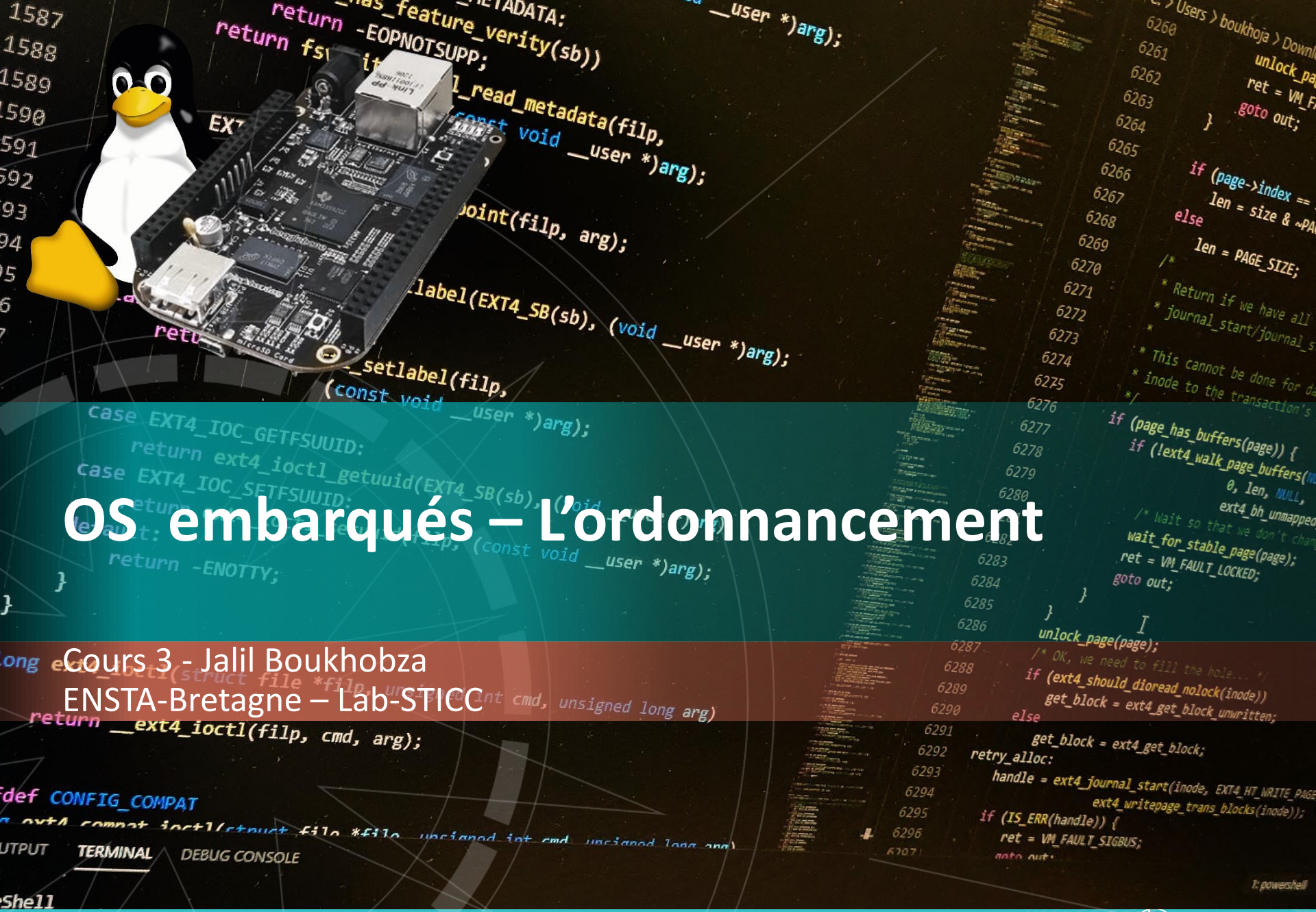


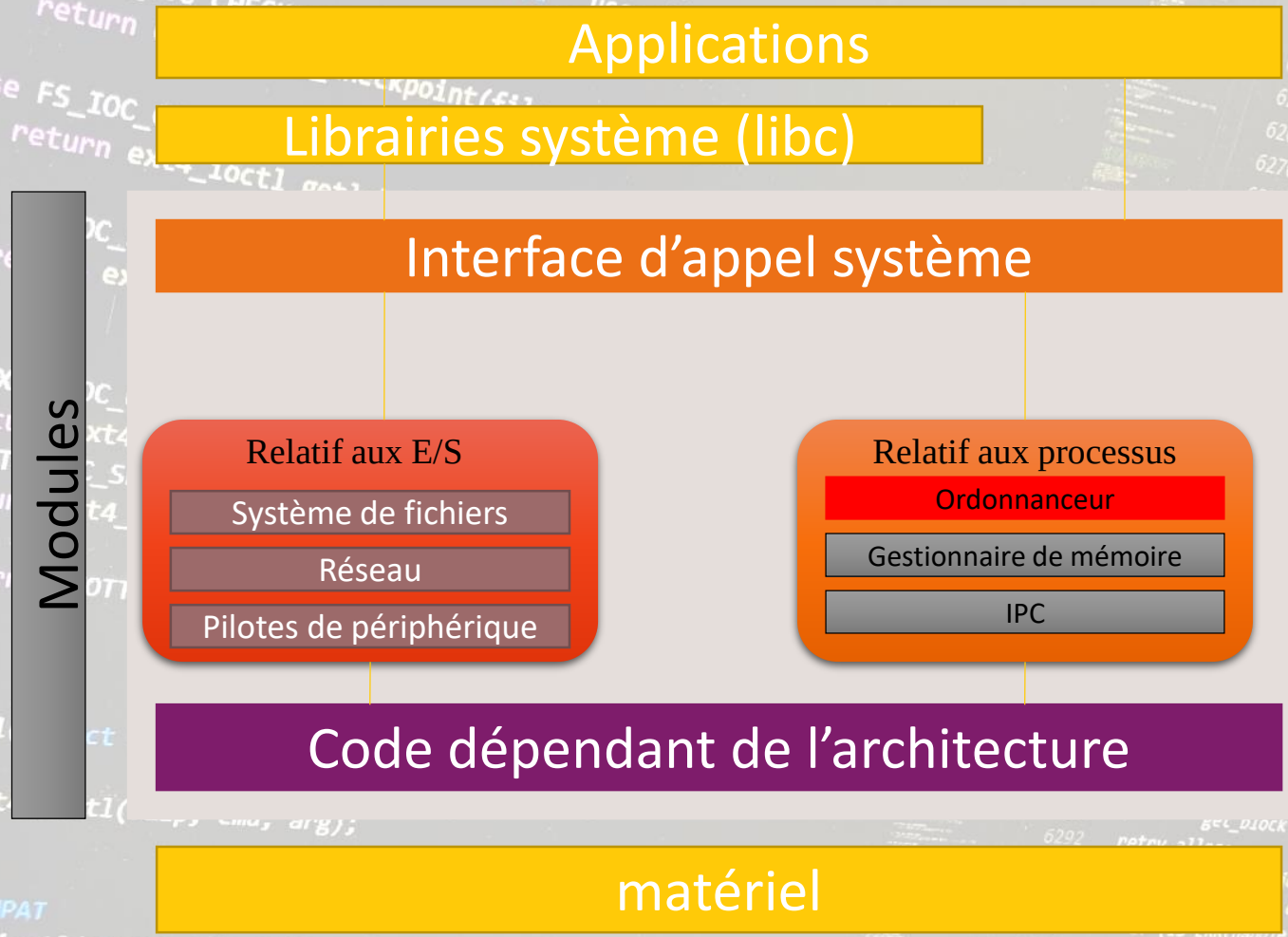
OS embarqués – L'ordonnancement

Cours 3 - Jalil Boukhobza
ENSTA-Bretagne – Lab-STICC

Plan

1. L'ordonnancement
   1. Définition
   2. Métriques
   3. Objectifs
2. Les algorithmes d'ordonnancement
3. L'ordonnancement sous VxWorks, Jbed et Linux TimeSys
4. L'ordonnancement sous Linux 2.6.x.

Schéma d'un OS



Tâches de l'ordonnanceur

- L'ordonnanceur est le composant de l'OS qui détermine l'ordre et la durée des tâches qui s'exécutent sur le CPU.
- L'ordonnanceur dicte l'état dans lequel doivent se trouver les tâches.
- Il charge et décharge le bloc de contrôle de la tâche (descripteur de processus ou task_struct dans le cas de Linux).
- Certains OS utilisent un processus séparé pour allouer le CPU au processus nouvellement sélectionné, il s'appelle le **dispatcheur**.
- Tous les processus ne sont pas égaux
  - **L'interactivité** est importante
  - Les tâches de **fonds**… moins

L'ordonnancement

- Passer d'un processus à un autre est **coûteux**
  1. Sauvegarde de l'état du processus
  2. Chargement du nouvel état (registres, mémoire…)
  3. Démarrage du nouveau processus
  4. Invalidation probable du cache mémoire

- Les processus alternent souvent des phases de calcul avec des phases d'E/S
  - Si phases de calcul très importantes : *CPU-Bound*
  - Si E/S : *I/O-Bound*

Métriques clés

- **Temps de réponse de l'ordo:** temps pris par l'ordonnanceur pour effectuer le changement de contexte pour une tâche prête, ce temps inclut le temps d'attente de la tâche dans la file des tâches prêtes.
- Le temps pris par le processus pour compléter son exécution (***Turnaround time***).
- Le temps et les données dont a besoin l'ordonnanceur pour choisir la tâche à exécuter (***Overhead***).
- **Équité**: quels sont les facteurs déterminants quant au choix de la tâche à exécuter.
- **Débit**: nombre de tâches traitées en un temps donné.
- **Famine**: un ordonnanceur doit (dans certains cas) assurer que ce problème ne se produise pas…
- **Préemptivité et non préemptivité**

Ordonnancement (quelques définitions)

- **Ordonnancement**: détermine l'ordre et la durée avec d'exécution des tâches.
- **Dispatching**: le dispatcheur commence et arrête une tâche.
- Chaque OS implémente une **fonction** d'ordonnancement qui **n'est pas une tâche** en soit. Mais une fonction appelée à plusieurs endroits du noyau:
  - Fin de routine d'interruption
  - Lorsque les tâches sont mises en sommeil
  - Lorsque les tâches sont prêtes à s'exécuter.
- L'ordonnancement, c'est de la ***pure perte de temps***: plus l'algorithme est complexe, moins il est bon de l'implémenter.

Quand ordonnancer ?

Les décisions d'ordonnancement sont prises dans plusieurs circonstances:

- **Fork**
  - Qui choisir entre le père et le fils ?
- **Fin d'un processus**
  - En trouver un autre
  - Si aucun, un processus spécial est exécuté (*idle* ou *Processus inactif du système*)
- **Quand un processus devient bloqué**
  - Attente d'une E/S ou d'une condition
  - Peut influer sur le prochain à exécuter
- **Interruption E/S**
  - Si indique fin de traitement, réordonnancer le processus qui attendait

Ordonnancement non préemptif

Un processus est choisi et détient le CPU jusqu'à ce qu'il bloque ou se termine

Ordonnancement préemptif

Un processus n'a le CPU que pour une durée donnée (*timeslice*)

Différents besoins/différents ordonnancements

- Des systèmes ont des propriétés et besoins différents
- Il faut des ordonnanceurs adaptés
- **Batch**
  - Pas d'utilisateurs devant des écrans
  - Ordonnanceurs non préemptifs ou presque…
  - Changements de processus réduits
  - Ex: compilateur de langage, recherche dans des BDs, calcul scientifique, etc.
- **Interactif**
  - Préemption pour maintenir la réactivité
  - Ex: éditeur de texte, shell, etc.
- **Temps réel**
  - Préemption
  - Ex: Application vidéo, audio, collecte de données à partir d'un capteur.

Objectifs de l'ordonnancement

- Les objectifs des algorithmes d'ordonnancement dépendent du système considéré
  - Mais certaines propriétés sont communes
- **Ce qui est commun**
  - Respect des règles : permettre à certains processus d'avoir un ordonnancement particulier
  - Équilibre : Maximiser l'utilisation du système (alterner des E/S et du CPU)
  - « *Équité : chaque processus doit avoir accès au CPU de manière équitable* »
- **Batch**
  - Débit : maximiser le nombre de jobs par heure
  - Temps de rotation de l'appli: minimiser le temps entre la soumission et la complétion
  - CPU : maximiser l'utilisation du CPU

Objectifs (2)

- **Interactif**
  - Réactivité: répondre rapidement aux demandes
  - Proportionnalité
    - Les utilisateurs ont souvent une idée du temps que va prendre une opération (click == rapide…)
    - L'ordonnanceur doit choisir les processus pour être conforme à leurs attentes
- **Temps réel**
  - Respecter les contraintes temporelles
  - Prédictibilité : l'ordonnanceur doit être prévisible

- Certains de ces objectifs peuvent ne pas être maximisés en même temps (voire contradictoires)
  - Débit et temps de réponse par exemple…

Politique FCFS / exécution jusqu'à terminaison

- First Come First Served (premier arrivé, premier servi).
- Ordonnancement utilisé pour les batchs
- Approche non préemptive:
  - Pas de file d'attente de tâches bloquées
- Temps de réponse lent
  - Surtout s'il existe des tâches longues
  - Dans ce cas: pb d'équité (les tâches courtes attendent plus que les tâches longues)
- Famine: impossible.

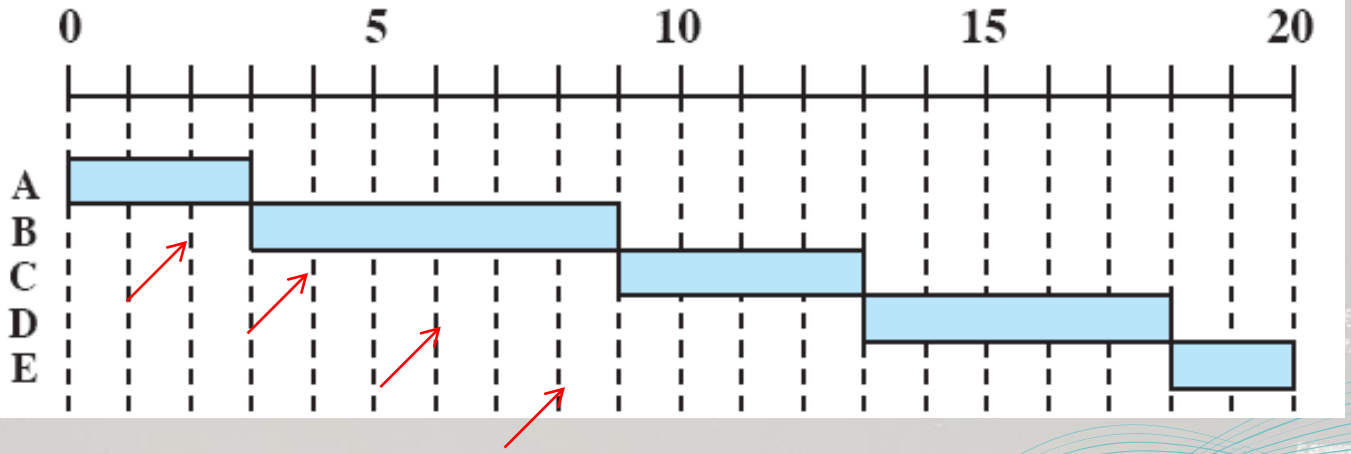
FCFS

W. Stallings, Operating Systems Internal Design and Principles, 6th ed. Pearson Int. Edition

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

First-Come-First Served (FCFS)



Le plus court d'abord (SPN/Shortest Process Next)

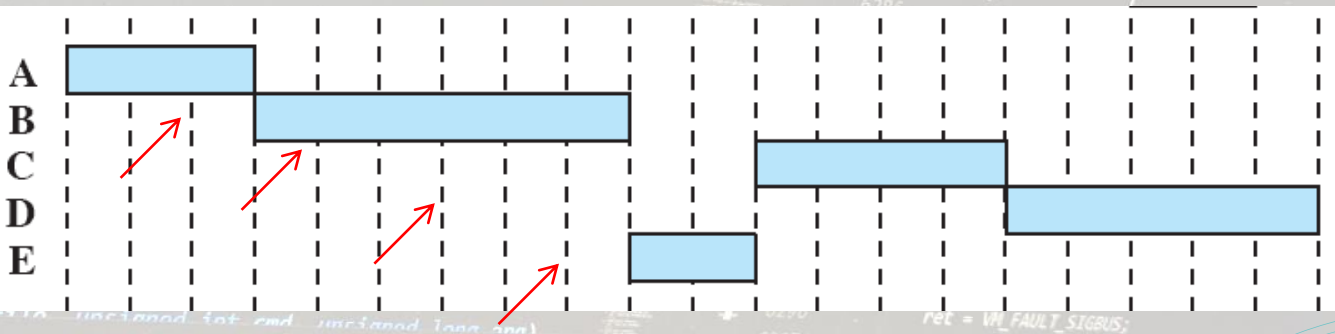
- On suppose que le temps d'exécution est connu à l'avance (dur dur !).
- La tâche la plus courte de l'ensemble des processus s'exécute en premier.
- Temps de réponse plus rapide que le précédent
- Famine possible
- Temps de calcul (de l'ordonnanceur) important: pour savoir quel processus doit s'exécuter

W. Stallings, Operating Systems Internal Design and Principles, 6th ed. Pearson Int. Edition

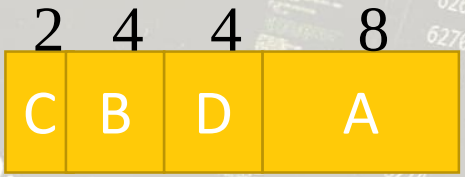
Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Shortest Process Next (SPN)



SPN exemple



- Délai de rotation/turnaround: 8, 12, 14, 18, moyenne de 13.5

Délai de rotation/turnaround: 2, 6, 10, 18, moyenne: 9

- Présuppose la disponibilité de toutes les tâches au moment de l'ordonnancement.
- Si 5 jobs: a, b, c, d, e avec temps d'exécution: 2, 4, 1, 1, 1 et temps d'arrivée de 0, 0, 0, 3, 3
  - SPN: c, a, d, e, b
  - Si tous dispos: c, d, e, a, b

Ordonnancement coopératif

- Les tâches préviennent l'OS lorsqu'elles peuvent être préemptées.
- Ceci peut avoir lieu avant la fin de l'exécution.
- Cet algorithme est implémenté en plus d'autres algorithmes FCFS et SPN.

- **Remarque**: OS avec ordonnanceur non préemptif ne force pas un changement de contexte si aucune tâche n'est prête même dans le cas d'un ordonnancement coopératif, contrairement à un ordonnanceur préemptif.

Tourniquet/FIFO/*Round Robin*

- File d'attente FIFO stocke les processus prêts.
- Exécution indépendante de la charge de travail et de l'interactivité.
- Quantum de temps égal pour tous
  - Durée maximale durant laquelle il peut s'exécuter
  - S'il atteint son quantum, il est préempté
  - S'il est bloqué avant, un autre processus est lancé
- Famine: impossible
- Quantum de temps:
  - Trop grand: trop d'attente par processus
  - Trop petit: temps de changement de contexte relativement important (en pourcentage).
  - En général, entre 10 et 50ms

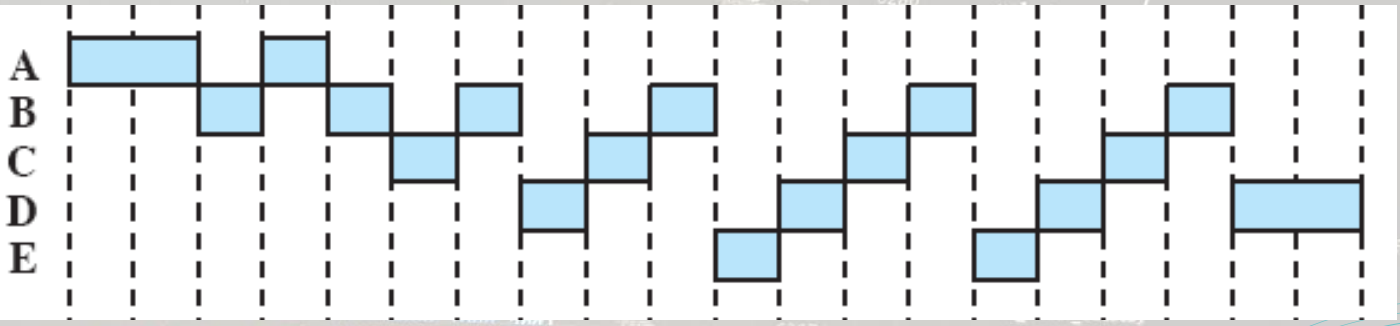
Tourniquet /Round Robin

W. Stallings, Operating Systems Internal Design and Principles, 6th ed. Pearson Int. Edition

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Round-Robin (RR), $q = 1$



La notion d'espace de priorités

- **2 possibilités**:
  - Certains OS laissent toutes les tâches (système et utilisateur) « vivre » dans **un même espace de priorités**.
  - D'autres **isolent l'espace des priorités** des processus système et utilisateurs (WindowsNT, Linux, UNIX)
    - Sous Unix par ex, les tâches utilisateurs voulant augmenter leur priorités peuvent utiliser l'appel à `nice()`
    - … mais sont tous préemptées par une tâche système.
- **L'espace noyau a 3 types de *« tâches » (en termes de priorités)*:**
  1. Interruptions: c'est une interruption hw (timer, clavier, réseau, etc). Ce type de tâche est appelé ISR (Interrupt Service Routine). Ce n'est pas vraiment une tâche mais une fonction exécutée ***indépendamment de l'ordonnanceur*** à chaque interruption. L'OS n'est pas impliqué.
  2. Fonctions *tasklets (minitâches)* et routines de service différées: fonctions qui peuvent être activées par n'importe quelle tâche du noyau (pour la mini tâche) ou par une fonction d'interruption (pour les DSR). Les interruptions sont actives lors de l'exécution de ces fonctions ***et l'OS est impliqué*** pour déterminer l'ordre d'exécution.
  3. Toutes les autres tâches du noyau: le niveau le moins prioritaire du noyau, préempte toutes les tâches utilisateurs.

Ordonnancement prioritaire

- **Principe**: tous les processus ne sont pas aussi importants les uns que les autres
- Chaque processus se voit attribuer une priorité
- Le processus de priorité supérieure préempte ceux de priorité inférieure.
- Pb de **famine** → solution:
  - Priorité dynamique: à chaque tic d'horloge la priorité du processus en exécution est réduite
  - Si inférieure à un autre processus, changement de contexte
- Optimiser les performances
  - Un processus qui fait beaucoup d'E/S devrait avoir une grande priorité car il consomme peu de CPU (linux 2.6 …)
  - Simple : la priorité est une fraction du quantum effectivement utilisé
    - Un processus qui tourne 1ms sur 50 a une priorité de 50
    - Un processus qui tourne 50ms a une priorité de 1
- Pb *d'inversion de priorité:* processus de priorité importante est en attente d'exécution d'un processus de priorité inférieure. Et les processus avec une priorité entre les 2 derniers sont en cours d'exécution.

EDF (Earliest Deadline First)

- Plus proche date limite d'abord (tâches périodiques et apériodiques).
- Trois paramètres à prendre en compte et connaître pour donner des priorités aux processus:
  - **Fréquence**: nombre de fois que le processus est exécuté
  - **Date limite**: quand le processus doit avoir terminé son exécution.
  - **Durée**: temps d'exécution du processus.
- Priorité dynamique (contrairement à RMS)

Ordonnancement préemptif et RTOS

- Si l'ordonnanceur respecte ses dates limites avec précision:
  - RTOS
  - Sinon EOS.
- RTOS:
  - Ordonnancement **doit être préemptif** (la tâche de plus haute priorité doit préempter celle en cours d'exécution)
- Exemple d'implémentation:
  - **VxWorks**: basé sur la priorité et tourniquet
  - **Jbed**: EDF
  - **Linux** (TimeSys): basé sur la priorité.

Ordonnancement sous VxWorks

- **Priorité préemptive**
  - C'est ce qui permet le temps réel.
- Tourniquet pour les tâches de la même priorité
- Possibilité de verrouiller l'ordonnanceur (pour qu'une tâche ne soit pas préemptée).
- Choix de la priorité: (0: plus haute priorité)

```
int taskSpawn(
   {Task Name},
   {Task Priority 0-255, related to scheduling},
   {Task Options – VX_FP_TASK, execute with floating point coprocessor
                   VX_PRIVATE_ENV, execute task with private environment
                   VX_UNBREAKABLE, disable breakpoints for task
                   VX_NO_STACK_FILL, do not fill task stack with 0xEE}
   {Stack Size}
   {Task address of entry point of program in memory– initial PC value}
   {Up to 10 arguments for task program entry routine}
```

Ordonnancement sous Jbed

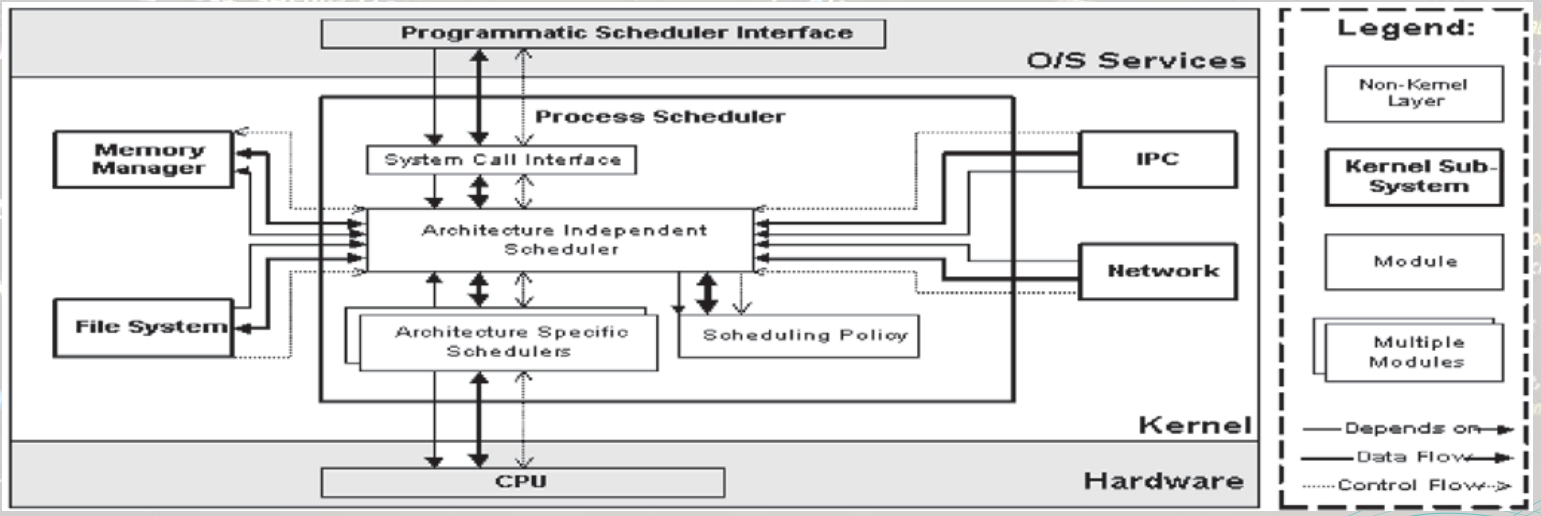
Lab-STICC

- Ordonnancement EDF.
- Toutes les tâches (6 types) ont 3 variables:
  - *duration*: le temps alloué à la tâche pour compléter son exécution
  - *allowance*: le temps alloué pour gérer l'exception lorsque le temps imparti est fini.
  - *deadline* : date limite de fin d'exécution de la tâche

Public Task(**Long duration,** **Long allowance,** **Long deadline,** **RealEvent event)** **Throws AdmissionFailure**	**Public Task(java.lang.String name,** **Long duration,** **Long allowance,** **Long deadline,** **RealEvent event)** **Throws AdmissionFailure**	**Public Task(java.lang.Runnable target,** **java.lang.String name,** **Long duration,** **Long allowance,** **Long deadline,** **RealEvent event)** **Throws AdmissionFailure**

Linux embarqué (Timesys)

- « L'ordonnanceur » est fait de 4 modules différents:
  - *Module d'interface d'appels systèmes:* interface entre processus utilisateur et les fonctionnalités du noyau
  - *Module de politique d'ordonnancement:* qui détermine quelle processus a accès au CPU.
  - *Module de l'ordonnanceur spécifique à l'architecture*: couche d'abstraction qui s'interface avec le matériel
  - *Module de l'ordonnanceur indépendant du matériel*: couche d'abstraction entre le module d'ordonnancement et le module spécifique à l'architecture.



Source: *Embedded Systems Architecture: A comprehensive guide for engineers and programmers*, Elsiever ed. T. Noergaard, 2005

Linux embarqué (2)

- La plupart des systèmes Linux (2.2/2.4) **ne sont pas préemptibles**, et ne sont **pas temps réel**.
- Linux embarqué de Timesys est basé sur les priorités et a été modifié pour permettre du temps réel:
  - Modification des ***timers*** qui sont trop grossiers sous Linux (basé sur le matériel)
  - Partie de la ***task_struct*** pour le temps réel …

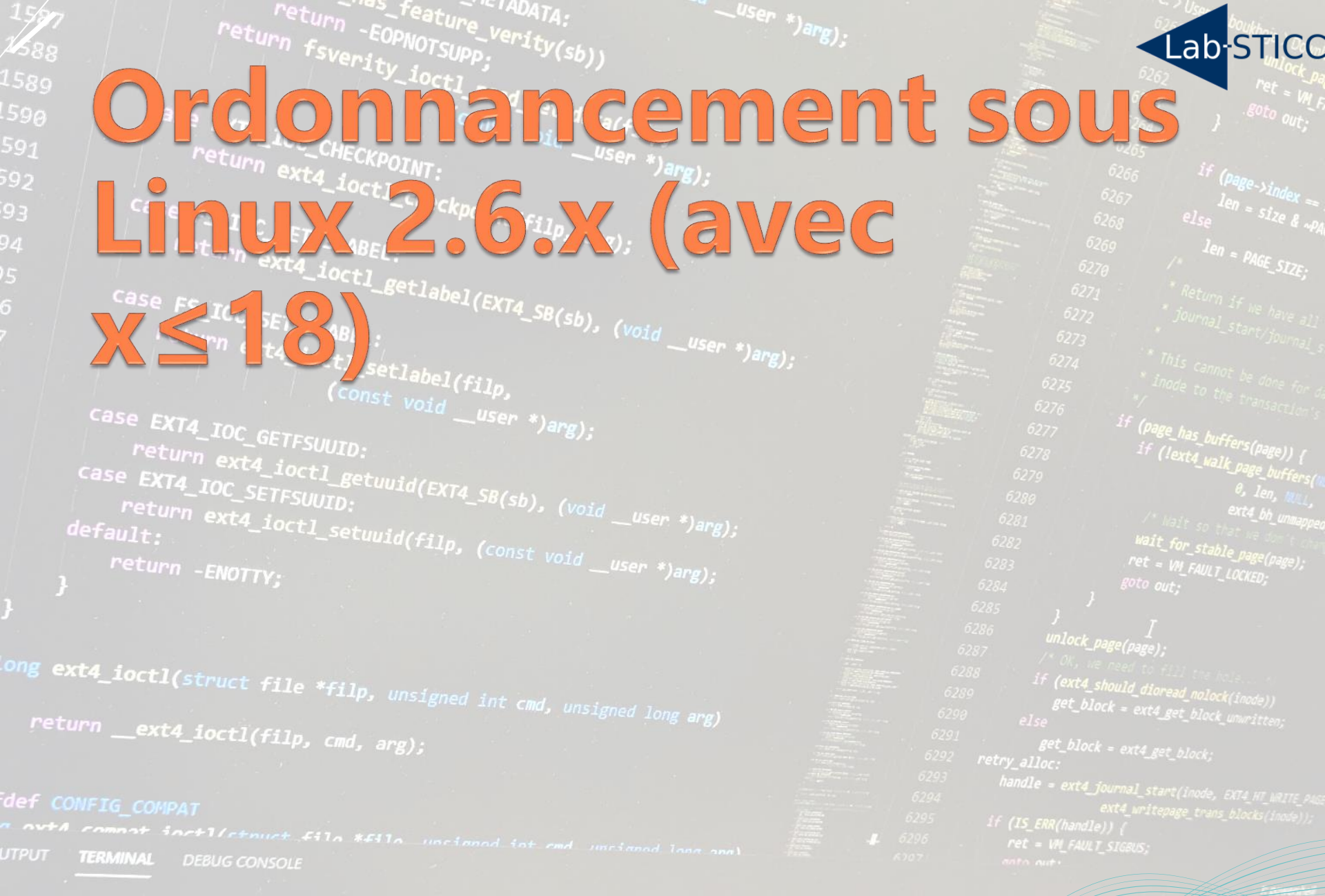
```
/* the scheduling policy, specifies which scheduling class the task
  belongs to, such as: SCHED_OTHER (traditional UNIX process),
  SCHED_FIFO (POSIX.1b FIFO realtime process - A FIFO realtime process
  will run until either

  a) it blocks on I/O,

  b) it explicitly yields the CPU or

  c) it is preempted by another realtime process with a higher
  p->rt_priority value.) and  SCHED_RR (POSIX round-robin realtime
  process SCHED_RR is the same as SCHED_FIFO, except that  when its
  timeslice expires it goes back to the end of the run queue).*/
unsigned long policy;
//realtime priority
unsigned long rt_priority;
```

Ordonnancement sous Linux 2.6.x (avec x≤18)



Politique d'ordonnancement

- L'ordonnancement de Linux fonctionne avec des quanta dynamiques
- Les processus sont ordonnés par priorité
  - La valeur de cette dernière indique à l'ordonnanceur quel processus exécuter en premier
- Les priorités sont dynamiques
  - Les processus qui n'ont pas eu le CPU voient leur priorité augmentée (moins de pb de famine).
- Linux reconnaît les processus temps réel (mou)
- Mais aucune notion « explicite » de batch vs interactif, … mais implicite:
  - Décision prise avec une heuristique basée sur le comportement antérieur des processus

Préemption

Les processus peuvent être préemptés:

- Quand un processus devient TASK_RUNNING
  - Le noyau vérifie si sa priorité est plus grande que celle du processus en cours d'exécution
  - Si oui, l'exécution du processus courant est interrompue et l'ordonnanceur est invoqué
- Quand un processus fini tout son quantum de temps
  - Drapeau (champs flag) `TIF_NEED_RESCHED` mis à 1 dans la structure `thread_info`
  - Quand interruption horloge → le noyau invoque l'ordonnanceur si le drapeau est à 1.
- Avant 2.6
  - Un processus en mode noyau ne pouvait pas être préempté

Exemple

- 2 processus
  - Un éditeur de texte → interactif → plus haute priorité dynamique
    - L'utilisateur alterne temps de réflexion et entrée des données.
    - Temps mis entre pressions sur le clavier est assez important
  - Un compilateur → batch → plus basse priorité.
- Dès qu'il y a une pression sur un bouton:
  - Une interruption (relative à la pression sur le bouton) est générée
  - Le noyau réveille le processus de l'éditeur de texte (sort de la *wait queue* et devient prêt -> TASK_RUNING) .
  - Le noyau détermine/voit aussi la priorité de l'éditeur qui est supérieure à celle du compilateur.
  - Il met le drapeau `TIF_NEED_RESCHED` à 1 (du compilateur) pour forcer l'ordonnanceur à s'activer lorsque le noyau aura fini de gérer l'interruption
  - L'ordonnanceur prend alors le relais et effectue le changement de processus
  - Le caractère est alors affiché sur l'écran.

Algorithme d'ordonnancement

- **Version simple (<2.6)**
  - A chaque changement de processus le noyau parcourt toute la liste des processus prêts
  - Calcule leur priorité
  - Et choisit le gagnant
  - Mais algorithme très coûteux si beaucoup de processus
- **Version 2.6**
  - Sélectionne les processus en un temps constant
  - Une file de processus prêts par CPU
  - Distingue **mieux** les processus interactifs des batchs

Classes d'ordonnancement

Les processus sont ordonnancés selon 3 classes

- **SCHED_FIFO** :
  - Processus temps réel en fifo
  - Ordonnancement à priorité
  - Quand l'ordonnanceur assigne le CPU à un processus, il ne change pas sa position dans la *runqueue* des processus prêts (jusqu'à terminaison de l'exécution, sauf si processus de priorité plus importante).
  - Si un autre processus de même classe et même priorité est prêt …tant pis !
- **SCHED_RR**
  - Processus temps réel en Round Robin (tourniquet)
  - Le descripteur de processus est mis en fin de liste une fois le CPU assigné
  - Permet à tous les processus temps réel de même priorité de partager le CPU
- **SCHED_NORM**
  - Processus conventionnel (temps partagé)
- **Q: Y en a-t-il d'autres dans la mise en place de Linux ?**

Ordonnancement de processus conventionnels: Priorité statique

- Chaque processus conventionnel a une **priorité statique** (PS)
  - Utilisée par l'ordonnanceur pour ordonner les **processus conventionnels** entre eux
  - Valeur de 100 (haute priorité) à 139 (basse priorité)
- Un nouveau processus hérite de la priorité de son père
- Quantum de base (QB en ms)
  - Déterminé à partir de la priorité statique
  - Assigné à un processus qui a épuisé son quantum précédent
    - Si PS < 120, QB = (140-PS)*20
    - Si PS >=120, QB = (140-PS)*5
- Un processus de basse priorité aura un quantum faible

Priorité dynamique

- Chaque processus a, en plus **une priorité dynamique** (PD), de 100 (plus haute) à 139 (plus basse)
- La priorité dynamique sert à l'ordonnanceur pour **choisir le processus à exécuter**
  - PD=max(100,min(PS-bonus +5, 139))
- ***bonus*** est une valeur entre 0 et 10
  - Une valeur **< 5 est une pénalité** qui baissera la PD
  - Valeur **>=5 augmente** la PD (donc baisse sa valeur)
- La valeur du bonus dépend de l'historique du processus
  - Son **temps moyen de sommeil**
- **Temps moyen de sommeil:**
  - Mesuré en nanosecondes, jamais plus grand que 1 seconde
  - Mesure différente selon l'état (TASK_INTERRUPTIBLE vs TASK_UNINTERRUPTIBLE)
  - Diminue quand le processus s'exécute
- Le temps moyen de sommeil sert à déterminer si le processus est **batch** ou **interactif**
  - Si ***PD ≤ 3 * PS/4 + 28*** alors **interactif**
  - Ce qui est équivalent à ***bonus – 5 ≥ PS/4 – 28***
  - ***PS/4-28*** est le **delta interactif**

Équivalence sommeil - bonus

Temps de sommeil moyen	Bonus
Entre 0 et 100ms	0
Entre 100 et 200ms	1
Entre 200 et 300ms	2
Entre 300 et 400ms	3
Entre 400 et 500ms	4
Entre 500 et 600ms	5
Entre 600 et 700ms	6
Entre 700 et 800ms	7
Entre 800 et 900ms	8
Entre 900 et 1 s	9
1 seconde	10

Exemple de valeurs de priorités pour des processus conventionnels

Description	PS	Valeur de *nice*	Quantum de base	Delta interactif
Priorité statique la plus haute	100	-20	800ms	-3
Haute priorité statique	110	-10	600ms	-1
Priorité normale	120	0	100ms	+2
Basse priorité	130	+10	50ms	+4
Priorité la plus basse	139	+19	5ms	+6

$$bonus - 5 \geq PS/4 - 28$$

Exemple

- Il est plus facile pour un processus de haute priorité statique d'être interactif.
- Un processus ayant une PS de 100
  - Est considéré comme interactif si la valeur du bonus >2
    - C'est-à-dire lorsque son temps moyen de sommeil est de 200ms.
- Un processus avec une PS de 139 ne peut être considéré comme interactif
  - ***bonus – 5 ≥ PS/4 – 28*** ⇨ **bonus ≥ *139/4-28+5***

  bonus ≥ 11 ce qui est impossible

Processus actifs et expirés

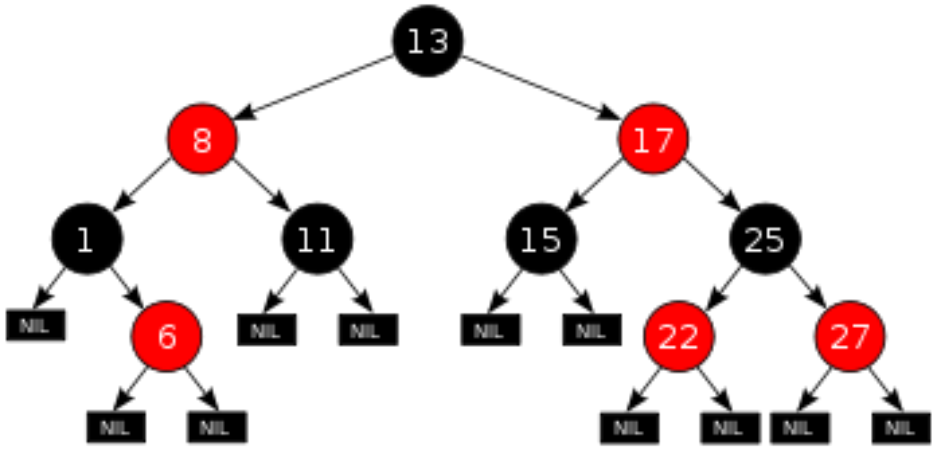
- Un processus conventionnel de haute priorité ne devrait pas empêcher ceux de basse priorité de tourner
- Quand un processus fini son quantum, il peut être remplacé par un processus de plus basse priorité qui n'a pas fini le sien
- L'ordonnanceur maintient **2 ensembles de processus**
  - Processus **Actifs** : ils n'ont pas encore fini leur quantum
  - Processus **expirés** : ont épuisé leur quantum et ne peuvent pas s'exécuter tant qu'ils restent des processus actifs
- Plus compliqué que cela en pratique:
  - Un processus batch qui finit son quantum devient **toujours** *expiré*
  - Un processus interactif qui finit son quantum reste **souvent** *actif:*
    - Sauf si le plus vieux processus expiré attend depuis « très longtemps »
    - Ou un processus expiré a une priorité statique plus élevée

Résumé

- **Priorité statique** ↗ : la taille du quantum de temps ↗
- **Priorité dynamique** ↗ : ordre d'exécution
  - PS
  - Bonus → temps moyen de sommeil
- **Interactivité** (delta interactif): reste dans la liste des processus actifs
  - Priorité statique ↗
  - Temps de sommeil ↗

Ordonnanceur CFS (Completely Fair Scheduler)

- Développé par Ingo Molnar: partage équitable du temps CPU (à partir de 2.6.23)
- Pas de liste d'attente par priorité → classement par rapport au « virtual runtime » par tâche (temps CPU consommé)
- Activation de la tâche ayant le plus manqué de temps (virtual runtime ⇲)
- Organisation des tâches en utilisant un arbre rouge-noir (ou bicolore), complexité O(log(N)) pour l'insertion



Source: Wikipedia

Priorité

Suite (CFS)

- Avantages:
  - Influence de la priorité statique par rapport au temps accordé est uniforme (multiplication de 1.1 si augmentation de priorité de 1)
  - Granularité à la nano seconde (timeslice)
  - (Ré) utilise la notion de « sleeper fairness », les processus endormi reçoivent leur quota temps (interactifs)
- Pb: multi –utilisateurs ou job multi-threadé → possibilité d'accaparement du CPU par « les plus nombreux » → pb résolu par la prise en compte des groupes de processus
  - Ex: 4 groupes, 4 taches:
    - Chacune dans 1 groupe différent: 25%CPU
    - 4 tâches dans un même groupe ~6% CPU chacune
  - Notion de groupe = processus du même terminal (2.6.38)

Ordonnancement temps réel

- Chaque processus temps réel a une priorité temps réel de 1 (plus basse) à 99
- L'ordonnanceur favorise toujours le processus temps réel de plus haute priorité
  - Aucun autre ne peut s'exécuter
- `sched_setparam()` et `sched_setscheduler( )`
- Les processus temps réel **sont toujours considérés comme « actifs »**
- Un processus temps réel est remplacé par un autre processus si:
  - Il est préempté par un autre de **plus haute priorité** temps réel
  - Il effectue une **opération bloquante** et est mis en sommeil (`TASK_INTERRUPTIBLE` ou `TASK_UNINTERRUPTIBLE`)
  - Il est **stoppé** (`TASK_STOPPED` ou `TASK_TRACED`) ou tué (`EXIT_ZOMBIE` ou `EXIT_DEAD`).
  - Il **rend volontairement le CPU** avec un appel à `sched_yield( )`
  - Il est **SCHED_RR** et a fini son quantum

Appels système relatifs à l'ordonnancement

- Conventionnels
  - **nice( ):** changement de la valeur de « gentillesse ». Maintenu pour des raisons de compatibilité et remplacé par `setpriority`()
    - Valeurs négatives: augmentation de priorité statique (superutilisateur)
  - **getpriority( ) et setpriority( ):** concerne un processus ou tout le groupe de processus (priorité statique max).
  - **sched_getaffinity( ) et sched_setaffinity( ):** quels sont les CPUs qui peuvent exécuter ce processus.
- Temps réel
  - **sched_getscheduler( ) et sched_setscheduler( ):** politique d'ordonnancement
  - **sched_getparam( ) et sched_setparam( ):** retourne les paramètres de la politique d'ordonnancement (priorité temps réel).
  - **sched_yield( ):** relâche le CPU.
  - **sched_get_priority_min( ) et sched_get_priority_max( ):** retourne la priorité min et max utilisable par le processus.
  - **sched_rr_get_interval( ):** retourne la taille du quantum de temps de la politique du tourniquet.