REDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

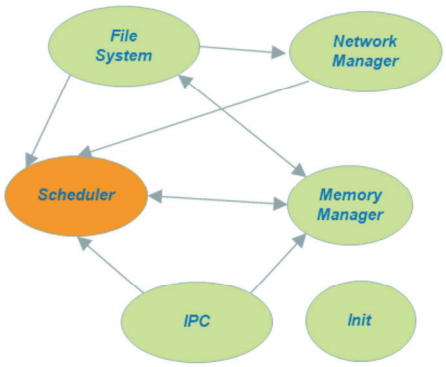
Systèmes d'exploitation (SYE)
Ordonnancement

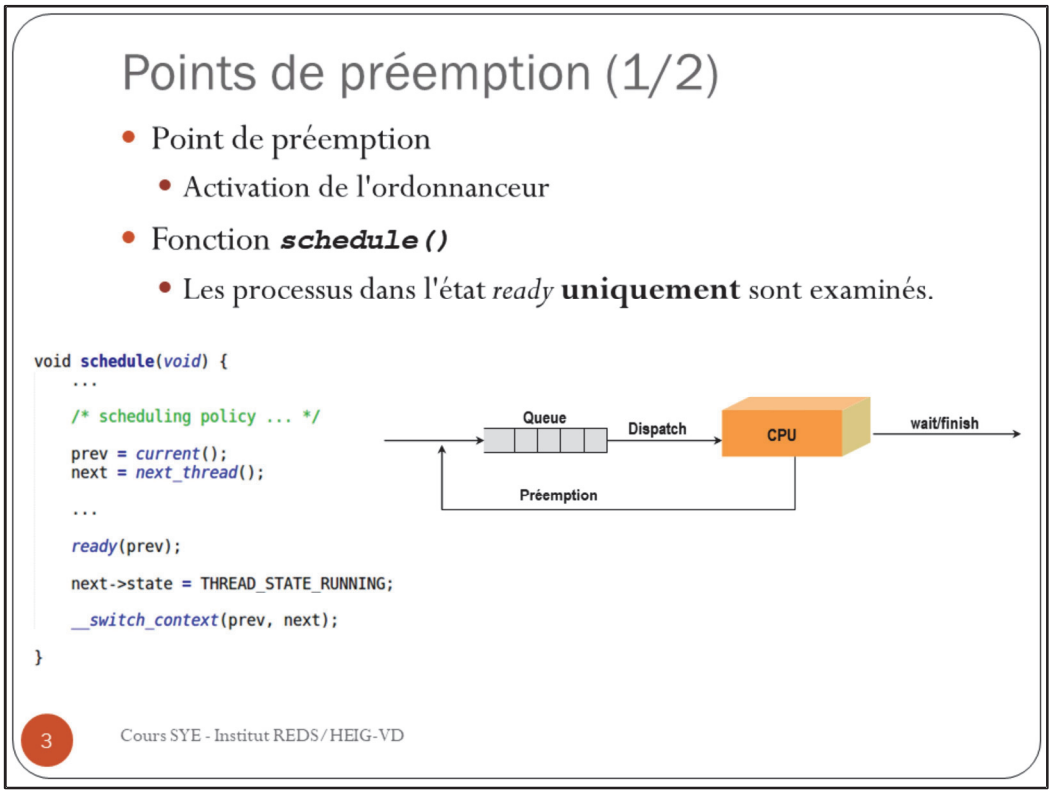
Profs Daniel Rossier, Alberto Dassatti, Salvatore Valenza

Version 3.3 (2017-2018)

Plan

- Points de préemption
- Critères et mesures
- Politiques d'ordonnancement

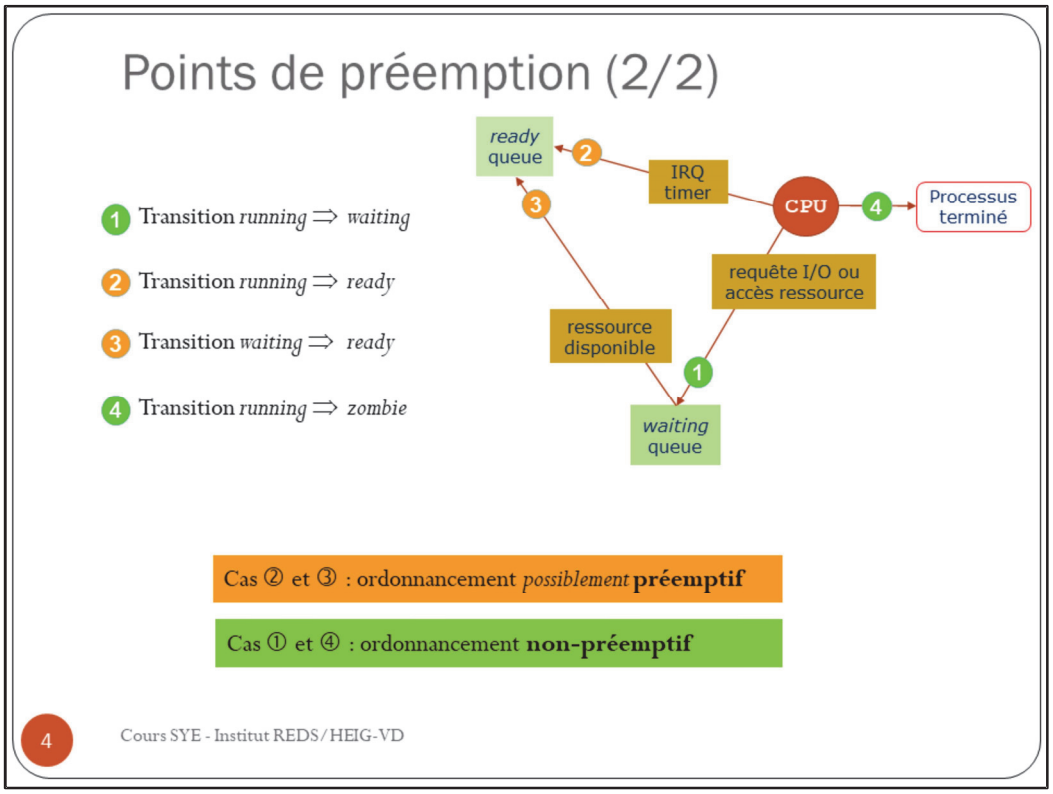




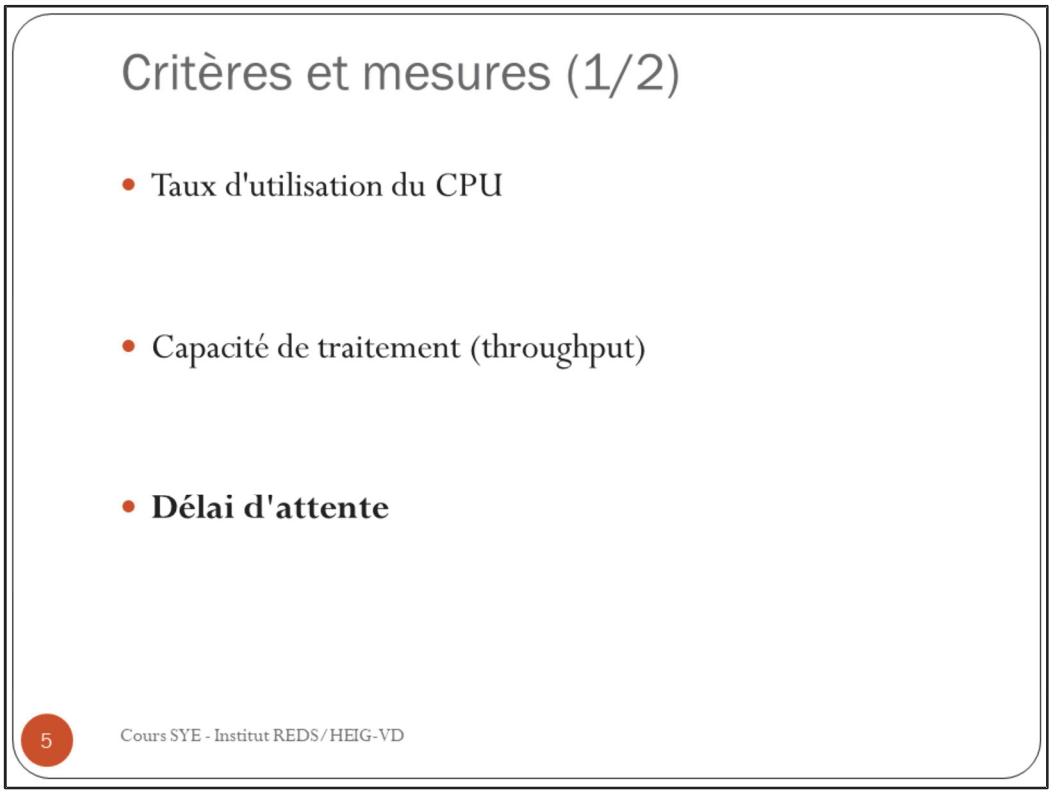
Lorsque plusieurs processus sont présents dans l'état *ready*, l'ordonnanceur doit procéder à la sélection de l'un de ceux-ci pour l'exécuter (passage à l'état *running*). Cette opération – entièrement gérée par le noyau – peut survenir à des instants différents, instants que l'on appelle aussi ***points de préemption***. On trouve des points de préemption généralement à la fin d'une routine de service, ou encore en fin de traitement d'un appel système. A chaque point de préemption, l'ordonnanceur est activé et peut "décider" de préempter éventuellement le processus en cours d'exécution (*running*). On parle également d'une approche *opportuniste.*

L'ordonnanceur est également responsable de *dispatcher* les processus (ou les *threads*) vers l'un ou l'autre processeur (CPU) en fonction de la charge de ceux-ci par exemple, ou si un processus doit impérativement tourner sur un processeur dédié (comme un processeur graphique GPU, ou autres co-processeurs). Les ordonnanceurs modernes peuvent, sous certaines conditions, migrer un ou plusieurs processus d'un CPU à l'autre.

Il est important de relever que l'ordonnanceur peut agir **uniquement** sur les processus dans l'état *ready* (et bien entendu *running* dans le cas d'une préemption).



Lorsque l'ordonnanceur s'exécute, il peut décider de remplacer le processus en cours d'exécution par un autre (par exemple, si le processus qui s'exécute à épuiser son *quantum* de temps). Dans ce cas, le processus dans l'état *running* passera dans l'état *ready* et le processus sélectionné de l'état *ready* vers *running*. Le passage de *running* à *ready* est appelé **préemption**. Un processus préempté est donc un processsus qui "quitte" le processeur à son insu; ce n'est pas le cas si une interruption survient et que celui-ci est *momentanément* suspendu par l'exécution d'une routine de service en mode noyau.

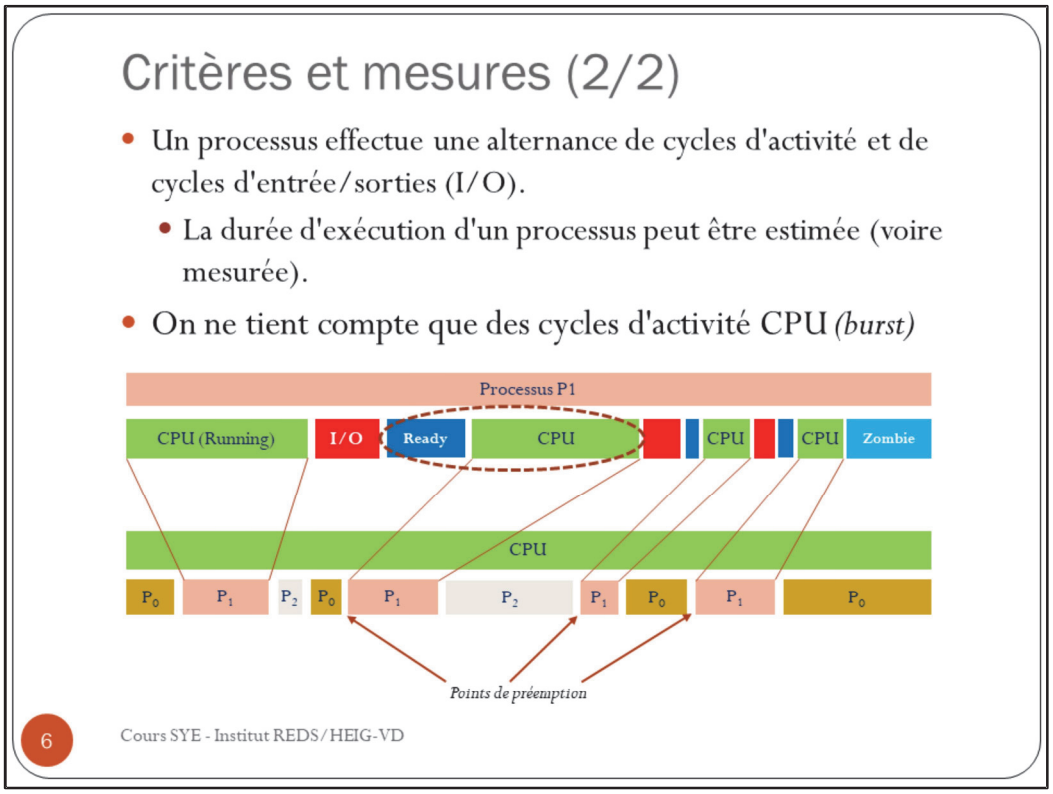


La performance d'une politique d'ordonnancement peut se mesurer avec différents critères:

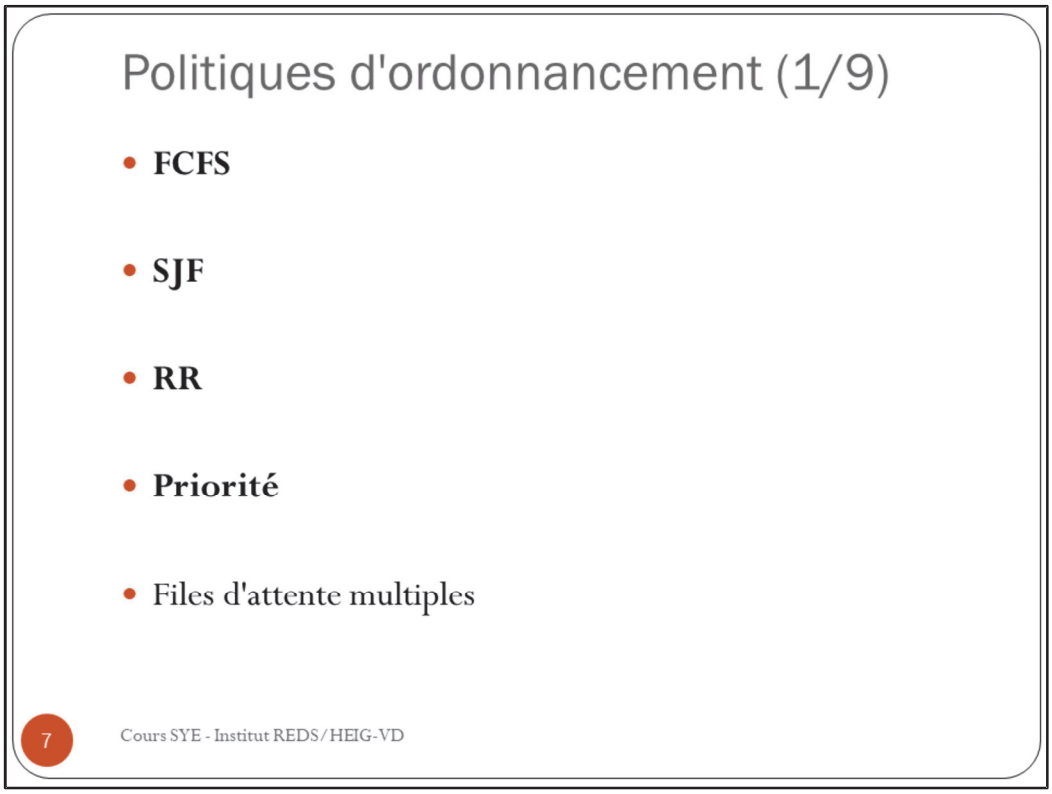
- Le **taux d'utilisation du CPU** qui consiste à calculer le rapport du temps utile (CPU) sur le temps écoulé. Ce critère est intéressant pour les processus de type *batch*, mais difficilement utilisable pour les processus interactifs et lorsqu'il y a beaucoup d'accès I/O (entrées/sorties)

- La **capacité de traitement** (*throughput*) consiste à analyser le débit de processus (combien de processus sont exécutés à la seconde, toutes les heures, etc.). Là aussi, la mesure peut s'avérer compliquée et les résultats sont durs à interpréter. Par exemple, on peut avoir un processus long, et beaucoup de petits processus, conduisant à une capacité de traitement élevée, alors qu'il y a un processus dont le temps de réponse n'est pas satisfaisant.

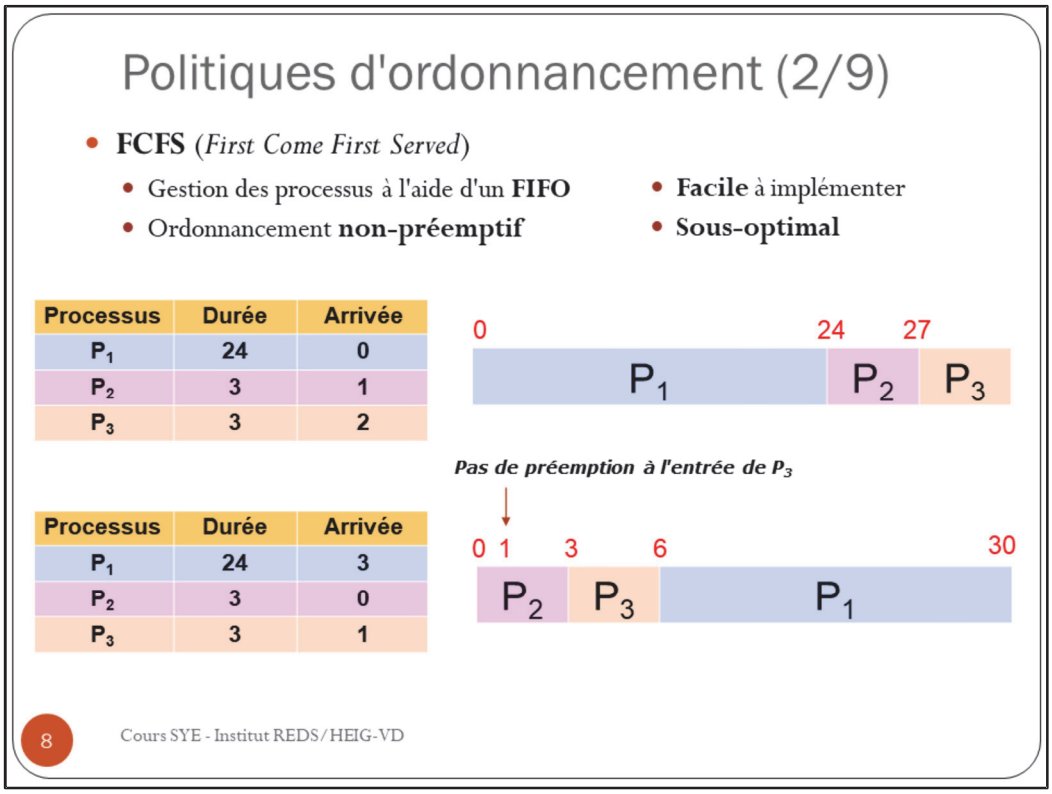
- Le **délai d'attente** consiste à mesurer le temps que passe un processus dans l'état *ready*. Cette mesure est pertinente car on se rappelle que la politique d'ordonnancement ne s'applique sur les processus dans l'était *ready*. Par conséquent, ce critère ne concerne pas les processus qui sont dans l'état *waiting*.



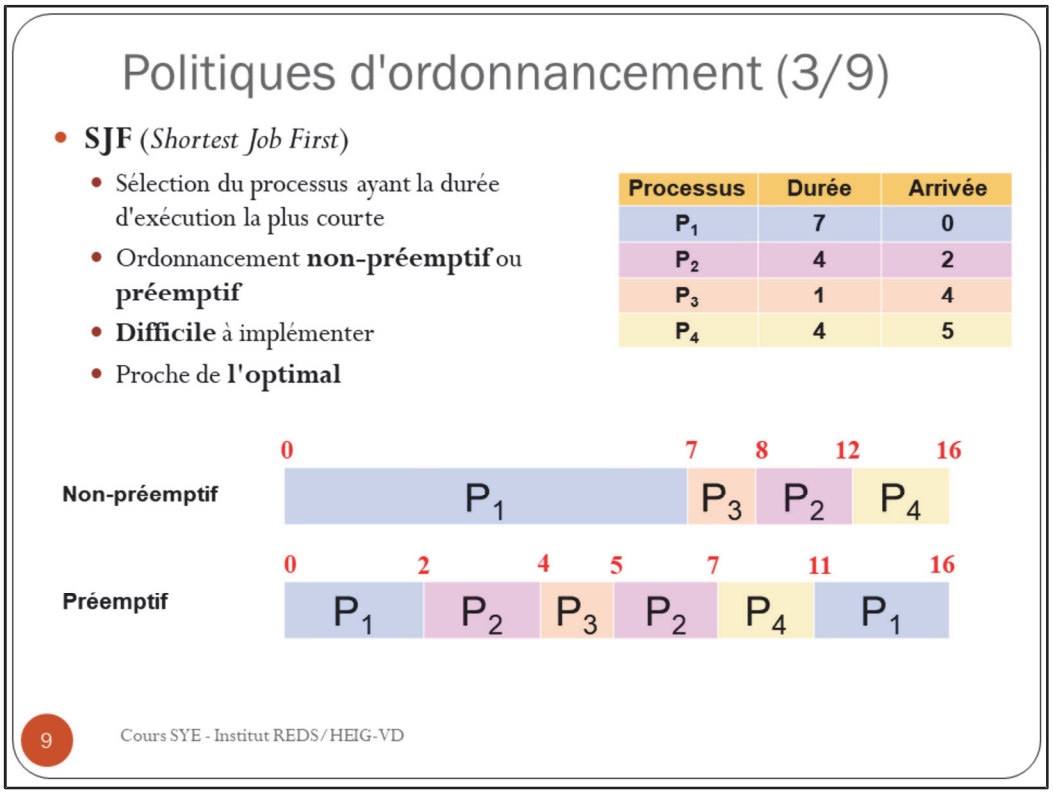
Un processus effectue une alternance de cycles CPU (aussi appelé *burst*) et de cycles I/O (interactions avec le matériel). Comme l'ordonnanceur n'a pas d'influence directe sur les processus dans l'état *waiting*, on s'intéressera particulièrement aux moments où les processus disposent du processeur, c-à-d du moment où ceux-ci se retrouvent dans l'état *running* et à ceux où les processus seront dans l'état *ready*. C'est pourquoi, le critère le plus intéressant pour la mesure de performance sera le *délai d'attente*.



Les différentes politiques d'ordonnancement de base que nous allons étudier constituent le fondement des politiques les plus évoluées que l'on retrouve dans les noyaux d'OS aujourd'hui. Hormis le fait qu'elles sont utilisés dans un noyau d'OS, ces politiques sont intéressantes à plus d'un titre et peuvent être utiles dans toutes applications nécessitant de l'ordonnancement. Les recherches dans ce domaine sont vastes et ont fait l'objet de plusieurs décennies de différents travaux; l'ordonnancement est un sujet très lié aux mathématiques.



La politique FCFS est la plus simple à implémenter, car elle ne nécessite que la gestion d'une queue de type FIFO. C'est aussi la moins bonne du point de vue des performances car elle ne tient pas compte de la nature des processus (priorité, durée, etc.). Par exemple, un processus long peut monopoliser le système durant longtemps au détriment de petits processus (par processus, ici, il faut comprendre la notion de *burst* tel que présenté précédemment).



La politique SJF repose sur l'analyse des durées d'exécution des *futurs bursts* à exécuter. Autrement dit, on examine les processus dans l'état *ready* et on sélectionne celui qui nécessitera le moins longtemps le processeur.

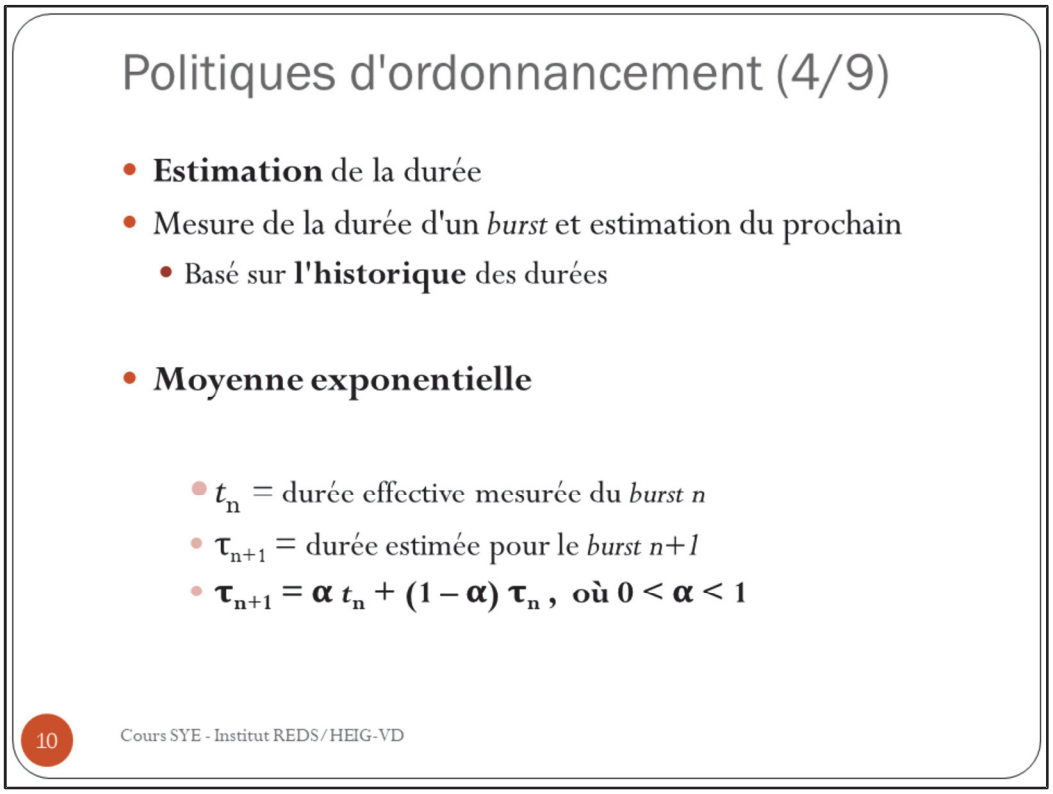
Le SJF se décline en deux versions: une version non-préemptive, c-à-d que l'ordonnanceur ne peut pas retirer un processus dans l'état *running* à son insu. La version préemptive le peut, et on considèrera alors la durée **restante** à exécuter lors de la prochaine sélection. Dans ces conditions, la politique SJF consiste à retenir le processus dont la durée restante est la plus courte (*shortest (remaining) job first).*

Du point de vue du délai d'attente, cette politique est optimale. En effet, si l'on considère 4 processus P_a, P_b, P_c, P_d, avec les durées respectives a, b, c et d et que, sans autres restrictions, on considère que a < b < c < d, alors un ordonnancement SJF consistera à sélectionner les processus dans cet ordre:
$P_a – P_b – P_c – P_d$ conduisant aux délais d'attente suivants:

- Pour P_a, le délai d'attente = 0
- Pour P_b, le délai d'attente = a
- Pour P_c, le délai d'attente = a+b
- Pour P_d, le délai d'attente = a+b+c

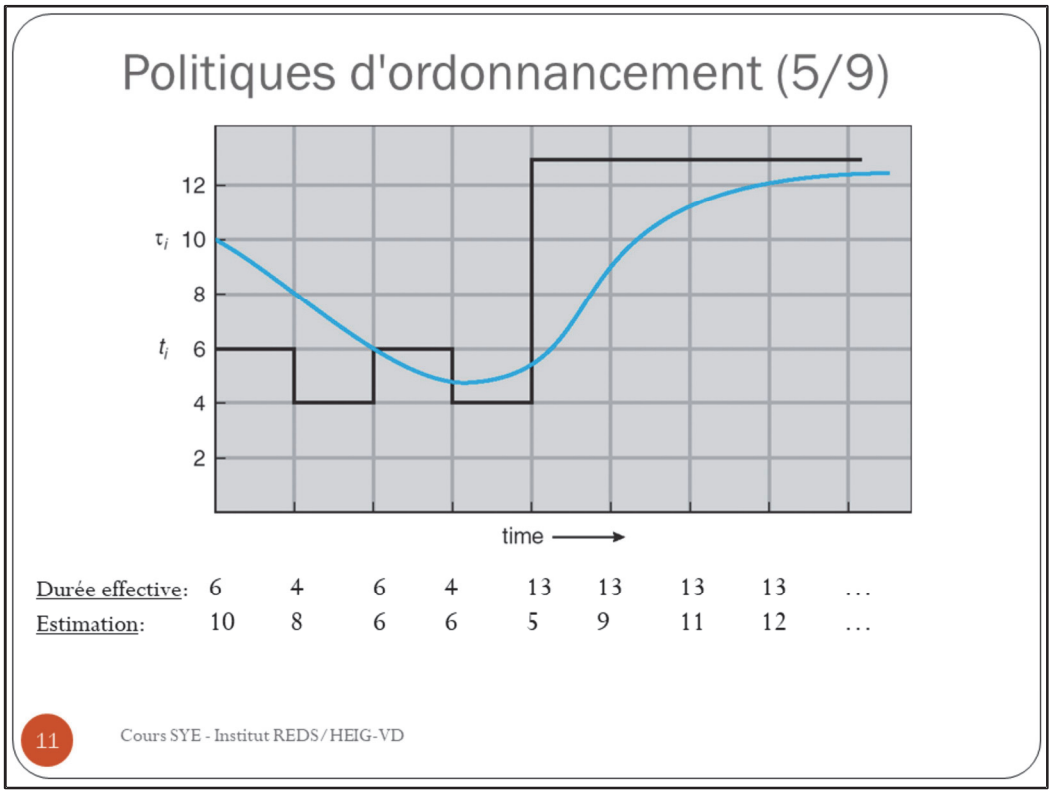
Délai d'attente moyen = $\frac{1}{4}(3a + 2b + c)$

Et comme a < b < c par hypothèse, ce délai ne peut pas être plus petit.



Un des problèmes majeurs de la politique SJF réside dans l'incapacité de connaître "à l'avance" les durées d'exécution des processus (et des *bursts* à fortiori). C'est pourquoi l'algorithme ne peut être utilisable que si l'on dispose d'une estimation de ces durées. L'utilisation d'une moyenne exponentielle est intéressante car elle permet de tenir compte des estimations précédentes tout en diminuant leur effet (de manière exponentielle) au fil du temps.

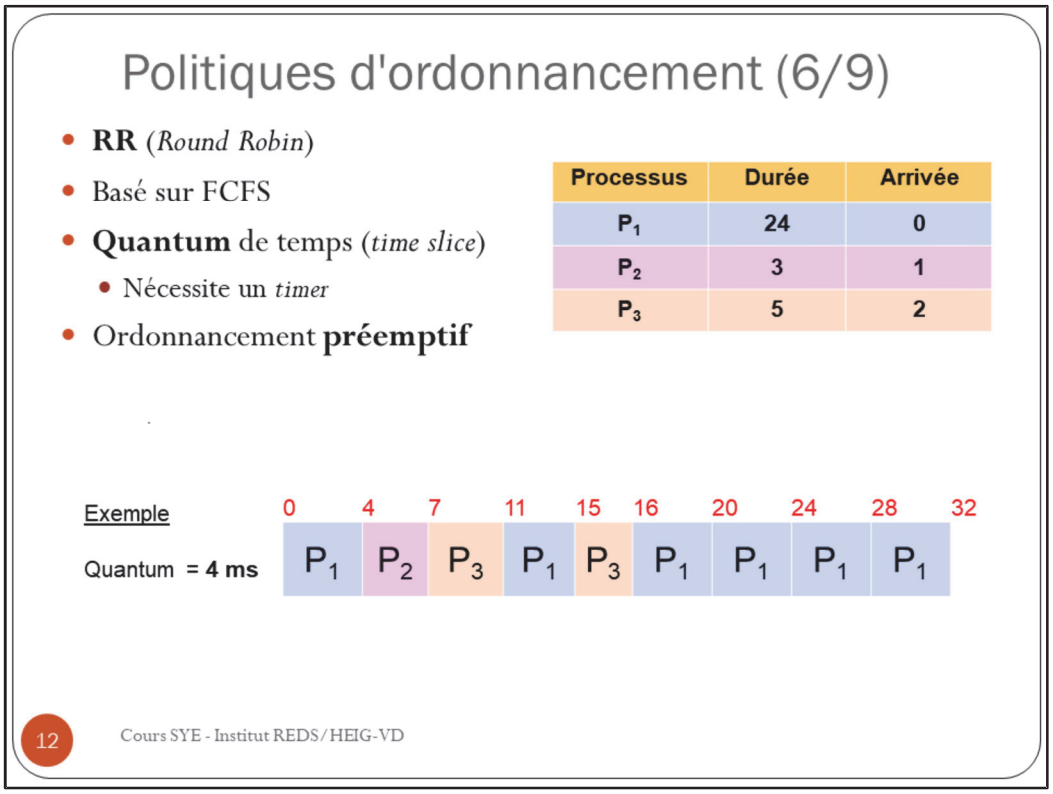
Le facteur α permet de pondérer l'importance donnée aux estimations précédentes par rapport à une nouvelle mesure. Si le facteur α est élevé (proche de 1), une mesure très différente prendra le dessus par rapport aux estimations précédentes (correction rapide); en revanche, si ce facteur est faible, une variation subite de la durée n'aura que peu d'incidence sur l'ensemble des valeurs.



Dans le cas de la figure ci-dessus, le facteur α vaut 0.5

On remarque que l'approximation reste fidèle à la réalité et rend utilisable la politique *SJF*.

Il n'en reste pas moins que la politique *SJF* est sujette au problème potentiel de **famine** des processus. En effet, si des processus de courtes durées d'exécution arrivent continuellement dans le système, les processus de durées supérieures ne seront jamais exécutées. C'est pourquoi cette politique comme d'autres sera combinée avec des techniques d'ordonnancement plus sophistiquées, basées notamment sur les priorités dynamiques, permettant de traiter ce problème. Ces techniques sont abordées dans ce chapitre.

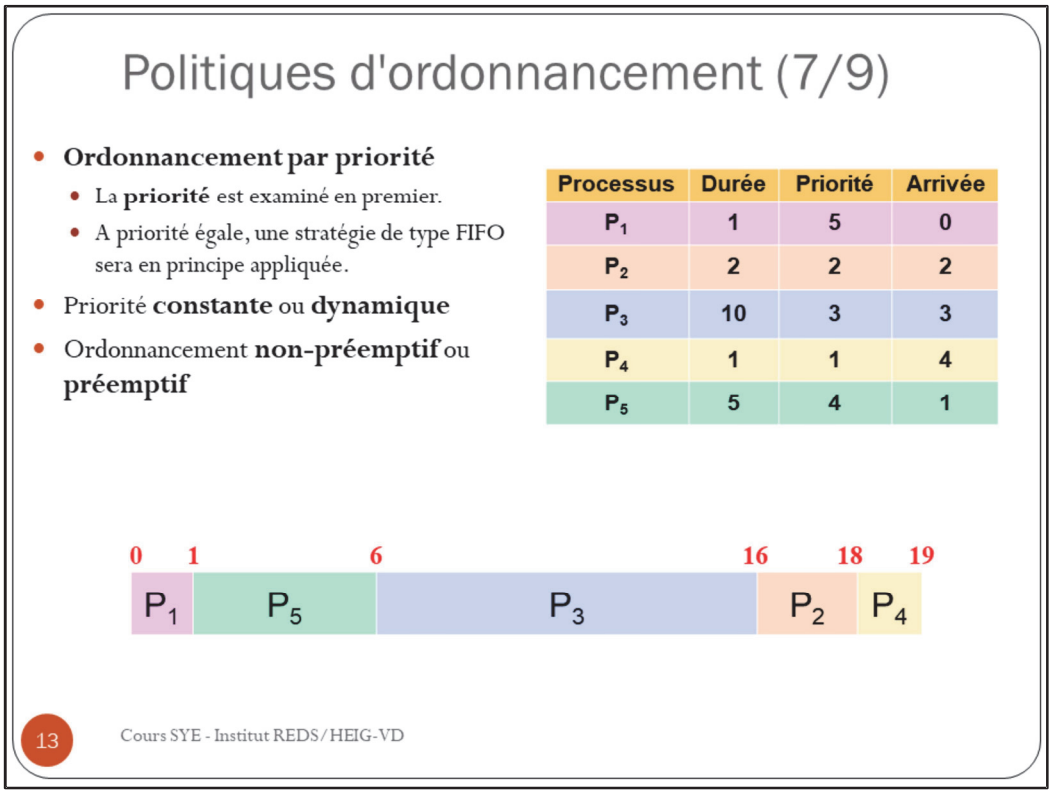


La politique RR est très utilisé en système d'exploitation et constitue la base des systèmes à temps partagé. Basé sur la notion de temps, chaque processus dispose d'un quantum de temps constant attribué à chaque tour d'exécution d'un processus. Arrivé à la fin de son quantum de temps, l'ordonnanceur préempte le processus en cours d'exécution et sélectionne le prochain dans la liste selon une politique FCFS.

Dans la version canonique de l'algorithme RR, le quantum de temps est fixe et ne varie pas. Une valeur trop faible du quantum conduira inexorablement à un nombre élevé de changement de contexte, ce qui conduira à *l'écroulement* du système puisque tout le temps processeur sera dévolu à du code noyau. A l'inverse, une valeur trop grande du quantum fera que la politique RR ressemblera à une politique FCFS et ne sera pas efficace. La valeur "idéale" du quantum est estimé de manière empirique de telle manière que 80% des *burst* se terminent durant ce quantum (20% conduisent à une préemption).

En principe, les valeurs typiques du quantum oscillent entre 10 et 200ms.

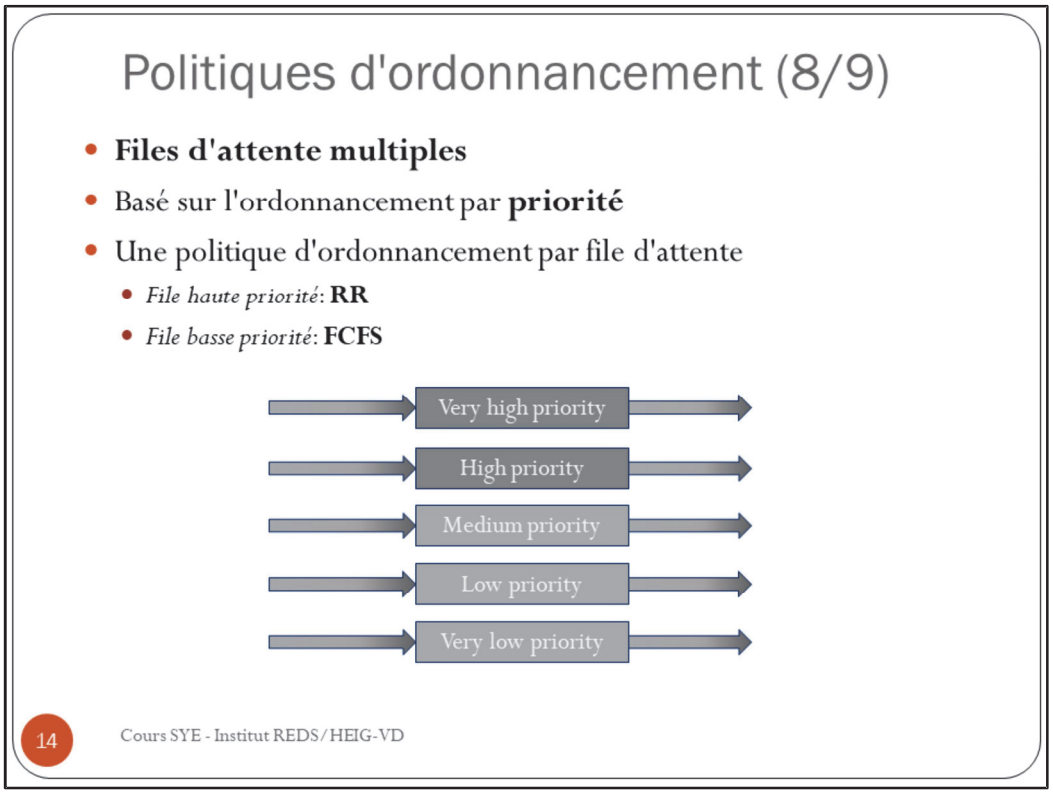
Les versions plus évoluées de la politique RR font que l'ordonnanceur peut décider de changer la valeur du quantum de manière dynamique, en fonction d'autres paramètres comme la priorité, la durée d'inactivité, etc. Ces variantes ne sont pas discutées dans ce cours.



Nous avons vu qu'un processus (ou un *thread*) peut disposer d'une priorité. Un ordonnancement par priorité consiste à examiner la priorité des processus dans l'état *ready* et à sélectionner celui dont la priorité est la plus élevée.

Différentes variantes existes: la priorité peut être attribué à la création du processus et rester constante tout au long de son existence. Cette approche peut conduire à une *famine* des processus à basse priorité. C'est pourquoi une augmentation graduelle de la priorité d'un processus inactif depuis un certain temps peut grandement améliorer les temps de réponses. Lorsque le processus a "réussi" à s'exécuter, il retrouvera sa priorité initiale.

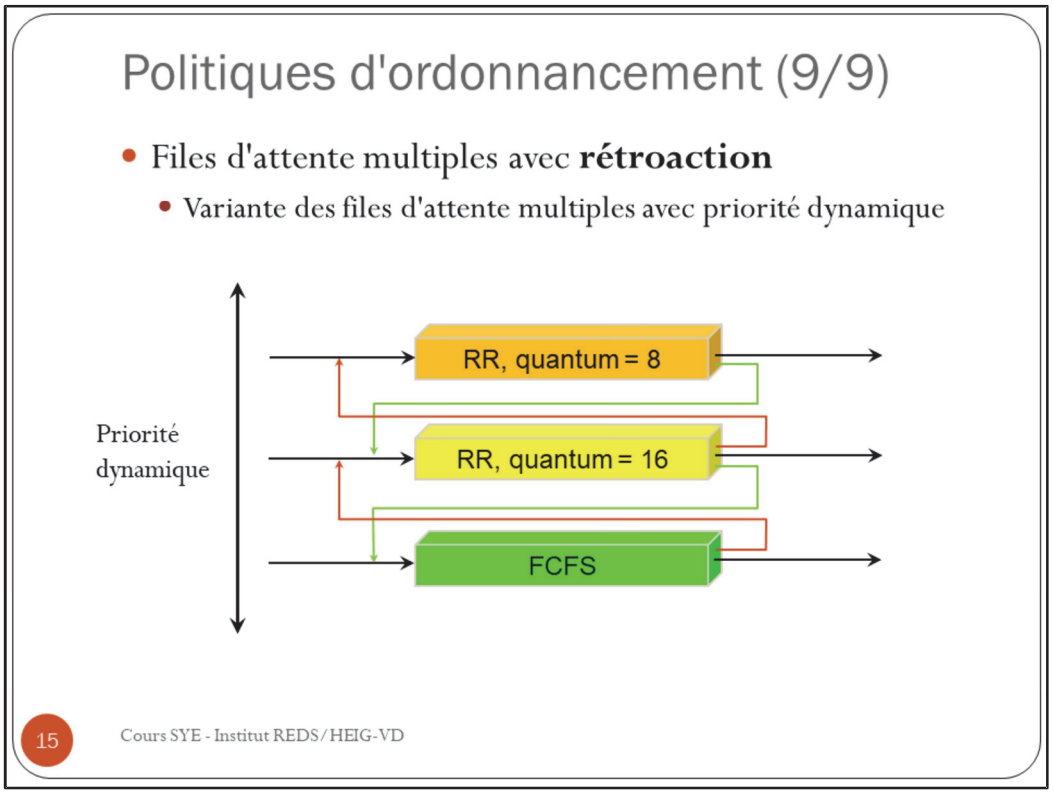
Lorsque l'on parle de priorité, il faut toujours s'interroger sur les niveaux de priorités, à savoir leur nombre et leur signification. Dans certains OS, une priorité de 1 signifie une priorité élevée, alors que d'autres considèreront cette valeur comme la priorité la plus basse. Dans ce cours, plus la valeur est élevée, plus la priorité est élevée.



La politique d'ordonnancement basée sur des files d'attente multiples consiste à ranger les processus dans une catégorie particulière, et chaque catégorie est liée à une file d'attente à laquelle **on attribue une priorité**. Il faut comprendre que la priorité dans ce cas ne concerne que la file, et **non le processus**: un processus à très basse priorité pourrait se retrouver dans une file à très haute priorité.

L'algorithme en premier lieu sélectionnera la file d'attente disposant de la priorité la plus élevée, puis l'ordonnanceur appliquera la politique interne à la file.

Le problème de cette politique réside dans le fait que les processus doivent être "catégorisés". Or, il n'est pas évident de savoir si un processus doit appartenir à l'une ou l'autre catégorie (par exemple, processus de type interactif, processus de type batch, processus temps-réel, etc.): le noyau ne dispose pas de ce type d'information. Par conséquent, on attribuera une politique d'ordonnancement à une file d'attente **en fonction de la priorité** de celle-ci plutôt qu'en fonction d'une catégorie de processus.



Si l'on considère une politique à files d'attente multiples comme précédemment, on retombe sur la problématique de *famine* si la priorité reste constante. Une solution avait alors été proposée pour une politique par priorité en augmentant graduellement la priorité des processus inactifs. C'est le même principe qui sera utilisé dans le cas des files d'attente multiples: si un processus n'a pas été exécuté depuis trop longtemps, celui-ci pourra *migrer* d'une file à une autre (de priorité supérieure) jusqu'à ce que le processus soit exécuté. Attention! Il s'agit de changer le processus de file, et non de changer la priorité de celui-ci.

Références

- A. Silberschatz et al.:
  "**Operating Systems Concepts**", 8th edition, Wiley

- Andrew Tanenbaum
  "**Systèmes d'exploitation**", 3ème édition