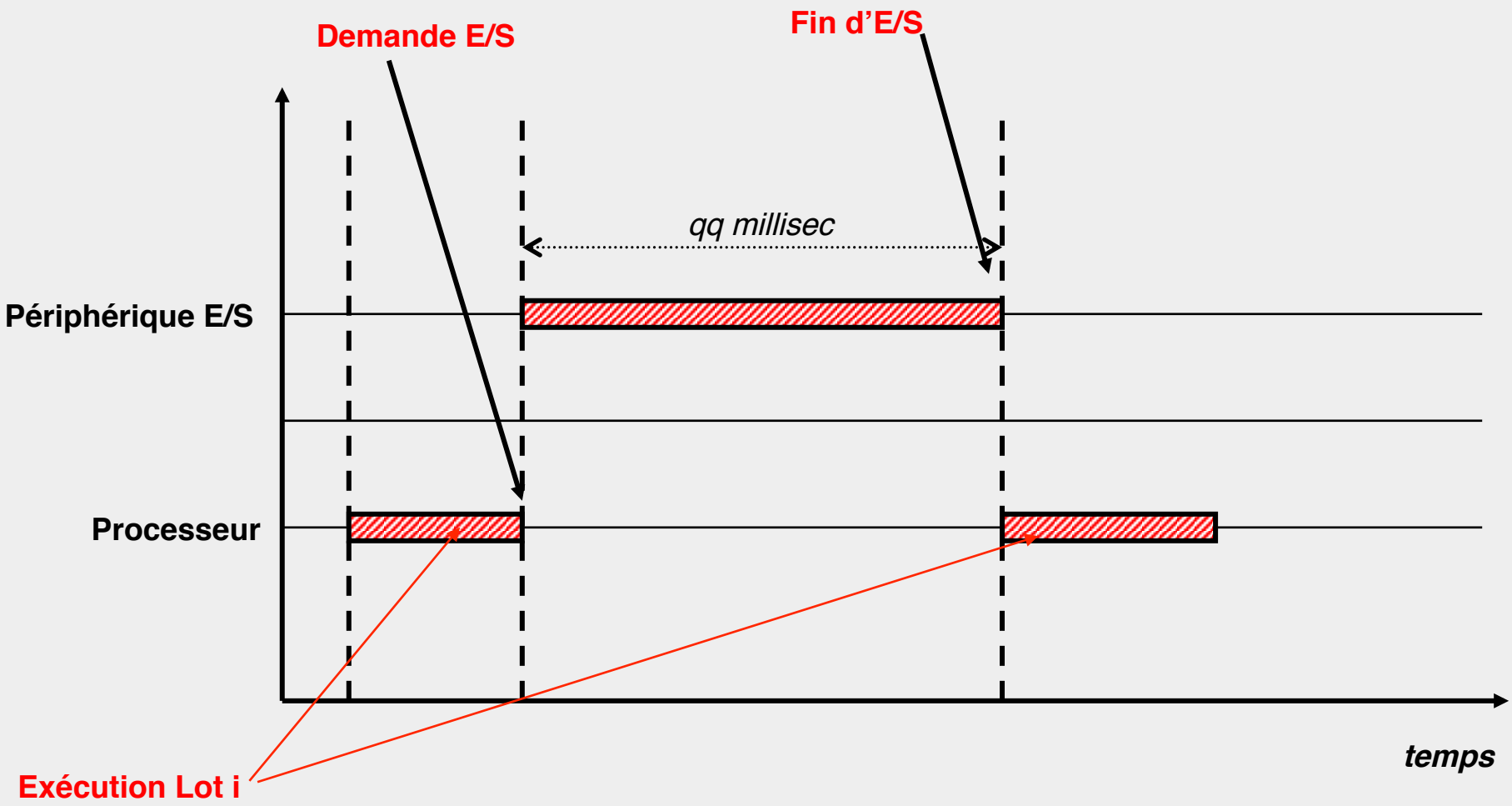
Premières générations de systèmes: mono-programmés

- **Ordinateurs "mainframes"**
  - Traitement par lots (cartes perforées)
  - L'enchainement des lots est décrit par une carte perforée spéciale (carte de controle)
  - Le SE se limite à un moniteur résident qui enchaine les lots

- **Inconvénients**
  - Lent
  - Peu interactif
  - Pas vraiment de concurrence



Mono-programmation



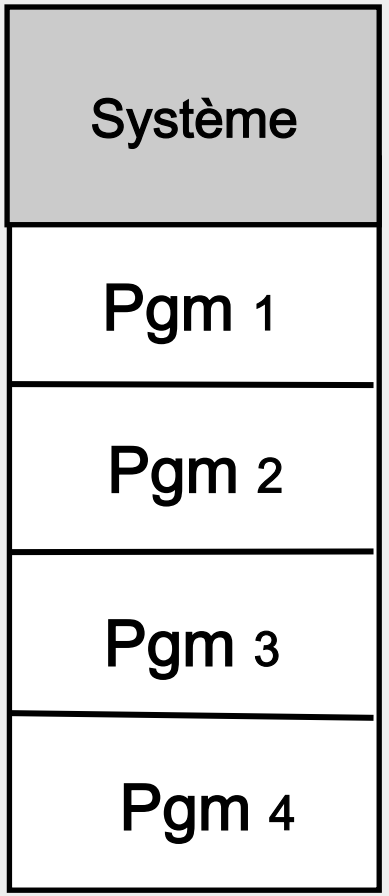
E/S tamponnées

- **Utilisation d'Unités d'Echange (UE) capables :**
  - de fonctionner simultanément avec l'UC (asynchronisme)
  - de lire/écrire dans des tampons (buffers d'octets)

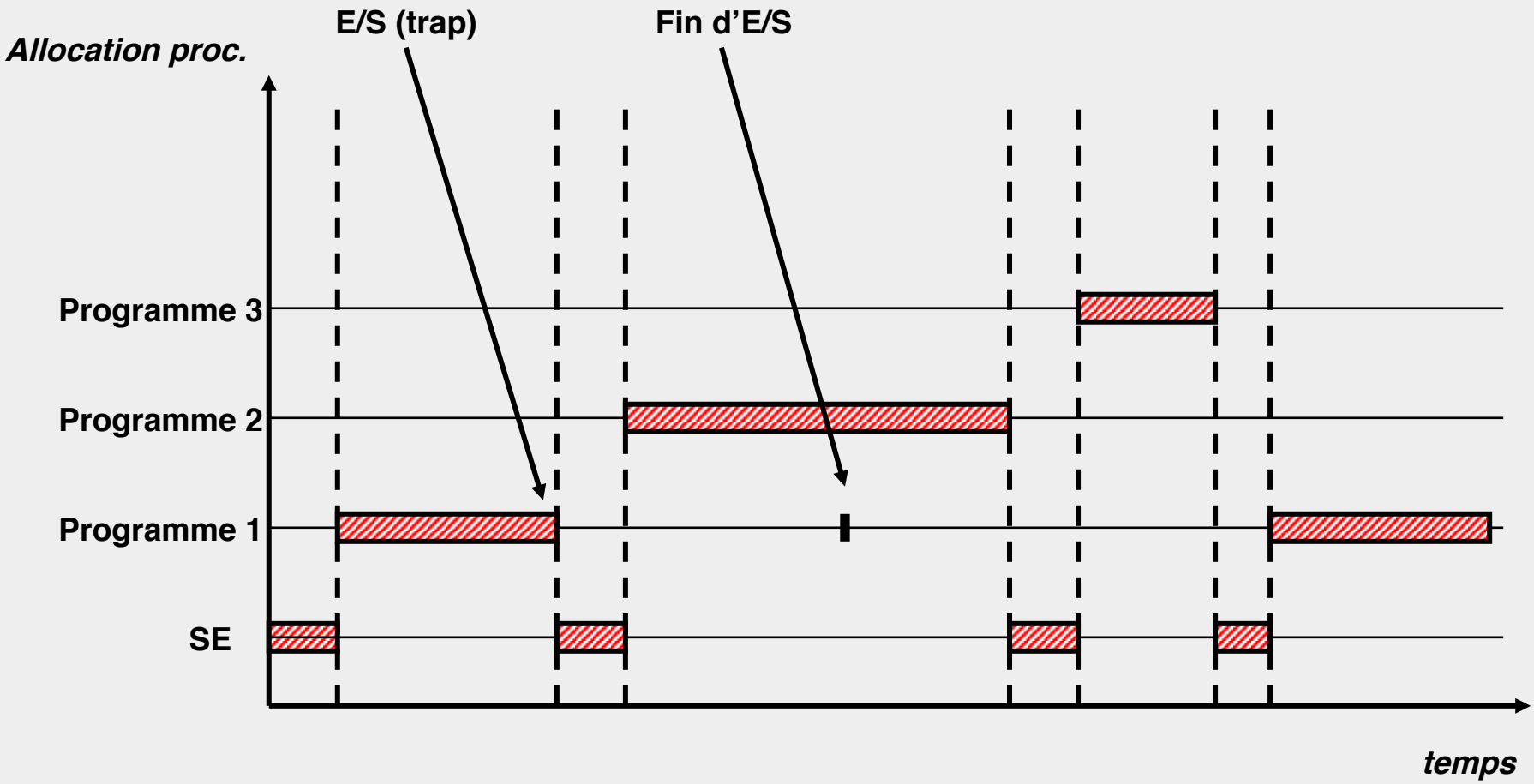
- **Permet de gagner en efficacité**
  - exécution processus $_{i}$ // chargement processus $_{i+1}$
  - Les cartes perforées sont lues par l'UE et stockées dans des tampons d'entrée
  - L'UC lit les données dans le tampon d'entrée, place ces données dans la mémoire, et produit le résultat dans un tampon de sortie

Multi-programmation (1960/1970)

- **Systèmes multi-programmés**
  - Plusieurs programmes en mémoire simultanément
  - Multiplexage du processeur entre les programmes
  - Perte du processeur lors des E/S
- **Avantages**
  - Meilleure utilisation de l'UC
  - On gagne sur les temps d'exécution globaux
- **Inconvénients**
  - Complexité
  - Taille de mémoire grande
  - Partage et protection des ressources



Multi-programmation



Multi-programmation et protection

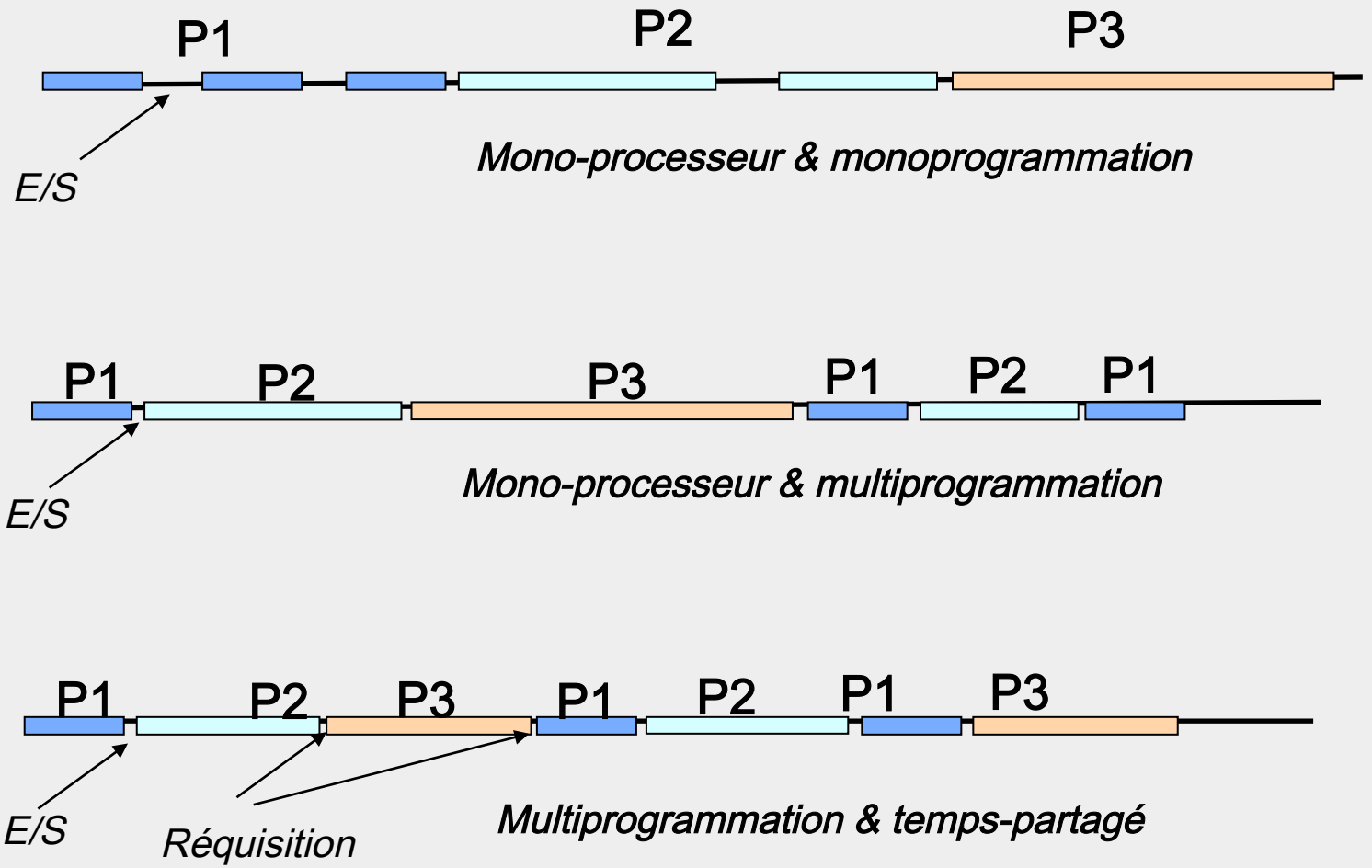
- Eviter qu'un programme en cours d'exécution puisse lire / écrire dans la zone mémoire affectée à un autre programme
- Eviter qu'un programme en cours d'exécution puisse manipuler la zone réservée au système autrement que par les appels système
- Eviter qu'un programme en cours d'exécution puisse lire / écrire dans les buffers d'E/S d'un autre programme

➔ La gestion des ressources (mémoire, E/S, …) devient une tâche complexe pour le système

Temps-partagé (1970)

- **Systèmes à temps partagé**
  - Partage du temps processeur entre les programmes en cours d'exécution (quantum de temps)
  - Processus en mémoire ou "swappé" sur disque
    - Plus grand nombre de programmes en cours
    - Une mémoire plus grande pour chaque programme en cours
- **Avantages**
  - Temps de réponse corrects pour programmes courts, même en présence de programmes longs et non interactifs
- **Inconvénients**
  - Complexité
  - L'utilisation du processeur peut être moins bonne

Systèmes mono/multiprogrammés et temps partagé



2- Notion de processus

- **Concept fourni par le système pour exécuter un programme**
  - On dit aussi qu'un processus correspond à un programme en cours d'exécution
  - Un processus est lancé à la demande du système ou d'un autre processus
  - Il est créé par un processus père (sauf processus initial)
  - Il est identifié de manière unique (*pid*)
  - Il peut être administré (démarré, suspendu, stoppé, ..)
- **Regroupe deux unités**
  - Flôt d'exécution : exécute la suite d'instruction qui compose le programme
  - Espace d'adressage : mémoire manipulable par les instructions exécutées
- **Propriété d'isolation**
  - Isolation des fautes : une faute dans un processus ne peut affecter un autre processus
  - Isolation mémoire : un processus ne peut en aucun cas accéder à l'espace d'adressage d'un autre processus

Echange de données entre processus

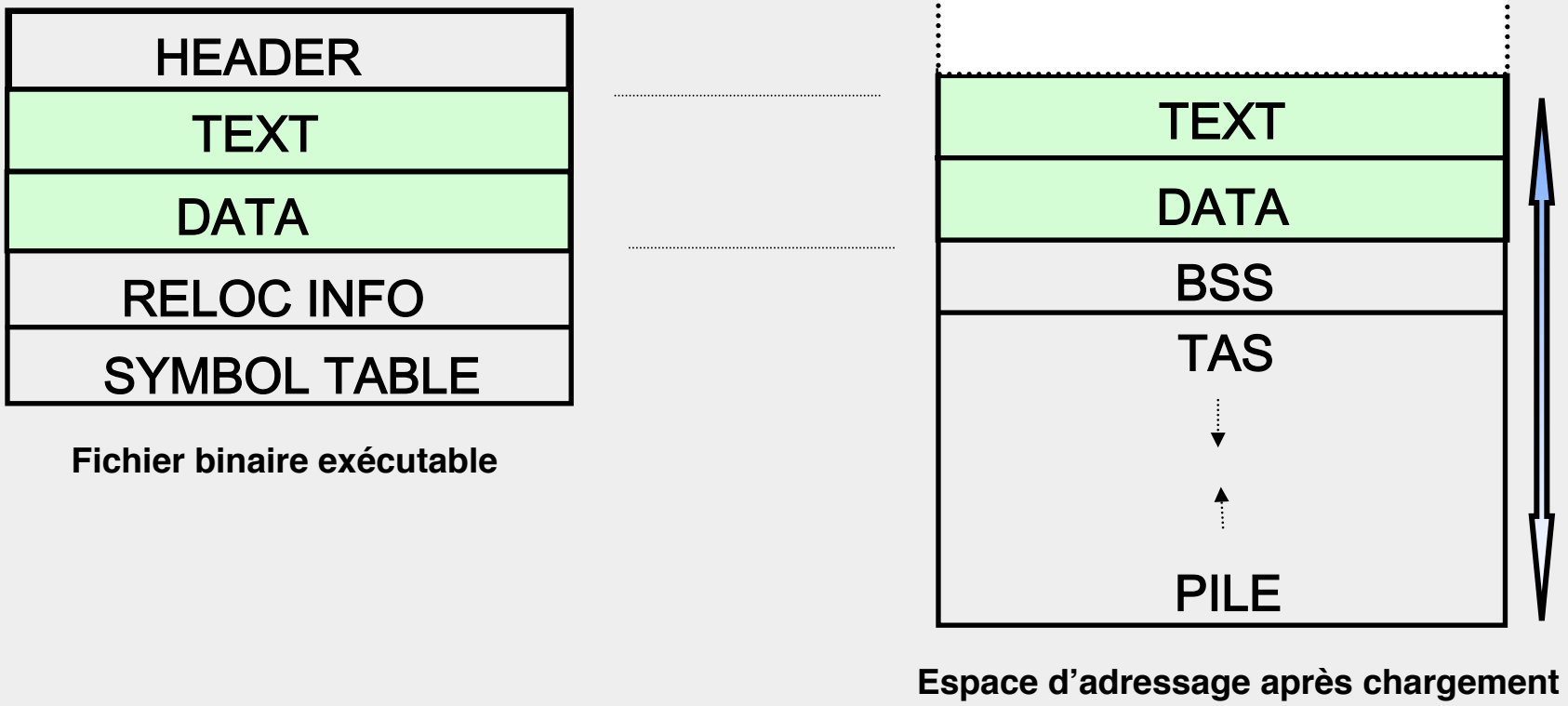
- **Au moment de la création**
  - Héritage de l'état du père
- **Via des messages (stream)**
  - Tubes et queues de message
- **Via des fichiers**

- *Via des segments de mémoire partagée*
  - *Portion partagée de l'espace d'adressage*
  - *Utilisé dans la programmation de niveau système*

(rappel) Exécution d'un programme

- **Lancement du programme via le shell (cas général)**
  - ./mypgm
- **Le système effectue un ensemble d'actions**
  - Allocation d'un espace d'adressage
  - Résolution des liens vers les bibliothèques dynamiques
  - Initialisation des registres
  - Placement des arguments du main sur la pile
  - Lancement de l'exécution
    - Exécution de la fonction _start() qui est dans le crt0.o de gcc
    - _start() appelle la fonction main(..) puis appelle la fonction exit()

(rappel) Espace d'adressage d'un processus



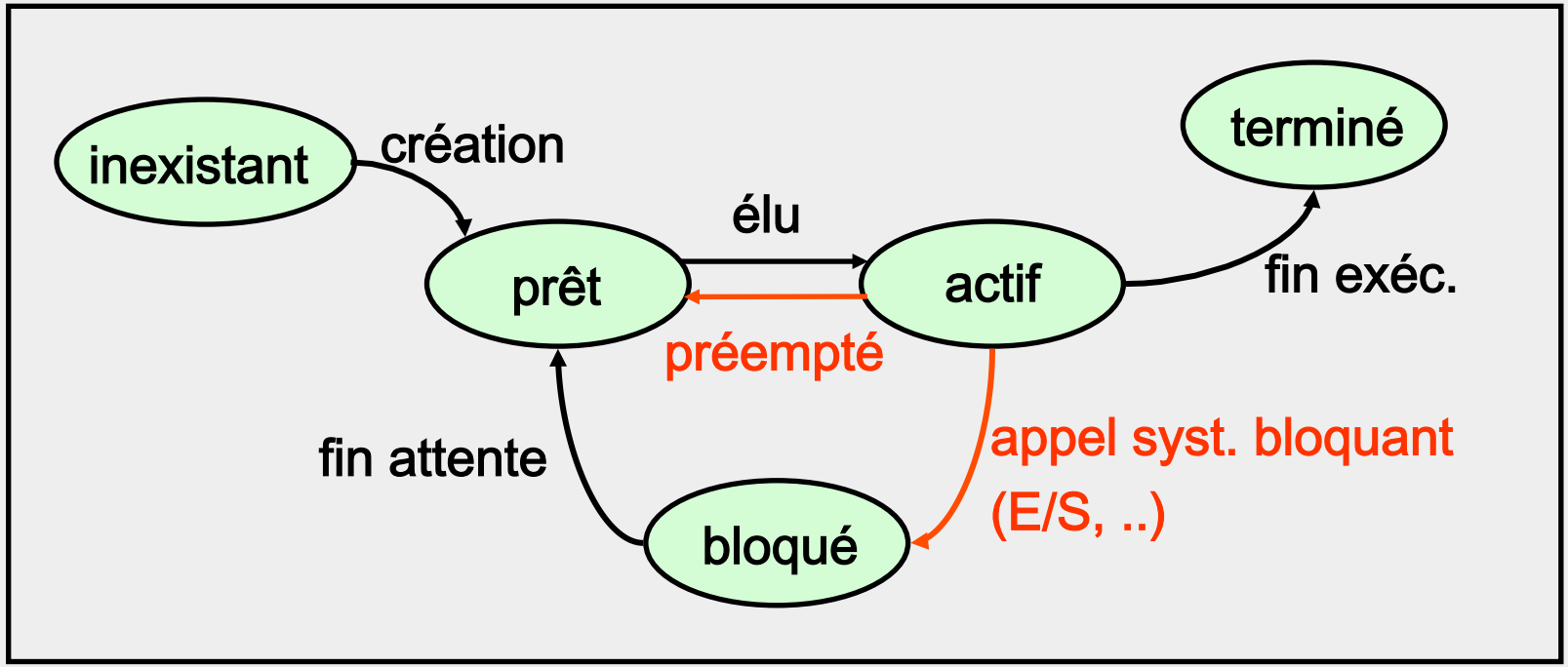
Fichier binaire exécutable

Espace d'adressage après chargement

Modes d'exécution

- **Mode utilisateur**
  - Accès réduit à l'espace d'adressage propre au processus
  - Instructions limitées
- **Mode superviseur**
  - Toutes les instructions autorisées et toute la mémoire accessible
  - Superviseur # SuperUser (shell)
- **Passage du mode utilisateur au mode superviseur**
  - Interruptions
  - Déroutements
  - Appels au superviseur

Cycle de vie



remarque : certains systèmes définissent des états supplémentaires (suspendu, …)

Gestion des processus par le Système

- **des Files de processus**
  - File des prêts (Ready queue)
  - File des bloqués sur E/S (Device queues)
  - File des bloqués sur conditions de synchronisation (Blocked queue)
  - ...
- **Un seul processus actif (hypothèse mono-coeur)**

→ **Le système fait migrer les processus entre les files**

→ **Il prend soin de conserver le contexte d'exécution pour chaque processus**

Contexte d'exécution

- **Etat courant du processus**
  - Etat de l'espace d'adressage
    - Code du programme exécuté
    - Données du programme exécuté
  - Etat des ressources utilisées par le processus
    - Registres (PC, SP, etc.,)
    - Liste des fichiers ouverts (descripteurs)
    - Variables d'environnement
    - …

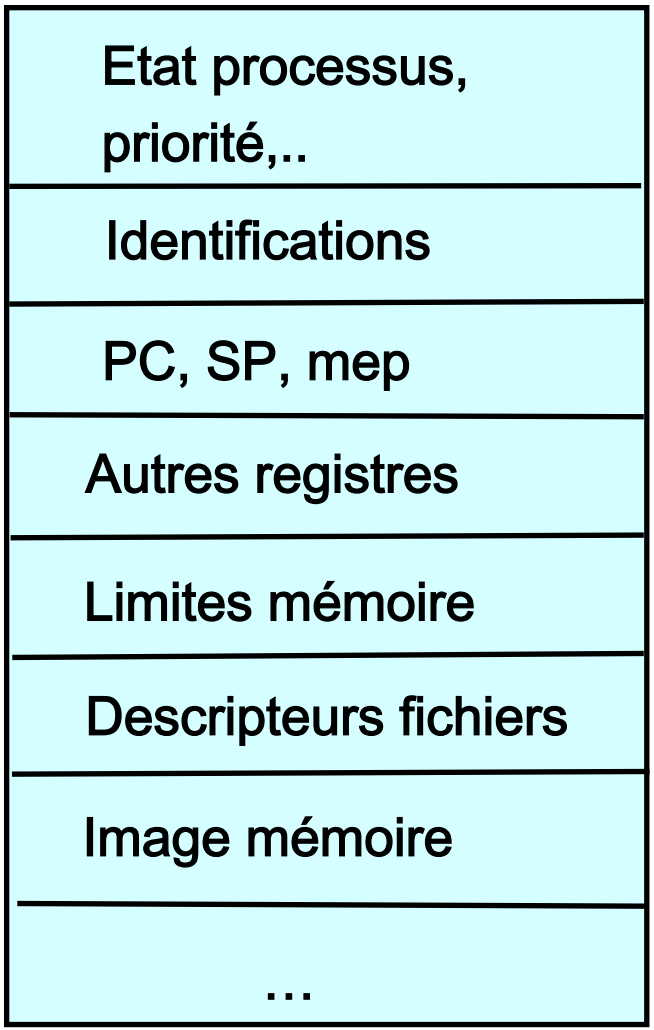
→ **Sauvegardé lorsque le processus est commuté**

→ **Restauré lorsque le processus reprend la main**

Structure de contrôle des processus

PCB (Process Control Block): informations permettant de gérer un processus

Table des Processus:
PCB [MAX-PROCESSUS]

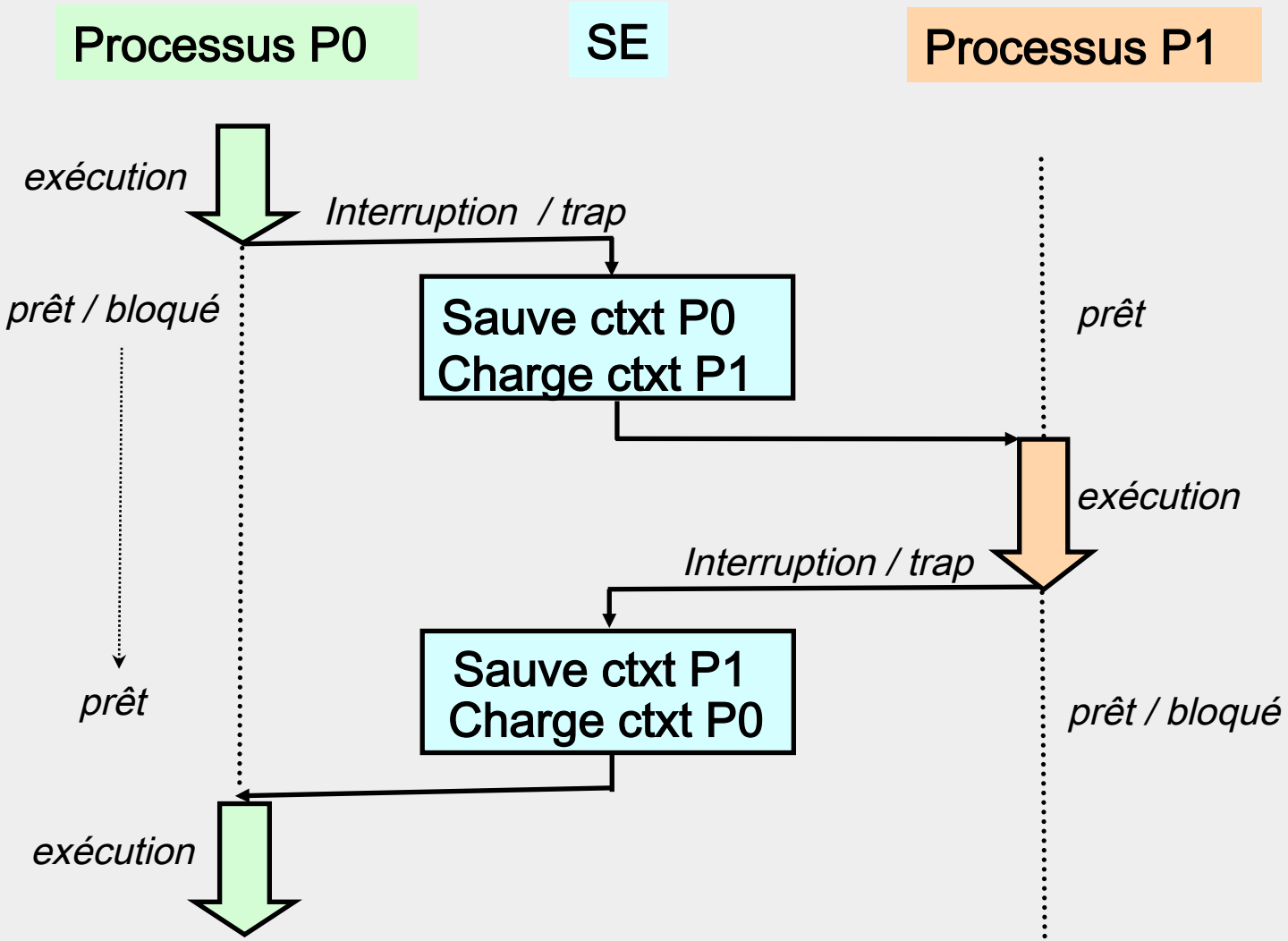


Opérations de base sur les processus

- Création / destruction
- Activation / suspension
- Suspension momentanée (sleep)
- Attente de la terminaison d'un fils

- *Commutation (opération de bas niveau)*

Commutation de processus



Création d'un processus

- **Allocation PID**
- **Allocation image mémoire (clone de l'image mémoire du père)**
- **Allocation + initialisation PCB**
- **Si préemption du père pour le fils**
  - Arrête le père et sauvegarde son contexte dans son PCB
  - Etat du père ← prêt
  - Ajoute le père dans la File des prêts
  - Etat du fils ← actif
  - Donne le processeur au fils (commutation)
- **Sinon**
  - Etat du fils ← prêt
  - Ajoute le fils dans la File des prêts
- **Renvoie le PID du processus créé**

Exercices

- **Exercice**
  - décrire les étapes principales pour les opérations suivantes
    - Destruction
    - Activation / Suspension
    - Suspension momentanée (sleep)
- **Objet de l'exercice**
  - Compréhension de la notion de processus
  - Compréhension du cycle de vie associé aux processus

Allocation du processeur au processus

- **L'ordonnanceur (scheduler) est la partie du système qui gère l'allocation du processeur**

- **Critères**
  - Équitable entre les processus
  - Efficace (usage optimal du temps processeur)
    - Ex: si les processus passent leur temps à commuter, l'usage du temps processeur n'est pas bon
  - Minimisant les temps de réponse des processus
    - Ex: si les processus ne commutent jamais, les temps de réponse seront mauvais
  - Maximisant le rendement (nombre de processus qui progressent par unité de temps)

Paramètres / Stratégies

- **Algorithme avec / sans préemption**
  - Transition *interrompu* du cycle de vie autorisée si préemption (temps partagé) et interdite sinon (multiprogrammation seule)
- **Choix du quantum**
  - Petit ou grand
- **Gestion de priorités**
  - Processus systèmes
  - Processus utilisateurs interactifs
  - Processus utilisateurs peu interactifs

Algorithmes classiques d'ordonnancement

- **Multiprogrammation**
  - FIFO / FCFS
  - PCTE / SJF (Plus Court Temps Exécution)

- **Multiprogrammation + préemption / temps partagé**
  - PCTE / SJF préemptif
  - Tourniquet / Round Robin
  - CFS (linux)

First-Come, First-Served (FIFO)

Processus	Temps d'exécution
P_1	24
P_2	3
P_3	3

- Supposons que les processus arrivent dans l'ordre : P_1 , P_2 , P_3

- Temps de réponse de P_1 = 24; P_2 = 27; P_3 = 30
- Temps moyen : (24 + 27 + 30)/3 = 27

Extrait et traduit de Sylberschatz

First-Come, First-Served

Supposons que les processus arrivent dans l'ordre : P_2 , P_3 , P_1 .

- **Temps de réponse de $P_1 = 30$; $P_2 = 3$, $P_3 = 6$**
- **Temps moyen : (30 + 3 + 6)/3 = 13**
- **Meilleur que le cas précédent**

➔ ***Traiter les processus courts avant les longs***

Extrait et traduit de Sylberschatz

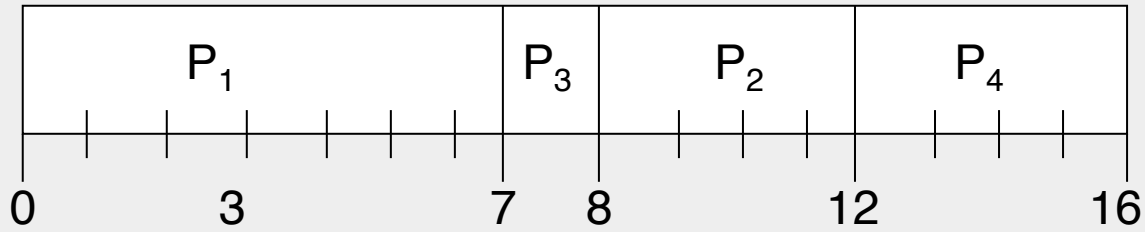
Shortest-Job-First (SJF)

- **Associe à chaque processus la durée de son exécution**
- **Exécute en priorité le processus dont la durée est la plus courte**
- **Deux possibilités**
  - Non préemptif
  - Préemptif (algorithme Shortest-Remaining-Time-First (SRTF))

Non-Preemptive SJF

Process	Arrivée	Temps d'exéc.
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptif)

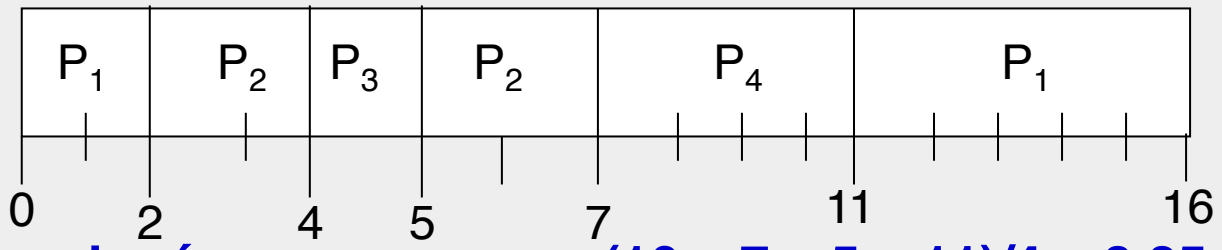


- Temps de réponse moyen = (7 + 8 + 12 + 16)/4 = 10,75

SJF Préemptif

Processus	Arrivée	Temps d'exec.
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptif)



- Temps de réponse moyen = (16 + 7 + 5 + 11)/4 = 8,25

Round Robin

- **Principes**
  - Les processus prêts sont dans une file de type FIFO
  - Lors d'une commutation, le processus le suivant dans la file est élu (*getFirst*())
  - Le processus préempté s'insère en fin de file (*putLast*())
- **Le choix du quantum est l'un des points critiques du Round Robin**

Round Robin avec priorités statiques

- **Principes**
  - Les processus ont des priorités
  - Une priorité est associée à un nombre de quantum (1, 2, 4, ..)
  - Plus la priorité est forte, plus le quantum est petit
  - Le processus le plus prioritaire est élu
- **Evaluation**
  - Bon compromis efficacité / rendement
  - Temps de réponse bons (travaux interactifs prioritaires)
  - Famine possible (les processus de faible priorité peuvent ne jamais s'exécuter)
    - *Aging Solution* ≡ augmenter la priorité d'un processus avec son âge

    → Système à priorités dynamiques

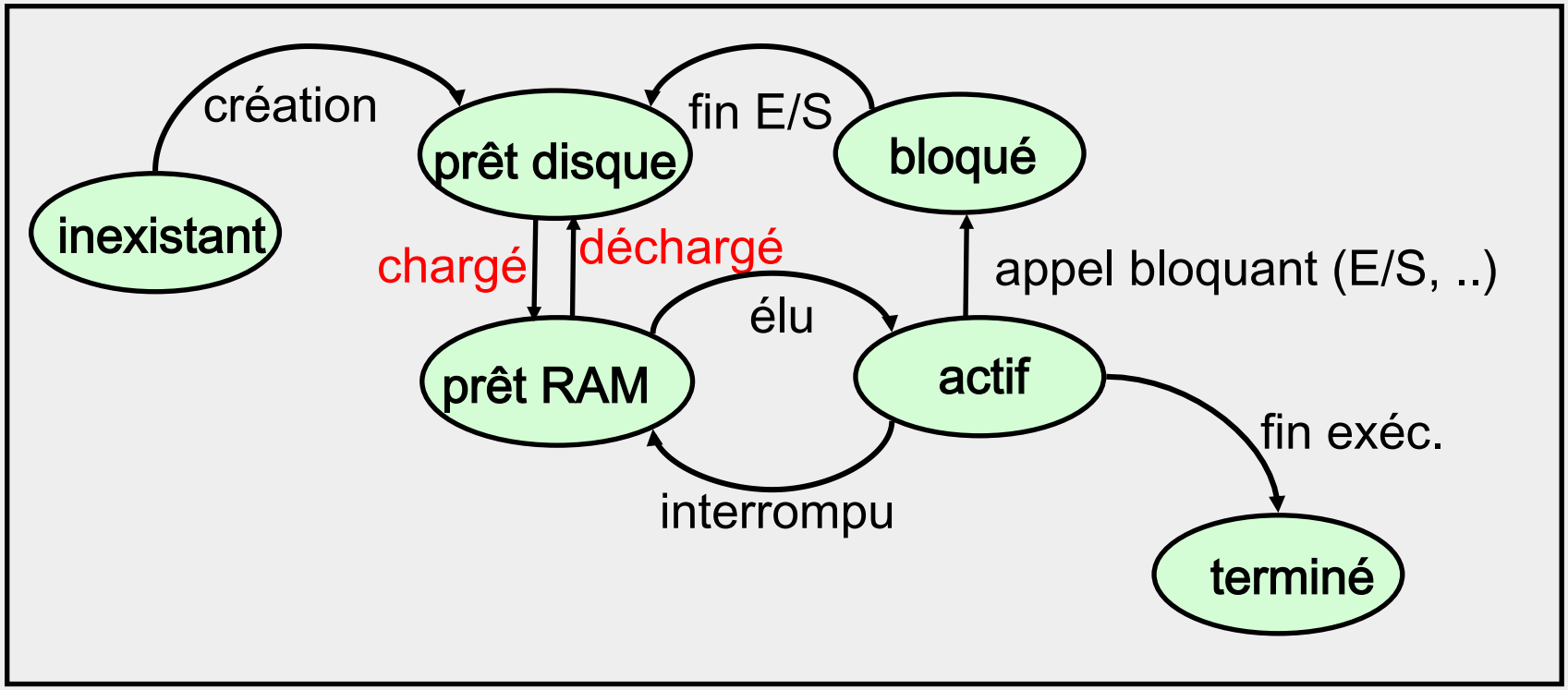
Round Robin avec priorités dynamiques

- **Principes**
  - Déterminer la nature d'un processus (interactif, calcul)
  - Monter la priorité d'un processus s'il est interactif ou âgé
- **Mise en oeuvre**
  - Un paramètre (une durée et un nombre d'interruptions, ou l'âge) permet d'augmenter ou diminuer la priorité des processus
  - Ou bien la priorité p est fonction de la fraction de quantum (f) utilisée (p = 1/f)

Algorithmes d'ordonnancement à plusieurs niveaux

- **Ensemble des processus prêts trop important pour tenir en mémoire centrale**
  - Certains sont déchargés sur disque, ce qui rend leur activation plus longue
  - Le processus élu est toujours pris parmi ceux chargés en mémoire
- **En parallèle, on utilise un deuxième algorithme d'ordonnancement pour gérer les déplacement des processus prêts entre le disque et la mémoire centrale**

Cycle de vie avec deux niveaux d'ordonnancement



RAM = Random Access Memory

Notion de Processus – points clés

- **Programme en cours d'exécution auquel est associé un contexte**
  - Contexte processeur: registres, mep, ..
  - Contexte mémoire: mémoire adressable, fichiers ouverts, ..
  - Le contexte est sauvegardé / restauré lors des commutations
  - Les commutations sont liées à la multi-programmation et au temps partagé

→ **Moyen d'obtenir du (pseudo) parallélisme**

→ **Moyen d'obtenir de l'isolation (mémoire et fautes)**

Aujourd'hui, on utilise des processus pour des raisons d'isolation

Si l'on veut du parallélisme sans isolation, on se tourne vers les threads

Usage des processus : exemple du Shell

- **Un processus qui crée d'autres processus pour exécuter des commandes (sauf pour les commandes *built-in)***

- **Les variables d'environnement du shell sont initialisées via des fichiers de configuration**
  - Fichiers typiques : .login, .logout, .cshrc ou .bashrc, ..
  - Variables typiques : HOME, PATH, TERM, CURRENT-DIR

- **Les variables d'environnement peuvent être modifiées et d'autres variables peuvent être créées pour le shell courant**
  - *setenv VAR value* (VAR=value en bash)
  - *Ou bien modifier le fichier de configuration et le « resourcer » pour le prendre en compte dans le shell courant (e.g., source ~/.cshrc ou . ~/.cshrc)*

- **Les variables sont héritées par un shell fils**
  - Mais elles sont « écrasées » par la relecture des fichiers de configuration
  - Sauf pour les variables exportées (visibles par les shells secondaires)
  - *Ex dans bash : export VAR*

Exemple du shell

xterm 1

[initialisation automatique des variables d'envt à partir du fichier .bashrc]
➢**env**
USERNAME=paul
PATH=.:/usr/local/bin:/usr/bin:/sbin:/bin
PWD=/home/paul
➢**PATH=$PATH:/usr/lib/jvm/jdk-8/bin**
➢**CLASSPATH=.**
➢**env**
USERNAME=paul
PATH=.:/usr/local/bin:/usr/bin:/sbin:/bin:/usr/lib/jvm/jdk-8/bin
PWD=/home/paul
CLASSPATH=.
>xterm &

.bashrc

USERNAME=paul
PATH=.:/usr/local/bin:……
PWD=/home/paul
…..

xterm 2

[initialisation automatique des variables d'envt à partir du fichier .bashrc]
➢**env**
USERNAME=paul
PATH=.:/usr/local/bin:/usr/bin:/sbin:/bin
PWD=/home/paul

CLASSPATH et PATH ne sont pas hérités dans le shell fils
Il aurait fallu les exporter dans le père pour ce faire

3- Notion de thread

- **Flot d'exécution "léger"**
  - Contexte allégé
    - Une partie partagée : mémoire adressable, fichiers ouverts, …
    - Une partie propre : pile, registres

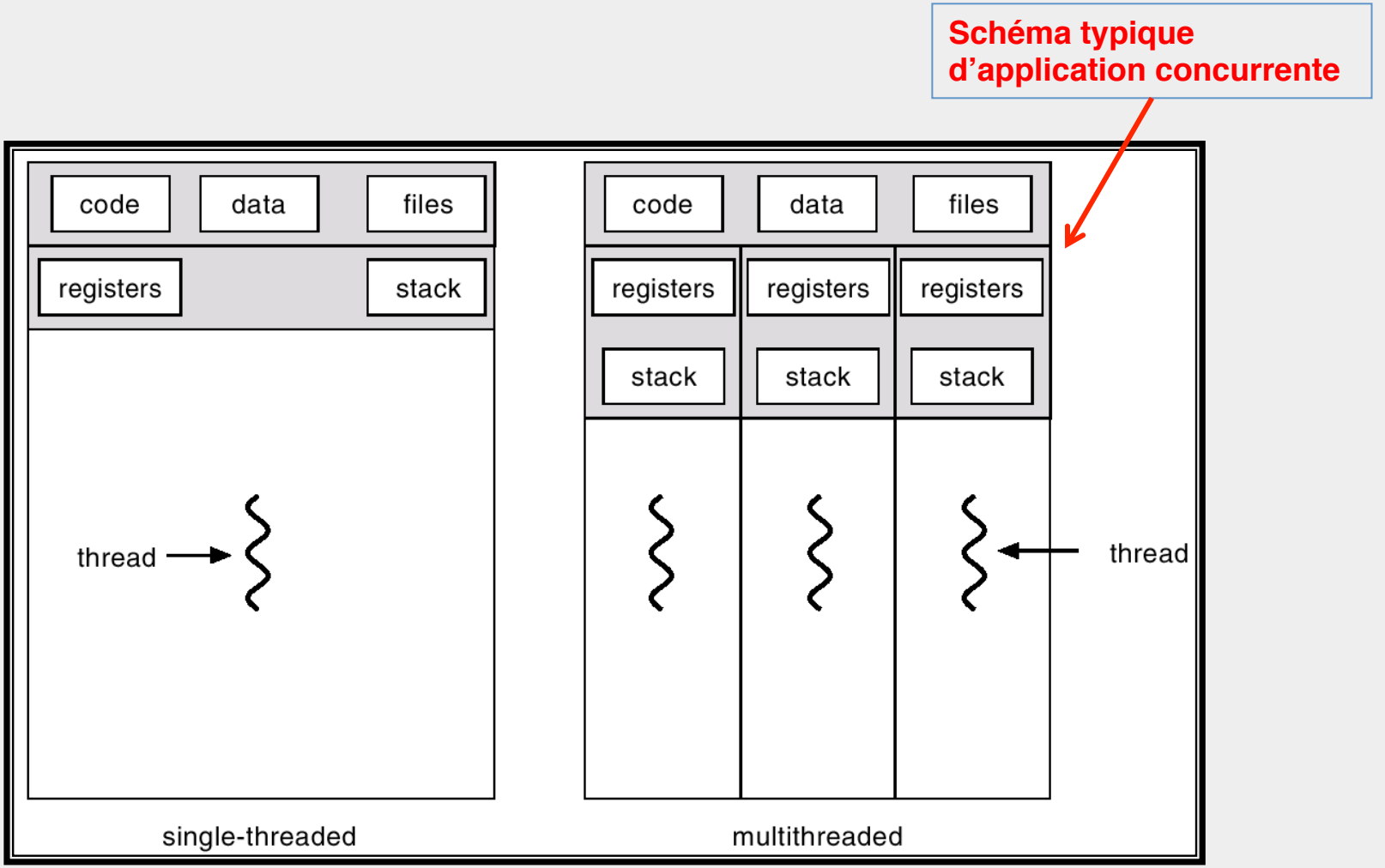
- **Commutations plus rapides**

→ **Efficacité d'exécution des applications concurrentes**

- **Partage de mémoire**

→ **Facilité de programmation des applications concurrentes**

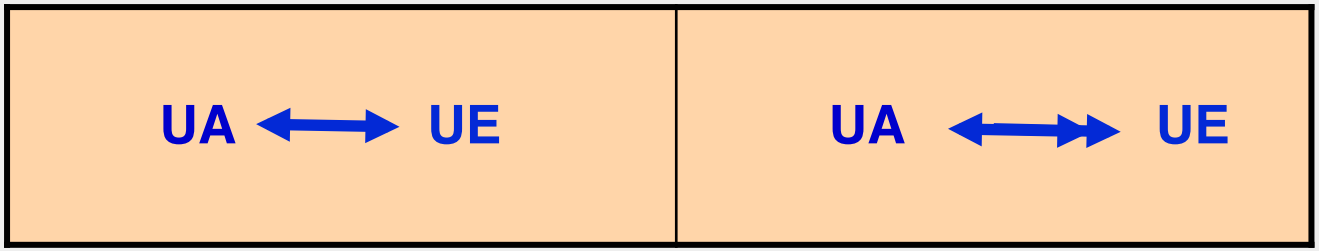
Processus single-threaded et multi-threaded



A.Sylberschatz

Historique des threads

- Unité d'adressage (UA)
- Unité d'exécution (UE)
- 2 notions historiquement couplées avec les processus, dissociées avec les threads



Juste pour concrêtiser : exemple basique (pthreads)

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

static void *task_a (void *p_data) {
   puts ("A  says: Hello world ");
}

static void *task_b (void *p_data) {
   puts (”B says: Hello universe");
}
```

```
int main (void) {
   pthread_t ta;
   pthread_t tb;

   puts ("main init");
   pthread_create (&ta, NULL, task_a, NULL);
   pthread_create (&tb, NULL, task_b, NULL);

   pthread_join (ta, NULL);
   pthread_join (tb, NULL);

   puts (“That’s all folk guys");
   return NULL;
}
```

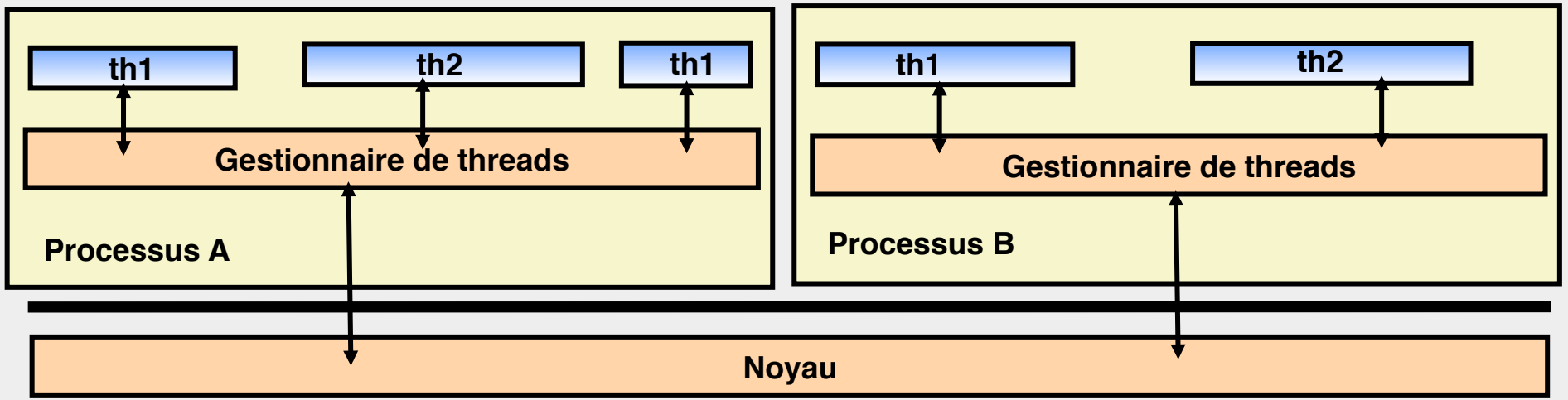
Types de threads

- Threads *User-level*
- Threads *Kernel*
- Solutions mixtes

Threads *user-level*

- Code de gestion des threads dans une librairie
- Pas de modification du noyau
- Le gestionnaire de threads ainsi que les threads s'exécutent au sein d'un processus utilisateur

Ex: Mach C-threads, Solaris threads, Threads Java (initialement)



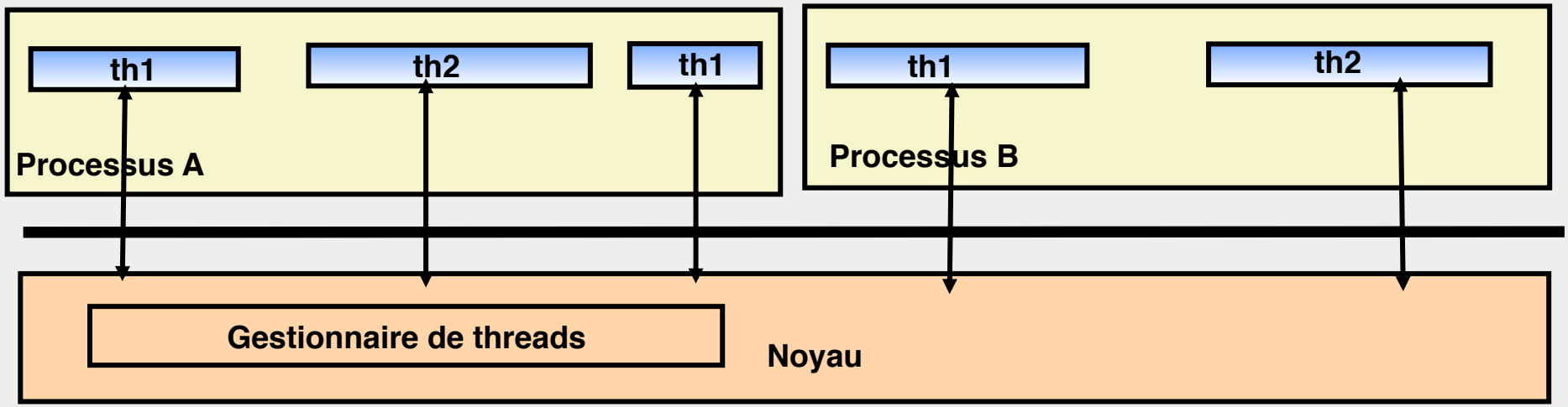
Avantages et inconvénients des threads
User-level

- **Efficacité (+)**
  - La commutation de contexte est rapide
- **Parallélisme (-)**
  - Pas de parallélisme réel entre les threads d'un EV
- **Appels systèmes bloquants (-)**
  - Le processus est bloqué au niveau du noyau
  - Tous les threads sont bloqués tant que l'appel système (ex: I/O) n'est pas terminé
  - Il n'y a donc pas de multiprogrammation dans l'application concurrente

Threads *kernel-level*

- **Notion de thread gérée par le noyau**
- **Lorsqu'un thread se bloque, le noyau alloue le processeur à un autre thread**

Examples: Windows 95/98/NT/2000, Solaris, Tru64 UNIX, Linux, Threads Java (HotSpot)



Avantages et inconvénients des threads
Kernel-level

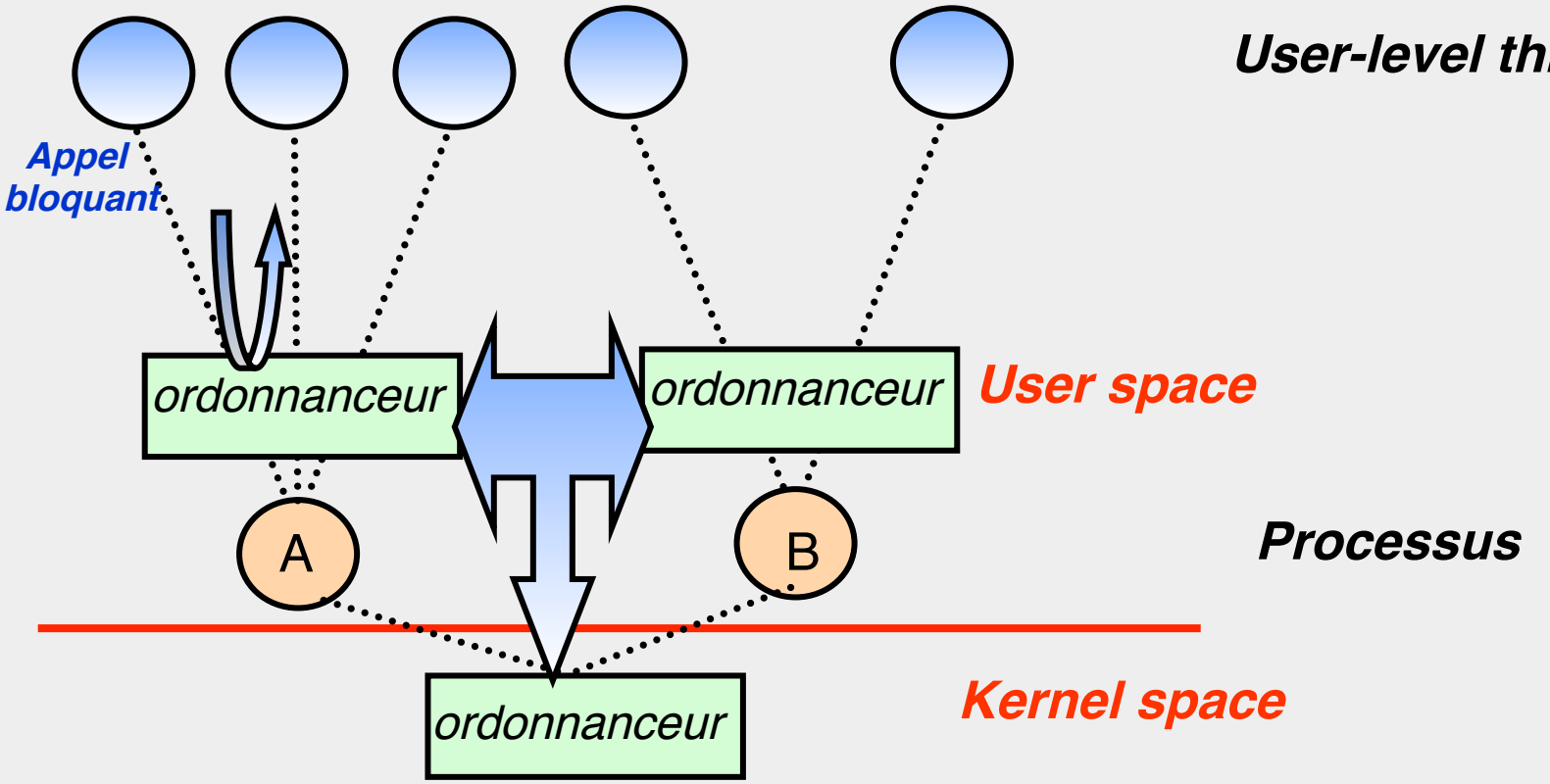
- **Appels systèmes bloquants (+)**
  - Pas de blocage des threads d'une application concurrente lors d'un appel système
- **Parallélisme réel (+)**
  - N threads d'une application concurrente peuvent s'exécuter sur K processeurs
- **Efficacité (-)**
  - Commutation plus chère que pour les threads User-level, car demande un passage en mode noyau.

Solutions hybrides permettant une commutation légère

- **Principes des threads User-level (commutations gérées au niveau utilisateur)**

- **Modification du noyau pour gérer les appels systèmes de manière à ne pas bloquer les threads**
  - Lorsqu'un thread fait un appel système bloquant, le noyau ne préempte pas le processeur
  - Un mécanisme de signaux permet de gérer la fin de l'appel bloquant
  - L'ordonnanceur est départagé entre l'espace système et l'espace utilisateur (ordonnanceurs coopérants)

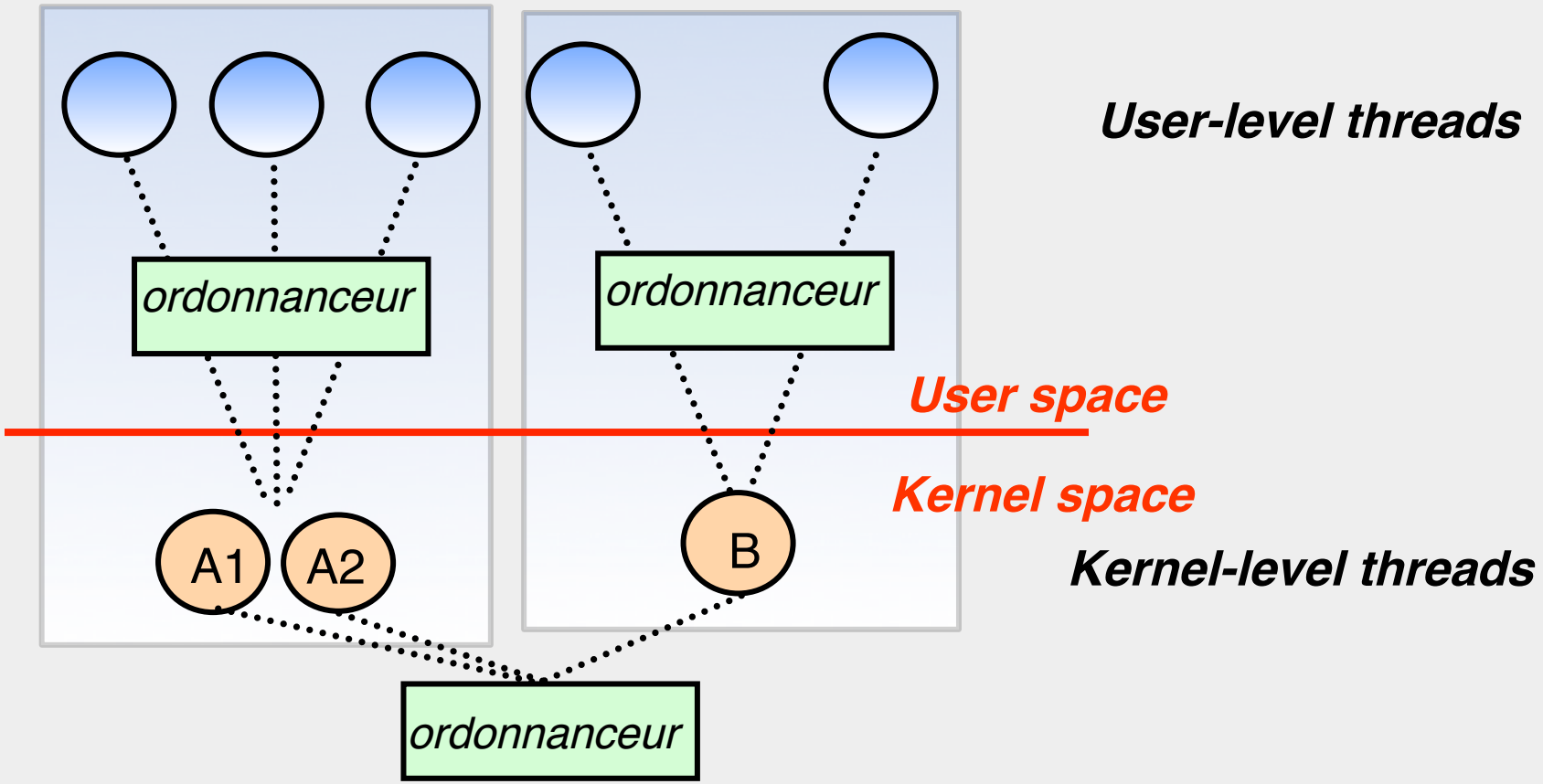
Solutions hybrides permettant une commutation légère



Autres solutions hybrides permettant un parallélisme réel

- **On utilise les deux types de threads**
- **Les threads Kernel-level fournissent la capacité de parallélisme**
- **Les threads User-level fournissent la capacité d'avoir des commutations légères**
  - On peut associer plusieurs threads User-level à un thread kernel-supported
  - Appels systèmes bloquants gérés ou non par le système

Autres solutions hybrides permettant un parallélisme réel

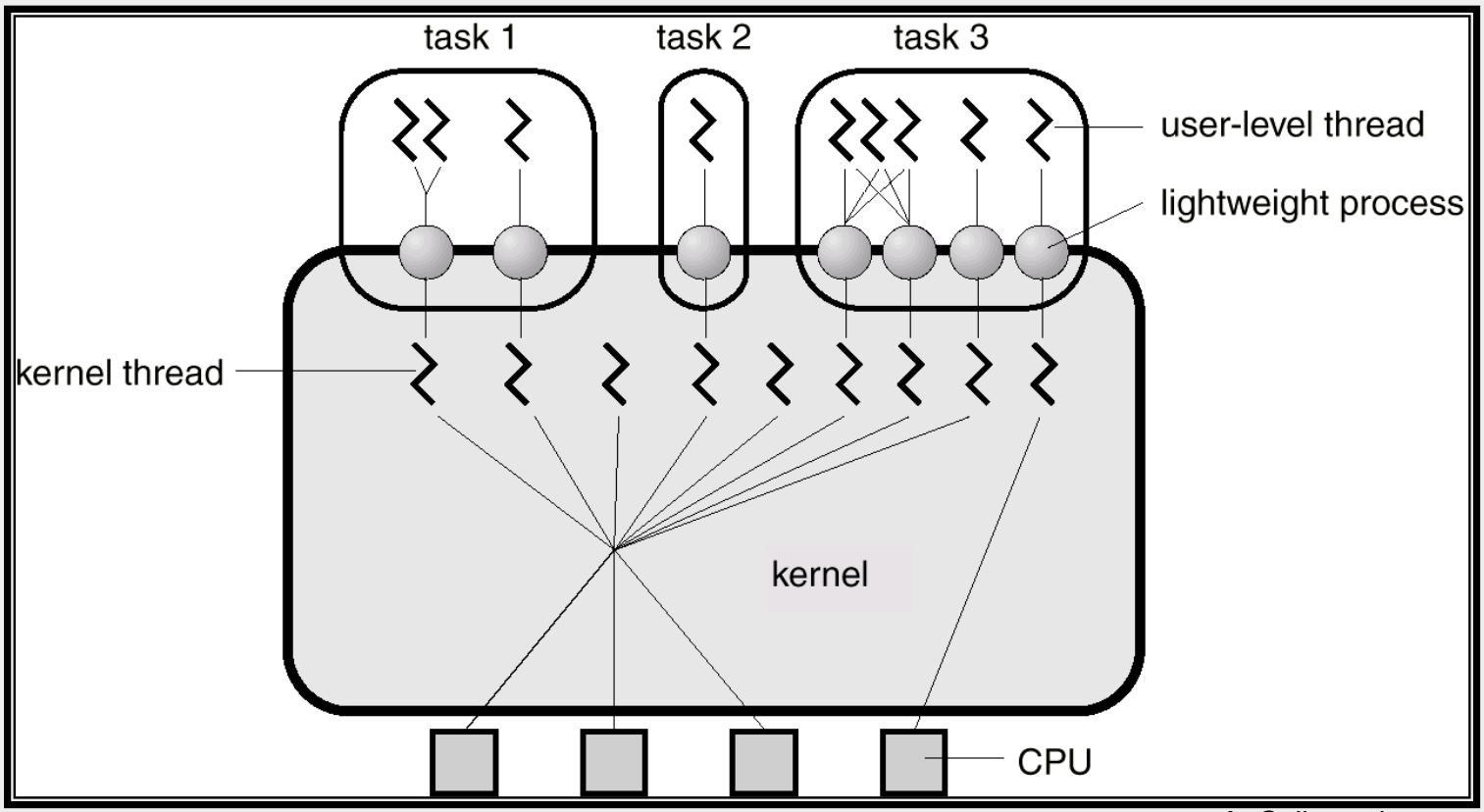


Modèles de Threads

- **Many-to-One**
  Un thread noyau ←→ plusieurs threads utilisateurs
- **One-to-One**
  Un thread noyau ←→ un thread utilisateur
- **Many-to-Many**

  Pool de threads noyau ←→ Pool de threads utilisateurs

Modèle de threads Many-to-Many



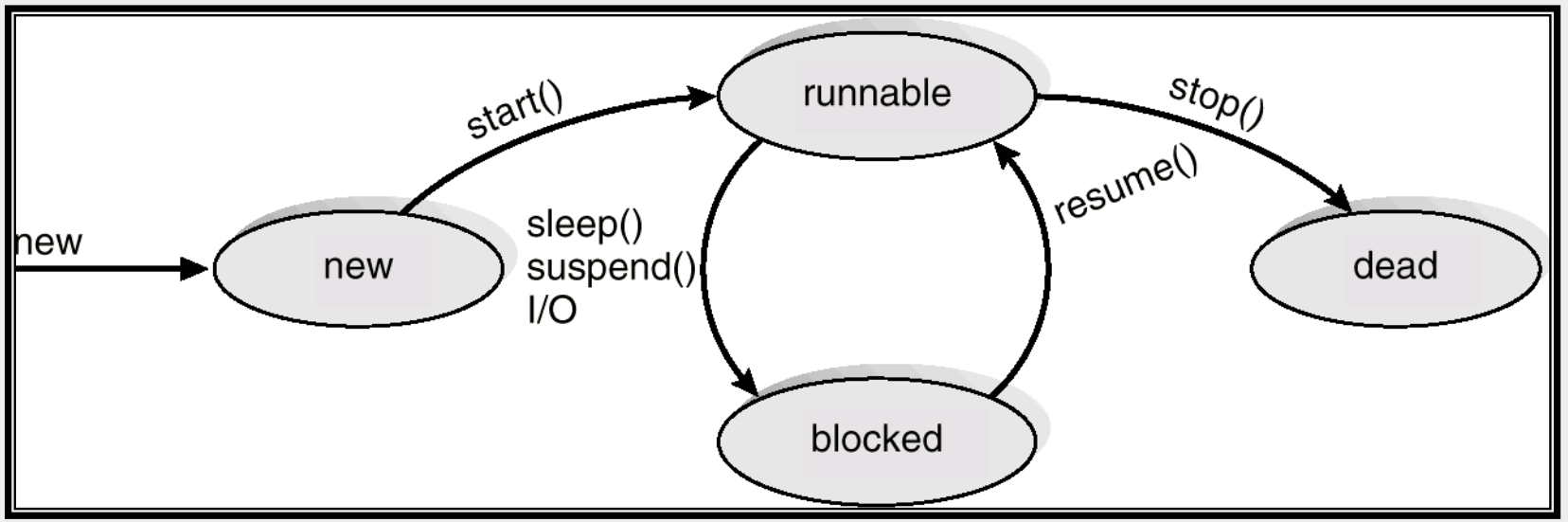
A. Sylberschatz

Principales familles de threads

- **Systèmes Unix**
  - POSIX Threads (IEEE POSIX 1003.1c-1995 standart) (appelés Pthreads, threads kernel)
  - DCE Threads
  - Solaris Threads
- **Microsoft-style threads (PCs)**
  - Win32 (Microsoft Windows 95 and Windows NT)
  - OS/2 threads (IBM)

Threads Java

- User-level / Kernel-level, selon l'implémentation de la JVM (généralement kernel-level)
- Fixed Priority Scheduling (round-robin basé sur priorités)



Threads Java

- **Deux méthodes de création**
  - Par héritage (extension de la classe Thread)
  - Par délégation (implémentation de l'interface Runnable)

    ```
    interface Runnable  { public abstract void run(); }
    ```

- **Dans les deux cas**
  - Définir la méthode *run()* qui définit les instructions exécutées par le thread

Définition et création de threads Java par héritage

```
// Defining a thread class
public class MyThread extends Thread {
        public void run() {
            <code à exécuter>
    }
    …
```

```
// Creating and starting a thread of class MyThread
….
MyThread thread1 = new MyThread();
thread1.start();
….
thread1.join();
…
```

Définition et création de threads Java par délégation

```
// Defining a thread class
public class MyThread implements Runnable {
        public void run() {
            <code à exécuter>
    }
    ...
```

```
// Creating and starting a thread of class MyThread
....
Thread thread1 = new Thread(new MyThread);
thread1.start();
....
thread1.join();
...
```

Définition et création par délégation via une classe anonyme

```
...
new Thread(
  new Runnable() {
      public void run() {
              <code à exécuter>
      }
  }).start();
...
```

Exemple

```
Collection c;
Thread[] threads = new Thread[..];
…
for (final Object o : c) {
   threads[i] = new Thread(new Runnable(){
            public void processObj(Object o) {
                     ...
            }
            public void run(){
                     processObj(o);
            }
     };
   threads[i++].start();
}
for (int j=0; j<l; j++)
            threads[j].join();
System.out.println("done");
…
```

Manipulation de threads Java

- **Des méthodes permettent de gérer le cycle de vie des threads**
- **Méthodes d'instances et des méthodes statiques**
- ***Have a look at javadoc* (java.lang.Thread)**

```
public class Thread implements Runnable {
 …
public Thread();
public Thread(String name);

public static Thread currentThread();
public static void sleep(long ms);
…
public void start();
public void join ();
…
```

Que fait un thread tout au long de sa vie ?

- **Il invoque des méthodes sur des objets (instances ou classes)**
- **Un objet est forcément crée par un thread**
- **Un thread manipule les objets qu'il a crée ou dont il a obtenu la référence**
  - Il suffit de partager un objet pour pouvoir échanger des données (références ou valeurs primaires)
  - Lors de leur création, les threads d'une même application peuvent recevoir en argument la référence d'un objet partagé initial

Terminaison d'un thread Java

- **Naturelle**
  - Le thread a fini d'exécuter sa dernière instruction
- **Par interruption**
  - Invocation de la méthode interrupt() sur le thread
  - Si celui-ci est dans une méthode bloquante, il sort de la méthode avec une *InterruptedException*
  - Sinon, interrupt() ne fait que positionner un drapeau (flag) dans le thread
  - Le thread concerné est censé tester régulièrement ce drapeau :

    if (Thread.interrupted()) …

Terminaison d'un programme Java

- **Quand tous les threads ont terminé leur exécution**
  - A l'exception des threads de type démon (méthode setDaemon() fournie par la classe Thread)
  - Concept fourni pour faciliter la gestion des threads qui exécutent une boucle infinie

```
// Example of a thread that prints the CPU usage every 10 ms
public void run(){
    while(true){
       printCPUUsage();
       sleep(10);
}
...
```

Point crucial à retenir

- **Une application concurrente (Java)**
  - Est composée d'un ensemble de threads qui partagent des objets
  - Ces threads Java sont gérés en multi-programmation et temps-partagés, ils peuvent donc être commutés n'importe quand
- **Il faut donc faire attention**
  - Un objet peut devenir incohérent si deux threads le modifient en parallèle