## Légende

SE : Système d’exploitation. E/S : Entré/Sortie. EXS : Exception système. DMA : Direct Memory Access. CPU : Central Processing Unit. PC : Program Counter. Pg : Programme. Pu : Processus. PCB : Pu control block. ORD : Ordonnanceur. PID : Pu Id. IPC : Interme-diary Pu Communications. MPC : Multiprocessors. TD : Thread. SJF : Shortest Job First. SRTF : Shortest Remaining Time First. SC : Section critique. FIFO : First in first out. IB : Interblocage.

*AVs* : Avantages – *DAVs* : Désavantages – Mots surlignés : Données manquantes.

Introduction

## Interruptions

Un SE est guidé par les interruptions (*trap* ou E/S) – Détermine quel type d’interruption par l’usage de polling ou de vecteurs d’interruption.

Après une E/S, le système retourne seulement quand la E/S est complétée : boucle d’attente active, une seule E/S à la fois – Retourne au Pg sans attendre la fin de l’E/S (requêtes au SE pour la suite, attend le OK du SE).

Gestion de la mémoire : stockage temporaire des données, mise en cache, utilisation spool.

## Stockage

Mémoire centrale : seul stockage auquel le CPU peut accéder directement (accès aléatoires, volatile) – Mémoire de masse : extension de la mémoire centrale, grande capacité, pas volatile – SSD – Caching : Accès aux données récurrentes rapidement en les stockant dans un espace dédié, coût moindre, complexité gestion – Registres.

## Architecture des systèmes

Un CPU – Multiprocesseurs (multi-core): plus haute performance, moindre coût, meilleure fiabilité. Deux types : asymétrique (un CPU principal gère plus les autres), symétrique (le plus populaire) – Systèmes en groupes (clusters) : comme multiprocesseurs, mais là, nous utilisons une architecture multisystèmes. Plus efficace, tolérance aux pannes, etc.

## Structure des SE

Multiprogrammation : organise le travail que le CPU ait toujours quelque chose à faire, sous-ensemble des tâches gardé en mémoire, choix d’exécution des tâches, parallélisme. Multitasking : plusieurs tâches prêtes en même temps. Interblocage, mémoire virtuelle (tâches partiellement en mémoire), swapping.

## Transition au mode noyau

Si E/S, passe en mode noyau. Utilisé afin d’éviter les boucles infinies. Repasse en mode utilisateur lorsque l’E/S a été gérée. Fonctions noyau : clear memory, turn off interrups, switch from user to kernel, set value of timer...

## Gestion des processus (intro)

Un programme est une entité passive. Un processus est une entité active.

Single-thread : PC qui indique l’adresse de la prochaine instruction. Multi-threads : Plusieurs PC.

SE est responsable de créer, détruire, suspendre, réveiller des Pu et doit fournir des mécanismes de synchronisation et de communication entre les Pu.

## Gestion de la mémoire (et +)

SE doit optimiser l’usage du CPU et le temps de réponse (gestion des données efficace) – SE monitore les accès mémoire (trace, allouer et désallouer les espaces mémoire) – SE fournit une vue abstraite des données (répertoires, fichiers, etc.) + gestion de cette vue. SE gère la sécurité (permissions).

## Services offerts par le SE

Exécution d’un Pg – I/O Opérations – Manipulation du système de fichiers – Communications entre différents Pu– Détection d’erreurs – Allocation de ressources – Comptabilité (garder une trace des opérations effectuées) – Protection et sécurité.

## Types d’interfaces

CLI (command line) – GUI (graphical user interface) – Appel système (CLI) : Pg avec paramètres disponibles à travers une API. Beaucoup de types de Pg – Pg système : Création, affichage, modification et suppression de fichiers (mkdir, vim, etc.). Compilateurs, assembleurs, interprètes. Services background. Applications.

## Différentes structures du SE

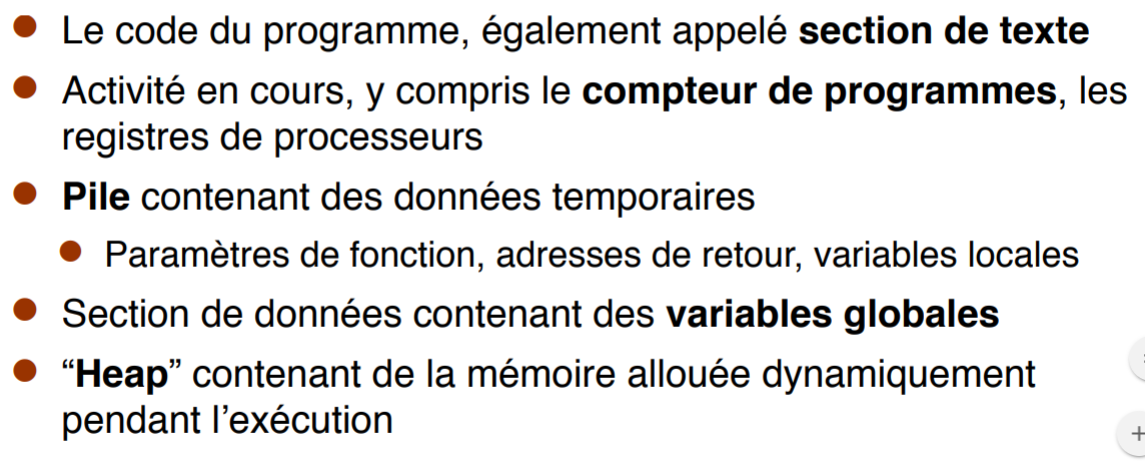
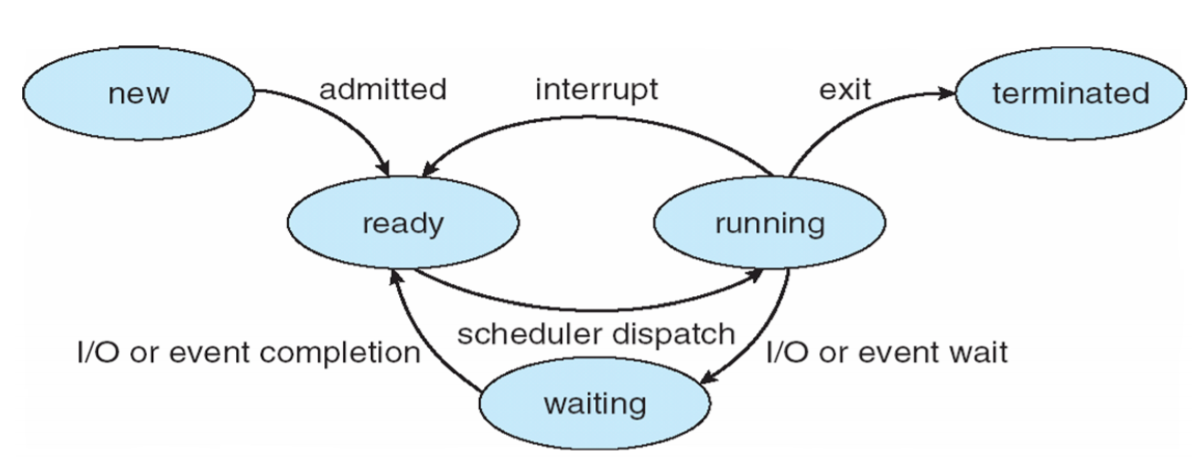
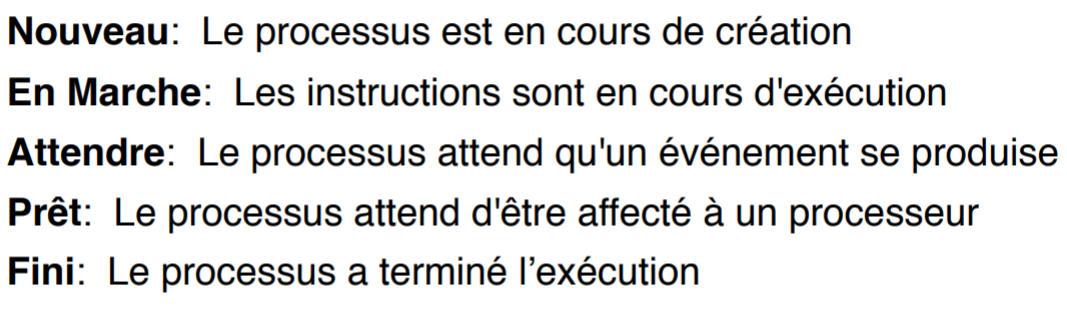
MS-DOS : Simple, non divisé en modules, fonctionnalités pas bien séparées – Unix : deux parties [Pg systèmes, Noyaux : entre système et matériel physique, fournit le système de fichier, la gestion de la mémoire, un grand nombre de fonctions, très peu de frais généraux dans l’interface d’appel système].

Approche en couches : Couche 0 (matériel), couche N (UI). Toutes les couches utilisent que des fonctionnalités inférieures – Microkernel: Commu-nications au niveau du noyau ET des modules de l’utilisateur. Mac OS X. *AVs* : Plus facile d’étendre un micro-noyau, plus facile de porter le SE à de nouvelles machines, plus fiable (moins de code s’exécute en mode noyau), plus sécurisé. *DAVs* : Espace utilisateur doit être plus performant pour permettre la communication entre modules – Modules : Chaque composant est séparé et peut être chargé (au besoin) dans le noyau (Linux, solaris, etc) – Systèmes Hybrides : SE modernes ne sont pas « purs », combine plusieurs approches.

## Débogage du système exploitation

Les SE génèrent des fichiers journaux contenant des informations d'erreur. Kernighan’s Law : Le débogage est 2x plus difficile que coder, alors si tu codes à ton max, tu n’es pas assez intelligent pour déboguer ton code.

Processus

Pu Job tâche. Pg (passif), Pu (actif). Pg devient Pu lorsque chargé en mémoire.  
   
  
  
Contenu du PCB : État du Pu, PC, Registres du Pu, Informations Pu (priorité, mémoire allouée, temps écoulé, fichiers ouverts, CPU utilisé, etc)

## Gestion des Pu

Changement de Contexte : SE sauvegarde l’état du Pu courant et remplace celui en chargeant l’état du nouveau Pu. Contexte = PCB. Le temps de charge-ment doit être minimal – Création des Pu : Les Pu parents créent des Pu enfants. Chaque Pu a un PID. Options de partage de ressources : Parents et enfants partagent les mêmes ressources, enfants partagent un sous-ensemble de ressources. Aucun partage. Options d’exécution : Le parent et les enfants s’exécutent simultanément, parent attend que les enfants se terminent. Possibilités : L’enfant est une copie du parent ou un nouveau Pu. – Terminaison des Pu : 4 possibilités – *(1)* Le Pu enfant termine via **exit()** et le parent reçoit les données via wait(). *(2)* Le parent termine les Pu enfants avec abort(). *(3)* Le Pu enfant termine pendant qu’aucun parent n’attend : Zombie. *(4)* Le parent termine, alors les enfants deviennent des Pu orphelins.

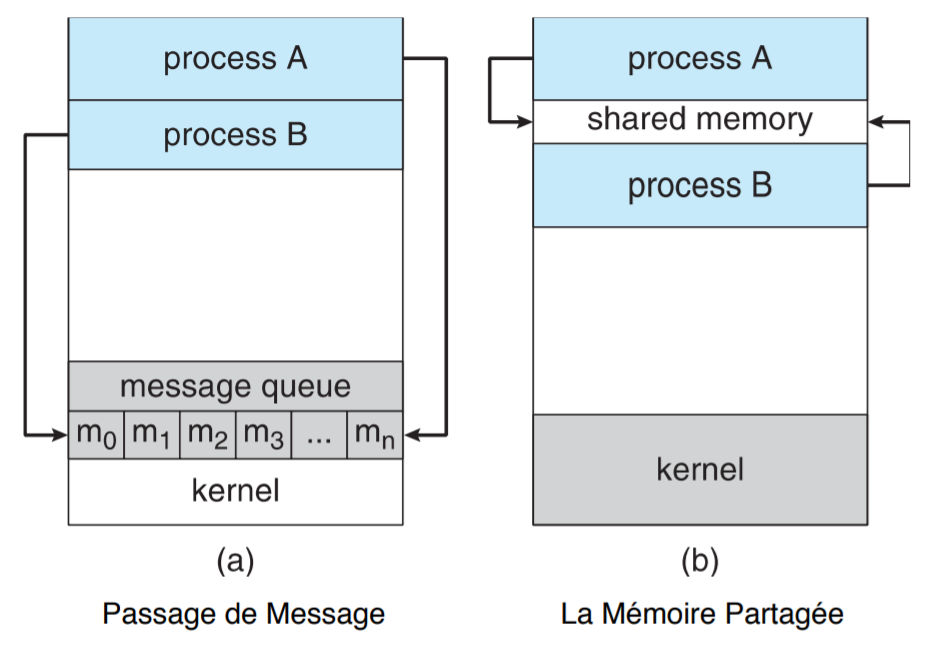
Processus (suite)

## Méthodes

fork() : Duplique le Pu courant. Cette méthode retourne *0* si le Pu courant est l’enfant, *-1* s’il y a une erreur et un *chiffre quelconque* si c’est le parent (Pu d’origine).

exec\*() : Exécute un Pg (avec paramètres)  
wait() : Attend après un Pu enfant.

## Intermediary Pu Communications (IPC)

Shared memory: unbounded-buffer (illimité), bounded-buffer (taille fixe). Message passing: send(TargetPu msg), receive(targetPu, msg). Un lien par pairs de Pu. Il est aussi possible de communiquer indirectement en transmettant des messages à des ports utilisés par des Pu. Les messages peuvent être (a)synchrones.

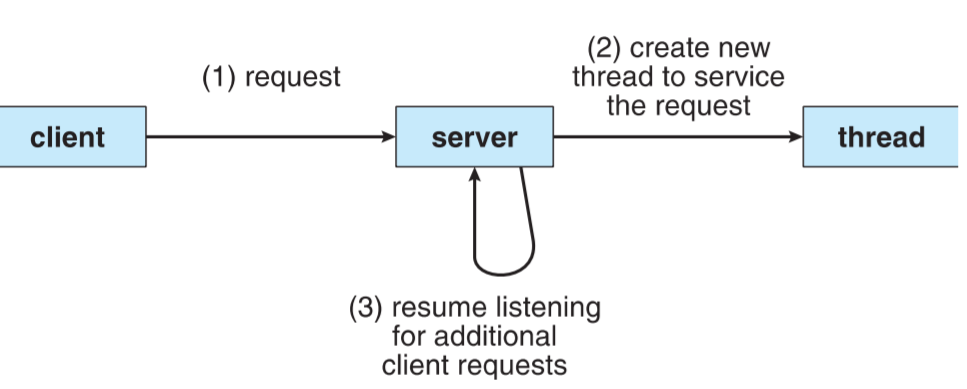
## Communications client-serveur

Socket : Le socket u.v.x.y:z fait référence au port z sur la machine u.v.x.y. UDP/TCP/Multiclast – Remote Procedure Calls (RPC) : Communication à distance. Les messages doivent être livrés exactement une fois (solution : timestamp, client attend accusé de réception). Comment le client peut savoir les numéros des ports? (solution : Le SE emploi un matchmaker) – Pipes (tuyaux): Permet la communication entre 2 Pu. (Uni/Bi)directionnel (write-end, read-end). Les tuyaux ordinaires sont unidirectionel, il faut donc 2 tuyaux pour aller dans les 2 sens.

## [\*\* Exercices TP \*\*]

Q1 | Client-serveur VS P2P : P2P, chaque station peut être le client ou le serveur – Que partagent les Pu lors d’un fork? Pile et tas (readonly), puis mémoire partagée.

Threads



Avs : Réactivité, partage de ressources, économie (moins cher que la création d’un Pu). Évolutivité : Le Pu peut bénéficier des multiprocesseurs.

## Concurrence et parallélisme

Concurrence : Illusion de parallélisme (prend en charge plus d’une tâche, intercalage des tâches) – Parallélisme : Plus d’une tâche simultanément. Deux types : *(1)* même opération distribuée sur plusieurs threads, *(2)* une opération unique par TD.

## Multi-Core/Multiprocesseurs (MPC)

Défis : Diviser les activités en tâches distinctes, Équilibre au niveau de la distribution des ressources, Éviter les collisions de données, Synchronisation des données, test et débogage.

## Threads

Deux types de TD : Utilisateurs (Java, Win32, POSIX Pthread) et noyaux (pris en charge par le noyau). Les TD utilisateurs ne peuvent pas s’exécuter tout seul (seulement par un TD du noyau). Aussi, ces TD peuvent être (a)synchrones.

Linux TD : Linux se réfère à eux comme des tâches plutôt que des TD. La méthode clone() permet à une tâche enfant de partager l'espace d'adressage de la tâche parente (Pu).

Modèles Multithreading (4) : *(1)* Many-to-One : Plusieurs TD utilisateurs attachés à un seul TD noyau (pas très populaire). *(2)* One-to-One : Plus d’accès simultanés. *(3)* Many-to-Many : Permet à de nombreux TD de niveau utilisateur d'être mappés vers de nombreux TD du noyau (plus petits ou égaux). *(4)* À deux niveaux : Similaire à Many-to-Many, sauf qu'il permet à un TD utilisateur d'être lié à un TD du noyau

Autres : Thread Pools : Garage à TD. Lorsque l’on veut exécuter une tâche en attend qu’un TD soit disponible dans le garage (pool). Plus rapide. – OpenMP : Ensemble de directives de compilation + API pour C, C++ et Fortran.

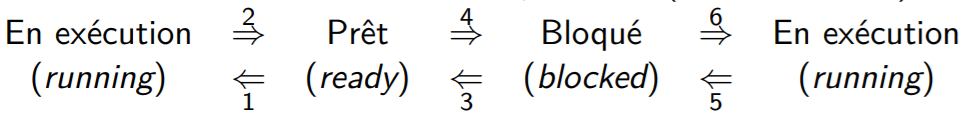
## Complications

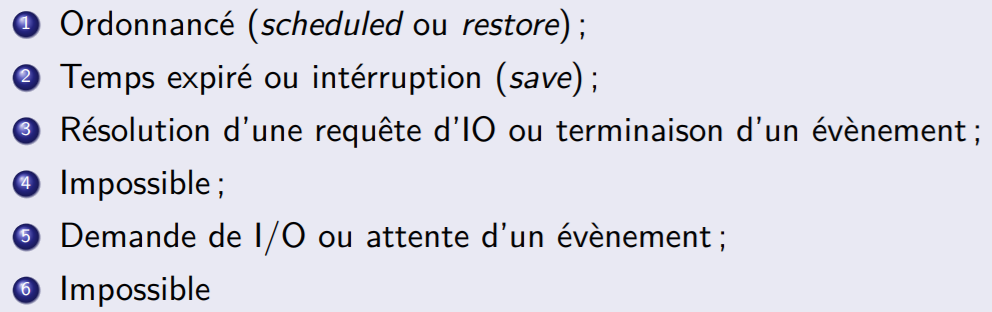
fork(): Certains SE ont 2 versions du fork (pour gérer différemment l’exécution pour les TD) – exec() : Comme d’habitude (remplace le Pu en cours …et tous les TD) – Traitement de signal : Comment notifier les TD qu’un événement particulier s’est produit (gestionnaire de signal assigné à chaque signal) – Annulation d’un TD : Deux moyens : *(1)* Annulation asynchrone (termine le TD immédiatement). *(2)* Annulation différée : permet au TD cible de vérifier périodiquement s'il doit être annulé – Partage des données : Thread-Local Storage : permet à chaque TD d'avoir sa propre copie de données (Utile lorsque l’on emploie les Pool).

## [\*\*Exercices TP\*\*]

Q1 | Qu’est-ce qui est partagé entre TD et un nouvel enfant TD? Le tas, le code et les variables globales.

# Q2 | Qu’est-ce qui ferait passer un TD d’un état à l’autre (si possible)? Ajoutez l’action effectuée.





# Q3 | Est-il possible d’améliorer une application multi user-TD en ajoutant des processeurs?

Non, car il est possible de charger qu’un seul TD utilisateur à la fois dans le noyau.

# Q4 | Est-il possible d’améliorer un Pg (qui ouvre un fichier, effectue des opérations qui sont cpu-bound, puis écrit les résultats à l’intérieur d’un fichier) en utilisant 2 processeurs double cœur?

Oui, mais seulement pour les opérations cpu-bound, car l’ouverture et l’écriture du fichier sont des actions bloquantes (maximum 4 TD).

# Q5 | Pourquoi utiliser des TD plutôt que des Pu (et inversement)?

Pu : *(1)* Ne fait pas planter d’autres Pu lorsque bug. *(2)* Sécurité des données (pas de partage avec les autres Pu).

TD : *(1)* Moins couteux que de créer un Pu. *(2)* Partage des ressources entre TD appartenant au même Pu.

# Q6 | Quand est-il utile d’utiliser une solution multi-TD pour un Pg sur une machine à un processeur?

Quand on veut améliorer le temps de réponse pour l’utilisateur (si un TD bloque…).

Synchronisation

## Section critique (SC)

Chaque Pu a une SC. Celle-ci est réservée pour effectuer des opérations critiques (écriture dans un fichier, etc).

Comment gérer ça? *(1)* Si un Pu est dans sa SC, alors les autres Pu ne peuvent pas interférer. *(2)* Si aucun Pu est dans sa SC, alors on choisit le Pu qui a besoin d’exécuter sa SC. *(3)* Imposer une limite sur le nombre de fois qu’un Pu peut entrer dans sa SC.

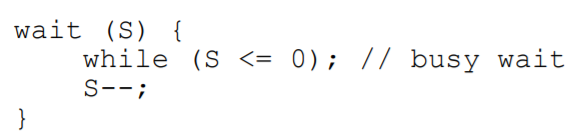
## Mutex

Mutex Locks? Utilisation de verrous pour effectuer les tâches de la SC. La méthode acquire() doit être utilisée pour activer le lock, puis nous utilisons la méthode release() pour enlever le lock

Suite à l’exécution de la méthode acquire(), tous Pu essayant d’entrer dans leur SC devront boucler jusqu’à tant que release() soit appelée.

## Sémaphore

[Youtube](https://www.youtube.com/watch?v=DvF3AsTglUU) – Doit garantir que 2 Pu ne peuvent accéder à la même méthode en même temps (wait, signal)





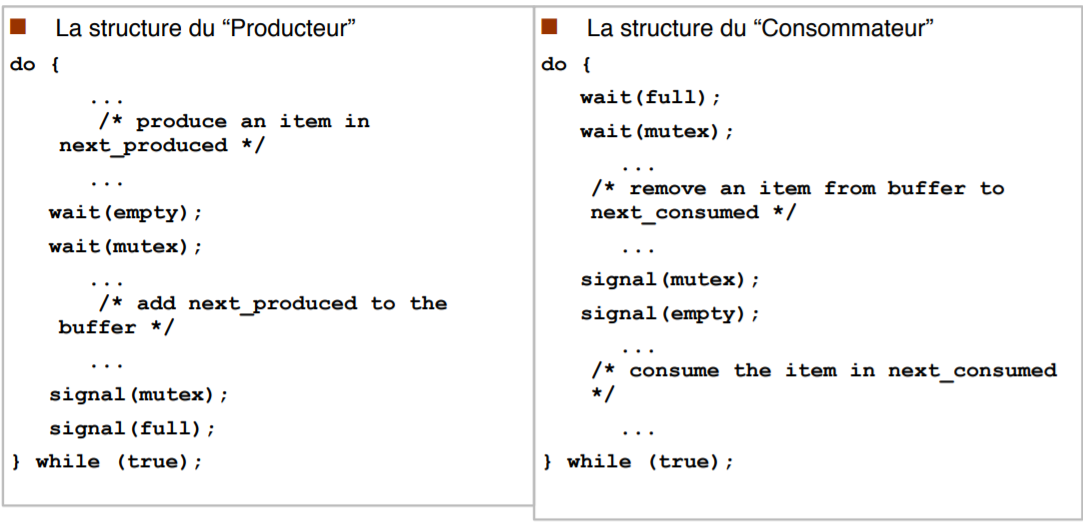
## Interblocage (IB)

Plus de verrous entraînent le risque que certains Pu tombent dans une attente infinie : les verrous sont tous actifs et les seuls Pu qui peuvent libérer les verrous sont tous en attente.

Famine : Blocage indéfini – Causé par des IB ou des problèmes de priorité.

## Problèmes classiques

# Producteur/Consommateur



# Lecteurs/Écrivains

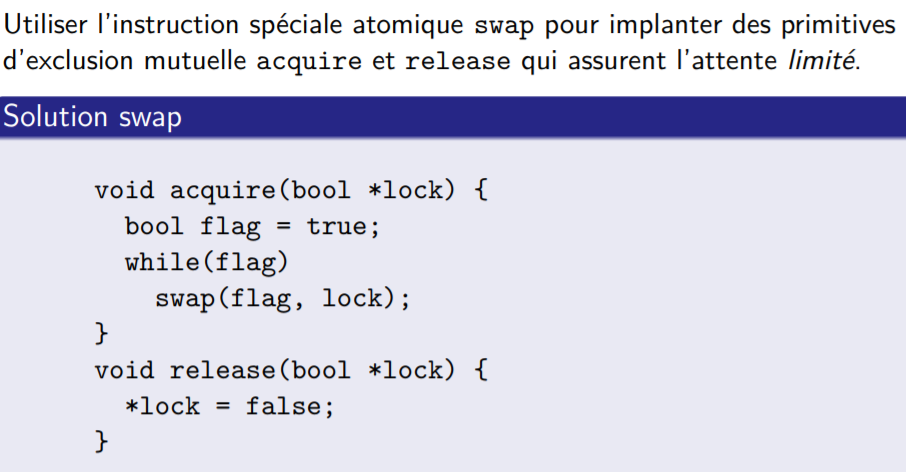
Certains TD ne peuvent que lire et certains TD ne peuvent qu’écrire. Problème de famine : Les lecteurs continuent d’arriver et les écrivains ne sont jamais traités (et vice-versa).

# Philosophes???

## Débogage

Il est très difficile de déboguer les mutex et les sémaphores. Nous pouvons utiliser des abstractions appelées : Moniteurs.

## [\*\* Exercices TP \*\*]

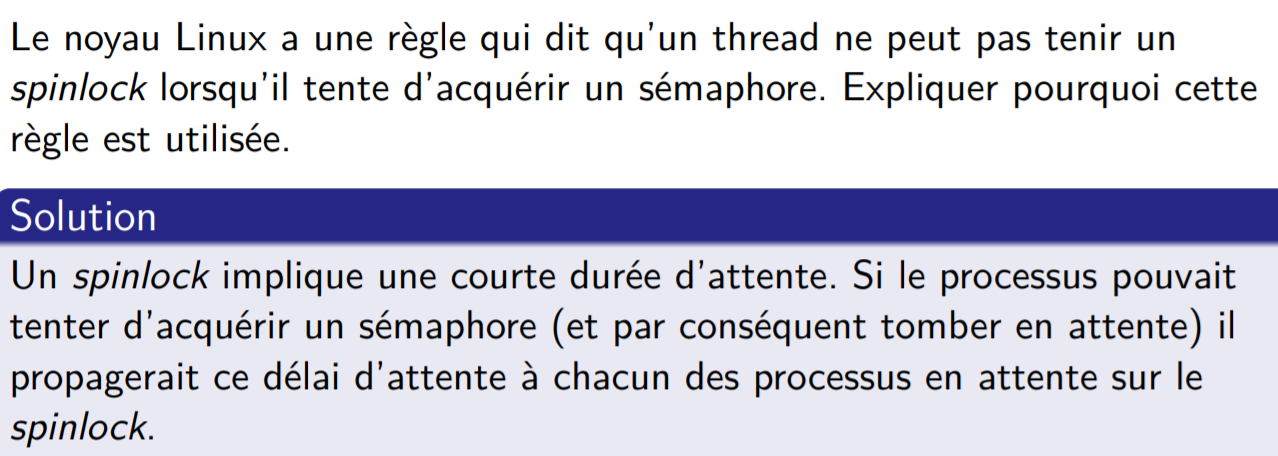


# Q2 | Que ce passe-t-il si les fonction wait et signal ne sont pas atomique ?

Deux TD pourraient lire en même temps la valeur du sémaphore et entrer dans leur SC en même temps.

# Q3 | Que signifie attente active ? Quelles autres formes d’attente existe-t-il dans un système informatique ? Peut-on complètement éviter l’attente active ?

*(1)* Attendre qu’un évènement se libère. *(2)* un Pu/TD peut bloquer. *(3)* Oui, mais il va falloir augmenter le nombre de changements de contexte.



Ordonnanceurs

## Révision (Introduction)

Les ORD choisissent quand exécuter quel Pu sur quel CPU. Utilisent des queues : job queue (tous les Pu), ready queue (les Pu), device queues (les Pu qui attendent un périphérique)

Type de ORD : Long terme – Sélectionne quels Pu doivent être placés dans la file d’attente, invoqué rarement (lent), contrôle le nombre de Pu en mémoire. Court terme – Gère les Pu, planificateur des tâches, invoqué fréquemment. Moyen terme – Contrôle le degré de multiprog (swapping).

Types de Pu : I/O-bound – Beaucoup de rafales du CPU (plus de E/S que des calculs). CPU-bound – Quelques rafales (beaucoup de calculs).

## Dispatcher

TODO

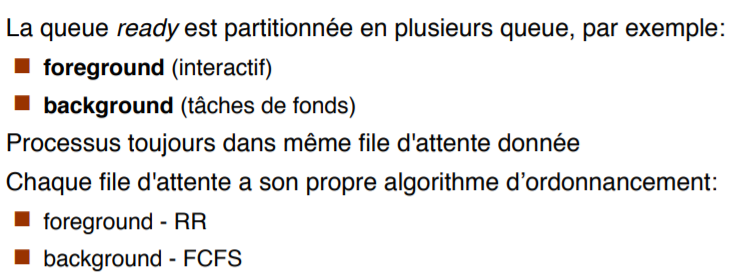
## Buts

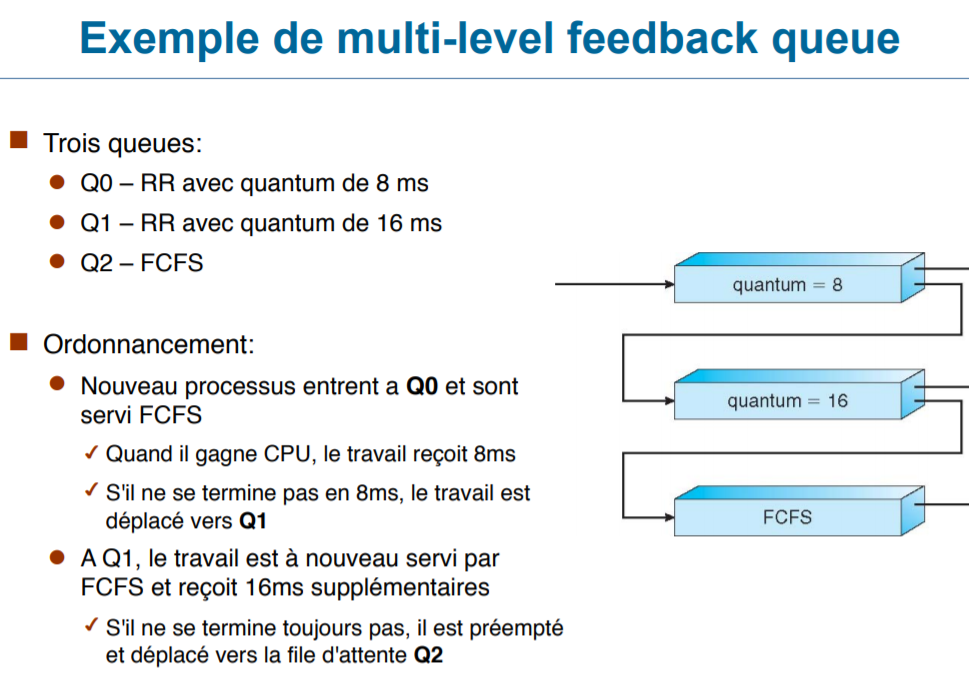
Maximiser l’utilisation du CPU – Maximiser le débit (travail/temps) – Minimiser le délai d’exécution (soumission/complétion) – Minimiser le temps d’attente des Pu dans la file Ready – Minimiser le temps entre une requête et sa réponse.

## Techniques d’ordonnancement

FIFO – Shortest-Job-First (SJF) : Pour déterminer le temps qu’une tâche va prendre, nous pouvons faire des estimations, puis on ajuste celle-ci avec l’expérience – Shortest-Remaining-Time-First (SRTF) : Même chose que pour SJF, sauf que là nous regardons le temps restant – Round Robin : À chaque *round*, on donne maximum q unités de *puissance* à chaque tâche (q trop haut => FIFO, q trop petit => trop de context switch).

## Queue multiniveau





## [\*\* Exercices TP \*\*]

# Q1 | Préemptif VS non-préemptif?

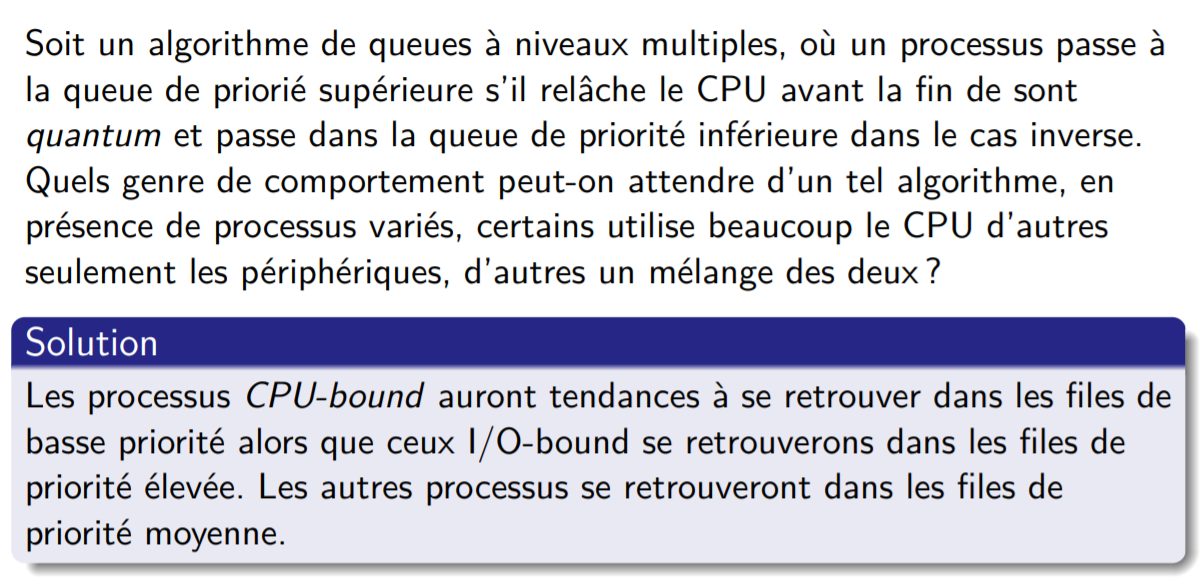
Préemptif : ORD peut arrêter un Pu à tous moments (utile dans un environnement interactif) – Non-préemptif : ORD attend que les Pu lui redonnent le contrôle des ressources allouées du CPU.

# Q2 | Quantum long vs court?

Long: Utile pour les applications qui demandent beaucoup de temps du processeur (sans interruptions). Tend vers le FIFO pour les Pu nécessitant pas beaucoup de quantum.  
Court : Utile pour les applications interactives. Beaucoup de changement de contexte

# Q3 | Soit une variante de RR dans laquelle les entrées dans la queue des Pu prêts à l’exécution sont des pointeurs vers des PCB. Quel serait l’effet de placer deux pointers vers le même PCB dans la queue ?

Augmente la priorité du Pu (double l’exécution du Pu).



Interblocages