Concurrence et parallélisme

- **Concurrence :** concerne plusieurs tâches qui commencent, s'exécutent et se terminent dans des périodes de temps qui se chevauchent, sans ordre précis.

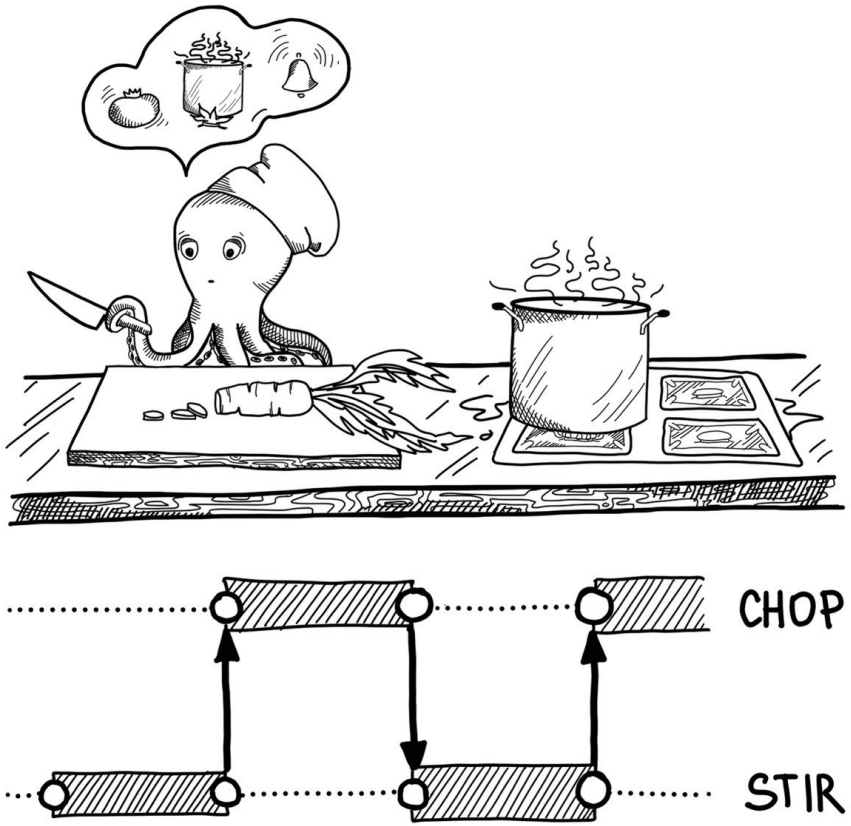
- **Parallélisme** : concerne plusieurs tâches ou sous-tâches qui s'exécutent en même temps sur un matériel doté de plusieurs ressources informatiques, comme un processeur multicœur.

Les deux concepts sont proches mais pas identiques.

- La **concurrence** est une **propriété sémantique d'un programme ou d'un système** : ne dépend pas de l'exécution réelle des tâches, mais de la façon dont un programme est conçu.

- Le **parallélisme** est une propriété de la mise en œuvre : il dépend de la couche matérielle.

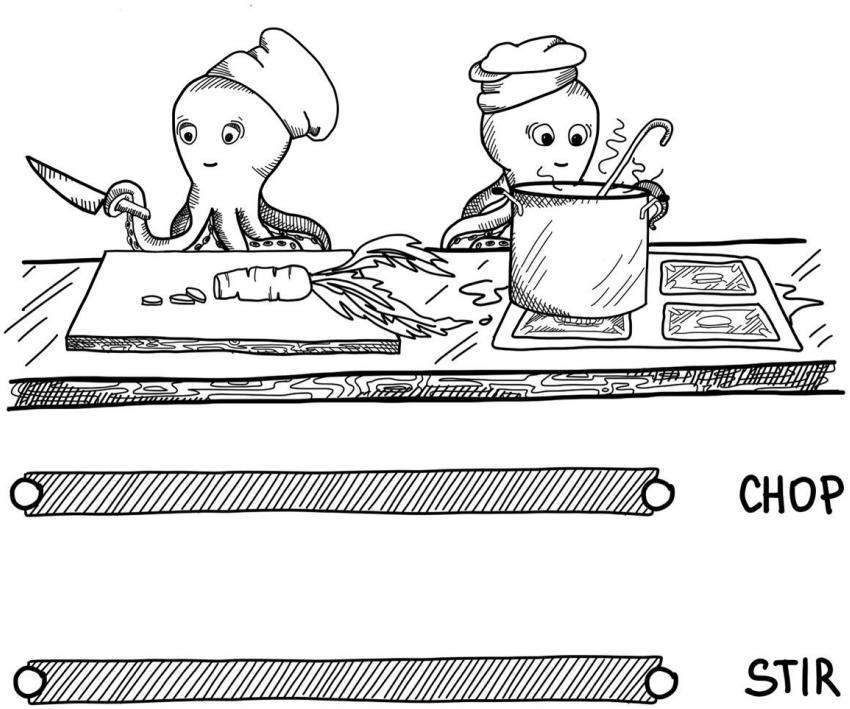
- Imaginez qu'un cuisinier soit en train de couper de la salade tout en remuant de temps en temps la soupe sur le feu. Il doit s'arrêter de hacher, vérifier la cuisinière, puis recommencer à hacher, et répéter ce processus jusqu'à ce que tout soit terminé.



- Une seule ressource de traitement, le chef, et la concurrence est principalement liée à la logistique ; sans concurrence, le chef doit attendre que la soupe sur la cuisinière soit prête pour couper la salade.

https://freecontent.manning.com/concurrency-vs-parallelism/

- Nous avons maintenant deux chefs, un qui peut remuer et un qui peut couper la salade. Nous avons divisé le travail en ayant une autre ressource de traitement un autre chef.



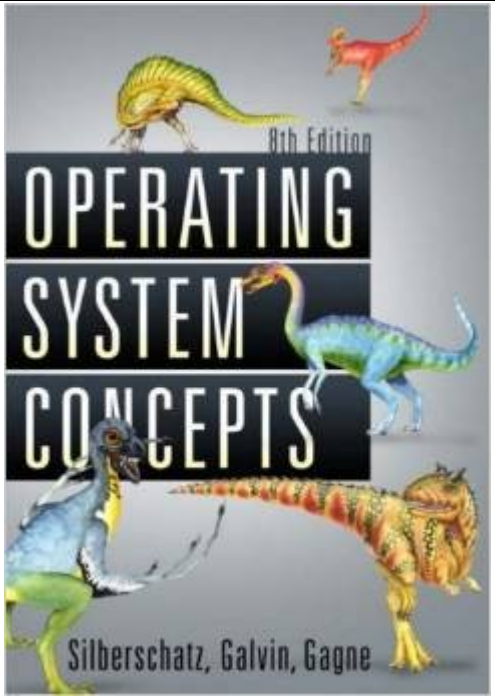
https://freecontent.manning.com/concurrency-vs-parallelism/

Le parallélisme est une sous-classe de la concurrence : avant de pouvoir effectuer plusieurs tâches à la fois, il faut d'abord gérer plusieurs tâches.

Rob Pike :

“Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.”

Ordonnancement des Processus



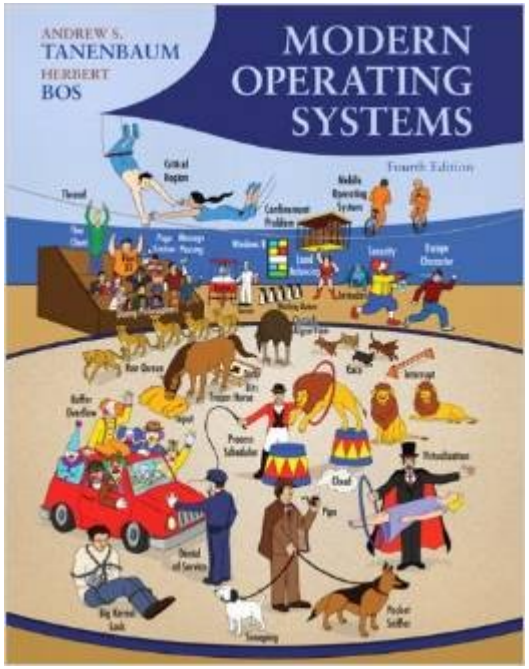
Operating System Concepts – 2008 Wiley

Abraham Silberschatz, Peter B. Galvin, Greg Gagne

Notes de cours de

Luigi Logrippo

http://w3.uqo.ca/luigi



Modern Operating Systems – 2014
Prentice Hall

Andrew S. Tanenbaum, Herbert Bos

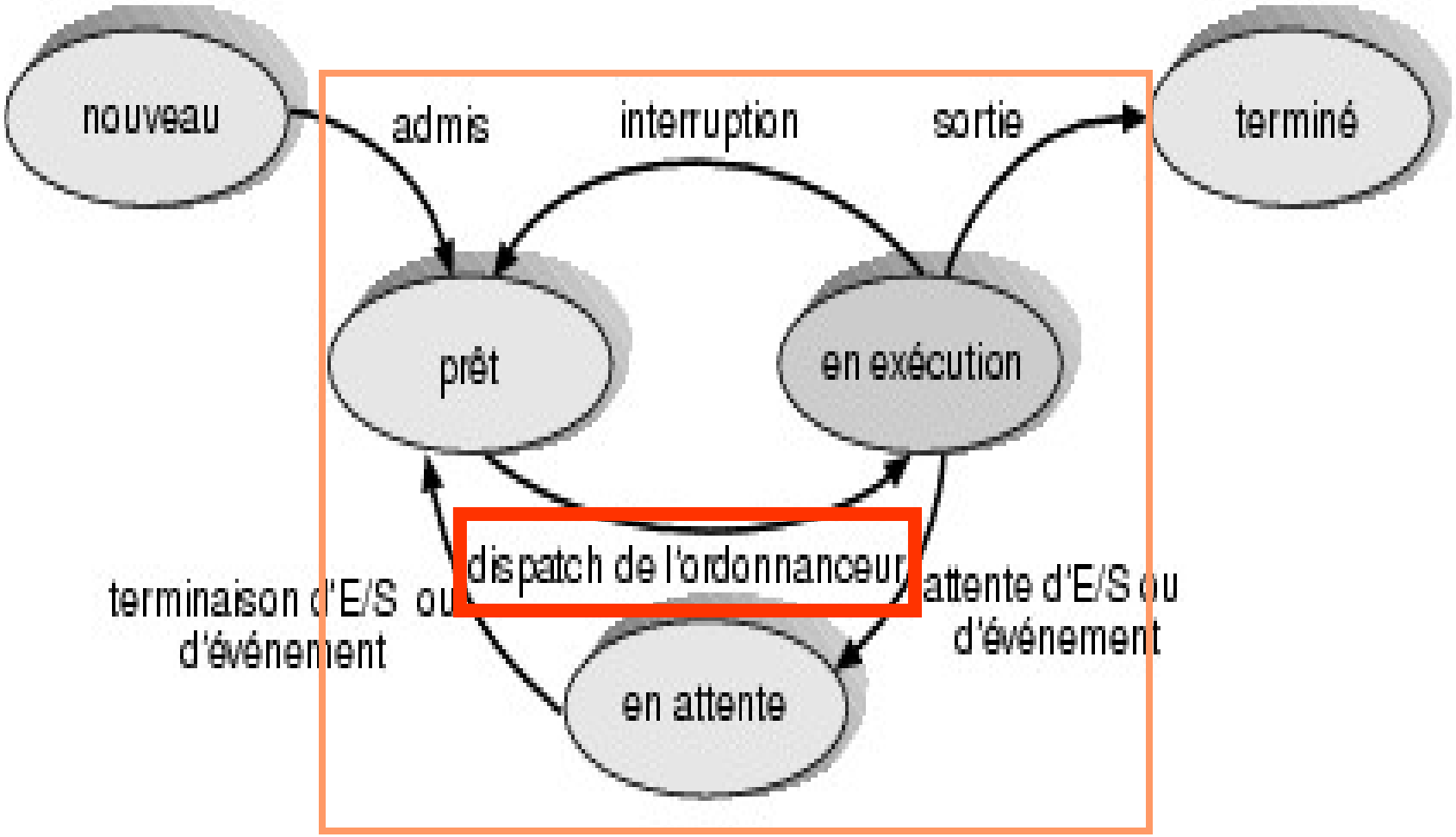
- **Les méthodes d'ordonnancement ont des nombreuses applications**
  - Et ont été très étudiées,surtout avec des méthodes probabilistes et de simulation
- **Non seulement en informatique, mais aussi en gestion:**
  - Supposons qu'on parle de différents tâches à exécuter dans une usine dans laquelle il y a des ressources qui sont des ouvriers et des outils

Importance dans l'informatique

- **L'utilisation de critères d'ordonnancement efficaces est surtout importante pour les grands centres d'informatique qui sont très chargés**
  - Google >2,7M, Microsoft: > 1M

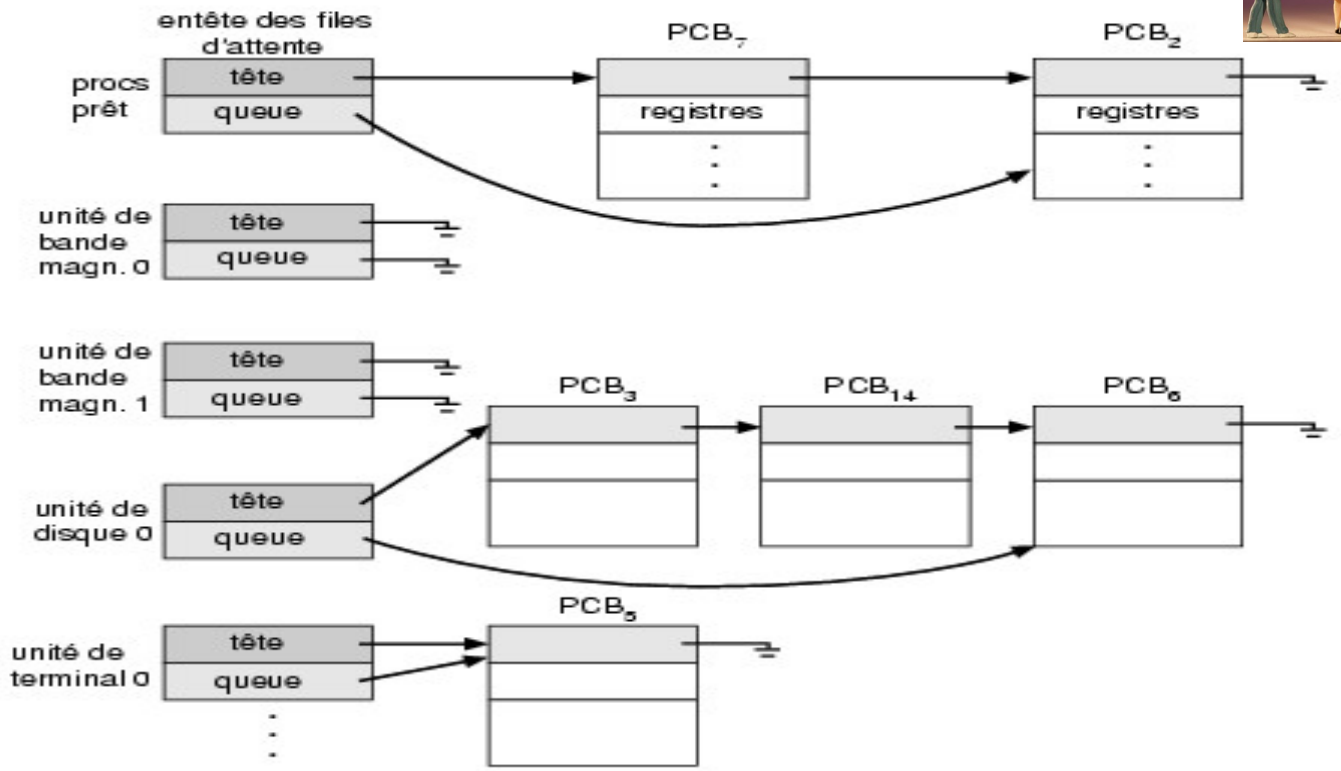
    (serveurs en 2015, chiffres cachés ensuite..)
  - Une petite économie en pourcentage peut économiser des sommes considérables (électricité, chauffage etc.)
  - **Il est important de garder les serveurs saturés de travail**

Diagramme de transition d'états d'un processus



Files d'attente de processus pour ordonnancement

File prêt

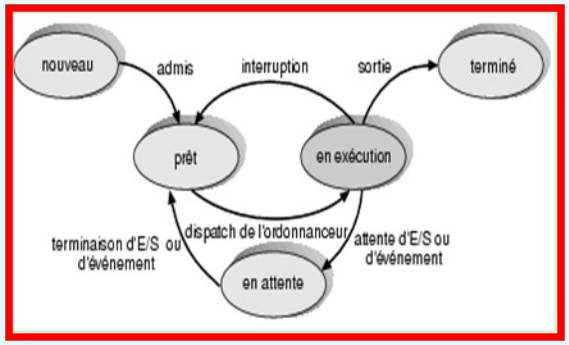


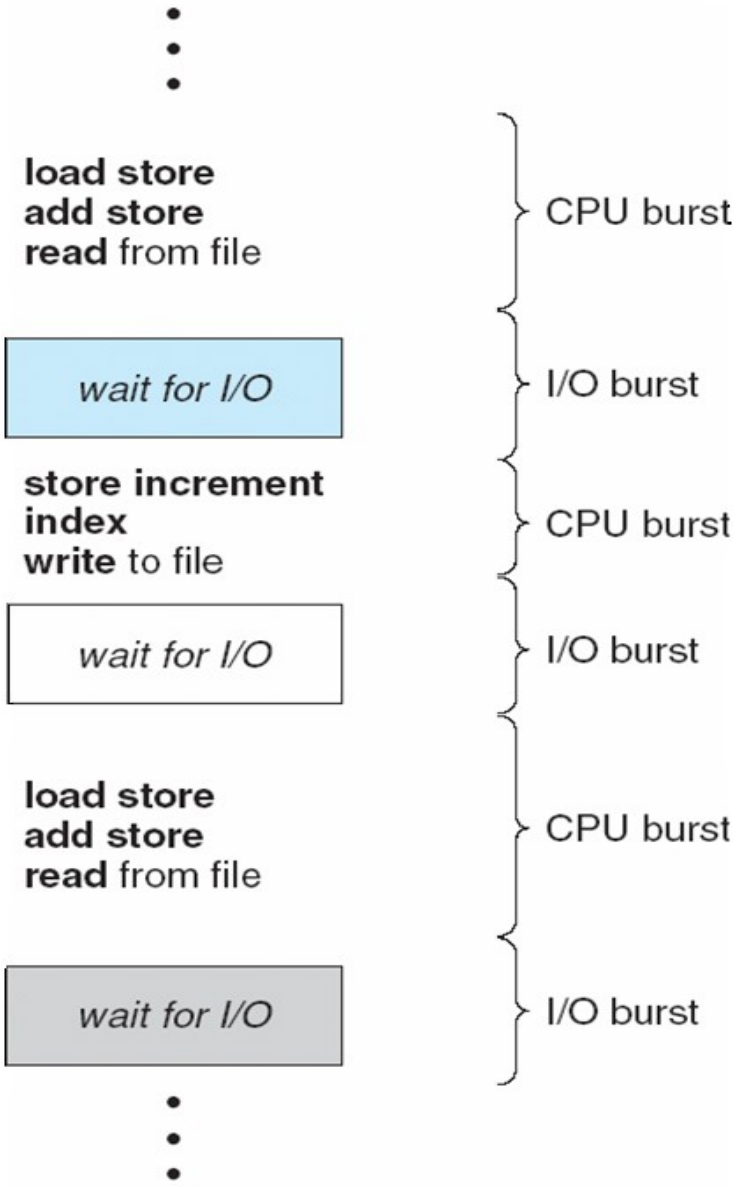
– Le premier processus dans une file est celui qui utilise la ressource: ici, proc7 s'exécute.
– Pour simplifier : une seule UCT (Unité Centrale de Traitement=CPU)

Concepts de base

- **On cherche**
  - utilisation optimale des ressources
  - et aussi un bon temps de réponse pour l'usager
- **L'UCT est la ressource la plus précieuse dans un ordinateur**
  - cependant, les principes que nous verrons s'appliquent aussi à l'ordonnancement des autres ressources (unités E/S, etc).
- **L'ordonnanceur UCT est la partie du SE qui décide quel processus dans la file ready/prêt obtient l'UCT quand celle-ci devient libre**

Les cycles d'un processus



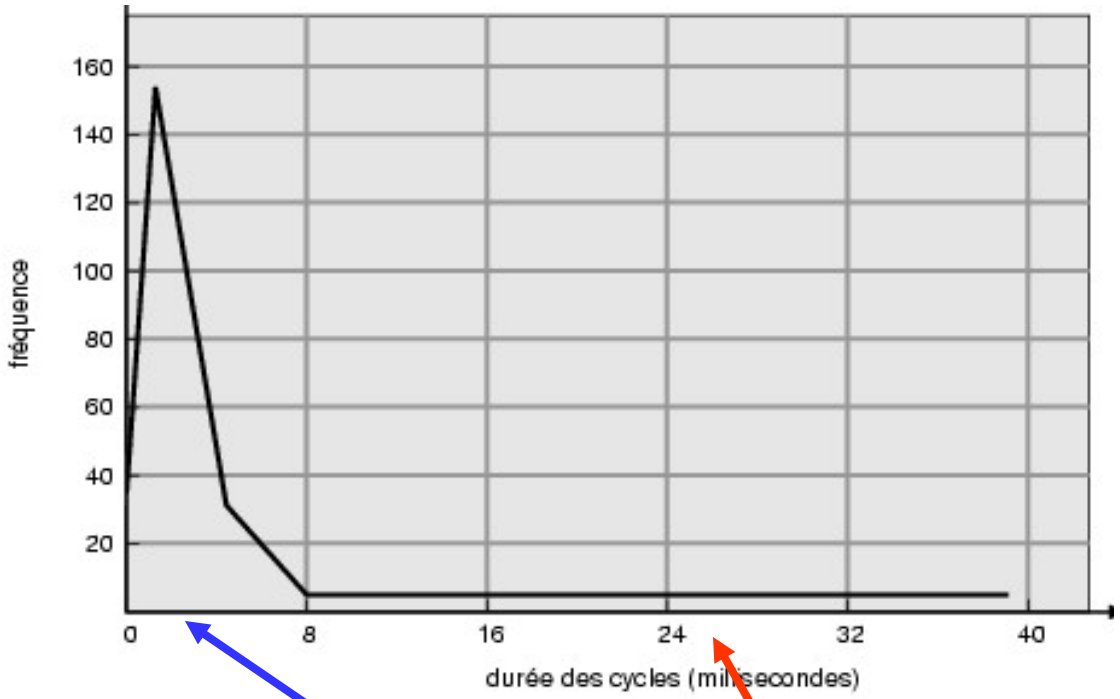


- **Cycles (bursts)** d'UCT (=CPU) et E/S (=I/O) : l'activité d'un processus est fait de séquences d'exécutions sur UCT et d'attentes: =E/S ou synchronisation avec autres processus

Durée normale des cycles

- Étant donnée la grande vitesse de l'UCT par rapport aux périphériques, la plupart des cycles d'UCT sont très courts
- Cependant il pourrait y avoir des longs cycles d'UCT quand on demande des calculs poussés
  - Calculs scientifiques etc.

Durée typique des cycles UCT



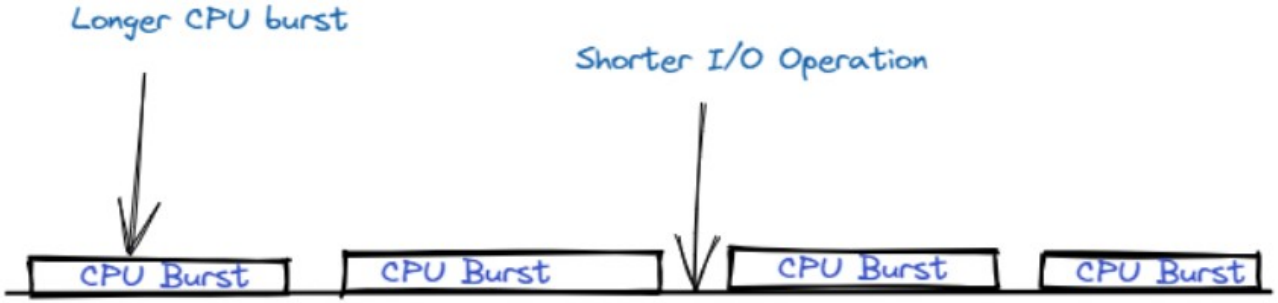
- **Observation expérimentale:**

  dans un système typique, nous observons un grand nombre de court cycles d'UCT, et un petit nombre de long cycles
- **Les programmes tributaires de l'UCT auront normalement un petit nombre de long cycles UCT**
- **Les programmes tributaires de l'E/S auront normalement un grand nombre de court cycles UCT**

CPU-Bound vs I/O-Bound

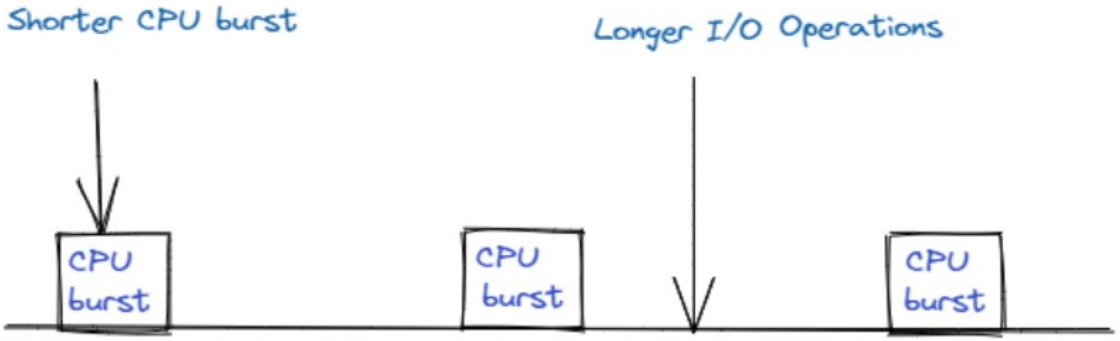
CPU-bound :

- un ordinateur qui utilise # 90 à 100 % de son temps pour des calculs



IO-bound :

- un ordinateur qui utilise son temps pour attendre des E/S (I/O)



Processus ou threads ?

En général, on utilise plutôt :

- **Threads** : IO-bound

- **Processus** : CPU-Bound

Remarque : sous Linux, en réalité tout est tâche (task) au niveau du noyau (primitive clone() avec des paramètres différents)

https://blog.mineiros.io/how-to-use-multithreading-and-multiprocessing-a-beginners-guide-to-parallel-and-concurrent-a69b9dd21e9d

Threads ?

En distingue plusieurs type de threads :

- **Green threads** :
  - dans l'espace **utilisateur** via une bibliothèque ou une VM (ex : autrefois Java)
  - Les multiples threads sont émulés dans un seul processus

+ Portables
+ Efficaces pour l'activation et la synchronisation
+ Transparents pour le noyau
- Si un thread bloque sur un appel système, tous les threads sont bloqués...typiquement sur une entrée/sortie.

Threads ?

- **Native threads** :
  - dans l'espace noyau
  - Les multiples threads peuvent utiliser des cœurs différents

Ce sont eux-qui sont le plus utilisés aujourd'hui, mais
- Plus compliqué pour le noyau

Fiber

- Threads très légères
- Utilisent généralement un changement de **contexte coopératif (yield)** :
    - coroutines (async IO en python...) , goroutines en Go

Modèles de threads

1:1 (threading au niveau du noyau)

Chaque thread utilisateur correspond à un thread noyau

+Possibilité d'utiliser plusieurs coeurs

Le plus utilisé aujourd'hui

N:1 (threading au niveau de l'utilisateur)

Tous les threads utilisateurs correspondent à un seul thread noyau

+ Le noyau n'a aucune connaissance des threads d'application.

+Commutation de contexte très rapide

- Pas de multi-cpu possible

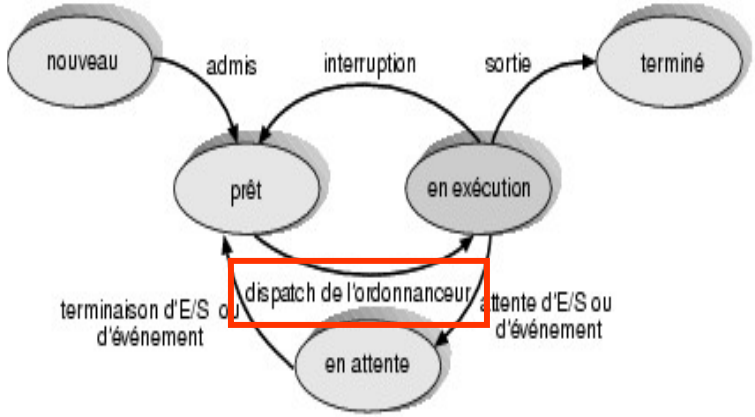
M:N (threading hybride)

M threads utilisateur correspond à N threads noyau

Compromis entre 1:1 et N:1

Plus complexe et plus coûteux à gérer !

Quand invoquer l'ordonnanceur UCT



- **L 'ordonnanceur UCT doit prendre la décision chaque fois que le processus exécutant est interrompu, c-à.-d.**
  1. un processus se présente en tant que nouveau ou se termine
  2. un processus exécutant devient bloqué en attente
  3. un processus change d'exécutant/running à prêt/ready
  4. un processus change de attente à prêt/ready
  - tout événement dans un système qui cause une interruption de l'UCT implique l'intervention de l'ordonnanceur,
  - qui devra prendre une décision concernant quel processus ou thread obtient l'UCT
- **Préemption (=réquisition) : on a préemption si on enlève l'UCT à un processus qui l'avait et ne l'a pas laissé de sa propre initiative**
  - On parle de **multitâche préemptif** (≠ **coopératif**)
- **Plusieurs pbs à résoudre dans le cas de préemption**

Dispatcheur (répartiteur)

- **Le processus qui donne le contrôle au processus choisi par l'ordonnanceur. Il doit se préoccuper de:**
  - changer de contexte
  - Passer en mode usager
  - réamorcer le processus choisi (emplacement mémoire...)
- **Attente du dispatcheur (dispatcher latency)**
  - le temps nécessaire pour exécuter les fonctions du dispatcheur
  - il est souvent négligé, il faut supposer qu'il est petit par rapport à la longueur d'un cycle

Critères d'ordonnancement

- **Il y aura normalement plusieurs processus dans la file prêt**
- **Quand l'UCT devient disponible, lequel choisir?**
- **Critères généraux:**
  - Bonne utilisation de l'UCT
  - Réponse rapide à l'usager
- **Mais ces critères peuvent être jugés différemment...**

Critères spécifiques d'ordonnancement

- Utilisation UCT: **pourcentage d'utilisation**
- **Débit = Throughput**: nombre de processus qui complètent dans l'unité de temps
- **Temps de rotation = turnaround**: le temps pris par un proc. de son arrivée à sa terminaison.
- **Temps d'attente**: attente dans la file prêt (somme de tout le temps passé en file prêt)
- **Temps de réponse**: le temps entre une demande de l'usager et la réponse

Critères d'ordonnancement: maximiser/minimiser

- **Utilisation UCT: pourcentage d'utilisation**
  - à maximiser
- **Débit = Throughput: nombre de processus qui complètent dans l 'unité de temps**
  - à maximiser
- **Temps de rotation (turnaround): temps terminaison moins temps arrivée**
  - à minimiser
- **Temps d'attente: attente dans la file prêt**
  - à minimiser
- **Temps de réponse (pour les systèmes interactifs): le temps entre une demande et la réponse**
  - à minimiser

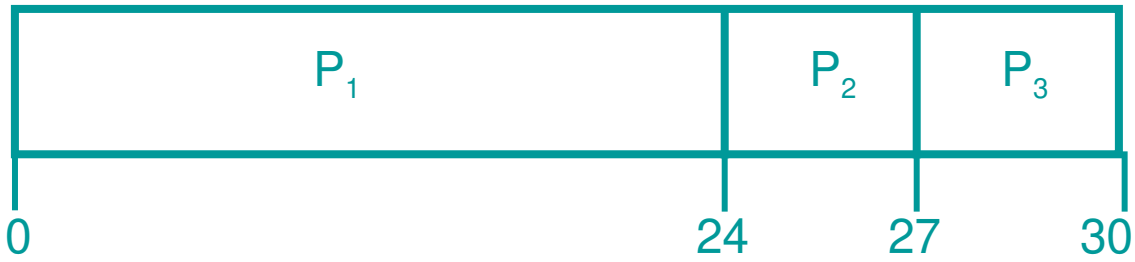
Premier arrivé, premier servi
(First come, first serve, **FCFS**)

Exemple:	Processus	Temps de cycle(Burst)
	P1	24
	P2	3
	P3	3

Si les processus arrivent au temps 0 dans l'ordre:

P1 , P2 , P3

Le diagramme de Gantt pour l'utilisation de l'UCT est:

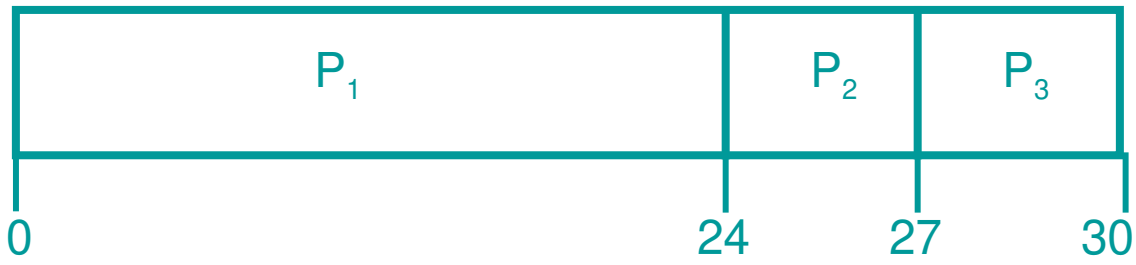


Temps d'attente pour P1= 0; P2= 24; P3= 27

Temps attente moyen: (0 + 24 + 27)/3 = 17

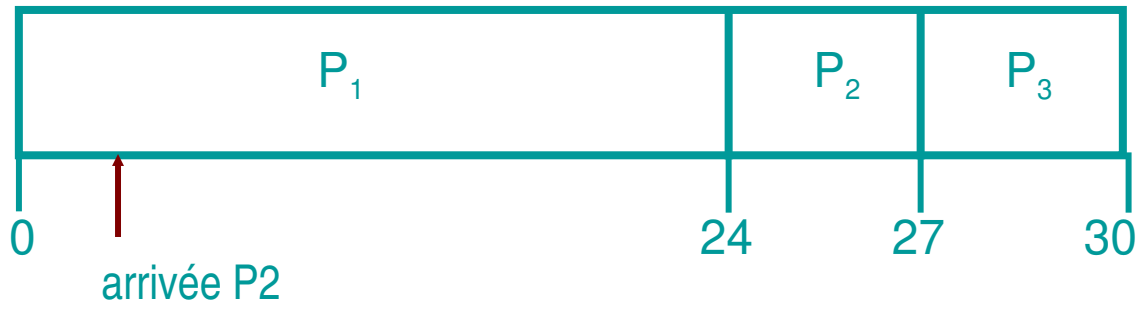
Premier arrive, premier servi

- **Utilisation UCT = 100%**
- **Débit = 3/30 = 0,1**
  - 3 processus complétés en 30 unités de temps
- **Temps de rotation moyen:**
  (24+27+30)/3 = 27



Tenir compte du temps d'arrivée!

- **Dans le cas où les processus arrivent à des moments différents, il faut soustraire les temps d'arrivée**
- **Exemple :**
  - P1 arrive à temps 0 et dure 24
  - P2 arrive à temps 2 et dure 3
  - P3 arrive à temps 5 et dure 3
- **Donc P1 attend 0 comme avant**
- **Mais P2 attend 24-2, etc.**

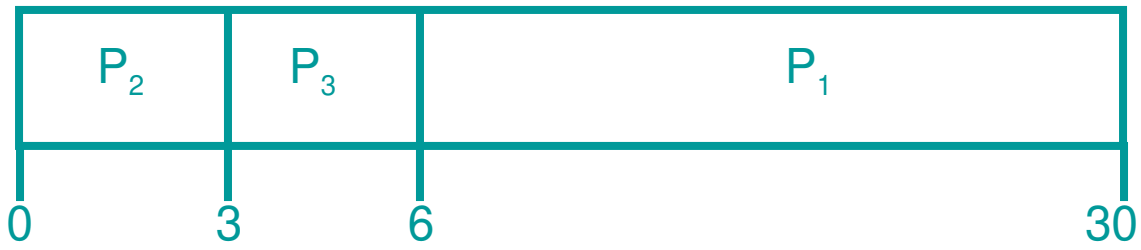


FCFS Scheduling

Si les mêmes processus arrivent à 0 mais dans l'ordre

P_2 , P_3 , P_1 .

Le diagramme de Gantt est:



- **Temps d'attente pour $P_1 = 6$ $P_2 = 0$ $P_3 = 3$**
- **Temps moyen d'attente: (6 + 0 + 3)/3 = 3 << 17**
- **Temps de rotation moyen: (3+6+30)/3 = 13 << 27**
- **Beaucoup mieux!**
- ***Les temps peuvent varier grandement* selon *l'ordre d'arrivée* de différent processus**
- ***Effet convoi* : des processus courts bloqués derrière un processus long**

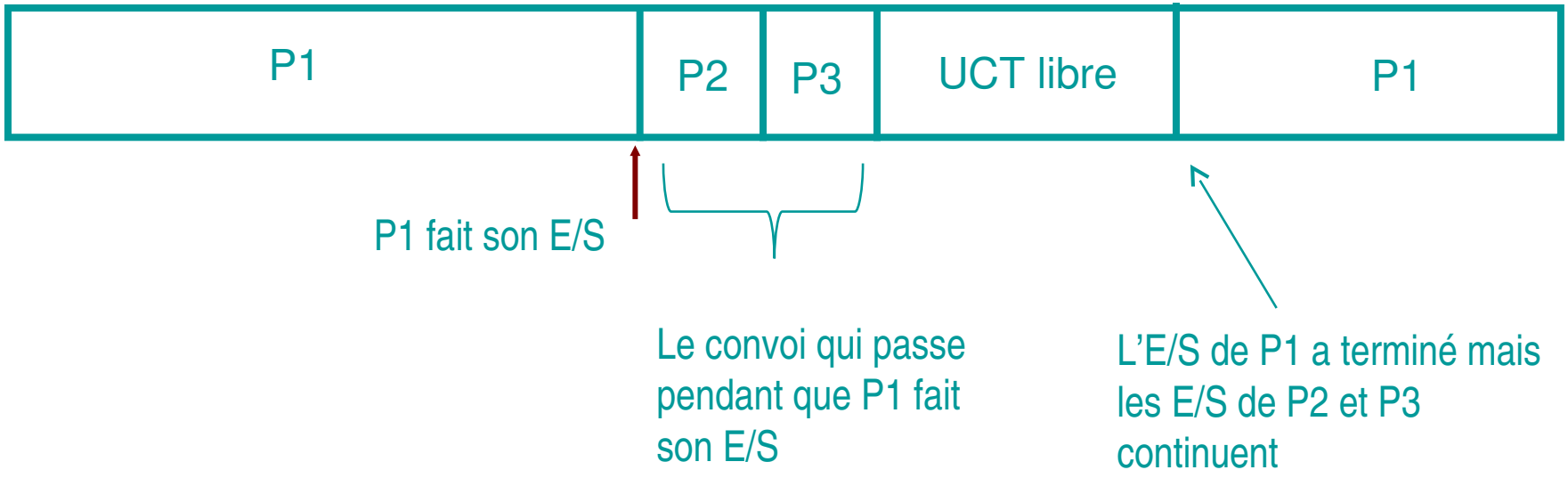
Effet d'accumulation (effet convoi) dans FCFS

- **Supposons**
  - un processus tributaire de l'UCT (cycles longs)
  - et plusieurs tributaires de l'E/S (cycles courts)
- **Les processus tributaires de l'E/S attendent l'UCT: E/S sous-utilisée (*)**
- **Le processus tributaire de l'UCT fait une E/S: les autres proc exécutent rapidement leur cycle UCT et retournent sur l'attente E/S (convoi qui passe): UCT sous-utilisée**
- **Processus tributaire de l'UCT finit son E/S, puis les autres aussi: retour à la situation (*)**
- Donc dans ce sens FCFS favorise les procs tributaires de l'UCT
- Et peut conduire à une mauvaise utilisation des ressources s'il y a apport continu de procs longs=tributaires de l'UCT

- Une possibilité: interrompre de temps en temps les proc tributaires de l'UCT pour permettre aux autres procs de s'exécuter (**préemption**)

Exemple

- P1: tributaire de l'UCT
- P2, P3: tributaires de l'E/S



Plus Court d'abord
Shortest Job First (SJF)

- **Le processus qui demande moins d'UCT part le premier**
- **Optimal du point de vue du temps d'attente moyen**

- **Mais comment savons-nous quel processus demande moins d'UCT!**
  - Supposons pour l'instant qu'on puisse le faire

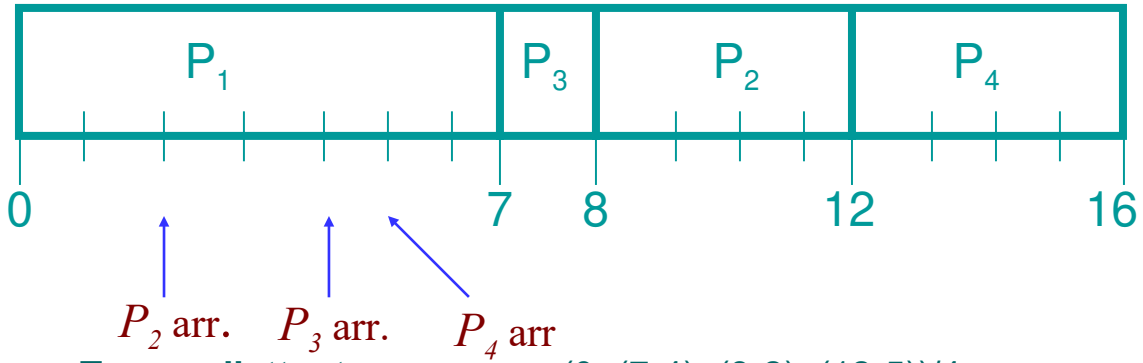
SJF avec préemption ou non

- **Avec *préemption*:**
  - si un processus qui dure *moins* que le *restant* du processus courant se présente
  - l'UCT *est enlevée* au processus courant et est donnée à ce nouveau processus
    - **SRTF: shortest remaining-time first**
- **Sans préemption: on permet au processus courant de terminer son cycle**

Example de SJF sans préemption

Processus	Arrivée	Cycle UCT
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (sans préemption)

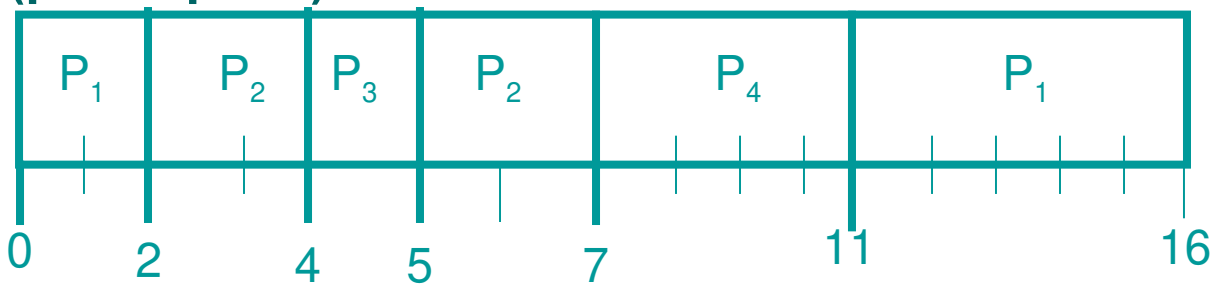


- Temps d'attente moyen = (0+(7-4)+(8-2)+(12-5))/4
  (0 + 6 + 3 + 7)/4 = 4
- Temps de rotation moyen = (7+(8-4+(12-2)+(16-5)) /4 = 8

Exemple de SJF avec préemption

Processus	Arrivée	Cycle
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (préemptive)



P_2 arr. P_3 arr. P_4 arr

- **Temps moyen d'attente = (9 + 1 + 0 +2)/4 = 3 < 4**
  - **P1 attend de 2 à 11, P2 de 4 à 5, P4 de 5 à 7**
- **Temps de rotation moyen =** 16+ (7-2) + (5-4) + (11-5) = 7 < 8

Plus en détail: file prêt

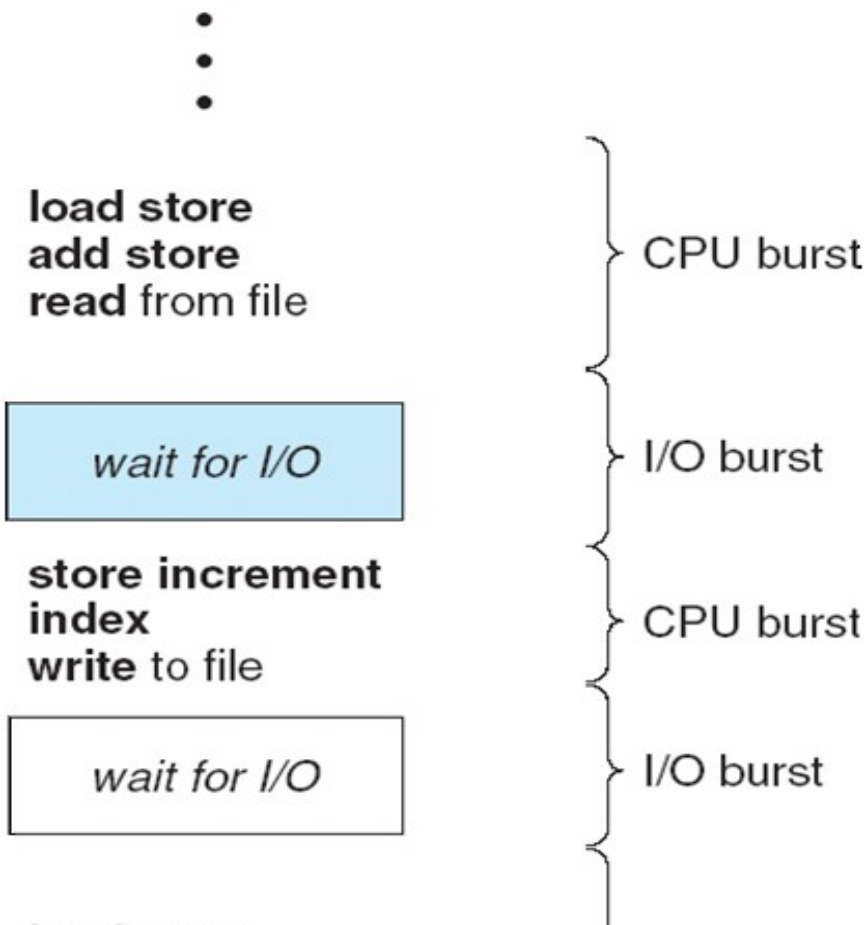
- **Temps des interruptions**
- **Temps 0:**
  - Le seul proc est P1, il est choisi
- **Temps 2: Interruption causée par l'arrivée de P2**
  - Deux procs dans la file:
    - P1 qui demande encore 5
    - P2 qui demande 4: ce dernier est choisi
- **Temps 4: Interruption causée par l'arrivée de P3**
  - Trois procs dans la file:
    - P1=5,
    - P2=2,
    - P3=1: ce dernier est choisi
- **Etc**

Préemption ou non

- **La préemption est le cas normal pour l'UCT car l'arrivée d'un nouveau proc cause une interruption et à ce moment là**
  - l'ordonnanceur devra prendre une décision sur le prochain proc à exécuter
- **Mais pour certaines ressources la préemption est impossible:**
  - P.ex. dans une imprimante il faut toujours terminer l'impression courante avant d'en amorcer une nouvelle

Comment déterminer la longueur des cycles à l'avance?

- **En utilisant le passé**
  - méthode de la moyenne exponentielle



Différentes méthodes en principe

- **Hypothèse de comportement constant d'un processus:**
  - Un processus qui a eu des cycles d'UCT de (par ex .) 3 millisecondes en moyenne continuera comme ça
- **Hypothèse de comportement variable d'un processus:**
  - Un processus qui avant avait des cycles d'UCT de 3ms, maintenant a allongé ces cycles, qui sont de 10ms
  - La durée des cycles les plus récents est considérée plus importante pour la prévision des prochains cycles

Estimation de la durée du prochain cycle hypothèse de comportement constant

- **T_i : la durée du *i*ème cycle de l'UCT pour ce processus**
- **S_i : la valeur *estimée* du *i*ème cycle de l'UCT pour ce processus.**
  - S_{n+1} l'estimée courante
  - S_n l'estimée précédente
- **Un choix simple est:**

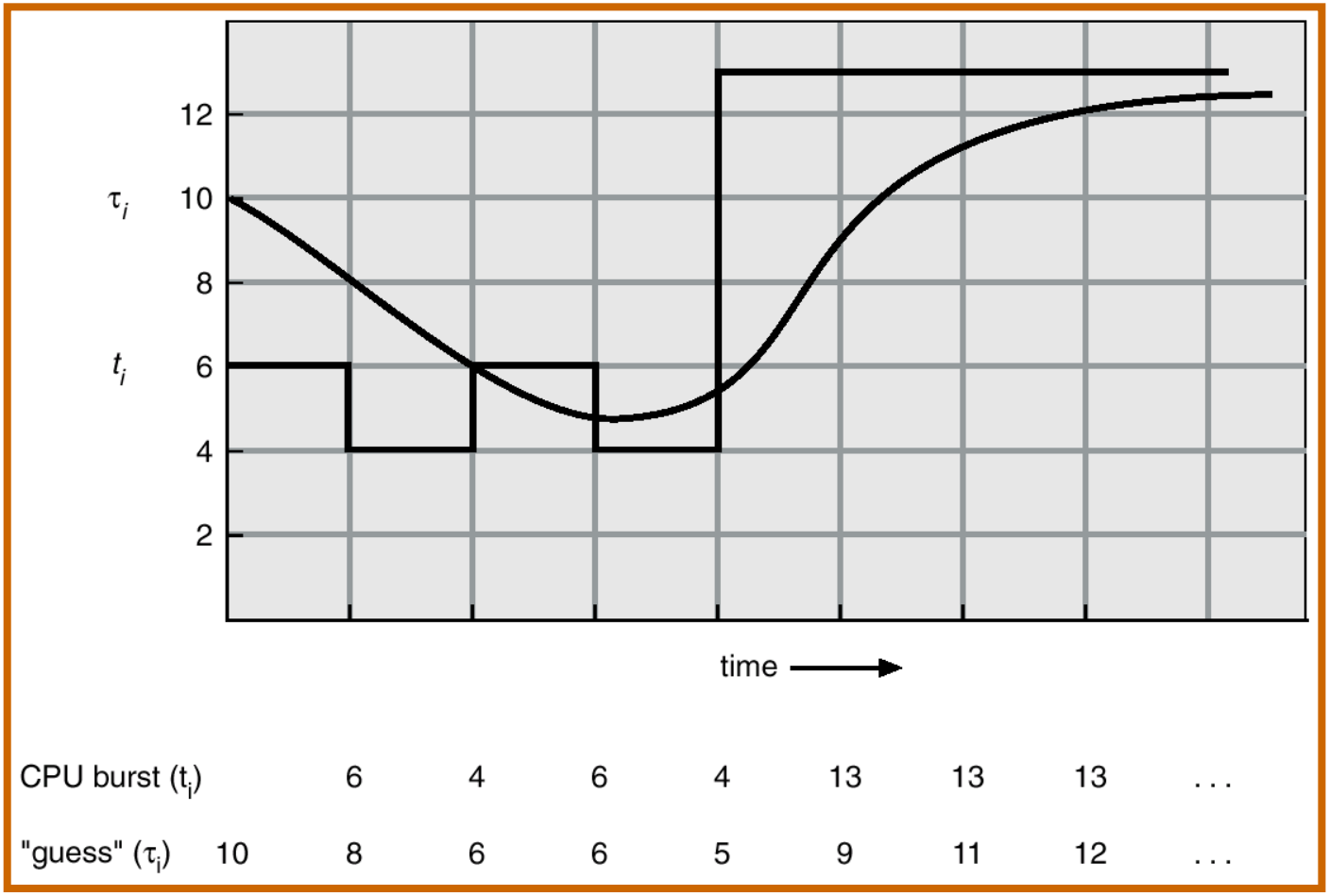
$$S_{n+1} = (1/n) \sum_{i=1 \text{ à } n} T_i \text{ (une simple moyenne)}$$

- **Nous pouvons éviter de recalculer la somme en récrivant:**

$$S_{n+1} = (1/n)\, T_n + ((n-1)/n)\, S_n$$

- **Ceci donne un poids identique à chaque cycle**

Prédiction de la Longueur du Prochain Temps CPU

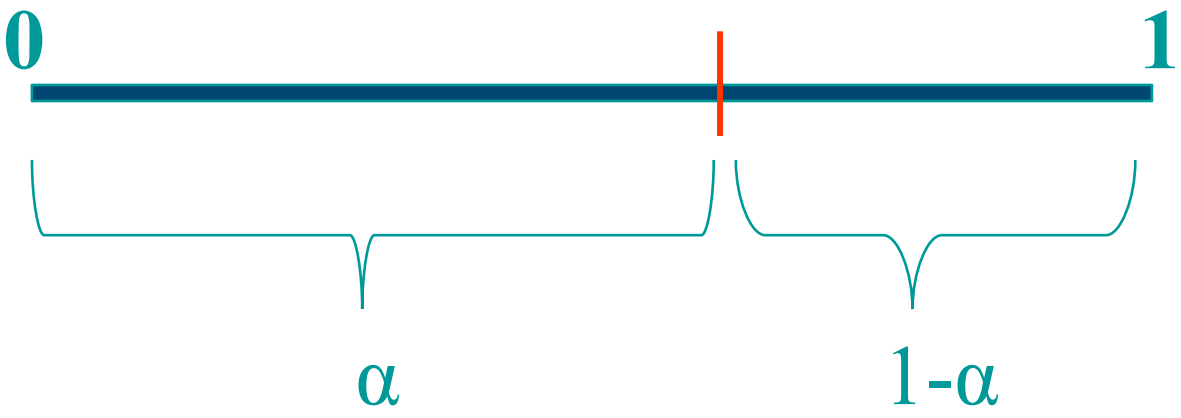


CPU burst (t_i)	6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Estimation de la durée du prochain cycle hypothèse de comportement variable

- **Nous devons décider quelle importance donner**
  - Aux changements plus récents
  - Par rapport aux observations précédentes

Coefficient α pour le poids



Importance du cycle le plus récent

Importance de l'estimée précédente

ex. si $\alpha = 0,7$, alors $1-\alpha = 0,3$

Estimation de la durée du prochain cycle hypothèse de comportement variable

- **Mais les cycles récents peuvent être plus représentatifs des comportements à venir**
- **La moyenne exponentielle permet de donner différents poids aux cycles plus ou moins récents:**

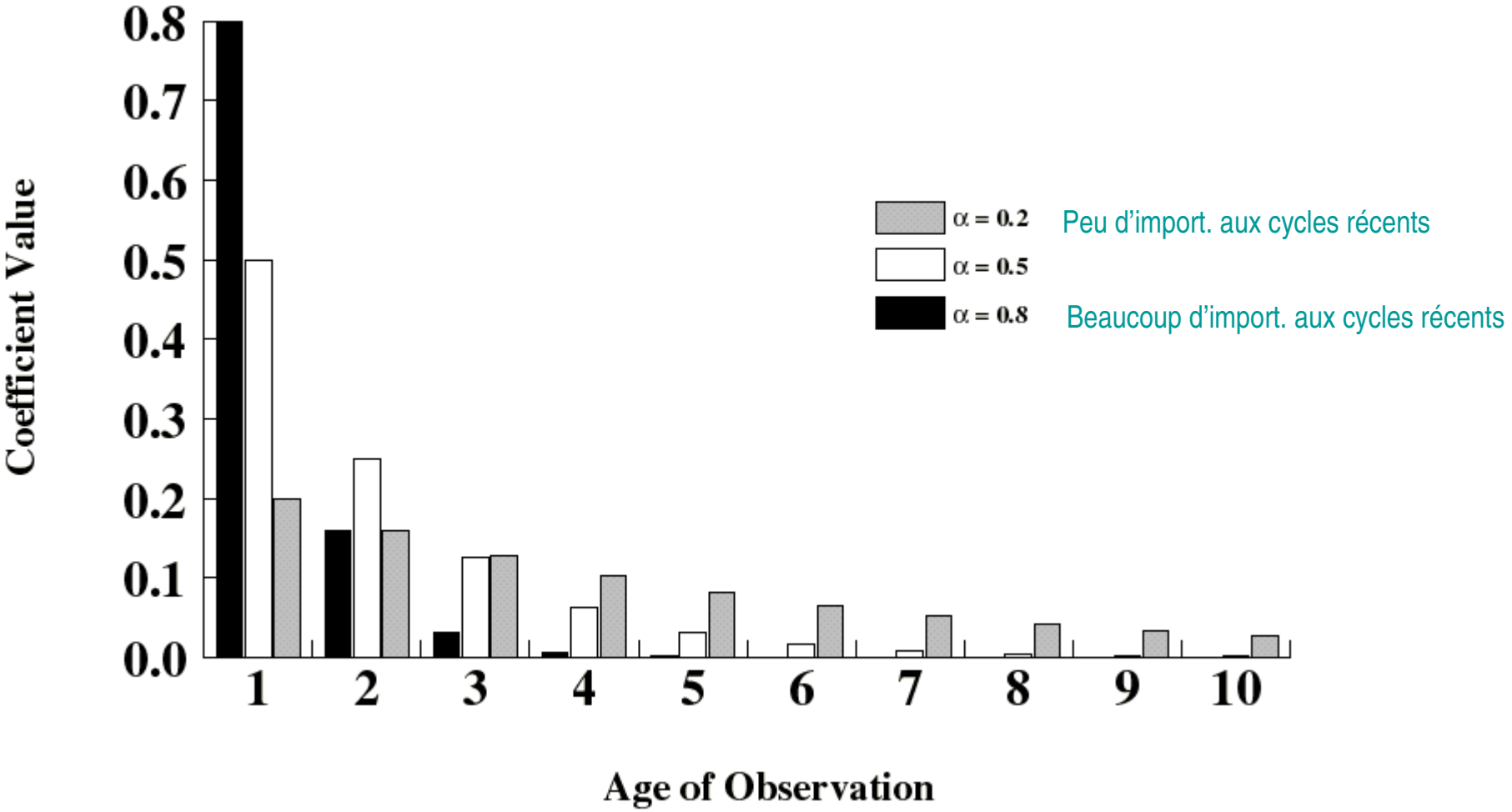
$$S_{n+1} = \alpha T_n + (1-\alpha) S_n \;\; ; \;\; 0 \leqslant \alpha \leqslant 1$$

- **α : le coefficient d'importance**
- **T_n: la durée du cycle le plus récent**
- **S : l'estimée**
  - S_{n+1} l'estimée courante (après le cycle T_n)
  - S_n l'estimée précédente

Pourquoi 'exponentielle'

- **Par développement de l'expression , nous voyons que le poids de chaque cycle décroît exponentiellement**
  - $S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_1$
- **la valeur estimée S_1 du 1er cycle peut être fixée à 0 pour donner priorité max. aux nouveaux processus**

Importance de différents valeurs de coefficients [Stallings]



La durée des vieux cycles perd de l'importance rapidement si on donne beaucoup d'importance aux cycles récents, et réciproquement…

Comment choisir le coefficient α

- **Un petit α assouplit les changements de comportement d'un processus**
  - Il donne moins d'importance aux cycles récents
  - Il est avantageux quand un processus peut avoir des anomalies de comportement, après lesquelles il reprend son comportement précédent
  - Cas limite: $\alpha = 0$
    - on reste sur l'estimée initiale
- **Un grand α réagit rapidement aux changements**
  - Il donne plus d'importance aux cycles récents
  - Est avantageux quand un processus est susceptible de changer rapidement de type d'activité
  - Cas limite: $\alpha = 1$: $S_{n+1} = T_n$
    - Le dernier cycle est le seul qui compte

Le plus court d'abord (SJF) : critique

- **Difficulté d'estimer la longueur à l'avance**
- **Plus pratique pour l'ordonnancement de travaux que pour l'ordonnancement de processus**
  - Normalement on peut plus facilement prévoir la durée d'un travail entier que la durée d'un seul cycle
- **Il y a assignation implicite de priorités: préférences aux travaux plus courts**
  - *Famine* possible pour les travaux aux cycles longs

Difficultés majeures avec les méthodes discutées

- **Premier arrivé, premier servi, FCFS:**
  - Temps moyen d'attente non-optimal
  - Mauvaise utilisation des ressources s'il y a apport continu de processus aux cycles longs (v. effet d'accumulation)

- **Plus court servi, SJF:**
  - Difficulté d'estimer les cycles
  - Famine s'il y a apport continu de processus aux cycles courts

- **Donc besoin d'une méthode *systématiquement préemptive***
  - Le **tourniquet**
  - si vous effectuez toujours les processus les plus courts en premier, vous pourriez avoir l'impression de faire beaucoup mais vous pourriez ne jamais arriver aux plus longs...
  - si vous effectuez les processus dans l'ordre d'arrivée, les longs pourraient vous bloquer pour longtemps
  - donc la solution est de donner un peu de temps à chacun, cycliquement

Le tourniquet

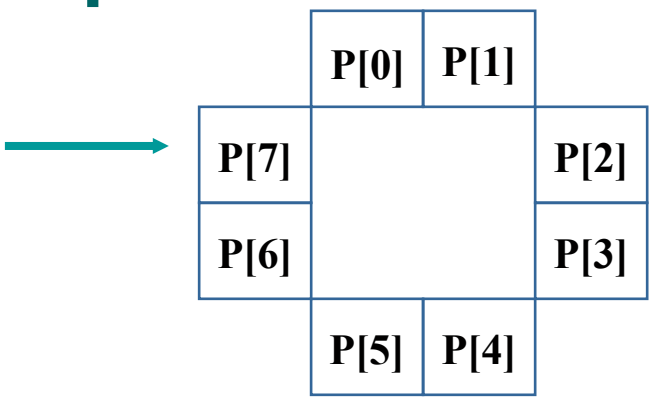
- **Si j'ai une seule grande pizza et plusieurs personnes affamées, je pourrais:**
  - L'offrir à chacun à son tour: attendre que chacun ait fini avant de passer au suivant
    - (méthodes précédentes)
  - Ou sinon offrir une tranche à la fois à chacun et permettre de revenir

Tourniquet = Round-Robin (RR)

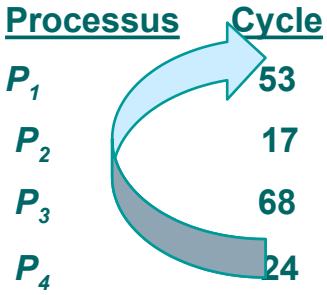
Le plus utilisé en pratique

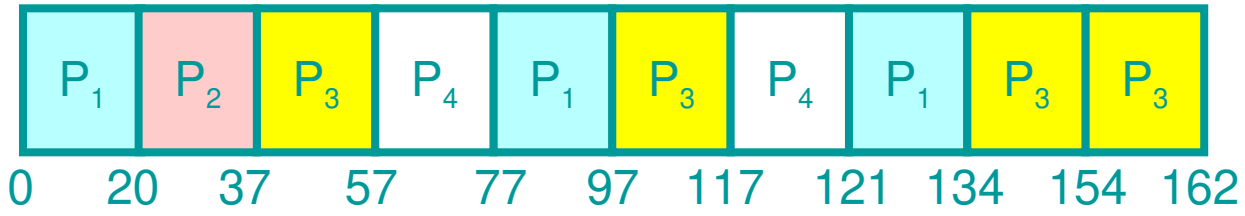
- A chaque processus est alloué une tranche de temps (p.ex. 10-100 millisecs.) pour s'exécuter
  - **Tranche aussi appelée *quantum***
- S'il exécute la tranche entière *sans autres interruptions*, il est interrompu par la minuterie et l 'UCT est donnée à un autre processus
- Le processus interrompu redevient prêt (à la fin de la file)
- Méthode **préemptive**

La file prêt est un cercle (d'où RR)



Exemple: Tourniquet tranche = 20



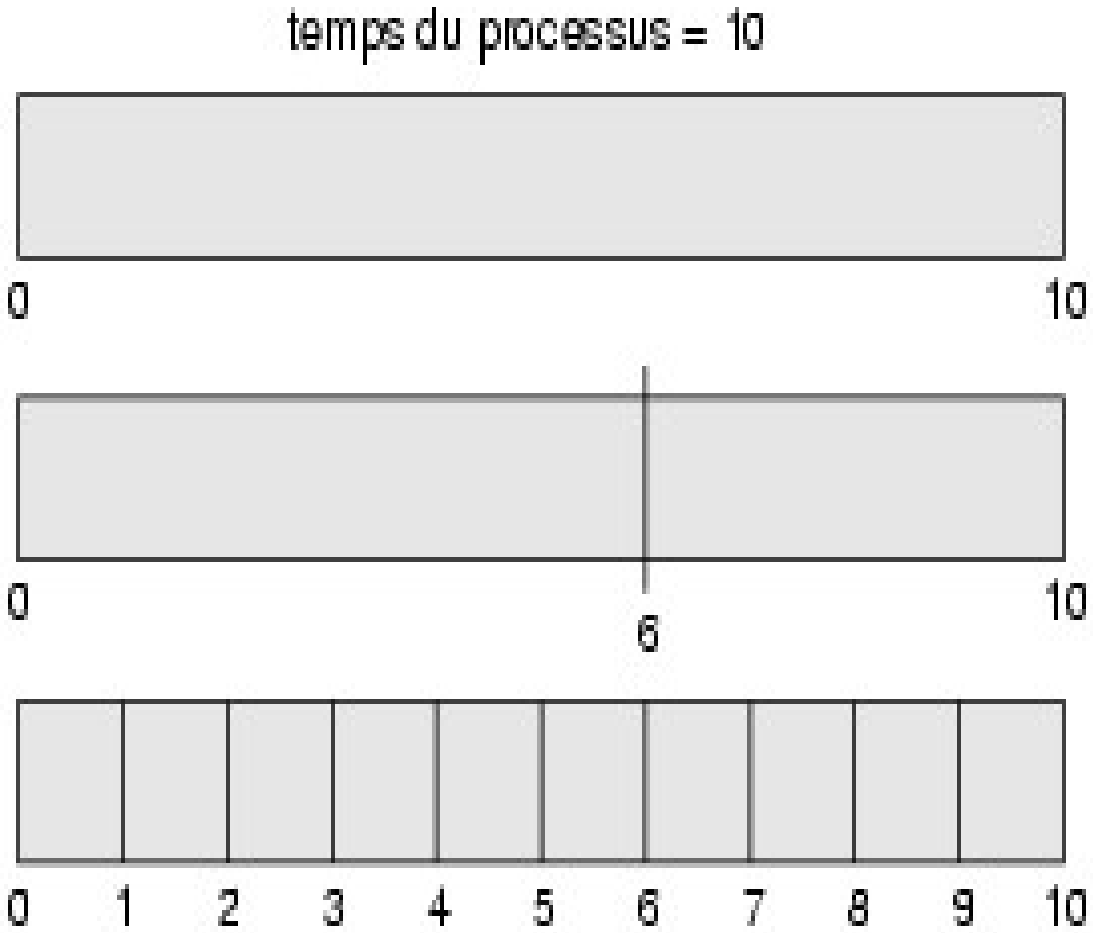


- temps de rotation et temps d'attente moyens beaucoup plus élevés que SJF
- mais *aucun processus n'est favorisé*

Performance du tourniquet

- Si *n* processus sont dans la file d'attente des processus prêts et le quantum est *q*, alors chaque processus reçoit 1/*n* du temps CPU par paquets de *q* unités. Aucun processus n'attend plus de (*n*-1)*q*.
- Si q grand ⇒ FCFS
- Si q petit … ⇒ q *doit être grand comparé au temps de commutation de tâche*, sinon l'overhead est trop grand

Une petite tranche augmente les commutations de contexte (temps de SE)



tranches de temps	changement de contexte
12	0
6	1
1	9

Comment déterminer la tranche en pratique

- **Évidemment la durée médiane/moyenne des cycles des processus en attente est impossible à déterminer avec certitude!**
- **En pratique, les observations passées peuvent être utilisées:**
  - le SE peut garder une trace des durées des cycles des processus récemment exécutés et ajuster la tranche périodiquement

Priorités

- **Affectation d'une priorité à chaque processus (p.ex. un nombre entier)**
  - souvent les petits chiffres dénotent des hautes priorités
    - 0 la plus haute
- **L'UCT est donnée au processus prêt avec la plus haute priorité**
  - avec ou sans préemption
  - il y a une file *prêt* pour chaque priorité

Problème possible avec les priorités

- **Famine: les processus moins prioritaires n'arrivent jamais à s'exécuter**
- **Solution: vieillissement:**
  - modifier la priorité d 'un processus en fonction de son âge et de son historique d 'exécution
  - le processus change de file d'attente
- **La modification dynamique des priorités est une politique souvent utilisée**

Inversion de priorités

- **Situation dans laquelle un processus de haute priorité ne peut pas avoir accès au processeur car il est utilisé par un processus de plus faible priorité.**

- **Exemple :**
  - trois threads A, B, et C, un mutex.
  - Priorité A > priorité B > priorité C

Scénario 1 (2 threads)

- C acquiert le mutex X.
- Un événement (ex : signal) réveille le thread A qui préempte C qui n'a pas libéré le mutex
- A essaye d'obtenir le mutex ; comme il est déjà acquis par C, A est mise en file d’attente.

Le thread de haute priorité (A) n'a donc pas accès à l’UCT mais un de basse priorité (C) y a accès.

→ relacher les mutex rapidement !

Scénario 2 (3 threads)

- C s'exécute, et prend le mutex.
- Un événement réveille B, qui préempte C, qui n'a pas libéré le mutex.
- Un autre événement réveille A, qui préempte B.

  Quand A demande à prendre le mutex, elle est mise en attente, et B continue de s'exécuter.

  B continue de s'exécuter jusqu'à son terme, puis C, et enfin A : la tâche de plus haute priorité s'exécute en dernier !

Solutions

Héritage de Priorité (Priority Inheritance) – Solution Classique

(mutex avec option PTHREAD_PRIO_INHERIT).

Le thread bloquant hérite temporairement de la priorité de la tâche bloquée.

- Dans ex 1 : C (basse priorité) héritera temporairement de la priorité de A (haute priorité) lorsqu'il détient le mutex, permet à C de finir rapidement et de libérer le mutex pour que A puisse s'exécuter.
- Mais risque d’héritage en chaine, complexe à ordonnancer...

Solutions

Plafond de Priorité (Priority Ceiling) – Solution Préventive
(mutex avec option PTHREAD_PRIO_PROTECT).

- Le thread qui verrouille une ressource prend dès le départ la plus haute priorité possible.
- Ex 2 :
  - Quand C (basse priorité) verrouille la ressource critique, son niveau de priorité est élevé au maximum (celle de A).
  - Et B (priorité moyenne) ne pourra pas l'interrompre et C terminera vite son travail.
- Gaspille du temps UCT

Solutions

Exécution Non Préemptive (Turn Off Preemption) – Solution Radicale

- Le thread qui verrouille une ressource désactive temporairement l'ordonnanceur : plus d'interruption possible !
- Mais retarde l'exécution des threads les plus prioritaires.

Solution	Avantages	Inconvénients	Exemples de Systèmes
Priority Inheritance	Empêche l'inversion de priorité, garantit une exécution en temps réel des tâches critiques.	Augmente la complexité du système, peut entraîner une surcharge supplémentaire.	**QNX**, **VxWorks**, **RTEMS**
Priority Ceiling Protocol	Empêche toute inversion de priorité, bon contrôle des ressources partagées.	Peut causer des blocages inutiles, moins flexible.	**LynxOS**, **QNX**, **VxWorks**
Non-preemptive Scheduling	Simple à mettre en œuvre, faible surcharge.	Peut retarder l'exécution des processus à haute priorité.	**FreeRTOS (version non préemptive)**, systèmes embarqués simples

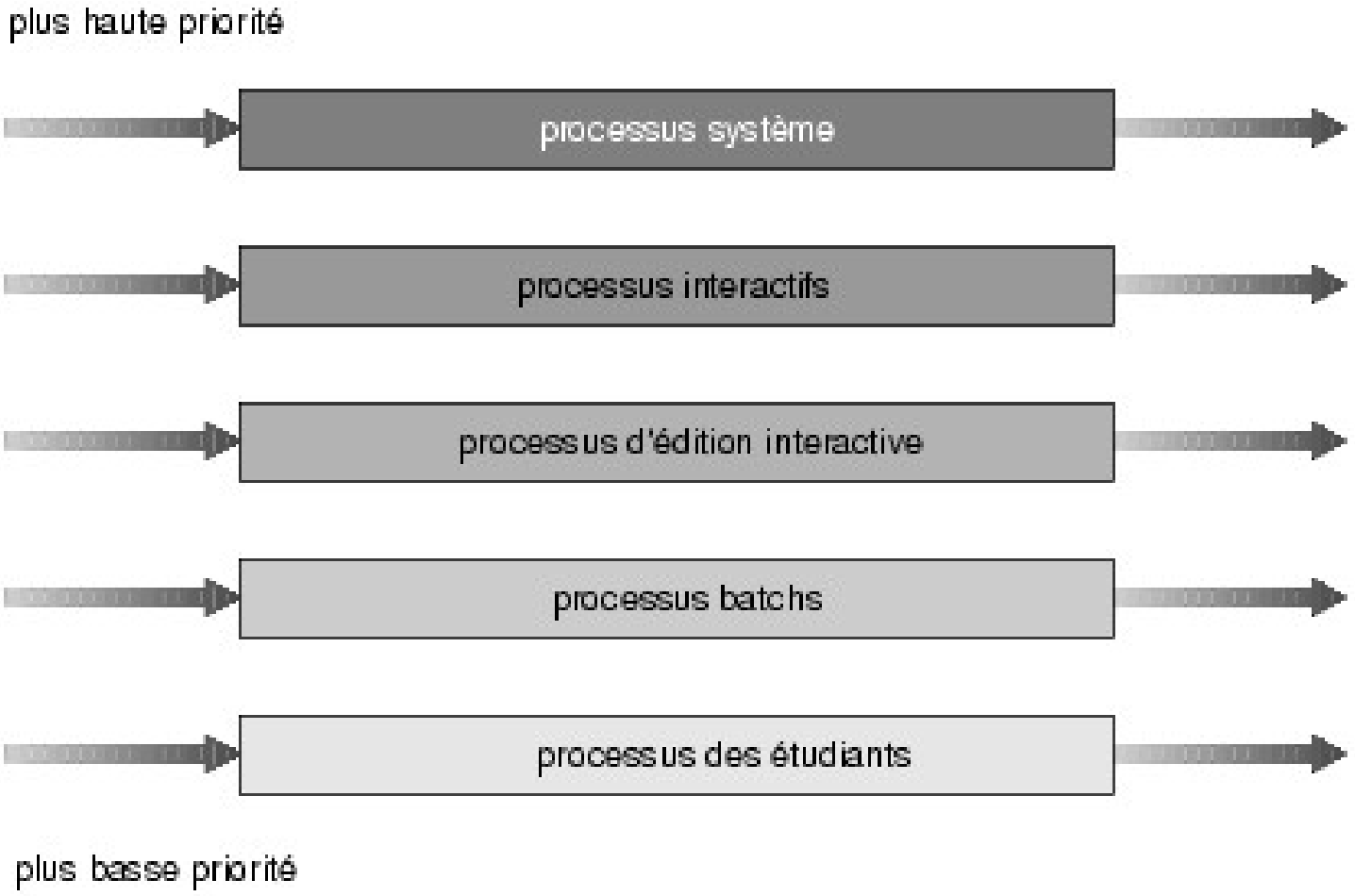
D’après ChatGPT

Remarque : il existe d’autres solutions….

Files à plusieurs niveaux (multiples)

- **La file *prêt* est séparée en plusieurs files, ex.**
  - travaux 'd'arrière-plan' (background - batch)
  - travaux 'de premier plan' (foreground - interactive)
- **Chaque file a son propre algorithme d'ordonnancement, ex.**
  - FCFS pour arrière-plan
  - tourniquet pour premier plan
- **Comment ordonnancer entre files?**
  - Priorité fixe à chaque file → famine possible, ou :
  - Chaque file reçoit un certain pourcentage de temps UCT, p.ex.
    - 80% pour arrière-plan
    - 20% pour premier plan

Ordonnancement avec files multiples
(ex.: serveur d'une université)

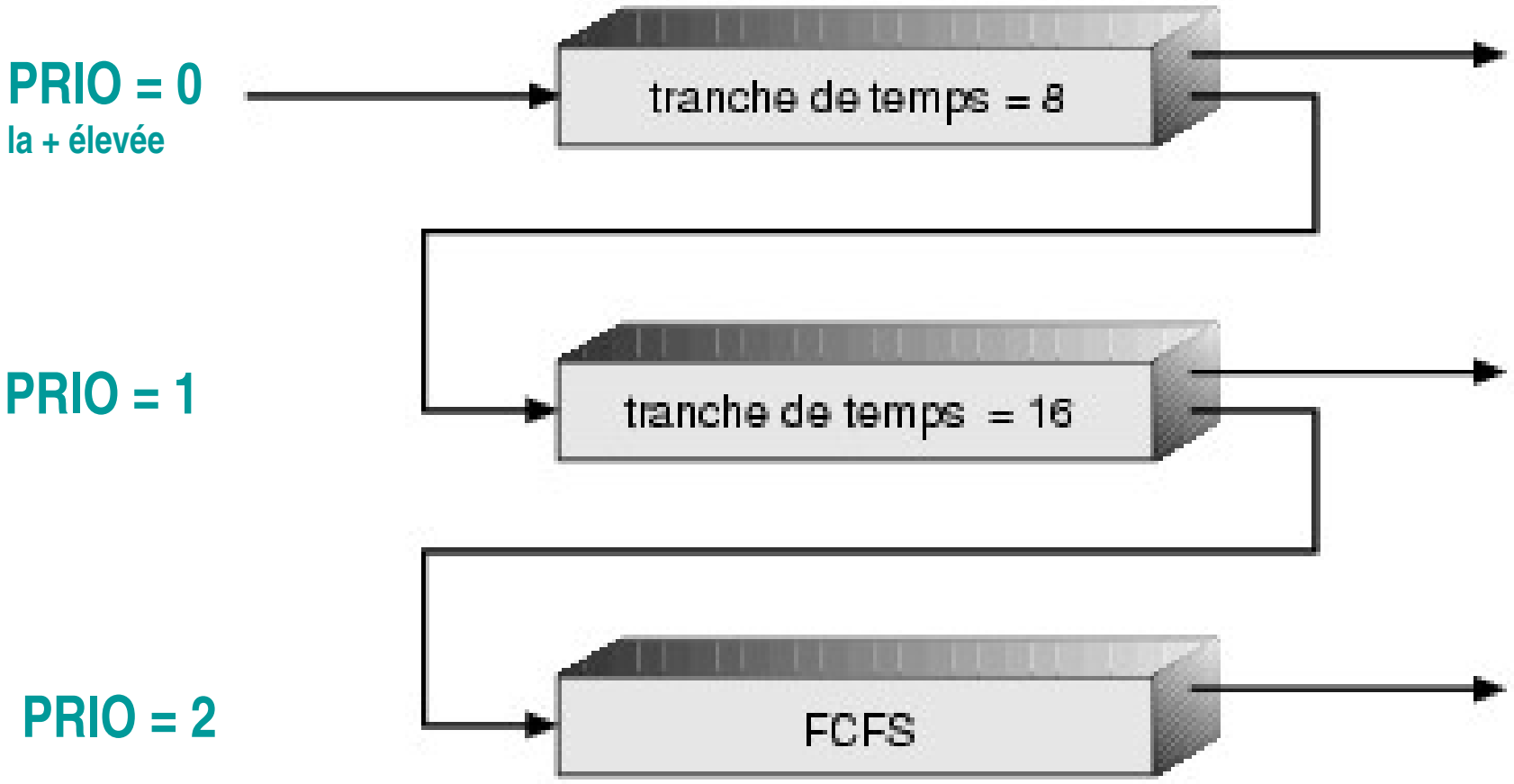


Un proc peut être servi seulement si toutes les files de priorités plus élevées sont vides

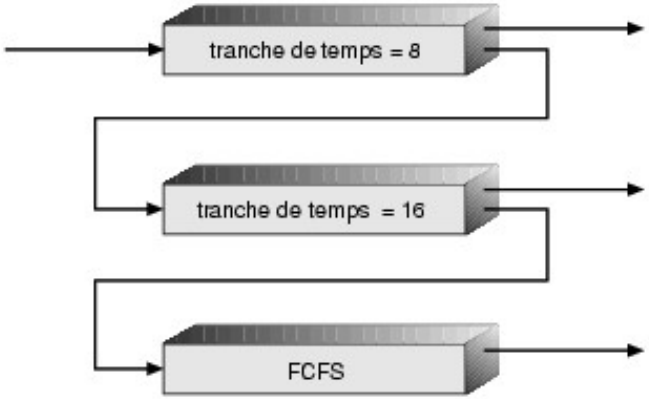
Files multiples et à retour

- **Un processus peut passer d'une file à l'autre, p.ex. quand il a passé trop de temps dans une file**
- **À déterminer:**
  - nombre de files
  - algorithmes d'ordonnancement pour chaque file
  - algorithmes pour décider quand un proc doit passer d 'une file à l'autre
  - algorithme pour déterminer, pour un proc qui devient prêt, la file sur laquelle il doit être mis

Files multiples et à retour



Exemple de files multiples à retour



- **Trois files:**
  - Q0: tourniquet, tranche = 8 msecs
  - Q1: tourniquet, tranche = 16 msecs
  - Q2: FCFS
- **Ordonnancement:**
  - Un nouveau processus entre dans Q0, il reçoit 8 msecs d ’UCT
  - S ’il ne finit pas dans les 8 msecs, il est mis dans Q1, il reçoit 16 msecs additionnels
  - S ’il ne finit pas encore, il est interrompu et mis dans Q2
  - Si plus tard il commence à avoir des cycles plus courts, il pourrait retourner à Q0 ou Q1

En pratique…

- Les méthodes que nous avons vu sont toutes utilisées
- Les SE sophistiqués fournissent aux gérants de grands systèmes des librairies de méthodes, qu'ils peuvent choisir et combiner au besoin après avoir observé le comportement du système
- Pour chaque méthode, plusieurs params sont disponibles: p.ex. durée des tranches, coefficients, etc.
- Ces méthodes évidemment sont importantes surtout pour les grands ordis qui ont des fortes charges de travail

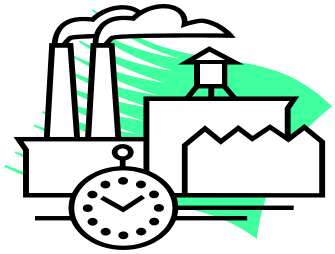
Ordonnancement Multiprocesseur

- L’ordonnancement CPU est plus complexe
- *Processeurs homogènes* dans un multiprocesseur
- *Partage de charge*
- *Multitraîtement asymétrique* – seulement un processeur accède aux structures de données systèmes, supprimant le besoin de partage de données (ex. Coeur haute performance + coeurs economes)

Plusieurs UCTs

- Dans le passé, la vitesse des UCT augmentait rapidement avec chaque génération d'ordinateurs
- Aujourd'hui, l'ingénierie rencontre des limites pour augmenter aussi rapidment la vitesse des UCTs
- Donc on se dirige vers la production de plus d'UCTs par ordinateur
- Cependant, augmenter le nombre d'UCT n'implique pas une augmentation proportionnelle de la puissance de l'ordi :
    - La charge de gestion de ces UCTs ralentit l'ordinateur

Systèmes temps réel

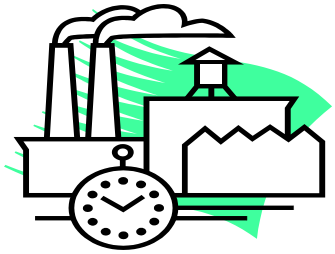


- **systèmes temps réel *rigides (hard):***
  - les échéances sont critiques (p.ex. contrôle d'une chaîne d`assemblage, animation graphique)
  - il est essentiel de connaître la durée des fonctions critiques
  - il doit être possible de **garantir** que ces fonctions sont effectivement exécutées dans ce temps (réservation de ressources)
  - ceci demande une structure de système très particulière
- **systèmes temps réel *souples (soft):***
  - les échéances sont importantes, mais ne sont pas toujours critiques (p.ex. systèmes téléphoniques)
  - les processus critiques reçoivent la *priorité*

Systèmes temps réel: Problèmes d'attente dans plus. systèmes (ex. UNIX)

- Dans UNIX 'classique' il n'est pas permis d'effectuer changement de contexte pendant un appel du système - et ces appels peuvent être longs
- Pour le temps réel il est nécessaire de permettre la préemption des appels systèmes ou du noyau en général
- Donc Unix 'classique' n'est pas considéré comme approprié pour le temps réel
- Mais des variétés appropriées de UNIX ont été conçues (RT)

Systèmes temps réel



Type de STR	Tolérance au retard	Exemples
Durs (Hard RTS)	❌ Intolérable (cause une panne ou un danger)	**Aviation, ABS, chirurgie robotique, fusées**
Fermes (Firm RTS)	⚠️ Toléré mais inutilisable si en retard	**IRM, calculs boursiers, tri logistique**
Souples (Soft RTS)	✅ Toléré mais diminue la qualité	**Streaming, jeux vidéo, navigation GPS**

Merci chatGPT;)

🔴 Systèmes Temps Réel Durs (Hard RTS)

1. **VxWorks** – Aéronautique, spatial (Boeing, Mars Rover, satellites).
2. **QNX** – Automobile (Tesla, BMW), médical, industrie.
3. **Integrity** – Défense, avionique certifiée DO-178C.

🟡 Systèmes Temps Réel Fermes (Firm RTS)

1. **RTLinux** – Finance, robotique, traitement d'images médicales.
2. **Xenomai** – Systèmes de contrôle industriels, robotique.
3. **LynxOS** – Défense, télécoms, ferroviaire.

🟢 Systèmes Temps Réel Souples (Soft RTS)

1. **PREEMPT-RT (Linux)** – Streaming vidéo/audio, jeux en réseau.
2. **FreeRTOS** – IoT, domotique, drones grand public.
3. **Windows CE** – Infotainment automobile, terminaux embarqués.