

# OS embarqués – L'ordonnement

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

```
long ext4_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    return __ext4_ioctl(filp, cmd, arg);
}

#define CONFIG_COMPAT
ext4_compat_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    return __ext4_ioctl(filp, cmd, arg);
}

OUTPUT TERMINAL DEBUG CONSOLE
Shell
```

## 1. L'ordonnancement

1. Définition
2. Métriques
3. Objectifs

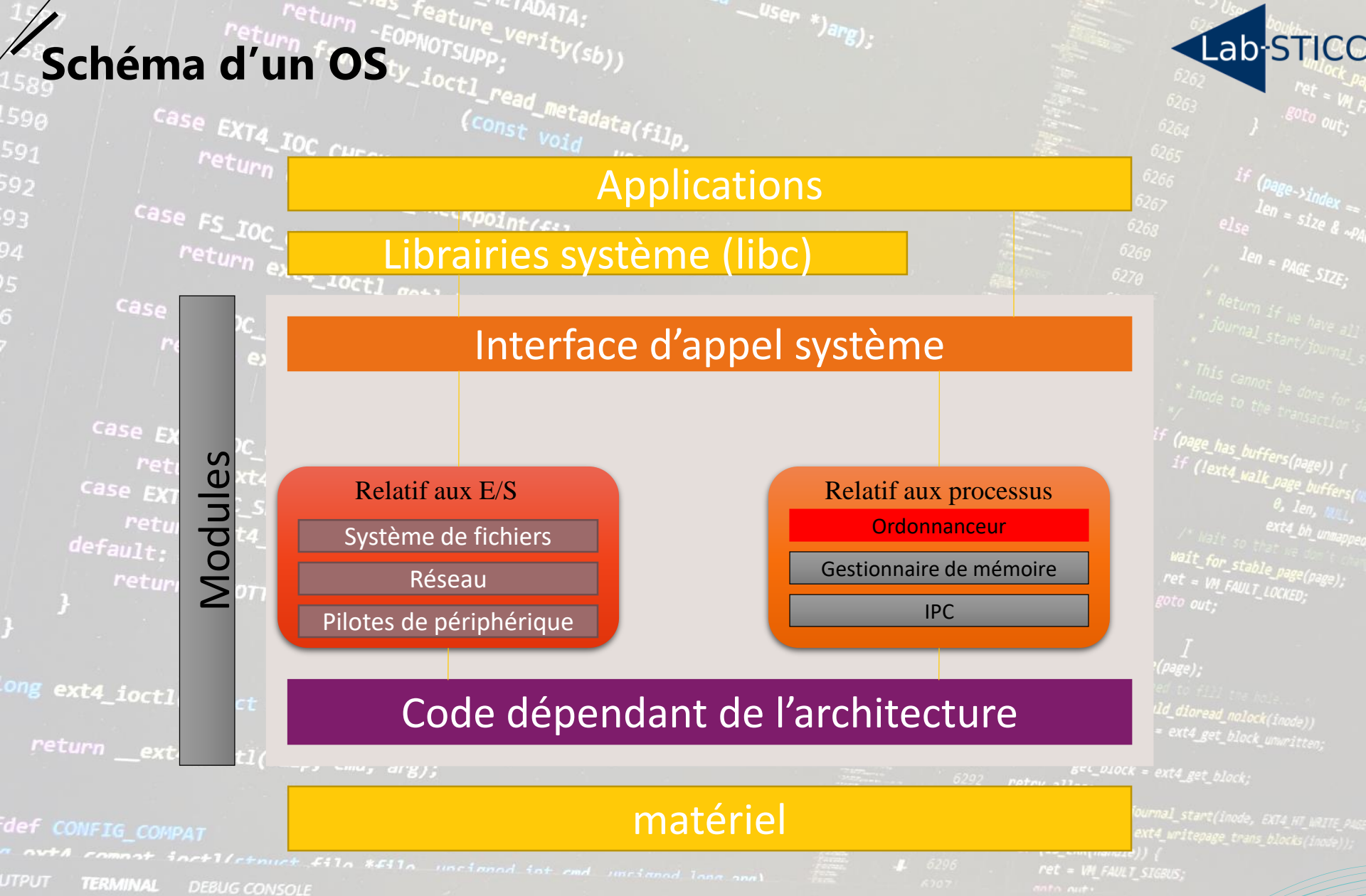
## 2. Les algorithmes d'ordonnancement

## 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 influencer 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 **batches**
- / 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.



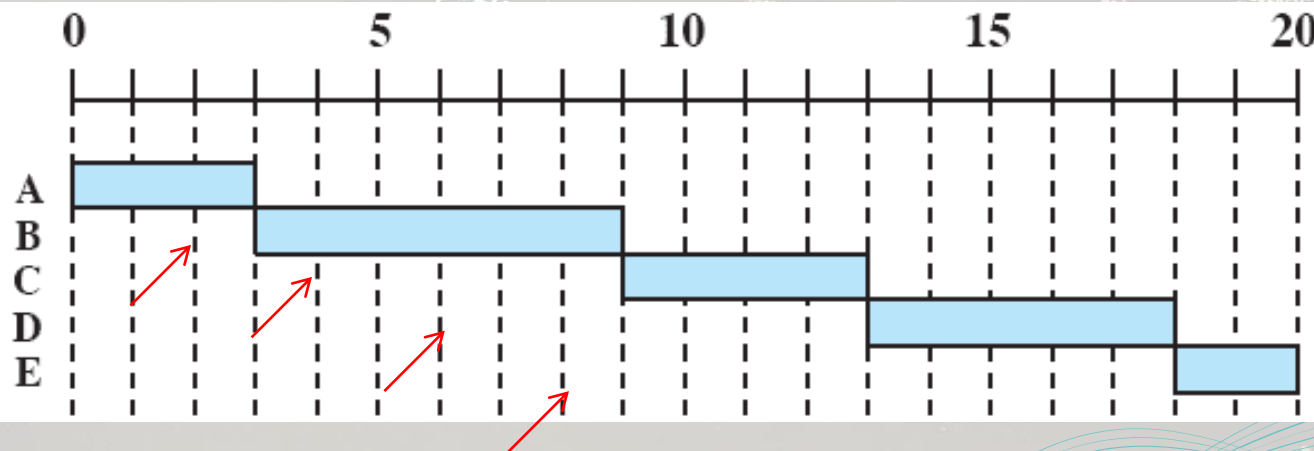
# FCFS

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)

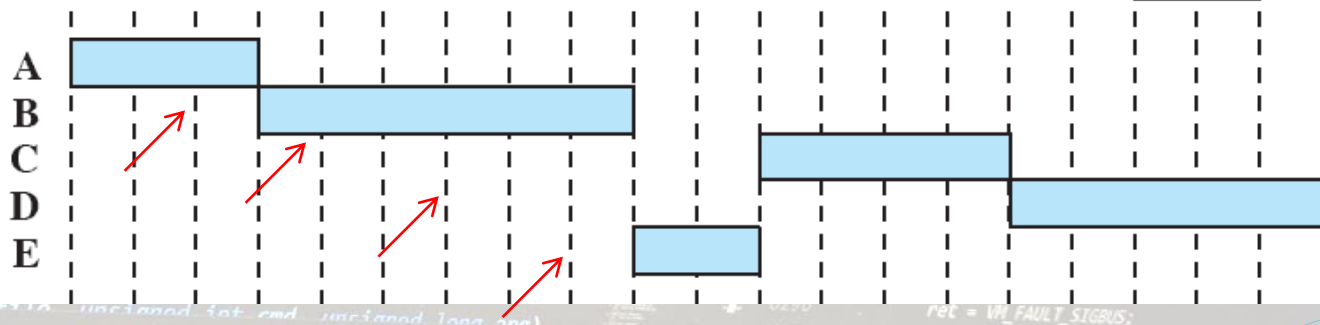
- / On suppose que le temps d'exécution est connu à l'avance (dur dur !).
- / La tâche la plus courte de l'ensemble des processus s'exécute en premier.
- / Temps de réponse **plus rapide** que le précédent
- / Famine possible
- / Temps de calcul (de l'ordonnanceur) important: pour savoir quel processus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

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

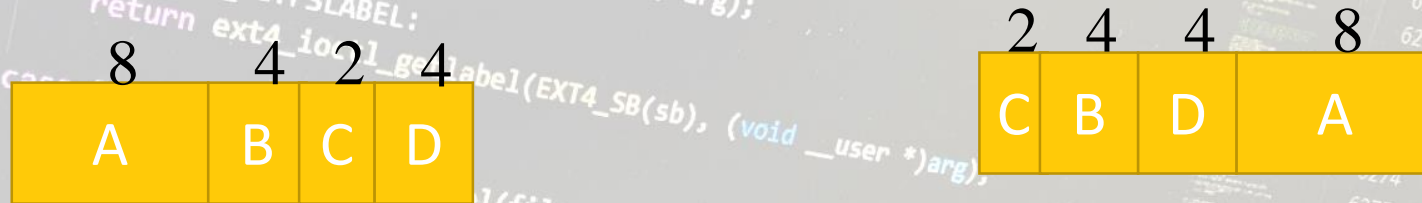
**Table 9.4 Process Scheduling Example**

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

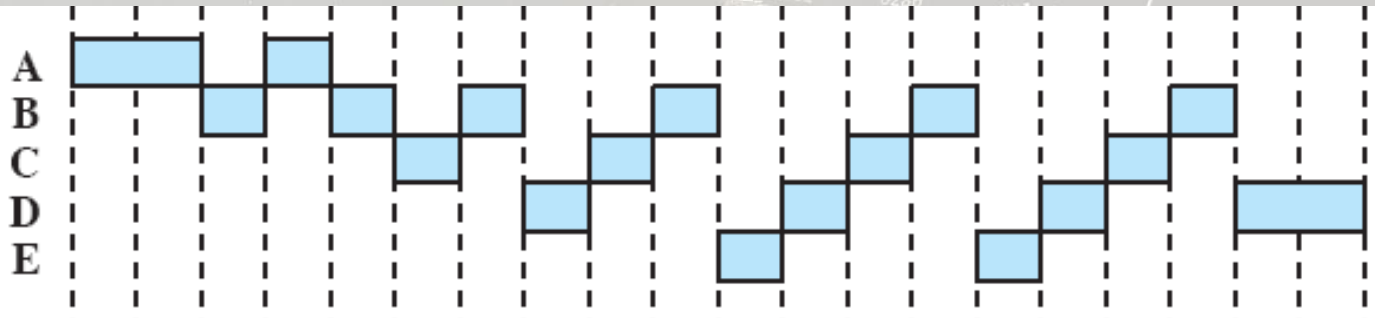
- / File d'attente **FIFO** stocke les processus prêts.
- / Exécution indépendante de la charge de travail et de l'interactivité.
- / **Quantum de temps** égal pour tous
  - / Durée maximale durant laquelle il peut s'exécuter
  - / S'il atteint son quantum, il est préempté
  - / S'il est bloqué avant, 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

**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

/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

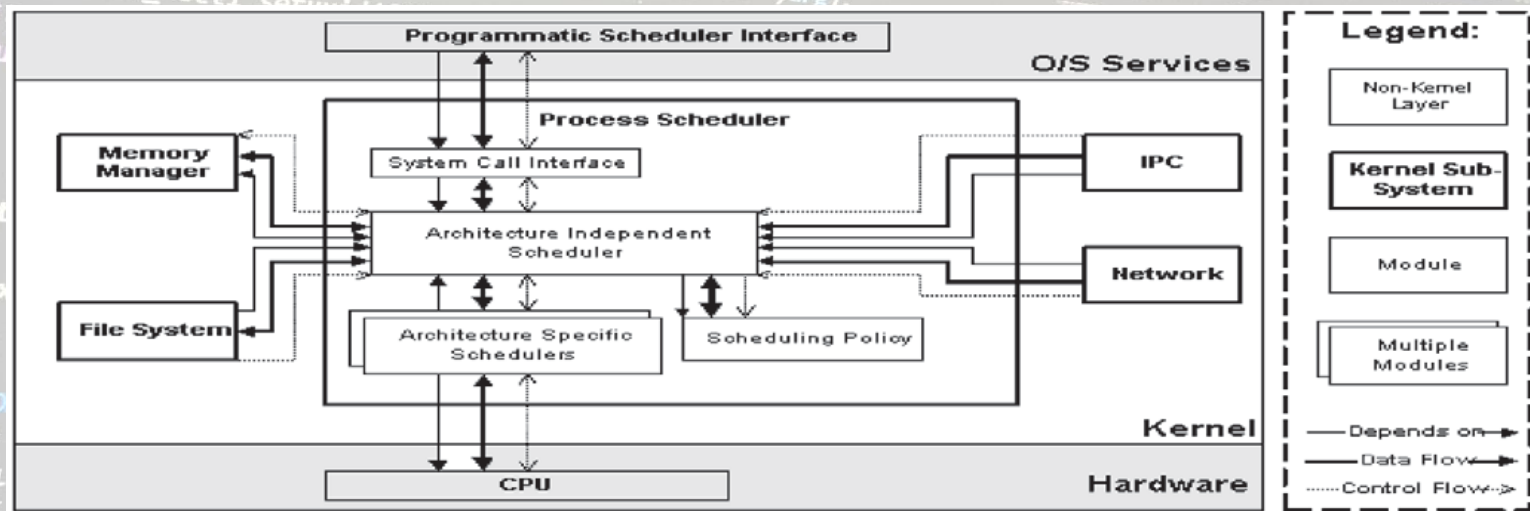
Public Task(java.lang.String name,  
 Long duration,  
 Long allowance,  
 Long deadline,  
 RealEvent event)  
 Throws AdmissionFailure

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'ordonnement*: 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'ordonnement et le module spécifique à l'architecture.



# 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 $x \leq 18$ )

```

1587 return __as_feature_verity(sb);
1588 return -EOPNOTSUPP;
1589 return fsverity_ioctl_getlabel(sb, (void __user *)arg);
1590
1591 return ext4_ioctl_checkpoint(filp, (void __user *)arg);
1592
1593 case EXT4_IOC_CHECKPOINT:
1594 return ext4_ioctl_checkpoint(filp, (void __user *)arg);
1595
1596 case EXT4_IOC_GETLABEL:
1597 return ext4_ioctl_getlabel(EXT4_SB(sb), (void __user *)arg);
1598
1599 case EXT4_IOC_SETLABEL:
1600 return ext4_ioctl_setlabel(filp, (const void __user *)arg);
1601
1602 case EXT4_IOC_GETFSUUID:
1603 return ext4_ioctl_getuuid(EXT4_SB(sb), (void __user *)arg);
1604
1605 case EXT4_IOC_SETFSUUID:
1606 return ext4_ioctl_setuuid(filp, (const void __user *)arg);
1607
1608 default:
1609 return -ENOTTY;
1610 }
1611 }
1612
1613 long ext4_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
1614 {
1615 return __ext4_ioctl(filp, cmd, arg);
1616 }
1617
1618 #ifdef CONFIG_COMPAT
1619 int ext4_compat_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
1620 {
1621 return __ext4_ioctl(file, cmd, arg);
1622 }
1623 #endif
1624
1625 /*
1626 * This cannot be done for data
1627 * inode to the transaction's
1628 */
1629 if (page_has_buffers(page)) {
1630 if (!ext4_walk_page_buffers(
1631 0, len, NULL,
1632 ext4_bh_unmapped))
1633 /* Wait so that we don't change
1634 wait_for_stable_page(page);
1635 ret = VM_FAULT_LOCKED;
1636 goto out;
1637 }
1638 }
1639
1640 I
1641 unlock_page(page);
1642 /* OK, we need to fill the hole...
1643 if (ext4_should_dioread_nolock(inode))
1644 get_block = ext4_get_block_unwritten;
1645 else
1646 get_block = ext4_get_block;
1647 retry_alloc:
1648 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1649 ext4_writepage_trans_blocks(inode));
1650 if (IS_ERR(handle)) {
1651 ret = VM_FAULT_SIGBUS;
1652 goto out;
1653 }
1654 }
1655
1656 /*
1657 * This cannot be done for data
1658 * inode to the transaction's
1659 */
1660 if (page_has_buffers(page)) {
1661 if (!ext4_walk_page_buffers(
1662 0, len, NULL,
1663 ext4_bh_unmapped))
1664 /* Wait so that we don't change
1665 wait_for_stable_page(page);
1666 ret = VM_FAULT_LOCKED;
1667 goto out;
1668 }
1669 }
1670
1671 I
1672 unlock_page(page);
1673 /* OK, we need to fill the hole...
1674 if (ext4_should_dioread_nolock(inode))
1675 get_block = ext4_get_block_unwritten;
1676 else
1677 get_block = ext4_get_block;
1678 retry_alloc:
1679 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1680 ext4_writepage_trans_blocks(inode));
1681 if (IS_ERR(handle)) {
1682 ret = VM_FAULT_SIGBUS;
1683 goto out;
1684 }
1685 }
1686
1687 /*
1688 * This cannot be done for data
1689 * inode to the transaction's
1690 */
1691 if (page_has_buffers(page)) {
1692 if (!ext4_walk_page_buffers(
1693 0, len, NULL,
1694 ext4_bh_unmapped))
1695 /* Wait so that we don't change
1696 wait_for_stable_page(page);
1697 ret = VM_FAULT_LOCKED;
1698 goto out;
1699 }
1700 }
1701
1702 I
1703 unlock_page(page);
1704 /* OK, we need to fill the hole...
1705 if (ext4_should_dioread_nolock(inode))
1706 get_block = ext4_get_block_unwritten;
1707 else
1708 get_block = ext4_get_block;
1709 retry_alloc:
1710 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1711 ext4_writepage_trans_blocks(inode));
1712 if (IS_ERR(handle)) {
1713 ret = VM_FAULT_SIGBUS;
1714 goto out;
1715 }
1716 }
1717
1718 /*
1719 * This cannot be done for data
1720 * inode to the transaction's
1721 */
1722 if (page_has_buffers(page)) {
1723 if (!ext4_walk_page_buffers(
1724 0, len, NULL,
1725 ext4_bh_unmapped))
1726 /* Wait so that we don't change
1727 wait_for_stable_page(page);
1728 ret = VM_FAULT_LOCKED;
1729 goto out;
1730 }
1731 }
1732
1733 I
1734 unlock_page(page);
1735 /* OK, we need to fill the hole...
1736 if (ext4_should_dioread_nolock(inode))
1737 get_block = ext4_get_block_unwritten;
1738 else
1739 get_block = ext4_get_block;
1740 retry_alloc:
1741 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1742 ext4_writepage_trans_blocks(inode));
1743 if (IS_ERR(handle)) {
1744 ret = VM_FAULT_SIGBUS;
1745 goto out;
1746 }
1747 }
1748
1749 /*
1750 * This cannot be done for data
1751 * inode to the transaction's
1752 */
1753 if (page_has_buffers(page)) {
1754 if (!ext4_walk_page_buffers(
1755 0, len, NULL,
1756 ext4_bh_unmapped))
1757 /* Wait so that we don't change
1758 wait_for_stable_page(page);
1759 ret = VM_FAULT_LOCKED;
1760 goto out;
1761 }
1762 }
1763
1764 I
1765 unlock_page(page);
1766 /* OK, we need to fill the hole...
1767 if (ext4_should_dioread_nolock(inode))
1768 get_block = ext4_get_block_unwritten;
1769 else
1770 get_block = ext4_get_block;
1771 retry_alloc:
1772 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1773 ext4_writepage_trans_blocks(inode));
1774 if (IS_ERR(handle)) {
1775 ret = VM_FAULT_SIGBUS;
1776 goto out;
1777 }
1778 }
1779
1780 /*
1781 * This cannot be done for data
1782 * inode to the transaction's
1783 */
1784 if (page_has_buffers(page)) {
1785 if (!ext4_walk_page_buffers(
1786 0, len, NULL,
1787 ext4_bh_unmapped))
1788 /* Wait so that we don't change
1789 wait_for_stable_page(page);
1790 ret = VM_FAULT_LOCKED;
1791 goto out;
1792 }
1793 }
1794
1795 I
1796 unlock_page(page);
1797 /* OK, we need to fill the hole...
1798 if (ext4_should_dioread_nolock(inode))
1799 get_block = ext4_get_block_unwritten;
1800 else
1801 get_block = ext4_get_block;
1802 retry_alloc:
1803 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1804 ext4_writepage_trans_blocks(inode));
1805 if (IS_ERR(handle)) {
1806 ret = VM_FAULT_SIGBUS;
1807 goto out;
1808 }
1809 }
1810
1811 /*
1812 * This cannot be done for data
1813 * inode to the transaction's
1814 */
1815 if (page_has_buffers(page)) {
1816 if (!ext4_walk_page_buffers(
1817 0, len, NULL,
1818 ext4_bh_unmapped))
1819 /* Wait so that we don't change
1820 wait_for_stable_page(page);
1821 ret = VM_FAULT_LOCKED;
1822 goto out;
1823 }
1824 }
1825
1826 I
1827 unlock_page(page);
1828 /* OK, we need to fill the hole...
1829 if (ext4_should_dioread_nolock(inode))
1830 get_block = ext4_get_block_unwritten;
1831 else
1832 get_block = ext4_get_block;
1833 retry_alloc:
1834 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1835 ext4_writepage_trans_blocks(inode));
1836 if (IS_ERR(handle)) {
1837 ret = VM_FAULT_SIGBUS;
1838 goto out;
1839 }
1840 }
1841
1842 /*
1843 * This cannot be done for data
1844 * inode to the transaction's
1845 */
1846 if (page_has_buffers(page)) {
1847 if (!ext4_walk_page_buffers(
1848 0, len, NULL,
1849 ext4_bh_unmapped))
1850 /* Wait so that we don't change
1851 wait_for_stable_page(page);
1852 ret = VM_FAULT_LOCKED;
1853 goto out;
1854 }
1855 }
1856
1857 I
1858 unlock_page(page);
1859 /* OK, we need to fill the hole...
1860 if (ext4_should_dioread_nolock(inode))
1861 get_block = ext4_get_block_unwritten;
1862 else
1863 get_block = ext4_get_block;
1864 retry_alloc:
1865 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1866 ext4_writepage_trans_blocks(inode));
1867 if (IS_ERR(handle)) {
1868 ret = VM_FAULT_SIGBUS;
1869 goto out;
1870 }
1871 }
1872
1873 /*
1874 * This cannot be done for data
1875 * inode to the transaction's
1876 */
1877 if (page_has_buffers(page)) {
1878 if (!ext4_walk_page_buffers(
1879 0, len, NULL,
1880 ext4_bh_unmapped))
1881 /* Wait so that we don't change
1882 wait_for_stable_page(page);
1883 ret = VM_FAULT_LOCKED;
1884 goto out;
1885 }
1886 }
1887
1888 I
1889 unlock_page(page);
1890 /* OK, we need to fill the hole...
1891 if (ext4_should_dioread_nolock(inode))
1892 get_block = ext4_get_block_unwritten;
1893 else
1894 get_block = ext4_get_block;
1895 retry_alloc:
1896 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1897 ext4_writepage_trans_blocks(inode));
1898 if (IS_ERR(handle)) {
1899 ret = VM_FAULT_SIGBUS;
1900 goto out;
1901 }
1902 }
1903
1904 /*
1905 * This cannot be done for data
1906 * inode to the transaction's
1907 */
1908 if (page_has_buffers(page)) {
1909 if (!ext4_walk_page_buffers(
1910 0, len, NULL,
1911 ext4_bh_unmapped))
1912 /* Wait so that we don't change
1913 wait_for_stable_page(page);
1914 ret = VM_FAULT_LOCKED;
1915 goto out;
1916 }
1917 }
1918
1919 I
1920 unlock_page(page);
1921 /* OK, we need to fill the hole...
1922 if (ext4_should_dioread_nolock(inode))
1923 get_block = ext4_get_block_unwritten;
1924 else
1925 get_block = ext4_get_block;
1926 retry_alloc:
1927 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1928 ext4_writepage_trans_blocks(inode));
1929 if (IS_ERR(handle)) {
1930 ret = VM_FAULT_SIGBUS;
1931 goto out;
1932 }
1933 }
1934
1935 /*
1936 * This cannot be done for data
1937 * inode to the transaction's
1938 */
1939 if (page_has_buffers(page)) {
1940 if (!ext4_walk_page_buffers(
1941 0, len, NULL,
1942 ext4_bh_unmapped))
1943 /* Wait so that we don't change
1944 wait_for_stable_page(page);
1945 ret = VM_FAULT_LOCKED;
1946 goto out;
1947 }
1948 }
1949
1950 I
1951 unlock_page(page);
1952 /* OK, we need to fill the hole...
1953 if (ext4_should_dioread_nolock(inode))
1954 get_block = ext4_get_block_unwritten;
1955 else
1956 get_block = ext4_get_block;
1957 retry_alloc:
1958 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1959 ext4_writepage_trans_blocks(inode));
1960 if (IS_ERR(handle)) {
1961 ret = VM_FAULT_SIGBUS;
1962 goto out;
1963 }
1964 }
1965
1966 /*
1967 * This cannot be done for data
1968 * inode to the transaction's
1969 */
1970 if (page_has_buffers(page)) {
1971 if (!ext4_walk_page_buffers(
1972 0, len, NULL,
1973 ext4_bh_unmapped))
1974 /* Wait so that we don't change
1975 wait_for_stable_page(page);
1976 ret = VM_FAULT_LOCKED;
1977 goto out;
1978 }
1979 }
1980
1981 I
1982 unlock_page(page);
1983 /* OK, we need to fill the hole...
1984 if (ext4_should_dioread_nolock(inode))
1985 get_block = ext4_get_block_unwritten;
1986 else
1987 get_block = ext4_get_block;
1988 retry_alloc:
1989 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
1990 ext4_writepage_trans_blocks(inode));
1991 if (IS_ERR(handle)) {
1992 ret = VM_FAULT_SIGBUS;
1993 goto out;
1994 }
1995 }
1996
1997 /*
1998 * This cannot be done for data
1999 * inode to the transaction's
2000 */
2001 if (page_has_buffers(page)) {
2002 if (!ext4_walk_page_buffers(
2003 0, len, NULL,
2004 ext4_bh_unmapped))
2005 /* Wait so that we don't change
2006 wait_for_stable_page(page);
2007 ret = VM_FAULT_LOCKED;
2008 goto out;
2009 }
2010 }
2011
2012 I
2013 unlock_page(page);
2014 /* OK, we need to fill the hole...
2015 if (ext4_should_dioread_nolock(inode))
2016 get_block = ext4_get_block_unwritten;
2017 else
2018 get_block = ext4_get_block;
2019 retry_alloc:
2020 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2021 ext4_writepage_trans_blocks(inode));
2022 if (IS_ERR(handle)) {
2023 ret = VM_FAULT_SIGBUS;
2024 goto out;
2025 }
2026 }
2027
2028 /*
2029 * This cannot be done for data
2030 * inode to the transaction's
2031 */
2032 if (page_has_buffers(page)) {
2033 if (!ext4_walk_page_buffers(
2034 0, len, NULL,
2035 ext4_bh_unmapped))
2036 /* Wait so that we don't change
2037 wait_for_stable_page(page);
2038 ret = VM_FAULT_LOCKED;
2039 goto out;
2040 }
2041 }
2042
2043 I
2044 unlock_page(page);
2045 /* OK, we need to fill the hole...
2046 if (ext4_should_dioread_nolock(inode))
2047 get_block = ext4_get_block_unwritten;
2048 else
2049 get_block = ext4_get_block;
2050 retry_alloc:
2051 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2052 ext4_writepage_trans_blocks(inode));
2053 if (IS_ERR(handle)) {
2054 ret = VM_FAULT_SIGBUS;
2055 goto out;
2056 }
2057 }
2058
2059 /*
2060 * This cannot be done for data
2061 * inode to the transaction's
2062 */
2063 if (page_has_buffers(page)) {
2064 if (!ext4_walk_page_buffers(
2065 0, len, NULL,
2066 ext4_bh_unmapped))
2067 /* Wait so that we don't change
2068 wait_for_stable_page(page);
2069 ret = VM_FAULT_LOCKED;
2070 goto out;
2071 }
2072 }
2073
2074 I
2075 unlock_page(page);
2076 /* OK, we need to fill the hole...
2077 if (ext4_should_dioread_nolock(inode))
2078 get_block = ext4_get_block_unwritten;
2079 else
2080 get_block = ext4_get_block;
2081 retry_alloc:
2082 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2083 ext4_writepage_trans_blocks(inode));
2084 if (IS_ERR(handle)) {
2085 ret = VM_FAULT_SIGBUS;
2086 goto out;
2087 }
2088 }
2089
2090 /*
2091 * This cannot be done for data
2092 * inode to the transaction's
2093 */
2094 if (page_has_buffers(page)) {
2095 if (!ext4_walk_page_buffers(
2096 0, len, NULL,
2097 ext4_bh_unmapped))
2098 /* Wait so that we don't change
2099 wait_for_stable_page(page);
2100 ret = VM_FAULT_LOCKED;
2101 goto out;
2102 }
2103 }
2104
2105 I
2106 unlock_page(page);
2107 /* OK, we need to fill the hole...
2108 if (ext4_should_dioread_nolock(inode))
2109 get_block = ext4_get_block_unwritten;
2110 else
2111 get_block = ext4_get_block;
2112 retry_alloc:
2113 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2114 ext4_writepage_trans_blocks(inode));
2115 if (IS_ERR(handle)) {
2116 ret = VM_FAULT_SIGBUS;
2117 goto out;
2118 }
2119 }
2120
2121 /*
2122 * This cannot be done for data
2123 * inode to the transaction's
2124 */
2125 if (page_has_buffers(page)) {
2126 if (!ext4_walk_page_buffers(
2127 0, len, NULL,
2128 ext4_bh_unmapped))
2129 /* Wait so that we don't change
2130 wait_for_stable_page(page);
2131 ret = VM_FAULT_LOCKED;
2132 goto out;
2133 }
2134 }
2135
2136 I
2137 unlock_page(page);
2138 /* OK, we need to fill the hole...
2139 if (ext4_should_dioread_nolock(inode))
2140 get_block = ext4_get_block_unwritten;
2141 else
2142 get_block = ext4_get_block;
2143 retry_alloc:
2144 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2145 ext4_writepage_trans_blocks(inode));
2146 if (IS_ERR(handle)) {
2147 ret = VM_FAULT_SIGBUS;
2148 goto out;
2149 }
2150 }
2151
2152 /*
2153 * This cannot be done for data
2154 * inode to the transaction's
2155 */
2156 if (page_has_buffers(page)) {
2157 if (!ext4_walk_page_buffers(
2158 0, len, NULL,
2159 ext4_bh_unmapped))
2160 /* Wait so that we don't change
2161 wait_for_stable_page(page);
2162 ret = VM_FAULT_LOCKED;
2163 goto out;
2164 }
2165 }
2166
2167 I
2168 unlock_page(page);
2169 /* OK, we need to fill the hole...
2170 if (ext4_should_dioread_nolock(inode))
2171 get_block = ext4_get_block_unwritten;
2172 else
2173 get_block = ext4_get_block;
2174 retry_alloc:
2175 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2176 ext4_writepage_trans_blocks(inode));
2177 if (IS_ERR(handle)) {
2178 ret = VM_FAULT_SIGBUS;
2179 goto out;
2180 }
2181 }
2182
2183 /*
2184 * This cannot be done for data
2185 * inode to the transaction's
2186 */
2187 if (page_has_buffers(page)) {
2188 if (!ext4_walk_page_buffers(
2189 0, len, NULL,
2190 ext4_bh_unmapped))
2191 /* Wait so that we don't change
2192 wait_for_stable_page(page);
2193 ret = VM_FAULT_LOCKED;
2194 goto out;
2195 }
2196 }
2197
2198 I
2199 unlock_page(page);
2200 /* OK, we need to fill the hole...
2201 if (ext4_should_dioread_nolock(inode))
2202 get_block = ext4_get_block_unwritten;
2203 else
2204 get_block = ext4_get_block;
2205 retry_alloc:
2206 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2207 ext4_writepage_trans_blocks(inode));
2208 if (IS_ERR(handle)) {
2209 ret = VM_FAULT_SIGBUS;
2210 goto out;
2211 }
2212 }
2213
2214 /*
2215 * This cannot be done for data
2216 * inode to the transaction's
2217 */
2218 if (page_has_buffers(page)) {
2219 if (!ext4_walk_page_buffers(
2220 0, len, NULL,
2221 ext4_bh_unmapped))
2222 /* Wait so that we don't change
2223 wait_for_stable_page(page);
2224 ret = VM_FAULT_LOCKED;
2225 goto out;
2226 }
2227 }
2228
2229 I
2230 unlock_page(page);
2231 /* OK, we need to fill the hole...
2232 if (ext4_should_dioread_nolock(inode))
2233 get_block = ext4_get_block_unwritten;
2234 else
2235 get_block = ext4_get_block;
2236 retry_alloc:
2237 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2238 ext4_writepage_trans_blocks(inode));
2239 if (IS_ERR(handle)) {
2240 ret = VM_FAULT_SIGBUS;
2241 goto out;
2242 }
2243 }
2244
2245 /*
2246 * This cannot be done for data
2247 * inode to the transaction's
2248 */
2249 if (page_has_buffers(page)) {
2250 if (!ext4_walk_page_buffers(
2251 0, len, NULL,
2252 ext4_bh_unmapped))
2253 /* Wait so that we don't change
2254 wait_for_stable_page(page);
2255 ret = VM_FAULT_LOCKED;
2256 goto out;
2257 }
2258 }
2259
2260 I
2261 unlock_page(page);
2262 /* OK, we need to fill the hole...
2263 if (ext4_should_dioread_nolock(inode))
2264 get_block = ext4_get_block_unwritten;
2265 else
2266 get_block = ext4_get_block;
2267 retry_alloc:
2268 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2269 ext4_writepage_trans_blocks(inode));
2270 if (IS_ERR(handle)) {
2271 ret = VM_FAULT_SIGBUS;
2272 goto out;
2273 }
2274 }
2275
2276 /*
2277 * This cannot be done for data
2278 * inode to the transaction's
2279 */
2280 if (page_has_buffers(page)) {
2281 if (!ext4_walk_page_buffers(
2282 0, len, NULL,
2283 ext4_bh_unmapped))
2284 /* Wait so that we don't change
2285 wait_for_stable_page(page);
2286 ret = VM_FAULT_LOCKED;
2287 goto out;
2288 }
2289 }
2290
2291 I
2292 unlock_page(page);
2293 /* OK, we need to fill the hole...
2294 if (ext4_should_dioread_nolock(inode))
2295 get_block = ext4_get_block_unwritten;
2296 else
2297 get_block = ext4_get_block;
2298 retry_alloc:
2299 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2300 ext4_writepage_trans_blocks(inode));
2301 if (IS_ERR(handle)) {
2302 ret = VM_FAULT_SIGBUS;
2303 goto out;
2304 }
2305 }
2306
2307 /*
2308 * This cannot be done for data
2309 * inode to the transaction's
2310 */
2311 if (page_has_buffers(page)) {
2312 if (!ext4_walk_page_buffers(
2313 0, len, NULL,
2314 ext4_bh_unmapped))
2315 /* Wait so that we don't change
2316 wait_for_stable_page(page);
2317 ret = VM_FAULT_LOCKED;
2318 goto out;
2319 }
2320 }
2321
2322 I
2323 unlock_page(page);
2324 /* OK, we need to fill the hole...
2325 if (ext4_should_dioread_nolock(inode))
2326 get_block = ext4_get_block_unwritten;
2327 else
2328 get_block = ext4_get_block;
2329 retry_alloc:
2330 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2331 ext4_writepage_trans_blocks(inode));
2332 if (IS_ERR(handle)) {
2333 ret = VM_FAULT_SIGBUS;
2334 goto out;
2335 }
2336 }
2337
2338 /*
2339 * This cannot be done for data
2340 * inode to the transaction's
2341 */
2342 if (page_has_buffers(page)) {
2343 if (!ext4_walk_page_buffers(
2344 0, len, NULL,
2345 ext4_bh_unmapped))
2346 /* Wait so that we don't change
2347 wait_for_stable_page(page);
2348 ret = VM_FAULT_LOCKED;
2349 goto out;
2350 }
2351 }
2352
2353 I
2354 unlock_page(page);
2355 /* OK, we need to fill the hole...
2356 if (ext4_should_dioread_nolock(inode))
2357 get_block = ext4_get_block_unwritten;
2358 else
2359 get_block = ext4_get_block;
2360 retry_alloc:
2361 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2362 ext4_writepage_trans_blocks(inode));
2363 if (IS_ERR(handle)) {
2364 ret = VM_FAULT_SIGBUS;
2365 goto out;
2366 }
2367 }
2368
2369 /*
2370 * This cannot be done for data
2371 * inode to the transaction's
2372 */
2373 if (page_has_buffers(page)) {
2374 if (!ext4_walk_page_buffers(
2375 0, len, NULL,
2376 ext4_bh_unmapped))
2377 /* Wait so that we don't change
2378 wait_for_stable_page(page);
2379 ret = VM_FAULT_LOCKED;
2380 goto out;
2381 }
2382 }
2383
2384 I
2385 unlock_page(page);
2386 /* OK, we need to fill the hole...
2387 if (ext4_should_dioread_nolock(inode))
2388 get_block = ext4_get_block_unwritten;
2389 else
2390 get_block = ext4_get_block;
2391 retry_alloc:
2392 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2393 ext4_writepage_trans_blocks(inode));
2394 if (IS_ERR(handle)) {
2395 ret = VM_FAULT_SIGBUS;
2396 goto out;
2397 }
2398 }
2399
2400 /*
2401 * This cannot be done for data
2402 * inode to the transaction's
2403 */
2404 if (page_has_buffers(page)) {
2405 if (!ext4_walk_page_buffers(
2406 0, len, NULL,
2407 ext4_bh_unmapped))
2408 /* Wait so that we don't change
2409 wait_for_stable_page(page);
2410 ret = VM_FAULT_LOCKED;
2411 goto out;
2412 }
2413 }
2414
2415 I
2416 unlock_page(page);
2417 /* OK, we need to fill the hole...
2418 if (ext4_should_dioread_nolock(inode))
2419 get_block = ext4_get_block_unwritten;
2420 else
2421 get_block = ext4_get_block;
2422 retry_alloc:
2423 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2424 ext4_writepage_trans_blocks(inode));
2425 if (IS_ERR(handle)) {
2426 ret = VM_FAULT_SIGBUS;
2427 goto out;
2428 }
2429 }
2430
2431 /*
2432 * This cannot be done for data
2433 * inode to the transaction's
2434 */
2435 if (page_has_buffers(page)) {
2436 if (!ext4_walk_page_buffers(
2437 0, len, NULL,
2438 ext4_bh_unmapped))
2439 /* Wait so that we don't change
2440 wait_for_stable_page(page);
2441 ret = VM_FAULT_LOCKED;
2442 goto out;
2443 }
2444 }
2445
2446 I
2447 unlock_page(page);
2448 /* OK, we need to fill the hole...
2449 if (ext4_should_dioread_nolock(inode))
2450 get_block = ext4_get_block_unwritten;
2451 else
2452 get_block = ext4_get_block;
2453 retry_alloc:
2454 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2455 ext4_writepage_trans_blocks(inode));
2456 if (IS_ERR(handle)) {
2457 ret = VM_FAULT_SIGBUS;
2458 goto out;
2459 }
2460 }
2461
2462 /*
2463 * This cannot be done for data
2464 * inode to the transaction's
2465 */
2466 if (page_has_buffers(page)) {
2467 if (!ext4_walk_page_buffers(
2468 0, len, NULL,
2469 ext4_bh_unmapped))
2470 /* Wait so that we don't change
2471 wait_for_stable_page(page);
2472 ret = VM_FAULT_LOCKED;
2473 goto out;
2474 }
2475 }
2476
2477 I
2478 unlock_page(page);
2479 /* OK, we need to fill the hole...
2480 if (ext4_should_dioread_nolock(inode))
2481 get_block = ext4_get_block_unwritten;
2482 else
2483 get_block = ext4_get_block;
2484 retry_alloc:
2485 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2486 ext4_writepage_trans_blocks(inode));
2487 if (IS_ERR(handle)) {
2488 ret = VM_FAULT_SIGBUS;
2489 goto out;
2490 }
2491 }
2492
2493 /*
2494 * This cannot be done for data
2495 * inode to the transaction's
2496 */
2497 if (page_has_buffers(page)) {
2498 if (!ext4_walk_page_buffers(
2499 0, len, NULL,
2500 ext4_bh_unmapped))
2501 /* Wait so that we don't change
2502 wait_for_stable_page(page);
2503 ret = VM_FAULT_LOCKED;
2504 goto out;
2505 }
2506 }
2507
2508 I
2509 unlock_page(page);
2510 /* OK, we need to fill the hole...
2511 if (ext4_should_dioread_nolock(inode))
2512 get_block = ext4_get_block_unwritten;
2513 else
2514 get_block = ext4_get_block;
2515 retry_alloc:
2516 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2517 ext4_writepage_trans_blocks(inode));
2518 if (IS_ERR(handle)) {
2519 ret = VM_FAULT_SIGBUS;
2520 goto out;
2521 }
2522 }
2523
2524 /*
2525 * This cannot be done for data
2526 * inode to the transaction's
2527 */
2528 if (page_has_buffers(page)) {
2529 if (!ext4_walk_page_buffers(
2530 0, len, NULL,
2531 ext4_bh_unmapped))
2532 /* Wait so that we don't change
2533 wait_for_stable_page(page);
2534 ret = VM_FAULT_LOCKED;
2535 goto out;
2536 }
2537 }
2538
2539 I
2540 unlock_page(page);
2541 /* OK, we need to fill the hole...
2542 if (ext4_should_dioread_nolock(inode))
2543 get_block = ext4_get_block_unwritten;
2544 else
2545 get_block = ext4_get_block;
2546 retry_alloc:
2547 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2548 ext4_writepage_trans_blocks(inode));
2549 if (IS_ERR(handle)) {
2550 ret = VM_FAULT_SIGBUS;
2551 goto out;
2552 }
2553 }
2554
2555 /*
2556 * This cannot be done for data
2557 * inode to the transaction's
2558 */
2559 if (page_has_buffers(page)) {
2560 if (!ext4_walk_page_buffers(
2561 0, len, NULL,
2562 ext4_bh_unmapped))
2563 /* Wait so that we don't change
2564 wait_for_stable_page(page);
2565 ret = VM_FAULT_LOCKED;
2566 goto out;
2567 }
2568 }
2569
2570 I
2571 unlock_page(page);
2572 /* OK, we need to fill the hole...
2573 if (ext4_should_dioread_nolock(inode))
2574 get_block = ext4_get_block_unwritten;
2575 else
2576 get_block = ext4_get_block;
2577 retry_alloc:
2578 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2579 ext4_writepage_trans_blocks(inode));
2580 if (IS_ERR(handle)) {
2581 ret = VM_FAULT_SIGBUS;
2582 goto out;
2583 }
2584 }
2585
2586 /*
2587 * This cannot be done for data
2588 * inode to the transaction's
2589 */
2590 if (page_has_buffers(page)) {
2591 if (!ext4_walk_page_buffers(
2592 0, len, NULL,
2593 ext4_bh_unmapped))
2594 /* Wait so that we don't change
2595 wait_for_stable_page(page);
2596 ret = VM_FAULT_LOCKED;
2597 goto out;
2598 }
2599 }
2600
2601 I
2602 unlock_page(page);
2603 /* OK, we need to fill the hole...
2604 if (ext4_should_dioread_nolock(inode))
2605 get_block = ext4_get_block_unwritten;
2606 else
2607 get_block = ext4_get_block;
2608 retry_alloc:
2609 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2610 ext4_writepage_trans_blocks(inode));
2611 if (IS_ERR(handle)) {
2612 ret = VM_FAULT_SIGBUS;
2613 goto out;
2614 }
2615 }
2616
2617 /*
2618 * This cannot be done for data
2619 * inode to the transaction's
2620 */
2621 if (page_has_buffers(page)) {
2622 if (!ext4_walk_page_buffers(
2623 0, len, NULL,
2624 ext4_bh_unmapped))
2625 /* Wait so that we don't change
2626 wait_for_stable_page(page);
2627 ret = VM_FAULT_LOCKED;
2628 goto out;
2629 }
2630 }
2631
2632 I
2633 unlock_page(page);
2634 /* OK, we need to fill the hole...
2635 if (ext4_should_dioread_nolock(inode))
2636 get_block = ext4_get_block_unwritten;
2637 else
2638 get_block = ext4_get_block;
2639 retry_alloc:
2640 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2641 ext4_writepage_trans_blocks(inode));
2642 if (IS_ERR(handle)) {
2643 ret = VM_FAULT_SIGBUS;
2644 goto out;
2645 }
2646 }
2647
2648 /*
2649 * This cannot be done for data
2650 * inode to the transaction's
2651 */
2652 if (page_has_buffers(page)) {
2653 if (!ext4_walk_page_buffers(
2654 0, len, NULL,
2655 ext4_bh_unmapped))
2656 /* Wait so that we don't change
2657 wait_for_stable_page(page);
2658 ret = VM_FAULT_LOCKED;
2659 goto out;
2660 }
2661 }
2662
2663 I
2664 unlock_page(page);
2665 /* OK, we need to fill the hole...
2666 if (ext4_should_dioread_nolock(inode))
2667 get_block = ext4_get_block_unwritten;
2668 else
2669 get_block = ext4_get_block;
2670 retry_alloc:
2671 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2672 ext4_writepage_trans_blocks(inode));
2673 if (IS_ERR(handle)) {
2674 ret = VM_FAULT_SIGBUS;
2675 goto out;
2676 }
2677 }
2678
2679 /*
2680 * This cannot be done for data
2681 * inode to the transaction's
2682 */
2683 if (page_has_buffers(page)) {
2684 if (!ext4_walk_page_buffers(
2685 0, len, NULL,
2686 ext4_bh_unmapped))
2687 /* Wait so that we don't change
2688 wait_for_stable_page(page);
2689 ret = VM_FAULT_LOCKED;
2690 goto out;
2691 }
2692 }
2693
2694 I
2695 unlock_page(page);
2696 /* OK, we need to fill the hole...
2697 if (ext4_should_dioread_nolock(inode))
2698 get_block = ext4_get_block_unwritten;
2699 else
2700 get_block = ext4_get_block;
2701 retry_alloc:
2702 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2703 ext4_writepage_trans_blocks(inode));
2704 if (IS_ERR(handle)) {
2705 ret = VM_FAULT_SIGBUS;
2706 goto out;
2707 }
2708 }
2709
2710 /*
2711 * This cannot be done for data
2712 * inode to the transaction's
2713 */
2714 if (page_has_buffers(page)) {
2715 if (!ext4_walk_page_buffers(
2716 0, len, NULL,
2717 ext4_bh_unmapped))
2718 /* Wait so that we don't change
2719 wait_for_stable_page(page);
2720 ret = VM_FAULT_LOCKED;
2721 goto out;
2722 }
2723 }
2724
2725 I
2726 unlock_page(page);
2727 /* OK, we need to fill the hole...
2728 if (ext4_should_dioread_nolock(inode))
2729 get_block = ext4_get_block_unwritten;
2730 else
2731 get_block = ext4_get_block;
2732 retry_alloc:
2733 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2734 ext4_writepage_trans_blocks(inode));
2735 if (IS_ERR(handle)) {
2736 ret = VM_FAULT_SIGBUS;
2737 goto out;
2738 }
2739 }
2740
2741 /*
2742 * This cannot be done for data
2743 * inode to the transaction's
2744 */
2745 if (page_has_buffers(page)) {
2746 if (!ext4_walk_page_buffers(
2747 0, len, NULL,
2748 ext4_bh_unmapped))
2749 /* Wait so that we don't change
2750 wait_for_stable_page(page);
2751 ret = VM_FAULT_LOCKED;
2752 goto out;
2753 }
2754 }
2755
2756 I
2757 unlock_page(page);
2758 /* OK, we need to fill the hole...
2759 if (ext4_should_dioread_nolock(inode))
2760 get_block = ext4_get_block_unwritten;
2761 else
2762 get_block = ext4_get_block;
2763 retry_alloc:
2764 handle = ext4_journal_start(inode, EXT4_HT_WRITE_PAGE,
2765 ext4_writepage_trans_blocks(inode));
2766 if (IS_ERR(handle)) {
2767 ret = VM_FAULT_SIGBUS;
2768 goto out;
2769 }
2770 }
2771
2772 /*
2773 * This cannot be done for data
2774 * inode to the transaction's
2775 */
2776 if (page_has_buffers(page)) {
2777 if (!ext4_walk_page_buffers(
2778 0, len, NULL,
2779 ext4_bh_unmapped))
2780 /* Wait so
```



# 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\_RUNNING) .

- / 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'ordonnement

## / 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  $PS \geq 120$ ,  $QB = (140 - PS) * 5$
- / 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
  - /  $PD = \max(100, \min(PS - \text{bonus} + 5, 139))$
- / **bonus** est une valeur entre 0 et 10
  - / Une valeur **< 5 est une pénalité** qui baissera la PD
  - / Valeur  **$\geq 5$  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  $PD \leq 3 * PS/4 + 28$  alors interactif
  - / Ce qui est équivalent à  $\text{bonus} - 5 \geq PS/4 - 28$
  - /  $PS/4 - 28$  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 <i>nice</i>	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

$$\text{bonus} - 5 \geq PS/4 - 28$$

# 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
  - /  $\text{bonus} - 5 \geq \text{PS}/4 - 28 \Rightarrow \text{bonus} \geq 139/4 - 28 + 5$
  - $\text{bonus} \geq 11$  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

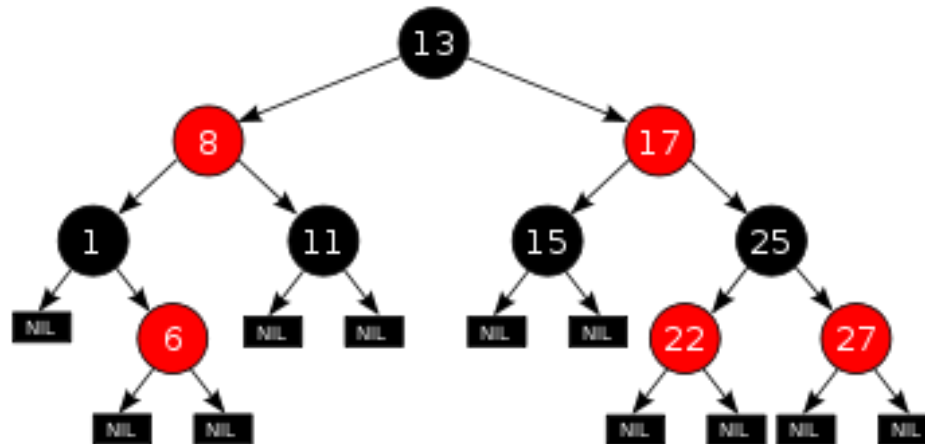
/ **Interactivité** (delta interactif): reste dans la liste des processus actifs

/ 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  $\searrow$ )
- / Organisation des tâches en utilisant un arbre rouge-noir (ou bicolore), complexité  $O(\log(N))$  pour l'insertion



Source: Wikipedia

# 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 tâches:
    - / 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

## / Conventiionnels

- / **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.