Cours sur l'Ordonnancement

MENRA W. Romial

PhD Student

Table des matières

Introduction Générale

Public Cible

Étudiants en informatique (Master, École d'ingénieur), Développeurs Système, Administrateurs Système avancés.

Prérequis

— Connaissances de base des systèmes d'exploitation (processus, threads, mémoire).
— Notions de programmation C (pour les exemples et TP).
— Utilisation basique de Linux en ligne de commande.

Objectifs du Cours

— Comprendre les concepts fondamentaux de l'ordonnancement.
— Maîtriser les mécanismes d'ordonnancement du noyau Linux (CFS, RT, Deadline).
— Appréhender les spécificités et défis de l'ordonnancement temps réel.
— Explorer les problématiques d'ordonnancement dans les systèmes distribués.
— Savoir analyser, configurer et optimiser le comportement de l'ordonnanceur Linux.

1

Introduction et Rappels Fondamentaux

L'ordonnancement est au cœur du fonctionnement des systèmes d'exploitation multitâches. Il s'agit du mécanisme qui permet de partager efficacement la ressource la plus critique d'un ordinateur, le processeur (CPU), entre plusieurs processus ou threads qui souhaitent s'exécuter simultanément. Ce chapitre introductif a pour but de poser les bases nécessaires à la compréhension des mécanismes d'ordonnancement plus complexes, notamment ceux implémentés dans le noyau Linux.

§1.1 Qu'est-ce que l'ordonnancement ?

L'ordonnancement, dans le contexte des systèmes d'exploitation, est l'ensemble des politiques et des mécanismes qui gouvernent l'ordre dans lequel les travaux (processus ou threads) soumis à un système sont exécutés. Son rôle principal est de décider quel processus doit s'exécuter à un instant donné, quand et pendant combien de temps.

Les **objectifs principaux** d'un bon algorithme d'ordonnancement peuvent varier en fonction du type de système, mais incluent généralement :

- **Équité (Fairness) :** Assurer que chaque processus reçoive une part équitable du temps CPU, évitant qu'un processus ne monopolise la ressource indéfiniment au détriment des autres.
- **Débit (Throughput) :** Maximiser le nombre de processus complétés par unité de temps. Ceci est particulièrement important pour les systèmes de traitement par lots (batch).
- **Temps de réponse (Response Time) :** Minimiser le temps écoulé entre la soumission d'une requête par un utilisateur (par exemple, une frappe au clavier) et l'affichage de la première réponse par le système. Crucial pour les systèmes interactifs.
- **Temps de rotation (Turnaround Time) :** Minimiser le temps total nécessaire à l'exécution d'un processus particulier, depuis sa soumission jusqu'à sa complétion.
- **Temps d'attente (Waiting Time) :** Minimiser le temps qu'un processus passe dans la file d'attente des processus prêts, attendant d'être élu par l'ordonnanceur.
- **Utilisation du CPU (CPU Utilization) :** Maintenir le CPU aussi occupé que possible, afin de ne pas gaspiller de cycles.
- **Respect des échéances (Deadline Adherence) :** Pour les systèmes temps réel, il est impératif que les tâches se terminent avant leur échéance.

Il est important de noter que certains de ces objectifs sont contradictoires (par exemple, minimiser le temps de réponse peut parfois se faire au détriment du débit global). Le choix d'un algorithme d'ordonnancement est donc souvent un compromis.

On distingue classiquement plusieurs **types d'ordonnancement** selon la nature du système :

- **Ordonnancement par lot (Batch Scheduling) :** Utilisé dans les systèmes où les travaux sont soumis en lots, sans interaction utilisateur directe pendant leur exécution (ex : calculs scientifiques intensifs). L'objectif principal est souvent le débit.
- **Ordonnancement interactif (Interactive/Timesharing Scheduling) :** Utilisé dans les systèmes d'exploitation modernes (PCs, serveurs) où plusieurs utilisateurs ou applications interagissent avec le système. L'objectif principal est un bon temps de réponse.
- **Ordonnancement temps réel (Real-time Scheduling) :** Utilisé dans les systèmes où le respect des contraintes temporelles est critique (ex : systèmes de contrôle industriel, avionique, multimédia). L'objectif principal est le respect des échéances.

§1.2 Processus et Threads sous Linux

Un **processus** est une instance d'un programme en cours d'exécution. Il est caractérisé par son propre espace d'adressage mémoire, un compteur ordinal (Program Counter - PC) qui indique la prochaine instruction à exécuter, un ensemble de registres, une pile, et des informations d'état gérées par le noyau (comme ses priorités, ses fichiers ouverts, etc.). Sous Linux, chaque processus est représenté par une structure de données complexe appelée `struct task_struct`.

Un **thread** (ou processus léger) est une unité d'exécution au sein d'un processus. Plusieurs threads peuvent coexister au sein d'un même processus et partagent son espace d'adressage, ses fichiers ouverts et d'autres ressources. Chaque thread possède cependant son propre compteur ordinal, sa pile et son ensemble de registres. L'utilisation de threads permet d'améliorer le parallélisme et la réactivité des applications. Pour l'ordonnanceur Linux, les threads sont les entités fondamentales à ordonnancer (Linux ne distingue pas fondamentalement les threads des processus au niveau de l'ordonnancement ; chaque thread est une "tâche" avec son propre `task_struct`).

• 1.2.1 Cycle de vie d'un processus/thread

Un processus passe par différents états au cours de son existence :

- **Nouveau (New) :** Le processus est en cours de création.
- **Prêt (Ready/Runnable) :** Le processus est prêt à s'exécuter et attend d'être choisi par l'ordonnanceur pour utiliser le CPU. Les processus prêts sont généralement maintenus dans une file d'attente (runqueue).
- **Élu/En exécution (Running) :** Le processus est en cours d'exécution sur un CPU.
- **Bloqué/En attente (Blocked/Waiting) :** Le processus ne peut pas continuer son exécution car il attend un événement externe (ex : fin d'une opération d'Entrée/Sortie,

disponibilité d'une ressource, réception d'un signal).
— **Terminé (Terminated/Zombie) :** Le processus a terminé son exécution mais sa structure `task_struct` existe encore pour que son processus parent puisse récupérer son code de retour. Une fois cette information récupérée (via l'appel système `wait()`), la structure est libérée.

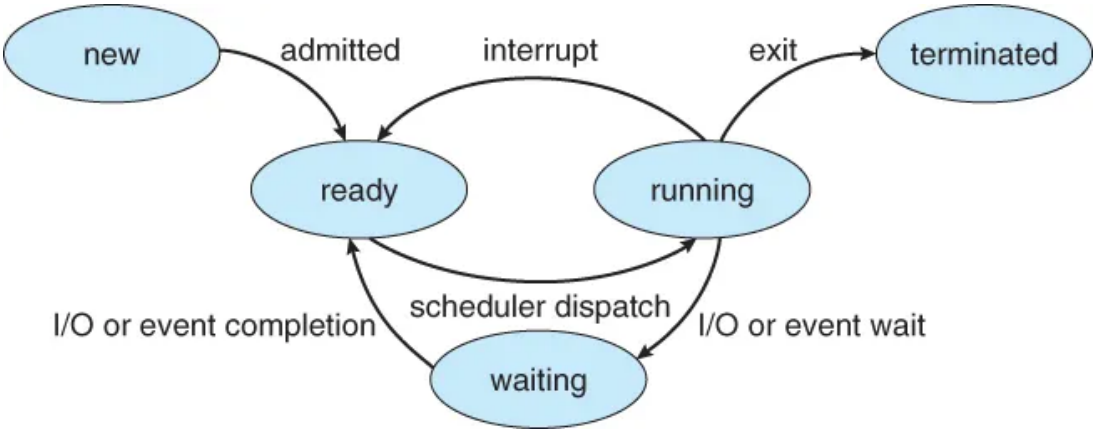

FIGURE 1.1 – Cycle de vie simplifié d'un processus.

1.2.2 Commutation de contexte (Context Switch)

La **commutation de contexte** est l'opération qui consiste pour le noyau à sauvegarder l'état (contexte CPU : registres, compteur ordinal, pointeur de pile, etc.) du processus qui s'exécute actuellement et à restaurer l'état d'un autre processus qui a été choisi par l'ordonnanceur. Cette opération est essentielle pour permettre la multiprogrammation et le partage du temps.

Une commutation de contexte peut survenir :
— Lorsqu'un processus termine son quantum de temps (dans un système préemptif).
— Lorsqu'un processus se bloque (e.g., en attente d'une E/S).
— Lorsqu'un processus se termine.
— Suite à une interruption qui rend un processus de plus haute priorité prêt à s'exécuter.

La commutation de contexte est une opération coûteuse en termes de temps CPU car elle implique :
— La sauvegarde et la restauration de nombreux registres.
— Des opérations sur les structures de données du noyau gérant les processus.
— Potentiellement, l'invalidation de caches (cache CPU, TLB - Translation Lookaside Buffer) car le nouveau processus s'exécute dans un contexte d'adressage différent, ce qui peut entraîner des défauts de cache et ralentir initialement l'exécution du nouveau processus.

Minimiser la fréquence et le coût des commutations de contexte est donc un objectif important, bien qu'il faille le balancer avec la nécessité d'assurer la réactivité du système.

§1.3 Critères d'évaluation d'un algorithme d'ordonnancement

Pour comparer et choisir un algorithme d'ordonnancement, plusieurs critères quantitatifs et qualitatifs sont utilisés. Ces critères sont souvent liés aux objectifs mentionnés précédemment :

- **Utilisation du CPU (CPU Utilization) :** Pourcentage de temps pendant lequel le CPU est occupé à exécuter des processus utilisateurs ou système, par opposition au temps où il est inactif (idle). On cherche à maximiser cette valeur, typiquement entre 40% (système peu chargé) et 90% (système très chargé).
- **Débit (Throughput) :** Nombre de processus complétés par unité de temps. On cherche à le maximiser.
- **Temps de rotation (Turnaround Time) :** Temps total écoulé entre la soumission d'un processus et sa complétion. C'est la somme du temps d'attente, du temps d'exécution et du temps passé en E/S. On cherche à le minimiser.
- **Temps d'attente (Waiting Time) :** Temps total qu'un processus passe dans la file des processus prêts. Il ne dépend que de l'algorithme d'ordonnancement et non du temps d'exécution ou des E/S. On cherche à le minimiser.
- **Temps de réponse (Response Time) :** Dans un système interactif, c'est le temps entre la soumission d'une commande et l'apparition de la première réponse, et non la sortie complète. On cherche à le minimiser pour une bonne interactivité.
- **Équité (Fairness) :** S'assurer qu'aucun processus n'est indéfiniment privé de ressources (famine) et que les processus similaires reçoivent un traitement similaire.
- **Prévisibilité (Predictability) :** Dans les systèmes temps réel, il est crucial que le comportement de l'ordonnanceur soit prévisible afin de garantir le respect des échéances.

Le choix des critères prioritaires dépend fortement de la nature et des objectifs du système d'exploitation.

§1.4 Ordonnancement préemptif vs. non-préemptif

Les algorithmes d'ordonnancement peuvent être classés en deux grandes catégories selon qu'ils autorisent ou non l'interruption d'un processus en cours d'exécution.

• 1.4.1 Ordonnancement non-préemptif (ou coopératif)

Dans un système d'ordonnancement **non-préemptif**, une fois qu'un processus a été élu et a commencé son exécution, il continue de s'exécuter jusqu'à ce qu'il :

- Se termine.
- Se bloque volontairement (par exemple, en attendant une opération d'E/S ou une ressource).

Le processus ne peut pas être interrompu de force par l'ordonnanceur pour laisser la place à un autre processus.

- **Avantages :** Simplicité d'implémentation, moins de commutations de contexte "for-

cées", moins de problèmes de synchronisation car un processus ne peut pas être interrompu à un moment inopportun au milieu d'une section critique (sauf par une interruption matérielle).
- **Inconvénients :** Un processus long ou bogué peut monopoliser le CPU, rendant le système non réactif. Ne convient pas aux systèmes interactifs ou temps réel où une réponse rapide est nécessaire.

Des exemples historiques incluent les premières versions de Windows (jusqu'à 3.x) ou MacOS (jusqu'à System 9).

1.4.2 Ordonnancement préemptif

Dans un système d'ordonnancement **préemptif**, l'ordonnanceur peut interrompre un processus en cours d'exécution, même s'il n'a pas terminé ou ne s'est pas bloqué volontairement, pour allouer le CPU à un autre processus. La préemption peut survenir :
- À la fin du quantum de temps alloué au processus.
- Lorsqu'un processus de plus haute priorité devient prêt.
- Suite à une interruption matérielle.
- **Avantages :** Meilleure réactivité du système, partage plus équitable du CPU, meilleure adaptation aux systèmes interactifs et temps réel.
- **Inconvénients :** Plus complexe à implémenter, coût des commutations de contexte plus fréquent, nécessité de mécanismes de synchronisation robustes pour protéger les données partagées contre les accès concurrents et les conditions de concurrence (race conditions) dues aux interruptions.

La plupart des systèmes d'exploitation modernes, y compris Linux, Windows (depuis NT), et macOS (depuis OS X), utilisent un ordonnancement préemptif.

§1.5 Notions de base

Quelques concepts clés sont récurrents dans l'étude de l'ordonnancement :
- **Quantum de temps (Timeslice ou Time Quantum) :** Dans les algorithmes d'ordonnancement préemptifs comme le Round-Robin (Tourniquet), le quantum de temps est la durée maximale pendant laquelle un processus peut s'exécuter avant d'être préempté et replacé dans la file des processus prêts (si d'autres processus sont en attente). Le choix de la taille du quantum est critique :
  - *Quantum trop petit :* Beaucoup de commutations de contexte, ce qui augmente la surcharge (overhead) du système et diminue l'efficacité.
  - *Quantum trop grand :* Le système peut devenir moins réactif, se rapprochant d'un comportement FCFS (First-Come, First-Served) pour les processus CPU-bound.

  La valeur typique d'un quantum se situe dans la plage de quelques millisecondes à quelques dizaines de millisecondes.
- **Priorités :** Une priorité est une valeur numérique associée à un processus, indiquant son importance relative par rapport aux autres processus. L'ordonnanceur utilisera cette information pour choisir le prochain processus à exécuter (généralement, le pro-

cessus prêt ayant la plus haute priorité). Les priorités peuvent être :
  - *Statiques :* Assignées à un processus lors de sa création (ou par un utilisateur privilégié) et ne changent pas pendant son exécution, sauf intervention manuelle.
  - *Dynamiques :* Ajustées par le système d'exploitation pendant l'exécution du processus, en fonction de son comportement (par exemple, un processus qui effectue beaucoup d'E/S peut voir sa priorité augmentée temporairement pour améliorer la réactivité).
- **Famine (Starvation) :** La famine est une situation où un processus prêt à s'exécuter est constamment ignoré par l'ordonnanceur et ne reçoit jamais de temps CPU, ou en reçoit très peu, souvent au profit de processus de plus haute priorité. C'est un problème potentiel dans les systèmes utilisant des priorités strictes sans mécanisme de compensation. Une solution courante est le *vieillissement (aging)*, où la priorité d'un processus augmente progressivement avec le temps passé en attente.
- **Inversion de priorité (Priority Inversion) :** C'est un scénario problématique dans les systèmes à priorités préemptifs où un processus de haute priorité se retrouve bloqué en attendant une ressource (par exemple, un sémaphore ou un mutex) détenue par un processus de basse priorité. Si un processus de priorité intermédiaire (plus élevée que le processus de basse priorité détenant la ressource, mais plus basse que le processus de haute priorité bloqué) devient prêt, il peut préempter le processus de basse priorité, empêchant ainsi ce dernier de libérer la ressource et prolongeant indéfiniment le blocage du processus de haute priorité. Des mécanismes comme l'*héritage de priorité (priority inheritance)* ou le *plafond de priorité (priority ceiling protocol)* sont utilisés pour contrer ce problème, particulièrement cruciaux dans les systèmes temps réel. Ce sujet sera exploré plus en détail dans le module sur l'ordonnancement temps réel.

Ce premier chapitre a introduit les concepts fondamentaux de l'ordonnancement. Le chapitre suivant se penchera sur les algorithmes d'ordonnancement classiques qui ont servi de base ou d'inspiration aux systèmes modernes.

2

Les Ordonnanceurs Classiques (Théorie)

Avant d'explorer les mécanismes d'ordonnancement sophistiqués du noyau Linux, il est essentiel de comprendre les algorithmes classiques qui ont servi de fondation et d'inspiration. Ces algorithmes, bien que parfois simplistes, illustrent les compromis fondamentaux inhérents à la gestion des processus. Ils sont souvent utilisés comme points de comparaison ou comme briques de base dans des systèmes plus complexes.

L'objectif de ce chapitre est de présenter ces algorithmes fondamentaux, d'analyser leurs caractéristiques, leurs avantages et leurs inconvénients, notamment au regard des critères d'évaluation définis au chapitre précédent.

§2.1 First-Come, First-Served (FCFS / FIFO)

[DEF] Définition

L'algorithme **First-Come, First-Served (FCFS)**, également connu sous le nom de **First-In, First-Out (FIFO)**, est l'algorithme d'ordonnancement le plus simple. Les processus sont servis dans l'ordre où ils arrivent dans la file d'attente des processus prêts. Lorsqu'un processus est élu, il s'exécute jusqu'à sa complétion ou jusqu'à ce qu'il se bloque (c'est donc un algorithme non-préemptif par nature).

Imaginez une file d'attente à la caisse d'un supermarché : le premier arrivé est le premier servi. C'est le principe du FCFS. La file d'attente des processus prêts est gérée comme une simple file FIFO.

• 2.1.1 Caractéristiques et Évaluation

— **Simplicité :** Facile à comprendre et à implémenter.
— **Non-préemptif :** Un processus garde le CPU tant qu'il ne le relâche pas volontairement.
— **Temps d'attente moyen :** Peut être très long, surtout si des processus courts arrivent après des processus longs. C'est le principal inconvénient.

- **Effet de convoi (Convoy Effect) :** Des processus courts peuvent être bloqués derrière un processus long qui monopolise le CPU. Cela dégrade significativement le temps de réponse moyen et l'utilisation des ressources (par exemple, si le processus long fait peu d'E/S tandis que les processus courts en attente pourraient en faire).
- **Débit :** Peut être faible si l'effet de convoi est prononcé.
- **Équité :** Semble équitable au premier abord ("premier arrivé, premier servi"), mais pas en termes de partage de la ressource si les besoins des processus sont hétérogènes.

[EX] Exemple

Considérons trois processus P1, P2, P3 arrivant simultanément avec les temps d'exécution (CPU burst) suivants :
- P1 : 24 ms
- P2 : 3 ms
- P3 : 3 ms

Si les processus arrivent dans l'ordre P1, P2, P3 :
- P1 s'exécute de 0 à 24. Temps d'attente de P1 = 0 ms.
- P2 s'exécute de 24 à 27. Temps d'attente de P2 = 24 ms.
- P3 s'exécute de 27 à 30. Temps d'attente de P3 = 27 ms.

Temps d'attente moyen = $(0 + 24 + 27)/3 = 17$ ms.
Si les processus arrivent dans l'ordre P2, P3, P1 :
- P2 s'exécute de 0 à 3. Temps d'attente de P2 = 0 ms.
- P3 s'exécute de 3 à 6. Temps d'attente de P3 = 3 ms.
- P1 s'exécute de 6 à 30. Temps d'attente de P1 = 6 ms.

Temps d'attente moyen = $(0+3+6)/3 = 3$ ms. Cet exemple illustre clairement l'impact de l'ordre d'arrivée et l'effet de convoi.

Le FCFS est rarement utilisé comme algorithme principal dans les systèmes modernes interactifs, mais peut être une composante de schémas plus complexes ou pour des files d'attente spécifiques (par exemple, pour certaines opérations d'E/S).

§2.2 Shortest Job First (SJF) / Shortest Remaining Time First (SRTF)

L'idée derrière l'algorithme **Shortest Job First (SJF)** est d'exécuter en priorité le processus qui a le plus petit temps d'exécution (CPU burst) suivant.

2.2.1 SJF Non-Préemptif

[DEF] Définition

Dans sa version **non-préemptive**, lorsque le CPU devient disponible, l'ordonnanceur choisit le processus dans la file d'attente des prêts qui a la durée d'exécution estimée la plus courte. Une fois élu, ce processus s'exécute jusqu'à sa complétion ou son blocage.

2.2.2 Shortest Remaining Time First (SRTF) - SJF Préemptif

[DEF] Définition

La version **préemptive** du SJF est appelée **Shortest Remaining Time First (SRTF)**. Si un nouveau processus arrive dans la file d'attente des prêts avec un temps d'exécution restant estimé plus court que le temps restant du processus actuellement en exécution, l'ordonnanceur préempte le processus en cours et élit le nouveau processus.

2.2.3 Caractéristiques et Évaluation

— **Optimalité :** SJF (et SRTF) est prouvé optimal pour minimiser le temps d'attente moyen pour un ensemble donné de processus.
— **Difficulté de prédiction :** Le principal défi est de connaître (ou d'estimer avec précision) la durée du prochain CPU burst. Pour les traitements par lot, cela peut être spécifié par l'utilisateur. Pour les systèmes interactifs, on utilise souvent des techniques d'estimation basées sur l'historique des exécutions précédentes (par exemple, une moyenne exponentielle).

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n \tag{2.1}$$

où t_n est la durée du n-ième CPU burst réel, τ_n est la durée prédite pour le n-ième CPU burst, et τ_{n+1} est la prédiction pour le prochain. α $(0 \leq \alpha \leq 1)$ contrôle l'importance relative des mesures récentes par rapport à l'historique.
— **Risque de famine (Starvation) :** Les processus longs peuvent être constamment reportés si des processus plus courts continuent d'arriver. Ceci est particulièrement vrai pour SRTF.
— **Temps de réponse :** SRTF tend à bien se comporter pour les processus courts, améliorant le temps de réponse perçu.

[EX] Exemple

Considérons les processus suivants arrivant à $t = 0$:
- P1 : Burst = 7 ms
- P2 : Burst = 4 ms
- P3 : Burst = 1 ms
- P4 : Burst = 4 ms

Avec SJF non-préemptif (supposons cet ordre si bursts égaux : P2 avant P4) : P3 (0-1), P2 (1-5), P4 (5-9), P1 (9-16). Temps d'attente : P3=0, P2=1, P4=5, P1=9. Moyenne = $(0 + 1 + 5 + 9)/4 = 3.75$ ms.
Maintenant, si P1 arrive à $t = 0$ (burst 8), P2 à $t = 1$ (burst 4), P3 à $t = 2$ (burst 9), P4 à $t = 3$ (burst 5). Avec SRTF :
- $t = 0$: P1 commence (reste 8).
- $t = 1$: P2 arrive (burst 4). P1 reste 7. P2 est plus court, P2 préempte P1. P2 commence (reste 4).
- $t = 2$: P3 arrive (burst 9). P2 reste 3. P2 continue.
- $t = 3$: P4 arrive (burst 5). P2 reste 2. P2 continue.
- $t = 5$: P2 termine. Processus en attente : P1 (reste 7), P3 (burst 9), P4 (burst 5). P4 est élu. P4 commence (reste 5).
- $t = 10$: P4 termine. Processus en attente : P1 (reste 7), P3 (burst 9). P1 est élu. P1 commence.
- $t = 17$: P1 termine. P3 est élu.
- $t = 26$: P3 termine.

Calculer les temps d'attente pour SRTF est plus complexe car il faut suivre les préemptions.

§2.3 Ordonnancement par Priorités

[DEF] Définition

L'algorithme d'ordonnancement par priorités associe une priorité (généralement un entier) à chaque processus. Le CPU est alloué au processus ayant la plus haute priorité (par convention, les petites valeurs numériques représentent souvent les plus hautes priorités, comme dans Linux). L'ordonnancement par priorités peut être préemptif ou non-préemptif.

- **Préemptif :** Si un nouveau processus arrive avec une priorité plus élevée que celle du processus en cours, ce dernier est préempté.
- **Non-préemptif :** Le nouveau processus à haute priorité est simplement placé en tête de la file d'attente des prêts.

SJF peut être vu comme un cas particulier d'ordonnancement par priorités, où la priorité est l'inverse de la durée du prochain CPU burst estimé.

2.3.1 Sources des Priorités

Les priorités peuvent être définies de manière :
— **Interne :** Basées sur des critères mesurables du processus (limites de temps, besoins en mémoire, ratio d'E/S par rapport au CPU, etc.).
— **Externe :** Fixées par des critères extérieurs au système d'exploitation (importance de l'utilisateur, type de tâche, paiement pour un service, etc.).

2.3.2 Problèmes et Solutions

— **Famine (Starvation) :** Le principal problème est la famine des processus à basse priorité, qui peuvent ne jamais s'exécuter si un flux constant de processus à haute priorité arrive.
— **Solution à la famine - Vieillissement (Aging) :** Une technique consiste à augmenter progressivement la priorité des processus qui attendent depuis longtemps dans le système. Ainsi, même un processus à basse priorité finira par atteindre une priorité suffisamment élevée pour être élu.

[!] Important / Attention

L'inversion de priorité, un problème majeur dans les systèmes à priorités préemptifs, a été introduite au chapitre précédent. Elle survient lorsqu'un processus de haute priorité est bloqué par un processus de plus basse priorité détenant une ressource nécessaire. Des mécanismes comme l'héritage de priorité sont essentiels, surtout en temps réel.

§2.4 Round-Robin (RR) - Tourniquet

[DEF] Définition

L'algorithme **Round-Robin (RR)**, ou Tourniquet, est conçu spécifiquement pour les systèmes en temps partagé. Il est similaire au FCFS, mais avec l'ajout de la préemption basée sur un **quantum de temps (time quantum ou timeslice)**. Chaque processus se voit allouer une petite unité de temps CPU, appelée quantum (typiquement 10-100 ms).

Le fonctionnement est le suivant :

1. La file d'attente des prêts est traitée comme une file circulaire (FIFO).
2. L'ordonnanceur choisit le premier processus de la file, lui alloue le CPU pour une durée maximale égale au quantum.
3. Si le processus termine son CPU burst avant la fin du quantum, il libère le CPU volontairement. L'ordonnanceur passe au processus suivant.
4. Si le processus utilise tout son quantum et est toujours en cours d'exécution, il est pré-

empté, et placé à la fin de la file d'attente des prêts. L'ordonnanceur passe au processus suivant.

2.4.1 Performance et Choix du Quantum

La performance de RR dépend fortement de la taille du quantum :

- **Quantum trop grand :** RR tend vers FCFS. Si le quantum est plus grand que le plus long CPU burst, RR se comporte exactement comme FCFS.
- **Quantum trop petit :** Le nombre de commutations de contexte devient excessif, ce qui augmente la surcharge (overhead) du système et réduit l'efficacité. Par exemple, si le quantum est de 1 ms et une commutation de contexte prend 0.1 ms, alors 10% du temps CPU est perdu en commutation. Le système apparaît rapide mais est inefficace.

Un bon quantum est un compromis : il doit être suffisamment long pour que la plupart des CPU bursts interactifs se terminent en un seul quantum, mais pas trop long pour ne pas nuire à la réactivité. Une règle empirique est que 80% des CPU bursts devraient être plus courts que le quantum.

- **Temps de rotation moyen :** Généralement meilleur que FCFS, mais peut être pire que SJF.
- **Temps de réponse :** RR offre un bon temps de réponse, ce qui est crucial pour les systèmes interactifs. Chaque processus obtient une part du CPU à intervalles réguliers.

[EX] Exemple

Considérons les mêmes processus que pour FCFS : P1 (24ms), P2 (3ms), P3 (3ms), et un quantum de 4ms. Diagramme de Gantt (simplifié, sans coût de commutation) : P1 (0-4), P2 (4-7, termine), P3 (7-10, termine), P1 (10-14), P1 (14-18), P1 (18-22), P1 (22-26), P1 (26-30, termine).
Le temps d'attente et le temps de rotation peuvent être plus longs qu'avec SJF, mais la perception de réactivité est meilleure car les petits processus ne sont pas bloqués indéfiniment.

§2.5 Files d'attente multiniveaux (Multilevel Queue Scheduling)

[DEF] Définition

L'algorithme des **files d'attente multiniveaux (Multilevel Queue Scheduling)** partitionne la file d'attente des prêts en plusieurs files distinctes. Chaque file est associée à une priorité (ou à une classe de processus). Les processus sont assignés de manière permanente à une file lors de leur entrée dans le système, généralement en fonction de certaines de leurs propriétés (type, besoins en mémoire, priorité utilisateur, etc.).

Exemples de files :

— **Processus système (System processes) :** Priorité la plus élevée.
— **Processus interactifs (Interactive processes) :** Priorité moyenne.
— **Processus de traitement par lots (Batch processes) :** Priorité la plus basse.

Chaque file peut avoir son propre algorithme d'ordonnancement. Par exemple :
— File des processus système : RR ou Priorités fixes.
— File des processus interactifs : RR.
— File des processus batch : FCFS.

L'ordonnancement entre les files est également nécessaire. Typiquement, on utilise un ordonnancement par priorités fixes préemptif : aucun processus d'une file de basse priorité ne peut s'exécuter tant qu'il y a des processus dans une file de plus haute priorité. Si un processus arrive dans une file de haute priorité alors qu'un processus d'une file de basse priorité s'exécute, ce dernier est préempté.

2.5.1 Avantages et Inconvénients

— **Avantage :** Faible surcharge d'ordonnancement car les processus ne changent pas de file. Permet d'appliquer des politiques différentes à des classes de processus différentes.
— **Inconvénient :** Peu flexible. Risque de famine pour les files de basse priorité si les files de haute priorité sont constamment alimentées.

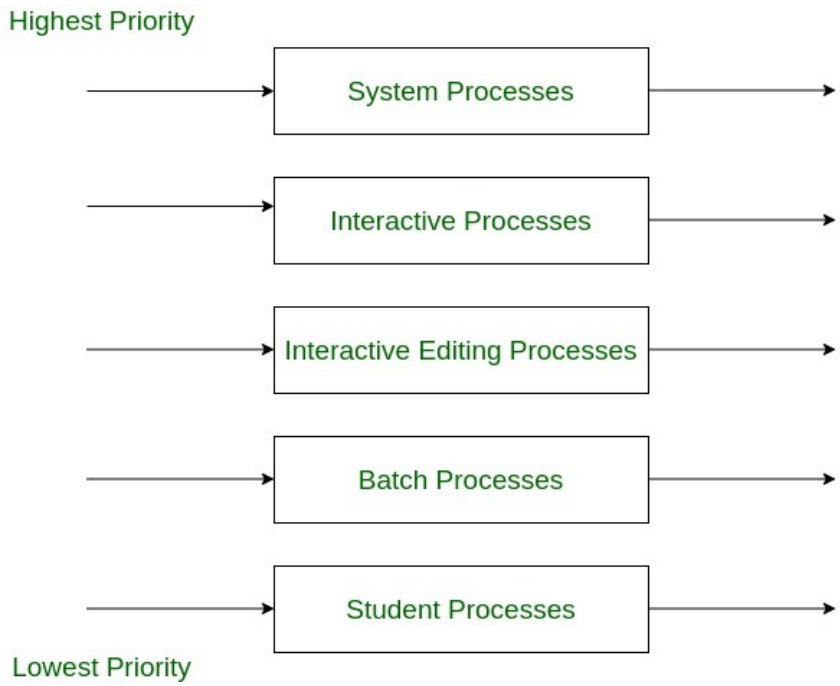

FIGURE 2.1 – Exemple de files d'attente multiniveaux.

§2.6 Files d'attente multiniveaux avec rétroaction (Multilevel Feedback Queue Scheduling)

[DEF] Définition

L'algorithme des **files d'attente multiniveaux avec rétroaction (Multilevel Feedback Queue Scheduling)** permet aux processus de migrer entre les différentes files d'attente. L'idée est de séparer les processus en fonction des caractéristiques de leurs CPU bursts. Si un processus utilise trop de temps CPU, il sera déplacé vers une file de plus basse priorité. Inversement, un processus qui attend trop longtemps dans une file de basse priorité peut être promu vers une file de plus haute priorité (mécanisme de vieillissement).

Cet algorithme est le plus général et aussi le plus complexe des algorithmes classiques. Il vise à :

- Favoriser les processus courts et interactifs (bon temps de réponse).
- Éviter la famine pour les processus longs ou de basse priorité.
- S'adapter au comportement changeant des processus.

Les paramètres définissant un ordonnanceur à files multiniveaux avec rétroaction sont :

- Le nombre de files.
- L'algorithme d'ordonnancement pour chaque file.
- La méthode utilisée pour déterminer quand promouvoir un processus vers une file de plus haute priorité.
- La méthode utilisée pour déterminer quand rétrograder un processus vers une file de plus basse priorité.
- La méthode utilisée pour déterminer dans quelle file un processus entrera initialement.

[EX] Exemple

Un exemple typique pourrait avoir trois files :

- **File 0 :** Quantum de 8 ms (e.g., RR). Si un processus ne termine pas, il passe à la File 1.
- **File 1 :** Quantum de 16 ms (e.g., RR). Si un processus ne termine pas, il passe à la File 2.
- **File 2 :** FCFS.

Un nouveau processus entre dans la File 0. Les processus dans la File 0 ont la plus haute priorité, suivis par ceux de la File 1, puis File 2. Un processus dans une file inférieure ne s'exécute que si les files supérieures sont vides. Un mécanisme de vieillissement peut ramener un processus de la File 2 à la File 1 ou 0 après un certain temps.

Cet algorithme est très configurable et peut être adapté pour obtenir un bon équilibre entre réactivité, débit et équité. De nombreux ordonnanceurs de systèmes d'exploitation modernes (y compris des versions antérieures de Unix et Linux) ont été inspirés par ce modèle.

[?] Question / Pour aller plus loin Comment les paramètres d'un ordonnanceur à files multiniveaux avec rétroaction (nombre de files, quanta, politiques de promotion/rétrogradation) influencent-ils les performances globales du système ? Quels sont les défis pour régler ces paramètres de manière optimale ?

Ce chapitre a couvert les principaux algorithmes d'ordonnancement théoriques. Ils fournissent un cadre essentiel pour comprendre les décisions de conception et les compromis rencontrés dans les ordonnanceurs réels, comme celui de Linux, que nous aborderons dans les chapitres suivants.

3

L'Ordonnancement dans le Noyau Linux

Après avoir exploré les algorithmes d'ordonnancement classiques, nous nous tournons maintenant vers l'implémentation concrète de ces concepts dans un système d'exploitation moderne et largement utilisé : Linux. L'ordonnanceur du noyau Linux est une pièce d'ingénierie logicielle complexe, en constante évolution, conçue pour offrir de bonnes performances sur une vaste gamme de charges de travail et d'architectures matérielles, des systèmes embarqués aux supercalculateurs.

Ce chapitre se concentre sur les mécanismes d'ordonnancement pour les processus normaux (non temps réel) et introduit les ordonnanceurs dédiés aux tâches temps réel, qui seront approfondis ultérieurement.

§3.1 Architecture Générale de l'Ordonnanceur Linux

L'ordonnanceur Linux est modulaire et basé sur un système de **classes d'ordonnancement (scheduling classes)**. Chaque classe d'ordonnancement implémente une politique d'ordonnancement spécifique. Lorsqu'une décision d'ordonnancement doit être prise (par exemple, lors d'un appel à la fonction `schedule()`), le noyau interroge les classes d'ordonnancement dans un ordre de priorité défini, de la plus haute à la plus basse, jusqu'à ce qu'une classe désigne une tâche à exécuter [7].

Les principales classes d'ordonnancement, par ordre de priorité décroissante, sont typiquement :

1. **Stop Class (`stop_sched_class`) :** Utilisée pour les tâches `stop_machine`, qui ont la priorité la plus élevée absolue pourarrêter les autres cœurs lors d'opérations critiques (comme la migration de tâches pour le hotplug CPU).
2. **Deadline Class (`dl_sched_class`) :** Implémente la politique `SCHED_DEADLINE` pour les tâches temps réel avec des échéances strictes.
3. **Real-Time Class (`rt_sched_class`) :** Implémente les politiques `SCHED_FIFO` et `SCHED_RR` pour les tâches temps réel traditionnelles.
4. **Fair Class (`fair_sched_class`) :** Implémente la politique `SCHED_NORMAL` (alias `SCHED_OTHER`), `SCHED_BATCH`, et `SCHED_IDLE`. C'est l'ordonnanceur par défaut, le **Completely Fair Scheduler (CFS)**.

5. **Idle Class (`idle_sched_class`) :** Utilisée pour la tâche idle de chaque CPU, qui s'exécute lorsqu'il n'y a aucune autre tâche prête.

Chaque CPU possède sa propre **file d'attente d'exécution (runqueue)**, représentée par la structure `struct rq`. Cette structure contient des sous-structures pour chaque classe d'ordonnancement (par exemple, `cfs_rq` pour CFS, `rt_rq` pour les tâches temps réel) [1]. L'utilisation de runqueues par CPU améliore la scalabilité sur les systèmes multiprocesseurs (SMP) en réduisant la contention sur les structures de données globales.

L'entité fondamentale ordonnançable est la tâche, représentée par `struct task_struct`. Cette structure contient de nombreuses informations, dont celles relatives à l'ordonnancement :

— `policy` : La politique d'ordonnancement de la tâche (e.g., `SCHED_NORMAL`, `SCHED_FIFO`).
— `prio` : Priorité statique (pour les tâches RT, `0-99`).
— `static_prio` : Priorité statique de base, influencée par la `nice value` pour les tâches CFS.
— `normal_prio` : Priorité dynamique pour les tâches CFS.
— `se` : `struct sched_entity`, entité d'ordonnancement pour CFS.
— `rt` : `struct sched_rt_entity`, entité d'ordonnancement pour les tâches RT.
— `dl` : `struct sched_dl_entity`, entité d'ordonnancement pour les tâches Deadline.
— `sched_class` : Pointeur vers la structure `struct sched_class` appropriée.

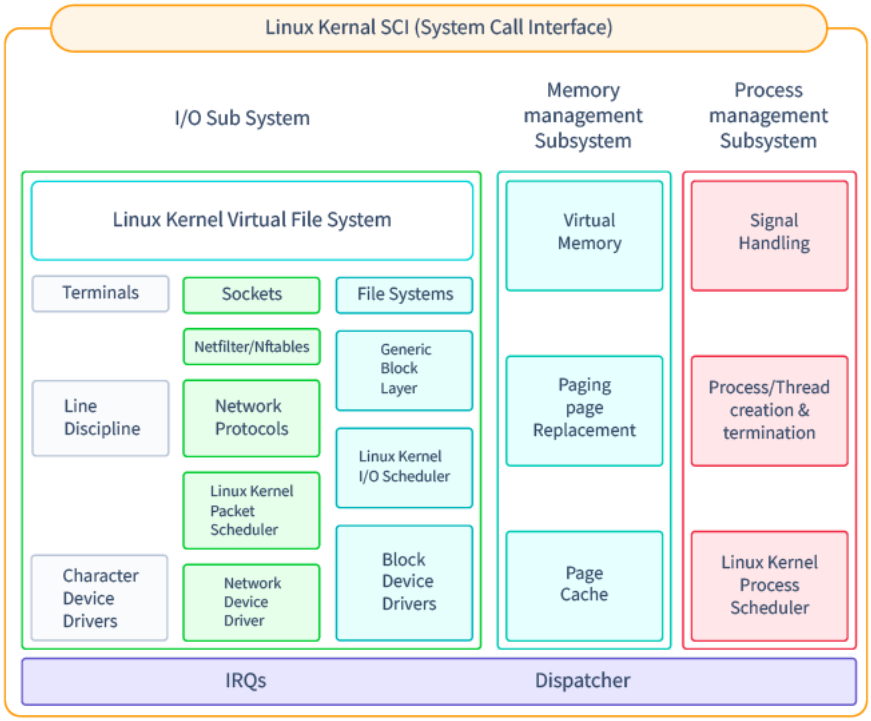

FIGURE 3.1 – Architecture générale simplifiée de l'ordonnanceur Linux.

§3.2 CFS (Completely Fair Scheduler) - Ordonnanceur par défaut

[DEF] Définition

Le **Completely Fair Scheduler (CFS)** est l'ordonnanceur par défaut pour les tâches normales (non temps réel) sous Linux, introduit dans le noyau 2.6.23 par Ingo Molnár [8]. Son objectif principal est d'assurer une **équité (fairness)** parfaite dans la distribution du temps CPU entre les tâches concurrentes. Il s'inspire des idées des ordonnanceurs à partage équitable (fair-share scheduling).

Au lieu d'utiliser des quanta de temps fixes comme l'algorithme Round-Robin classique,

CFS vise à donner à chaque tâche une part proportionnelle du processeur. L'idée est qu'un "processeur idéalement équitable" exécuterait chaque tâche en parallèle à une vitesse proportionnelle à sa part, sur un nombre infini de processeurs lents. CFS émule ce comportement sur un nombre limité de processeurs réels.

3.2.1 Virtual Runtime (`vruntime`)

La clé de CFS est le concept de **temps d'exécution virtuel (virtual runtime ou `vruntime`)**. Chaque tâche se voit associer une valeur de `vruntime` qui représente le temps CPU qu'elle a consommé, normalisé par sa part (poids). CFS choisit toujours la tâche qui a accumulé le moins de `vruntime` (celle qui a le `vruntime` le plus petit).

Le `vruntime` d'une tâche progresse au fur et à mesure de son exécution. Lorsque le `vruntime` d'une tâche augmente, elle est moins susceptible d'être choisie, laissant la place à d'autres tâches ayant un `vruntime` plus faible.

$$\Delta\text{vruntime} = \frac{\text{Temps réel exécuté}}{\text{Poids de la tâche}} \times \text{Poids total des tâches prêtes} \tag{3.1}$$

En réalité, la mise à jour du `vruntime` est plus directement liée au temps d'exécution réel, pondéré par le "poids" de la tâche, qui est dérivé de sa `nice value`. Une tâche avec un poids plus élevé (correspondant à une `nice value` plus basse, donc une plus haute priorité) verra son `vruntime` progresser plus lentement pour un même temps d'exécution réel, lui permettant ainsi de s'exécuter plus longtemps ou plus souvent.

3.2.2 Arbres Rouge-Noir

Les tâches prêtes pour CFS sur une runqueue sont stockées dans un **arbre rouge-noir (red-black tree)** ordonné par `vruntime` croissant. L'arbre rouge-noir est une structure de données auto-équilibrante qui permet des opérations d'insertion, de suppression et de recherche du minimum (la tâche avec le plus petit `vruntime`, située à l'extrême gauche de l'arbre) en temps $O(\log N)$, où N est le nombre de tâches prêtes [7].

3.2.3 Nice Values et Poids

Les utilisateurs peuvent influencer la part de CPU allouée à leurs processus via la `nice value`, un entier allant de -20 (plus haute priorité, plus grande part de CPU) à +19 (plus basse priorité, plus petite part de CPU). La valeur par défaut est 0. Chaque `nice value` est mappée à un **poids (weight)**. Une différence d'une unité dans la `nice value` correspond approximativement à une différence de 10% dans la part de CPU. Par exemple, une tâche avec `nice = -1` obtiendra environ 10% de CPU en plus qu'une tâche avec `nice = 0`.

3.2.4 Paramètres Clés de CFS

CFS est paramétré par plusieurs valeurs configurables via `/proc/sys/kernel/` ou `sysctl` :

— `sched_latency_ns` : La période cible pendant laquelle chaque tâche prête devrait s'exécuter au moins une fois. Typiquement quelques dizaines de millisecondes.

— `sched_min_granularity_ns` : La durée minimale pendant laquelle une tâche s'exécutera une fois élue, pour éviter des commutations de contexte excessives.

— `sched_wakeup_granularity_ns` : Seuil à partir duquel une tâche qui se réveille peut préempter la tâche en cours. Si le `vruntime` de la tâche qui se réveille est "suffisamment" inférieur à celui de la tâche courante (la différence est supérieure à ce seuil), la préemption a lieu.

Le quantum de temps effectif d'une tâche sous CFS n'est pas fixe mais dynamique. Il est calculé en fonction de `sched_latency_ns`, du nombre de tâches prêtes, et du poids de la tâche :

$$\text{Quantum} = \frac{\text{Poids de la tâche}}{\text{Poids total des tâches prêtes}} \times \texttt{sched\_latency\_ns} \tag{3.2}$$

Ce quantum est ensuite borné par `sched_min_granularity_ns`.

3.2.5 Group Scheduling (`cgroups` CPU)

CFS supporte le **group scheduling** via les `cgroups` (control groups) du sous-système CPU. Cela permet d'allouer des parts de CPU à des groupes de tâches (par exemple, tous les processus d'un utilisateur spécifique ou d'un service conteneurisé), et CFS assure alors l'équité entre ces groupes, puis entre les tâches au sein de chaque groupe [13].

[!] Important / Attention

Bien que CFS vise une "équité parfaite", le terme "fair" est relatif à la charge de travail et aux poids des tâches. Une tâche avec une `nice value` de -20 sera traitée de manière très "injuste" (positivement pour elle) par rapport à une tâche avec `nice` = +19. L'équité est proportionnelle aux poids.

§3.3 Ordonnanceurs Temps Réel (RT Schedulers)

Linux fournit également des politiques d'ordonnancement pour les tâches temps réel, qui ont une priorité plus élevée que les tâches CFS. Ces politiques sont cruciales pour les applications nécessitant des garanties temporelles.

3.3.1 `SCHED_FIFO` (First-In, First-Out)

[DEF] Définition

`SCHED_FIFO` est une politique d'ordonnancement temps réel simple, non préemptée par le temps (pas de quantum). Une tâche `SCHED_FIFO` s'exécute jusqu'à ce qu'elle :

— Se bloque (e.g., attente d'E/S).
— Libère volontairement le CPU (e.g., via `sched_yield()`).
— Soit préemptée par une tâche de plus haute priorité (RT ou Deadline).

Les tâches `SCHED_FIFO` de même priorité sont ordonnancées selon leur ordre d'arrivée dans la file des prêts. Une tâche `SCHED_FIFO` de priorité P s'exécutera toujours avant toute

tâche SCHED_FIFO de priorité $< P$, et avant toute tâche CFS.

3.3.2 SCHED_RR (Round-Robin)

[DEF] Définition

SCHED_RR est similaire à SCHED_FIFO, mais elle ajoute un quantum de temps. Une tâche SCHED_RR s'exécute jusqu'à ce qu'elle se bloque, libère le CPU, soit préemptée par une tâche de plus haute priorité, ou que son quantum de temps expire. Si son quantum expire, elle est placée à la fin de la liste des tâches prêtes de sa priorité.

Cela permet un partage du temps entre plusieurs tâches temps réel de même priorité.

3.3.3 Priorités Temps Réel

Les tâches SCHED_FIFO et SCHED_RR ont une priorité statique allant de 1 (la plus basse priorité RT) à 99 (la plus haute priorité RT). Ces priorités sont distinctes et supérieures aux priorités des tâches normales gérées par CFS [7].

[!] Important / Attention

Une mauvaise utilisation des politiques temps réel (par exemple, une tâche SCHED_FIFO avec une haute priorité qui boucle indéfiniment sans se bloquer) peut complètement geler le système pour les tâches de plus basse priorité, y compris les tâches système essentielles. Il faut les utiliser avec une extrême prudence. Le noyau Linux implémente un mécanisme de "RT Throttling" pour éviter qu'un groupe de tâches RT ne monopolise 100% du CPU indéfiniment, en leur réservant par défaut 95% du temps CPU (configurable via `sched_rt_runtime_us` et `sched_rt_period_us`).

§3.4 Ordonnanceur Deadline (SCHED_DEADLINE)

[DEF] Définition

Introduit dans le noyau Linux 3.14, SCHED_DEADLINE est une politique d'ordonnancement temps réel basée sur l'algorithme **Earliest Deadline First (EDF)** conjointement avec un mécanisme de **Constant Bandwidth Server (CBS)**. Chaque tâche SCHED_DEADLINE est caractérisée par trois paramètres :

- Runtime (Q_i) : Le temps CPU maximal dont la tâche a besoin.
- Period (P_i) : La période à laquelle la tâche est activée.
- Deadline (D_i) : L'échéance absolue par rapport au début de la période, avant laquelle la tâche doit terminer son Runtime. Typiquement $D_i \leq P_i$.

L'ordonnanceur choisit la tâche dont l'échéance absolue est la plus proche. Le CBS assure que chaque tâche ne consomme pas plus que sa bande passante réservée (Q_i/P_i), évitant ainsi qu'une tâche ne perturbe les autres en cas de dépassement de son budget.

`SCHED_DEADLINE` offre des garanties d'ordonnançabilité plus fortes que `SCHED_FIFO/RR`, en particulier pour les systèmes avec des tâches apériodiques ou sporadiques, à condition qu'un **contrôle d'admission** soit effectué pour s'assurer que la charge totale demandée ne dépasse pas la capacité du processeur ($\sum(Q_i/P_i) \leq 1$ pour un monoprocesseur, ou $\leq M$ pour M processeurs avec des mécanismes de partitionnement ou de migration appropriés).

`SCHED_DEADLINE` a la plus haute priorité parmi les politiques d'ordonnancement (avant RT et CFS) [13].

[?] Question / Pour aller plus loin Comment le mécanisme de "bandwidth reclaiming" (récupération de bande passante inutilisée) dans `SCHED_DEADLINE` ou des mécanismes similaires dans d'autres ordonnanceurs temps réel peuvent-ils améliorer l'utilisation du CPU tout en maintenant les garanties temporelles ?

§3.5 Ordonnanceurs Idle (`SCHED_IDLE`) et Batch (`SCHED_BATCH`)

Linux fournit deux autres politiques gérées par la classe `fair_sched_class` (CFS) mais avec des comportements spécifiques :

- `SCHED_BATCH` **:** Conçu pour les tâches non interactives, gourmandes en CPU (CPU-bound) où le débit est plus important que la latence. Les tâches `SCHED_BATCH` sont traitées comme des tâches CFS avec une `nice value` de +19 par défaut, mais elles sont moins susceptibles de préempter d'autres tâches et sont moins souvent préemptées elles-mêmes, ce qui réduit les commutations de contexte et peut améliorer l'utilisation des caches.
- `SCHED_IDLE` **:** Représente la priorité la plus basse possible, encore plus basse que `nice = +19`. Les tâches `SCHED_IDLE` ne s'exécutent que lorsque le CPU est autrement inactif. Utile pour des travaux de fond de très basse priorité qui ne doivent absolument pas impacter les autres activités du système.

§3.6 Interaction avec les Interruptions et les Softirqs/Tasklets

Les décisions d'ordonnancement peuvent être déclenchées par divers événements, notamment les interruptions.

- **Interruptions Matérielles (Hard IRQs) :** Elles préemptent toute tâche en cours (y

compris RT ou Deadline) et s'exécutent immédiatement. Leurs handlers (ISRs) doivent être aussi courts que possible.
- **Interruptions Logicielles (Softirqs, Tasklets, Timers) :** Ce sont des travaux différés déclenchés par les hard IRQs ou le noyau. Ils s'exécutent dans un contexte d'interruption mais avec les interruptions matérielles réactivées. Ils ont une priorité plus élevée que les tâches ordonnancées, y compris `SCHED_DEADLINE`.

Une exécution prolongée dans les handlers d'interruptions (hard ou soft) peut introduire des latences significatives et impacter la prévisibilité de l'ordonnancement, un défi majeur pour les systèmes temps réel (voir Chapitre 4).

§3.7 Préemption dans le Noyau Linux

La capacité du noyau à préempter une tâche s'exécutant en mode noyau est cruciale pour la réactivité. Linux offre plusieurs niveaux de préemption du noyau, configurables à la compilation :
- `CONFIG_PREEMPT_NONE` **:** Pas de préemption du noyau. Une tâche en mode noyau s'exécute jusqu'à ce qu'elle se bloque, retourne en mode utilisateur, ou appelle explicitement `schedule()`. Offre le meilleur débit mais la pire latence.
- `CONFIG_PREEMPT_VOLUNTARY` **:** Le noyau contient des points de préemption explicites (`cond_resched()`) à des endroits stratégiques où il est sûr de préempter. Réduit la latence par rapport à `NONE` sans trop impacter le débit.
- `CONFIG_PREEMPT` **(ou** `CONFIG_PREEMPT_DESKTOP`**) :** Préemption complète du noyau. Une tâche en mode noyau peut être préemptée à tout moment (sauf si elle détient un spinlock ou est dans une section non-préemptible), dès qu'une tâche de plus haute priorité se réveille. Optimisé pour la latence et la réactivité des applications de bureau.
- `CONFIG_PREEMPT_RT` **(anciennement** `PREEMPT_RT_FULL`**) :** Le patch "Real-Time Preemption" qui transforme la plupart des spinlocks en mutex préemptibles, permet aux handlers d'interruptions de s'exécuter dans des threads, etc., pour un déterminisme accru (voir Chapitre 4).

Le choix du modèle de préemption est un compromis entre débit, latence, et complexité.

[TP] Exercice / TP

Module 3 - Exploration de l'Ordonnanceur Linux

- Observer l'effet des `nice values` sur des processus concurrents (e.g., deux boucles `while true; do :; done` lancées avec des `nice` différentes).

[CODE] Bloc de Code

```
# Lancer dans deux terminaux diff rents
nice -n 0 while true; do :; done &
nice -n 10 while true; do :; done &
# Observer l'utilisation CPU avec top ou htop

```

Listing 3.1 – Lancer des tâches avec nice

— Utiliser `chrt` pour changer la politique et la priorité de processus (e.g., passer une tâche en `SCHED_RR` avec une priorité RT).

[CODE] Bloc de Code

```
# Lancer une t che en SCHED_RR, priorit  50
sudo chrt -r -p 50 $$ # $$ est le PID du shell
    courant
# Attention :    utiliser avec pr caution !
# Pour une commande sp cifique :
# sudo chrt -r -p 50 votre_commande

```

Listing 3.2 – Utilisation de chrt

— Explorer `/proc/[pid]/sched` pour voir les statistiques d'ordonnancement d'un processus (e.g., `se.vruntime`, `nr_switches`).
— Explorer `/proc/sched_debug` pour une vue globale de l'état des runqueues et des paramètres de l'ordonnanceur.

[CODE] Bloc de Code

```
cat /proc/self/sched
cat /proc/sched_debug

```

Listing 3.3 – Explorer /proc

— (Optionnel Avancé) Introduction à `cgroups` (v1 ou v2) pour la gestion des parts CPU. Créer un cgroup, lui assigner des parts CPU (e.g., `cpu.shares`), et y déplacer des processus.

Ce chapitre a fourni un aperçu des principaux mécanismes d'ordonnancement du noyau Linux. Les politiques temps réel, en particulier `SCHED_DEADLINE` et les défis liés à `PREEMPT_RT`, seront examinés plus en détail dans le prochain chapitre.

4

Systèmes et Ordonnancement Temps Réel

Les systèmes d'exploitation généralistes, comme Linux dans sa configuration standard, sont optimisés pour l'équité et le débit moyen. Cependant, de nombreuses applications, allant du contrôle industriel à la robotique, en passant par les télécommunications et le multimédia, exigent des garanties temporelles strictes. Pour ces **systèmes temps réel (Real-Time Systems - RTS)**, la correction du système ne dépend pas seulement du résultat logique du calcul, mais aussi du moment où ce résultat est produit.

Ce chapitre explore les concepts fondamentaux des systèmes temps réel, les défis que pose leur implémentation sur un OS comme Linux, les solutions apportées (notamment le patch `PREEMPT_RT`), et les techniques d'analyse de l'ordonnançabilité.

§4.1 Définitions et Concepts du Temps Réel

[DEF] Définition

Un **système temps réel** est un système informatique dont le bon fonctionnement dépend de sa capacité à respecter des contraintes temporelles, souvent appelées **échéances (deadlines)**. Une défaillance à respecter une échéance peut entraîner des conséquences allant d'une simple dégradation de la qualité de service à des issues catastrophiques (par exemple, dans un système de contrôle de vol).

On distingue classiquement plusieurs catégories de systèmes temps réel en fonction de la criticité du respect des échéances [2] :

- **Systèmes temps réel stricts (Hard Real-Time Systems - HRTS) :** Le non-respect d'une seule échéance est considéré comme une défaillance du système. Les conséquences peuvent être graves. Exemples : systèmes de contrôle de processus critiques (centrales nucléaires, usines chimiques), avionique, systèmes d'armes.
- **Systèmes temps réel souples (Soft Real-Time Systems - SRTS) :** Le non-respect occasionnel d'une échéance est tolérable et conduit à une dégradation de la performance ou de la qualité de service, mais pas à une défaillance complète du système. L'utilité d'un résultat diminue après son échéance. Exemples : streaming vidéo/audio,

jeux en ligne, systèmes de transaction.
- **Systèmes temps réel fermes (Firm Real-Time Systems) :** Un résultat produit après son échéance est inutile (valeur nulle), mais le système ne subit pas de défaillance catastrophique. C'est un cas intermédiaire entre strict et souple. Exemple : systèmes de prédiction où une information tardive n'a plus de valeur.

Les principales contraintes temporelles dans les RTS sont :
- **Échéance (Deadline) :** Instant auquel une tâche doit avoir terminé son exécution.
- **Gigue (Jitter) :** Variation de la latence d'un événement ou de la période d'exécution d'une tâche. Une faible gigue est souvent souhaitable pour la régularité.
- **Latence (Latency) :** Temps écoulé entre un événement (stimulus) et la réponse du système.

Un concept clé est le **déterminisme temporel** : la capacité du système à prédire et garantir le comportement temporel de ses opérations. Un système déterministe aura des latences bornées et prévisibles.

§4.2 Sources de Latence dans un OS Généraliste comme Linux

Un système d'exploitation généraliste comme Linux, même avec ses politiques d'ordonnancement temps réel `SCHED_FIFO/RR/DEADLINE`, présente intrinsèquement plusieurs sources de latence qui peuvent compromettre le déterminisme requis par les applications temps réel strictes [7] :
- **Interruptions Matérielles (Hard IRQs) :** Les handlers d'interruptions (ISRs) s'exécutent avec une priorité supérieure à toute tâche, y compris les tâches temps réel. Si un ISR est long, il retarde l'exécution des tâches RT.
- **Interruptions Logicielles (Softirqs, Tasklets, Timers) :** Ces travaux différés s'exécutent également à haute priorité (généralement après les hard IRQs mais avant les tâches ordonnancées). De longues chaînes de softirqs peuvent introduire des latences importantes.
- **Sections Critiques Non-Préemptibles dans le Noyau :** Certaines parties du code noyau sont protégées par des spinlocks ou des sections où la préemption est explicitement désactivée (`preempt_disable()`). Si une tâche RT se réveille alors que le CPU est dans une telle section (exécutée par une tâche de plus basse priorité ou dans un contexte d'interruption), elle doit attendre la fin de cette section.
- **Synchronisation (Spinlocks, Mutex) :**
  - **Spinlocks :** Conçus pour des sections critiques très courtes. Si une tâche RT essaie d'acquérir un spinlock détenu par une tâche de basse priorité (sur un système monoprocesseur ou si la tâche de basse priorité est sur un autre CPU mais tarde à le libérer), la tâche RT "spins" (boucle activement), gaspillant du temps CPU.
  - **Mutex et Sémaphores :** Peuvent mener à une **inversion de priorité** si une tâche de basse priorité détient un verrou nécessaire à une tâche de haute priorité, et qu'une tâche de priorité intermédiaire préempte la tâche de basse priorité.
- **Gestion de la Mémoire :** Les défauts de page, la pagination (swapping), ou les opéra-

tions du TLB (Translation Lookaside Buffer) peuvent introduire des latences non bornées.
— **Pilotes de Périphériques (Device Drivers) :** Des pilotes mal écrits ou effectuant de longues opérations avec les interruptions désactivées peuvent être une source majeure de latence.

[!] Important / Attention

La somme de ces sources de latence, même si chacune est petite, peut s'accumuler et rendre difficile, voire impossible, de garantir des échéances strictes pour les applications temps réel sur un noyau Linux standard. La latence maximale ("worst-case latency") est le principal indicateur pour les systèmes HRTS.

§4.3 Le Patch `PREEMPT_RT` (Real-Time Preemption)

Pour transformer Linux en un système d'exploitation capable de répondre aux exigences des systèmes temps réel stricts (ou quasi-stricts), le projet `PREEMPT_RT` (anciennement connu sous le nom de "patch temps réel") a été développé [9]. Il s'agit d'un ensemble de modifications profondes du noyau visant à minimiser les sources de latence non déterministes. Depuis plusieurs années, une grande partie de ce patch a été progressivement intégrée dans le noyau Linux principal ("mainline"), mais certaines fonctionnalités restent en dehors ou nécessitent une configuration spécifique (`CONFIG_PREEMPT_RT`).

Les principales modifications apportées par `PREEMPT_RT` incluent :

— **Préemption quasi-totale du noyau :** La plupart des sections du noyau qui étaient non-préemptibles deviennent préemptibles. Ceci est réalisé en rendant la plupart des spinlocks préemptibles.
— **Spinlocks transformés en Mutex préemptibles (PI-Mutex) :** Les spinlocks (`raw_spinlock_t` restant pour les cas critiques) sont remplacés par des mutex qui supportent l'héritage de priorité (`rtmutex`). Cela permet à une tâche de haute priorité d'être bloquée sur un mutex au lieu de "spinner", et si le mutex est détenu par une tâche de plus basse priorité, cette dernière hérite temporairement de la priorité de la tâche bloquée pour accélérer la libération du verrou.
— **Handlers d'Interruptions exécutés dans des Threads Noyau (Threaded IRQs) :** La majorité du code des handlers d'interruptions matérielles est déplacée dans des threads noyau dédiés, qui sont ordonnancés comme des tâches normales (avec des priorités RT élevées). Seule une partie minimale critique du handler s'exécute encore dans le contexte d'interruption "dur". Cela permet aux tâches utilisateur RT de préempter même les handlers d'IRQ (leurs threads).
— **High-Resolution Timers (HRT) :** Amélioration de la granularité et de la précision des temporisateurs du noyau, essentiels pour les applications temps réel. (Cette partie est largement mainline maintenant).
— **Réduction des sections `local_irq_disable()` :** Remplacement par des mécanismes de verrouillage plus fins lorsque c'est possible.

[DEF] Définition

L'objectif de `PREEMPT_RT` est de rendre les latences du noyau bornées et aussi petites que possible, afin que Linux puisse être utilisé dans des scénarios HRTS où il n'était traditionnellement pas adapté.

La configuration du noyau avec `CONFIG_PREEMPT_RT` (anciennement `CONFIG_PREEMPT_RT_FULL`) active l'ensemble de ces fonctionnalités. Il y a un coût en termes de débit et de complexité, mais pour les applications RT, les gains en déterminisme sont primordiaux.

§4.4 Défis de l'Ordonnancement Temps Réel

Même avec `PREEMPT_RT`, plusieurs défis subsistent pour l'ordonnancement temps réel.

• 4.4.1 Inversion de Priorité

L'inversion de priorité, comme mentionné précédemment, est un problème critique.

- **Problème :** Une tâche H (haute priorité) attend une ressource R détenue par une tâche L (basse priorité). Une tâche M (moyenne priorité, $L < M < H$) devient prête et préempte L, empêchant L de libérer R et prolongeant le blocage de H.
- **Solutions classiques :**
  - **Héritage de Priorité (Priority Inheritance - PI) :** Si H bloque sur R détenue par L, L hérite temporairement de la priorité de H jusqu'à ce qu'elle libère R. C'est ce qu'implémentent les `rtmutex` de Linux.
  - **Plafond de Priorité (Priority Ceiling Protocol - PCP) :** Chaque ressource partagée se voit assigner un "plafond de priorité", qui est la priorité la plus élevée de toutes les tâches susceptibles d'utiliser cette ressource. Une tâche ne peut acquérir une ressource que si sa priorité est strictement supérieure aux plafonds de toutes les ressources actuellement détenues par d'autres tâches. Cela prévient les inversions de priorité et les interblocages, mais est plus complexe à implémenter. Linux n'implémente pas PCP de manière standard pour les mutex utilisateurs, mais le concept existe pour certains mécanismes noyau.

• 4.4.2 Analyse de l'Ordonnançabilité (Schedulability Analysis)

[DEF] Définition

L'analyse de l'ordonnançabilité est le processus qui consiste à déterminer mathématiquement si un ensemble donné de tâches temps réel peut respecter toutes ses échéances sous une politique d'ordonnancement donnée et sur une architecture matérielle spécifique.

Pour les systèmes HRTS, cette analyse est cruciale. Quelques tests classiques pour des

tâches périodiques indépendantes sur un monoprocesseur :

— **Rate Monotonic Scheduling (RMS) :** Une politique à priorités fixes où les tâches avec des périodes plus courtes ont des priorités plus élevées. Le test de Liu & Layland (condition suffisante mais pas nécessaire en général) stipule qu'un ensemble de n tâches est ordonnançable si l'utilisation du CPU $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ (où C_i est le temps d'exécution pire-cas et T_i est la période) est inférieure ou égale à $n(2^{1/n} - 1)$ [6].

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{4.1}$$

Pour $n \to \infty$, cette limite tend vers $\ln(2) \approx 0.693$. Un test exact pour RMS est l'analyse du temps de réponse (Response Time Analysis - RTA).

— **Earliest Deadline First (EDF) :** Une politique à priorités dynamiques où la tâche avec l'échéance absolue la plus proche a la plus haute priorité. EDF est optimal pour les systèmes monoprocesseurs. Un ensemble de tâches périodiques est ordonnançable par EDF si et seulement si l'utilisation du CPU $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ est inférieure ou égale à 1 [6].

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{4.2}$$

C'est la base de `SCHED_DEADLINE` sous Linux.

L'analyse pour les systèmes multiprocesseurs, avec des tâches dépendantes, ou des tâches apériodiques/sporadiques est beaucoup plus complexe [2].

4.4.3 Gestion de la Surcharge (Overload Management)

Que se passe-t-il si la condition d'ordonnançabilité n'est pas respectée (surcharge) ?

— Pour RMS, certaines tâches (typiquement celles de plus basse priorité) manqueront leurs échéances.

— Pour EDF, en cas de surcharge, le comportement peut devenir chaotique (effet domino où de nombreuses tâches manquent leurs échéances). Des mécanismes comme le CBS dans `SCHED_DEADLINE` sont conçus pour isoler les tâches et fournir une dégradation contrôlée.

§4.5 Comparaison : Linux + `PREEMPT_RT` vs. RTOS dédiés

Pendant longtemps, pour les applications HRTS, les développeurs se tournaient vers des **RTOS (Real-Time Operating Systems)** dédiés comme VxWorks, QNX, FreeRTOS, Zephyr, etc.

TABLE 4.1 – Comparaison Linux + PREEMPT_RT vs. RTOS dédiés.

Caractéristique	**Linux +** PREEMPT_RT	**RTOS Dédié (e.g., QNX, VxWorks)**
Déterminisme / Latence	Bon à très bon (microsecondes), en amélioration constante. Peut encore avoir des "outliers".	Excellent, souvent garanti (nanosecondes à microsecondes). Conçu pour le temps réel dès le départ.
Empreinte mémoire / Ressources	Plus importante (noyau complet).	Très petite (micro-noyau ou noyau configurable). Adapté aux systèmes embarqués contraints.
Écosystème / Fonctionnalités	Immense (pilotes, bibliothèques, outils, communauté). OS complet.	Plus limité, mais focalisé sur les besoins RT. Peut nécessiter plus de développement spécifique.
Coût (Licence)	Open Source (gratuit).	Souvent propriétaire et coûteux (pour les plus performants). Certains sont open source (FreeRTOS, Zephyr).
Complexité	Élevée. Configurer et valider un système RT peut être complexe.	Variable, souvent plus simple pour les fonctionnalités de base RT.
Certification	Plus difficile à certifier pour des normes de sécurité critiques (e.g., DO-178C).	Certains RTOS sont conçus pour la certification et l'offrent.

[i] Note

Le choix entre Linux + PREEMPT_RT et un RTOS dédié dépend fortement des exigences spécifiques de l'application (criticité, contraintes de ressources, besoin d'un OS complet vs. focalisé RT, budget, expertise disponible). Linux + PREEMPT_RT gagne en popularité pour des systèmes "mixed-criticality" où des charges de travail RT et non-RT coexistent.

[TP] Exercice / TP

Module 4 - Expérimentation Temps Réel

— **Mesurer les latences avec** `cyclictest` **:** C'est l'outil standard pour mesurer les latences d'ordonnancement sur un système Linux, en particulier avec PREEMPT_RT.

[CODE] Bloc de Code

```
# Installer rt-tests (qui contient cyclictest)
# sudo apt install rt-tests (Debian/Ubuntu)
# sudo dnf install rt-tests (Fedora)

# Ex cuter cyclictest avec une haute priorit , sur tous les c urs ,
# intervalle de 200us, pendant 5 minutes.
sudo cyclictest -a -t -p90 -i200 -m -n -d300
# -a: tous les coeurs
# -t: un thread par coeur
# -p90: priorit  RT 90
# -i200: intervalle de base de 200 microsecondes
# -m: verrouiller la m moire pour  viter  les d fauts de page
# -n: utiliser clock_nanosleep pour plus de pr cision
# -d300: ex cuter pendant 300 secondes (5 minutes)
# Interpr ter les r sultats: Min, Avg, Max latences.
# L'objectif est d'avoir une latence Max aussi faible que possible.

```

Listing 4.1 – Utilisation de cyclictest

— **(Théorique/Si environnement `PREEMPT_RT` disponible) Observer l'inversion de priorité et l'effet de l'héritage de priorité :** Cela nécessite généralement un code C spécifique simulant le scénario (tâche H, M, L et un mutex).

— **Programmer des tâches avec `SCHED_FIFO` ou `SCHED_DEADLINE` :** Écrire un petit programme C qui crée des threads et leur assigne ces politiques via `pthread_setschedparam()` ou `sched_setattr()`. Observer leur comportement face à des tâches CFS.

[CODE] Bloc de Code

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

void *thread_func(void *arg) {
    // Boucle consommant du CPU
    while(1) { /* ... travail ... */ } // Placeholder pour le travail r el
    return NULL;
}

int main() {
    pthread_t tid;
    struct sched_param param;
    int policy = SCHED_FIFO; // Ou SCHED_RR
    int ret;

    // Cr er le thread
    pthread_create(&tid, NULL, thread_func, NULL);

    // D finir la politique et la priorit
    // Utiliser sched_get_priority_max est une bonne pratique
    param.sched_priority = sched_get_priority_max(policy) - 10; // Ex: prio 89 pour SCHED_FIFO
    ret = pthread_setschedparam(tid, policy,  m );
    if (ret != 0) {
        perror("pthread_setschedparam");
        // N cessite des privil ges root ou CAP_SYS_NICE
    }

    pthread_join(tid, NULL); // Attendre la fin du thread (ne se terminera jamais ici)
    return 0;
}
```

Listing 4.2 – Exemple simple SCHED$_FIFO(conceptuel)$

Ce chapitre a mis en lumière les aspects critiques de l'ordonnancement temps réel. Le prochain module s'éloignera des contraintes d'un seul nœud pour explorer les défis de l'ordonnancement dans les vastes étendues des systèmes distribués.

5

Ordonnancement dans les Systèmes Distribués

Jusqu'à présent, notre étude de l'ordonnancement s'est concentrée sur les mécanismes au sein d'un unique système d'exploitation, gérant les processus sur un ou plusieurs cœurs d'un même nœud. Cependant, de nombreuses applications modernes, des calculs scientifiques à haute performance (HPC) aux services web à grande échelle et aux plateformes cloud, s'exécutent sur des **systèmes distribués**. Dans ce contexte, l'ordonnancement prend une dimension supplémentaire : il ne s'agit plus seulement de choisir quelle tâche exécuter sur un CPU local, mais aussi de décider *sur quel nœud* du système distribué une tâche (ou un ensemble de tâches) devrait s'exécuter.

Ce chapitre introduit les concepts, les défis et les approches de l'ordonnancement dans les systèmes distribués. Il est important de noter que l'ordonnanceur local de chaque nœud (comme CFS ou les ordonnanceurs RT de Linux) joue toujours un rôle crucial, mais il est souvent subordonné à un ordonnanceur distribué de plus haut niveau.

§5.1 Introduction aux Systèmes Distribués

[DEF] Définition

Un **système distribué** est un ensemble d'ordinateurs autonomes (nœuds) connectés par un réseau et équipés d'un logiciel de coordination qui permet aux machines de communiquer et de collaborer pour atteindre un objectif commun. Pour l'utilisateur, le système distribué peut apparaître comme une seule et unique entité cohérente.

Les caractéristiques fondamentales des systèmes distribués qui impactent l'ordonnancement incluent [3] :

- **Absence d'horloge globale et de temps global précis :** La synchronisation des horloges entre les nœuds est un défi, rendant difficile la détermination d'un ordre global strict des événements.
- **Absence de mémoire partagée (généralement) :** Chaque nœud possède sa propre

mémoire locale. La communication et le partage de données se font par passage de messages sur le réseau.
- **Concurrence :** Plusieurs processus s'exécutent simultanément sur différents nœuds.
- **Pannes partielles :** Un ou plusieurs nœuds, ou des liens réseau, peuvent tomber en panne indépendamment des autres, tandis que le reste du système continue de fonctionner. La tolérance aux pannes est une préoccupation majeure.
- **Hétérogénéité :** Les nœuds peuvent avoir des capacités de calcul, de mémoire, de stockage ou des architectures différentes.
- **Scalabilité :** La capacité du système à maintenir ses performances lorsque le nombre de nœuds et d'utilisateurs augmente.

Les **objectifs de l'ordonnancement distribué** vont au-delà de ceux de l'ordonnancement local et incluent typiquement :
- **Répartition de charge (Load Balancing/Sharing) :** Distribuer les tâches de manière équilibrée entre les nœuds pour éviter la surcharge de certains et la sous-utilisation d'autres, optimisant ainsi l'utilisation globale des ressources.
- **Parallélisme et Accélération (Speedup) :** Exécuter les différentes parties d'une application ou différentes tâches en parallèle sur plusieurs nœuds pour réduire le temps total d'exécution.
- **Tolérance aux pannes (Fault Tolerance) :** Assurer la continuité du service ou la complétion des tâches même en cas de défaillance de certains nœuds, par exemple en migrant ou en répliquant les tâches.
- **Localité des données (Data Locality) :** Placer les tâches au plus près des données qu'elles manipulent pour minimiser les transferts réseau coûteux.
- **Minimisation des coûts de communication :** Réduire le volume et la latence des échanges de messages entre les nœuds.
- **Respect des contraintes utilisateur :** Telles que les échéances, les budgets, les besoins en ressources spécifiques.

§5.2 Taxonomie des Ordonnanceurs Distribués

Les ordonnanceurs distribués peuvent être classifiés selon plusieurs dimensions [11] (adapté) :
- **Degré de centralisation :**
  - **Centralisé :** Un seul nœud (le coordinateur ou maître) prend toutes les décisions d'ordonnancement. Simple à implémenter, mais constitue un point unique de défaillance (Single Point Of Failure - SPOF) et un goulot d'étranglement potentiel.
  - **Décentralisé (ou entièrement distribué) :** La logique d'ordonnancement est répartie entre tous les nœuds. Chaque nœud prend des décisions basées sur des informations locales et/ou échangées avec ses voisins. Plus robuste et scalable, mais plus complexe à concevoir (problèmes de consensus, d'information partielle).
  - **Hiérarchique :** Une structure d'ordonnancement à plusieurs niveaux, combinant des aspects centralisés et décentralisés. Par exemple, des coordinateurs régionaux gérant des groupes de nœuds.
- **Nature de la décision :**

  - **Statique :** Les décisions d'affectation des tâches aux nœuds sont prises avant l'exécution et ne changent pas. Adapté si les caractéristiques des tâches et du système sont bien connues et stables.
  - **Dynamique :** Les décisions sont prises ou ajustées pendant l'exécution, en fonction de l'état actuel du système (charge, disponibilité des nœuds). Plus adaptable mais plus complexe et introduit une surcharge.
- **Optimalité :**
  - **Optimal :** Cherche à trouver la meilleure solution possible selon un critère donné. Souvent irréalisable en pratique en raison de la complexité NP-difficile du problème d'ordonnancement distribué général.
  - **Sous-optimal (ou Heuristique) :** Utilise des règles ou des approximations pour trouver une "bonne" solution en un temps raisonnable, sans garantir l'optimalité. C'est l'approche la plus courante.
- **Mode de coopération :**
  - **Coopératif :** Les nœuds collaborent pour atteindre un objectif système global (par exemple, minimiser le temps de réponse moyen pour toutes les tâches).
  - **Non-coopératif (ou compétitif) :** Chaque nœud (ou utilisateur) essaie d'optimiser ses propres objectifs, potentiellement au détriment des autres. Relève de la théorie des jeux.

§5.3 Migration de Tâches et Répartition de Charge (Load Balancing)

La **répartition de charge** est un aspect fondamental de l'ordonnancement distribué. Elle vise à transférer du travail des nœuds surchargés vers des nœuds sous-utilisés. Cela implique souvent la **migration de tâches**, c'est-à-dire le déplacement d'un processus ou d'une tâche d'un nœud à un autre pendant son exécution.

[DEF] Définition

La **migration de tâches** peut être **préemptive** (la tâche est suspendue, son état est transféré, puis elle est reprise sur le nœud cible) ou **non-préemptive** (une tâche est affectée à un nœud au démarrage et y reste, ou seule la décision de lancer de nouvelles tâches sur des nœuds moins chargés est prise). La migration préemptive est plus flexible mais aussi plus coûteuse (transfert de l'état, de l'espace mémoire).

Les politiques de répartition de charge doivent répondre à plusieurs questions :

- **Politique d'information :** Comment et quand collecter les informations sur la charge des nœuds ?
- **Politique de déclenchement :** Quand initier une tentative de répartition de charge ? (périodiquement, sur seuil de charge, etc.)
- **Politique de sélection du nœud :** Comment choisir le nœud source (surchargé) et le nœud cible (sous-utilisé) ?

— **Politique de sélection de la tâche :** Quelle(s) tâche(s) migrer ? (taille, temps restant estimé, dépendances, etc.)

Quelques exemples d'algorithmes de répartition de charge dynamique :

— **Initié par l'émetteur (Sender-initiated) :** Un nœud surchargé cherche activement un nœud sous-utilisé pour lui transférer une tâche.
— **Initié par le récepteur (Receiver-initiated) :** Un nœud sous-utilisé annonce sa disponibilité ou demande du travail à des nœuds potentiellement surchargés.
— **Symétrique :** Combine les deux approches.

[i] Note

Sous Linux, au niveau d'un seul nœud multiprocesseur (SMP) ou NUMA, les "scheduler domains" (domaines d'ordonnancement) implémentent des mécanismes de répartition de charge entre les CPUs ou les nœuds NUMA. Ceci peut être vu comme une forme locale et spécialisée de répartition de charge, où l'ordonnanceur essaie de maintenir les cœurs équitablement chargés tout en considérant la localité mémoire (NUMA-awareness) [7].

§5.4 Ordonnancement pour Applications Parallèles (HPC)

Dans le domaine du Calcul Haute Performance (HPC), les applications sont souvent des programmes parallèles massifs (e.g., utilisant MPI - Message Passing Interface) qui s'exécutent sur des clusters de centaines ou milliers de nœuds.

Les **ordonnanceurs de batch (Batch Schedulers)** sont couramment utilisés dans cet environnement. Exemples :

— **Slurm (Simple Linux Utility for Resource Management)** : Un ordonnanceur open-source populaire et puissant.
— **PBS (Portable Batch System)** et ses dérivés (Torque, OpenPBS).
— **LSF (Load Sharing Facility)** : Une solution commerciale.

Ces systèmes gèrent des files d'attente de "jobs" soumis par les utilisateurs. Un job spécifie les ressources nécessaires (nombre de nœuds, cœurs par nœud, mémoire, temps d'exécution estimé). L'ordonnanceur décide quand et sur quelles ressources allouer chaque job, en fonction des politiques du site (priorités des utilisateurs, partage équitable, réservations).

Des stratégies d'ordonnancement spécifiques aux applications parallèles incluent :

— **Gang Scheduling (ou Co-scheduling) :** Vise à exécuter tous les processus/threads d'une même application parallèle simultanément sur différents processeurs. Si un processus du "gang" est bloqué, les autres peuvent aussi être suspendus pour éviter le gaspillage de ressources et les problèmes de synchronisation.
— **Backfilling :** Une technique pour améliorer l'utilisation des ressources. Si un job volumineux est en attente mais ne peut pas démarrer immédiatement, l'ordonnanceur peut autoriser des jobs plus petits (qui peuvent se terminer avant que le gros job ne soit prêt à démarrer) à s'exécuter dans les "trous" du planning.

§5.5 Ordonnancement dans les Architectures Cloud et Conteneurisées

L'avènement du cloud computing et des technologies de conteneurisation (comme Docker) a introduit de nouveaux défis et solutions pour l'ordonnancement distribué.

[DEF] Définition

Les **orchestrateurs de conteneurs** sont des systèmes qui automatisent le déploiement, la mise à l'échelle, la gestion et la mise en réseau des applications conteneurisées sur un cluster de machines.

Exemples d'orchestrateurs et leurs ordonnanceurs :

- **Kubernetes :** L'orchestrateur open-source le plus populaire.
  - Son composant d'ordonnancement principal est `kube-scheduler`.
  - `kube-scheduler` décide sur quel nœud du cluster un **Pod** (la plus petite unité déployable, contenant un ou plusieurs conteneurs) doit s'exécuter.
  - Les décisions sont basées sur les demandes de ressources du Pod (CPU, mémoire), les ressources disponibles sur les nœuds, les politiques d'affinité/anti-affinité (e.g., placer des pods ensemble ou séparément), les taints et tolérances des nœuds, etc.
  - Kubernetes peut être étendu avec des ordonnanceurs personnalisés.
- **Docker Swarm :** L'orchestrateur natif de Docker.
- **Apache Mesos :** Un gestionnaire de cluster plus généraliste qui peut exécuter divers "frameworks" d'ordonnancement (e.g., Marathon pour les applications long-running, Chronos pour les jobs batch). Mesos utilise un modèle d'allocation de ressources à deux niveaux.

[!] Important / Attention

Dans ces systèmes, l'ordonnanceur distribué (e.g., `kube-scheduler`) prend la décision de haut niveau d'affecter un conteneur (ou Pod) à un nœud. Ensuite, sur ce nœud, c'est l'ordonnanceur local du noyau Linux (CFS, RT) qui gère l'exécution effective des processus à l'intérieur du conteneur, potentiellement en respectant les limites de ressources (CPU quotas, shares) définies par les `cgroups` et configurées par l'orchestrateur.

Un autre domaine pertinent est l'ordonnancement pour les fonctions **Serverless (Function-as-a-Service - FaaS)**. Le défi est d'instancier et d'exécuter rapidement des fonctions éphémères en réponse à des événements, tout en minimisant les coûts et les latences de "cold start" (démarrage à froid d'une instance de fonction).

§5.6 Ordonnancement et Tolérance aux Pannes

La tolérance aux pannes est cruciale dans les systèmes distribués. L'ordonnanceur peut y contribuer :

- **Réplication de tâches :** Exécuter plusieurs copies d'une tâche sur différents nœuds. Si un nœud tombe en panne, une autre copie peut continuer.
- **Checkpointing et Redémarrage :** Sauvegarder périodiquement l'état d'une tâche (checkpoint). En cas de panne, la tâche peut être redémarrée à partir du dernier checkpoint sur un autre nœud, au lieu de tout recommencer.
- **Migration de tâches en cas de détection de panne imminente ou avérée.**
- **Utilisation d'algorithmes de consensus distribué** (e.g., Paxos, Raft) pour permettre à un groupe de nœuds de s'accorder sur l'état du système ou sur les décisions d'ordonnancement, même en présence de pannes (pour les ordonnanceurs décentralisés ou les maîtres répliqués).

§5.7 Défis Spécifiques

Outre ceux déjà mentionnés, l'ordonnancement distribué fait face à d'autres défis :

- **Hétérogénéité des ressources :** Comment ordonnancer efficacement lorsque les nœuds ont des capacités CPU, mémoire, GPU, ou des architectures différentes ?
- **Considérations réseau :** La latence et la bande passante du réseau entre les nœuds peuvent avoir un impact majeur. Les décisions d'ordonnancement doivent en tenir compte (co-localisation des tâches communicantes, minimisation des transferts de données).
- **Sécurité :** Comment s'assurer que les tâches s'exécutent dans des environnements sécurisés et isolés, et que les décisions d'ordonnancement ne peuvent pas être compromises ?
- **Coût et Consommation énergétique :** Pour les grands datacenters, l'ordonnancement peut viser à minimiser les coûts opérationnels ou la consommation d'énergie (e.g., en consolidant les tâches sur moins de nœuds pendant les périodes de faible charge).

[?] Question / Pour aller plus loin Comment les techniques d'apprentissage automatique (Machine Learning) pourraient-elles être utilisées pour améliorer les décisions d'ordonnancement dans des systèmes distribués complexes et dynamiques, par exemple en prédisant les besoins en ressources des tâches ou en identifiant les meilleurs placements ?

[TP] Exercice / TP

Module 5 (Partie Systèmes Distribués) - Études de Cas / Concepts

- **(Conceptuel) Conception d'un algorithme simple de répartition de charge :** Imaginez un système de 3 nœuds. Proposez une politique simple (e.g., initiée par l'émetteur avec des seuils) pour transférer des tâches si un nœud dépasse un certain seuil de charge CPU. Quels sont les paramètres à considérer ? Quels sont les potentiels problèmes ?
- **(Si environnement disponible ou étude documentaire) Kubernetes :** Déployer une application simple (e.g., un serveur web) sur un cluster Minikube (ou un vrai cluster). Observer comment `kube-scheduler` place les pods. Modifier les demandes de ressources CPU/mémoire d'un pod et voir si cela affecte son placement ou son redéploiement. Explorer les concepts d'affinité/anti-affinité de pods.
- **Analyse des logs d'un ordonnanceur HPC (e.g., Slurm) :** Si accès à un cluster HPC, soumettre un job simple et essayer de comprendre à partir des logs de Slurm (ou des commandes comme `squeue`, `sinfo`, `sacct`) comment il a été ordonnancé. Quelles informations l'ordonnanceur utilise-t-il ?
- **Discussion :** Comment l'ordonnanceur local du noyau Linux (CFS, RT) sur un nœud d'un cluster Kubernetes interagit-il avec les décisions de `kube-scheduler` ? Quelles sont les limites imposées par `cgroups` et comment CFS les respecte-t-il ?

L'ordonnancement dans les systèmes distribués est un domaine vaste et actif de recherche et développement. Les solutions varient considérablement en fonction des objectifs spécifiques du système, qu'il s'agisse de maximiser le débit des calculs scientifiques, d'assurer la haute disponibilité des services web, ou de gérer efficacement les ressources dans le cloud. Le chapitre suivant se concentrera sur les outils et techniques pour analyser et optimiser l'ordonnancement, principalement au niveau d'un nœud Linux.

6

Outils, Monitoring et Optimisation

Comprendre la théorie de l'ordonnancement et les mécanismes internes du noyau Linux est essentiel, mais pour un administrateur système ou un développeur, la capacité à observer, analyser et influencer activement le comportement de l'ordonnanceur est tout aussi cruciale. Heureusement, Linux offre une panoplie d'outils et d'interfaces pour le monitoring et l'optimisation des performances de l'ordonnancement.

Ce chapitre présente les principaux outils en ligne de commande, les interfaces du noyau via le système de fichiers `/proc`, les techniques de traçage avancées, l'utilisation des `cgroups` pour la gestion des ressources CPU, et des stratégies générales d'optimisation.

§6.1 Outils en Ligne de Commande pour l'Analyse

De nombreux utilitaires en ligne de commande fournissent des informations précieuses sur l'état des processus et l'utilisation du système, y compris des aspects liés à l'ordonnancement.

- `top` **et** `htop` **:** Ces outils interactifs affichent une vue dynamique et en temps réel des processus en cours d'exécution.
  - `top` : L'outil classique, présent sur la plupart des systèmes Unix-like. Affiche l'utilisation CPU (%CPU), la mémoire, les priorités (`PR`, `NI` pour nice value), l'état des processus (`S`).
  - `htop` : Une alternative plus conviviale et visuelle à `top`, offrant des fonctionnalités améliorées comme le défilement, la vue arborescente, et la manipulation facile des processus (e.g., changer la 'nice value', tuer un processus).
- `ps` **:** L'outil `ps` (process status) affiche des informations sur les processus actifs. Il est extrêmement flexible grâce à ses nombreuses options.

[CODE] Bloc de Code

```
# Afficher tous les processus avec des d tails complets (format BSD)
ps aux

# Afficher les processus avec des informations d'ordonnancement (format POSIX/GNU)
ps -eo pid,tid,class,rtprio,ni,pri,psr,pcpu,stat,wchan:14,comm

# pid: Process ID
# tid: Thread ID
# class: Classe d'ordonnancement (TS pour CFS, FF pour FIFO, RR pour RR, DLN pour DEADLINE)
# rtprio: Priorit  temps r el
# ni: Nice value
# pri: Priorit  dynamique du noyau
# psr: Processeur sur lequel la t che s'ex cute
# pcpu: Utilisation CPU en pourcentage
# stat:  tat  du processus (R: running, S: sleeping, D: uninterruptible sleep, etc.)
# wchan: Adresse de la fonction noyau dans laquelle le processus est endormi
# comm: Nom de la commande

```

Listing 6.1 – Exemples d'utilisation de ps

— `vmstat` **et** `sar` **:**
  — `vmstat` (virtual memory statistics) : Rapporte des informations sur les processus, la mémoire, la pagination, les E/S bloc, les interruptions, et l'activité CPU (e.g., `r` : nombre de processus en attente d'exécution ; `cs` : commutations de contexte par seconde).
  — `sar` (system activity reporter), du paquet `sysstat` : Collecte, rapporte et sauvegarde l'activité du système. Peut fournir des statistiques détaillées sur l'utilisation CPU par cœur, les commutations de contexte (`sar -w`), la charge des runqueues (`sar -q`), etc., sur des intervalles de temps.
— `taskset` **:** Permet de définir ou de récupérer l'affinité CPU d'un processus ou d'une tâche. L'affinité CPU lie l'exécution d'un processus à un ensemble spécifique de cœurs CPU.

[CODE] Bloc de Code

```
# Afficher l'affinit  CPU du processus avec PID <pid>
taskset -p <pid>

# Lancer une commande sur les CPUs 0 et 1 uniquement
taskset -c 0,1 votre_commande

# Changer l'affinit  du processus <pid> pour le CPU 2
taskset -pc 2 <pid>

```

Listing 6.2 – Utilisation de taskset

— `chrt` **:** Permet de manipuler les attributs d'ordonnancement temps réel des processus (politique et priorité). Déjà vu dans les TPs précédents.
— `nice` **et** `renice` **:**
  — `nice` : Lance une commande avec une 'nice value' modifiée.
  — `renice` : Modifie la 'nice value' d'un ou plusieurs processus en cours d'exécution.

§6.2 Interfaces du Noyau via `/proc`

Le système de fichiers virtuel `/proc` expose de nombreuses informations internes du noyau, y compris des détails sur l'ordonnanceur et les processus.

— `/proc/sched_debug` **:** Affiche des informations de débogage très détaillées sur l'état de l'ordonnanceur pour chaque CPU. Cela inclut les paramètres de CFS (`sysctl_sched_latency`, `sysctl_sched_min_granularity`, etc.), les statistiques des runqueues (longueur, `nr_running`, `nr_switches`), et des informations sur les tâches dans chaque runqueue CFS, RT, et Deadline [13]. C'est une mine d'or pour comprendre l'état interne de l'ordonnanceur.
— `/proc/[pid]/sched` **:** Fournit des statistiques d'ordonnancement spécifiques au processus (ou thread) avec l'identifiant `[pid]`. On y trouve des informations comme le `se.vruntime` (pour CFS), le nombre de commutations de contexte (`nr_switches`), le temps total passé en exécution sur le CPU (`sum_exec_runtime`), etc.
— `/proc/[pid]/stat` **et** `/proc/[pid]/status` **:** Contiennent également des informations relatives à l'ordonnancement, comme la priorité, la 'nice value', la politique d'ordonnancement, et l'état du processus. `/proc/[pid]/stat` est formaté pour une analyse par des programmes, tandis que `/proc/[pid]/status` est plus lisible par un humain.
— `/proc/interrupts` **et** `/proc/softirqs` **:** Affichent le nombre d'interruptions matérielles et logicielles qui se sont produites pour chaque type d'interruption et sur chaque CPU. Utile pour identifier des sources potentielles de latence.
— `/proc/sys/kernel/` **(via** `sysctl`**) :** De nombreux paramètres de l'ordonnanceur sont configurables à chaud via cette interface. Exemples :

- `sched_latency_ns`, `sched_min_granularity_ns`, `sched_wakeup_granularity_ns` pour CFS.
- `sched_rt_period_us`, `sched_rt_runtime_us` pour le throttling des tâches RT.
- `sched_migration_cost_ns` : Coût estimé d'une migration de tâche, utilisé par l'équilibreur de charge.
- `sched_autogroup_enabled` : Active/désactive la création automatique de groupes de tâches pour CFS, basée sur les sessions.

On peut lire ces valeurs avec `cat` ou `sysctl`, et les modifier (en tant que root) avec `echo ... >` ou `sysctl -w`.

§6.3 Traçage et Profilage Avancé

Pour une analyse plus fine du comportement de l'ordonnanceur et des performances applicatives, des outils de traçage et de profilage avancés sont nécessaires.

- `perf` **:** L'outil `perf` (parfois appelé "perf_events") est un puissant outil de profilage de performance pour Linux, capable d'utiliser les compteurs de performance matériels (PMCs), les tracepoints du noyau, et d'autres sources d'événements. Pour l'ordonnancement, `perf sched` est particulièrement utile [5].
  - `perf sched record <commande>` : Enregistre les événements d'ordonnancement pendant l'exécution de `<commande>`.
  - `perf sched latency` : Analyse les latences d'ordonnancement (temps entre le réveil d'une tâche et son exécution effective). Affiche des statistiques et un histogramme des latences.
  - `perf sched map` : Affiche une carte colorée montrant quels CPUs exécutaient quelles tâches au fil du temps.
  - `perf sched script` : Affiche les événements d'ordonnancement enregistrés sous forme de texte.
  - `perf top` : Profilage en temps réel similaire à `top`, mais basé sur les événements `perf`.
- `ftrace` **:** `ftrace` est un framework de traçage intégré au noyau Linux. Il permet de tracer des fonctions du noyau, des événements (tracepoints), des latences, etc., avec une très faible surcharge. C'est un outil de choix pour les développeurs noyau et pour le débogage de problèmes de performance complexes [12].
  - Les événements d'ordonnancement (comme `sched_switch`, `sched_wakeup`, `sched_migrate_task`) sont des tracepoints que `ftrace` peut capturer.
  - L'interface se fait principalement via le système de fichiers `debugfs`, généralement monté sur `/sys/kernel/debug/tracing/`.
  - Des outils frontends comme `trace-cmd` (de Steven Rostedt, le mainteneur de ftrace) et `KernelShark` (une GUI pour visualiser les traces `ftrace`) simplifient grandement son utilisation.

[CODE] Bloc de Code

```
# Enregistrer les événements sched_switch pendant 5 secondes
sudo trace-cmd record -e sched_switch sleep 5

# Afficher le rapport
trace-cmd report

```

Listing 6.3 – Exemple simple avec trace-cmd

— **eBPF (extended Berkeley Packet Filter) :** eBPF est une technologie révolutionnaire permettant d'exécuter des programmes personnalisés (écrits en C restreint) directement dans le noyau Linux de manière sûre et efficace, sans avoir à recompiler le noyau ou à charger des modules. eBPF peut être utilisé pour le traçage, le monitoring, le réseau, et la sécurité. Des outils basés sur eBPF, comme ceux de la suite BCC (BPF Compiler Collection) ou bpftrace, permettent de créer des scripts de traçage très puissants pour analyser l'ordonnancement (e.g., mesurer des latences spécifiques, suivre le comportement de tâches particulières) [4].

§6.4 `cgroups` (Control Groups) pour la Gestion des Ressources CPU

Les **Control Groups (`cgroups`)** sont un mécanisme du noyau Linux qui permet d'organiser les processus hiérarchiquement et de distribuer (et limiter) les ressources système entre ces hiérarchies. Le sous-système (ou contrôleur) CPU des `cgroups` est particulièrement pertinent pour l'ordonnancement.

Il existe deux versions de `cgroups` (v1 et v2), avec des interfaces légèrement différentes. `cgroups v2` est la version unifiée et recommandée.

— **CPU Shares (Poids Relatifs) :** Pour les tâches CFS, on peut assigner des poids relatifs aux groupes.
  — `cgroups v1` : Fichier `cpu.shares` dans le répertoire du cgroup. La valeur par défaut est 1024. Une valeur de 2048 donne au groupe deux fois plus de parts CPU qu'un groupe avec 1024.
  — `cgroups v2` : Fichier `cpu.weight`. La plage va de 1 à 10000, avec une valeur par défaut de 100.

  Ces poids sont utilisés par CFS pour la répartition équitable du temps CPU entre les cgroups concurrents.

— **CPU Quotas (Limites Absolues) :** Permet de définir une limite absolue de temps CPU qu'un groupe peut consommer sur une période donnée.
  — `cgroups v1` : Fichiers `cpu.cfs_period_us` (période en microsecondes, e.g., 100000 pour 100ms) et `cpu.cfs_quota_us` (quota de temps CPU en microsecondes pen-

dant cette période). Si `quota` est -1, il n'y a pas de limite.
  - `cgroups v2` : Fichier `cpu.max`. Format : `"<quota> <period>"`. Par exemple, `"50000 100000"` signifie 50ms de CPU toutes les 100ms (soit 50% d'un cœur). `"max"` pour `<quota>` signifie pas de limite.
- **Contrôle des Tâches Temps Réel :** `cgroups` permet également de gérer les budgets de temps CPU pour les tâches temps réel (`SCHED_FIFO/RR`) au sein d'un groupe, via les fichiers `cpu.rt_period_us` et `cpu.rt_runtime_us` (v1) ou des mécanismes similaires en v2. Cela est lié au RT Throttling.

Les orchestrateurs de conteneurs comme Docker et Kubernetes utilisent intensivement les `cgroups` CPU pour isoler et gérer les ressources des conteneurs.

§6.5 Optimisation des Performances de l'Ordonnancement

L'optimisation de l'ordonnancement est souvent un exercice d'équilibrage dépendant de la charge de travail spécifique. Voici quelques stratégies générales :

- **Ajustement des paramètres noyau (`sysctl`) :** Modifier les valeurs de `sched_latency_ns`, `sched_min_granularity_ns`, etc., peut avoir un impact. Par exemple, pour des charges de travail nécessitant un haut débit (HPC, compilation), augmenter légèrement ces valeurs peut réduire la surcharge de commutation. Pour des postes de travail interactifs, des valeurs plus faibles peuvent améliorer la réactivité (au détriment d'un peu de débit). À faire avec prudence et après des tests approfondis.
- **Utilisation de l'affinité CPU (`taskset`, `cgroups cpuset`) :** Pour les applications sensibles à la latence ou pour éviter l'interférence des caches, lier des processus critiques à des cœurs CPU spécifiques (isolation de cœurs) peut être bénéfique. Le contrôleur `cpuset` des `cgroups` offre un contrôle plus fin sur l'affectation des CPUs et des nœuds mémoire NUMA aux groupes de processus.
- **Choix de la bonne politique d'ordonnancement :** Utiliser `SCHED_FIFO/RR` uniquement lorsque c'est absolument nécessaire et avec des priorités appropriées. Pour les tâches de fond non critiques, `SCHED_BATCH` ou `SCHED_IDLE` peuvent être préférables.
- **Utilisation des 'nice values' et des poids `cgroup` :** Donner des priorités relatives appropriées aux différentes applications ou services.
- **Optimisation du code applicatif :** Souvent, la meilleure optimisation vient de l'application elle-même (réduction des sections critiques, E/S asynchrones, meilleur parallélisme).
- **Cas spécifique des architectures NUMA (Non-Uniform Memory Access) :** Sur les systèmes NUMA, l'accès à la mémoire locale d'un nœud est plus rapide que l'accès à la mémoire d'un autre nœud. L'ordonnanceur Linux est conscient de NUMA (`NUMA-aware scheduling`) et essaie de maintenir les processus et leurs données sur le même nœud NUMA autant que possible. Cependant, des déséquilibres de charge peuvent nécessiter des migrations inter-nœuds. Des outils comme `numactl` permettent de contrôler la politique NUMA des processus.
- **Identifier et résoudre les goulots d'étranglement :** Utiliser les outils de profilage (`perf`, `ftrace`, eBPF) pour trouver où le temps est réellement passé (CPU, E/S, ver-

rous, etc.) et concentrer les efforts d'optimisation là où ils auront le plus d'impact.

[!] Important / Attention

L'optimisation est un processus itératif : Mesurer → Analyser → Modifier → Mesurer à nouveau. Ne pas optimiser prématurément ou sans données concrètes.

[TP] Exercice / TP

Module 6 - Analyse et Optimisation Pratique

- **Utiliser `perf sched` pour analyser une charge de travail :** Lancer une charge de travail mixte (e.g., compilation en fond, navigation web, lecture vidéo). Utiliser `perf sched record` puis `perf sched latency` et `perf sched map` pour visualiser le comportement de l'ordonnanceur. Identifier les tâches qui subissent le plus de latence.
- **Utiliser `ftrace` (via `trace-cmd`) pour observer les événements `sched_switch` :** Filtrer les événements pour une tâche spécifique ou un CPU. Essayer de comprendre pourquoi une tâche est préemptée ou quand elle est réveillée.
- **Configurer des `cgroups` (v2 de préférence) pour limiter l'utilisation CPU :** Créer un cgroup, lui assigner un quota CPU (e.g., 20% d'un cœur via `cpu.max`), y déplacer un processus gourmand en CPU, et vérifier avec `top` ou `htop` que la limite est respectée. Tester également l'effet des poids (`cpu.weight`) entre deux cgroups.
- **(Défi) Diagnostiquer un problème de latence simple :** Créer un scénario simple où une tâche "importante" subit des latences à cause d'une autre tâche. Utiliser les outils présentés pour identifier la cause et proposer une solution (e.g., changer la 'nice value', utiliser l'affinité CPU, `cgroups`).

Ce chapitre a équipé l'étudiant d'un arsenal d'outils et de techniques pour interagir avec l'ordonnanceur Linux. Le chapitre suivant illustrera l'application de ces connaissances à travers des études de cas concrètes.

7

Études de Cas et Bonnes Pratiques

Après avoir exploré en détail la théorie de l'ordonnancement, les mécanismes spécifiques du noyau Linux, et les outils d'analyse et d'optimisation, ce chapitre vise à illustrer l'application de ces connaissances à travers plusieurs études de cas. Chaque scénario présentera des défis d'ordonnancement distincts et des approches pour les aborder. Nous conclurons par un résumé des bonnes pratiques et des erreurs courantes à éviter.

L'objectif est de montrer comment choisir les bonnes politiques, configurer les paramètres appropriés, et utiliser les outils adéquats pour atteindre les objectifs de performance désirés dans des contextes variés.

§7.1 Cas 1 : Optimisation d'un Serveur Web à Fort Trafic

[i] Note

Scénario : Un serveur web (e.g., Nginx, Apache) hébergeant un site dynamique populaire subit une forte charge, avec de nombreuses requêtes HTTP concurrentes. Les temps de réponse commencent à se dégrader pendant les pics de trafic.

• 7.1.1 Défis d'Ordonnancement

- **Gestion d'un grand nombre de connexions/processus légers :** Les serveurs web modernes utilisent souvent un modèle événementiel (e.g., Nginx) ou un pool de threads/processus workers (e.g., Apache avec MPM worker ou event). L'ordonnanceur doit gérer efficacement ces nombreuses entités.
- **Latence des requêtes :** Chaque requête doit être traitée rapidement pour assurer une bonne expérience utilisateur. Les tâches traitant les requêtes doivent obtenir le CPU rapidement.
- **Mix de tâches CPU-bound et I/O-bound :** Certaines requêtes peuvent nécessiter des calculs (CPU-bound, e.g., rendu de templates, exécution de scripts PHP/Python), tandis que d'autres sont principalement en attente d'E/S (lecture de fichiers disque, requêtes base de données, attente de réponses réseau).

— **Interférences entre les requêtes et les tâches de fond :** Des tâches de maintenance du serveur (logs, backups) ne doivent pas impacter les performances des requêtes.

7.1.2 Stratégies d'Optimisation de l'Ordonnancement

1. **Affinité CPU pour les processus workers :** Si le serveur web utilise plusieurs processus workers, on peut envisager de lier chaque worker (ou groupe de workers) à des cœurs CPU spécifiques en utilisant `taskset` ou les `cpusets` de `cgroups`.
   — **Avantage :** Améliore la localité du cache CPU (moins de "cache thrashing" si un worker reste sur le même cœur) et peut réduire la contention sur les verrous des runqueues.
   — **Précaution :** S'assurer que la répartition de la charge entre les workers reste équilibrée. Peut nécessiter une configuration au niveau du serveur web lui-même (e.g., nombre de workers égal au nombre de cœurs dédiés).
2. **Affinité des interruptions réseau :** Les interruptions des cartes réseau peuvent être concentrées sur des cœurs spécifiques (non utilisés par les workers principaux) pour réduire la gigue et la charge CPU sur les cœurs traitant les requêtes. Des outils comme `irqbalance` tentent de le faire automatiquement, mais une configuration manuelle via `/proc/irq/[num]/smp_affinity` peut être nécessaire pour des optimisations fines.
3. **Ajustement des 'nice values' et des priorités** `cgroup` **:**
   — Donner une priorité légèrement plus élevée (une 'nice value' plus basse, e.g., -5) aux processus workers du serveur web par rapport aux autres tâches moins critiques.
   — Utiliser les `cgroups` CPU pour garantir une part minimale de CPU aux workers du serveur web (`cpu.shares` ou `cpu.weight`) et potentiellement limiter les tâches de fond (`cpu.max` ou `cpu.cfs_quota_us`).
4. **Paramètres de l'ordonnanceur CFS :** Pour un serveur web axé sur la latence, on pourrait envisager de réduire légèrement `sched_latency_ns` et `sched_min_granularity_ns`. Cependant, cela doit être fait avec des tests rigoureux, car des valeurs trop basses peuvent augmenter la surcharge de commutation. Souvent, les valeurs par défaut de CFS sont déjà bien optimisées. `sched_autogroup_enabled=1` (souvent par défaut) peut aider à isoler les sessions utilisateurs, ce qui peut être bénéfique si différentes applications tournent sur le même serveur, mais peut être moins pertinent pour un serveur dédié.
5. **Monitoring :** Utiliser `top`, `htop`, `vmstat`, `sar` pour surveiller la charge CPU, les commutations de contexte, la longueur des runqueues. Utiliser `perf top` ou `perf record` pour identifier les "hot spots" dans le code du serveur web ou du noyau. Analyser les latences d'ordonnancement avec `perf sched latency`.

[!] Important / Attention

Pour les serveurs web, l'optimisation de l'ordonnancement est souvent une partie d'un ensemble plus large d'optimisations, incluant la configuration du serveur web lui-même, la base de données, le réseau, et le code applicatif.

§7.2 Cas 2 : Application Temps Réel Strict (e.g., Contrôle Robotique)

[i] Note

Scénario : Un système de contrôle pour un bras robotique doit lire les données des capteurs, effectuer des calculs de trajectoire, et envoyer des commandes aux actionneurs, le tout en respectant des échéances strictes de quelques millisecondes pour éviter des mouvements erratiques ou dangereux.

• 7.2.1 Défis d'Ordonnancement

- **Déterminisme et latences bornées :** La priorité absolue est de garantir que les tâches critiques se terminent avant leurs échéances, avec une gigue minimale.
- **Isolation des tâches critiques :** Les tâches non critiques (e.g., logging, interface utilisateur) ne doivent pas interférer avec les boucles de contrôle temps réel.
- **Gestion des E/S temps réel :** L'accès aux capteurs et actionneurs doit être rapide et prévisible.

• 7.2.2 Stratégies d'Optimisation de l'Ordonnancement

1. **Utilisation du noyau `PREEMPT_RT` :** Indispensable pour minimiser les sources de latence du noyau (interruptions threadées, spinlocks préemptibles, etc.). Configurer le noyau avec `CONFIG_PREEMPT_RT`.
2. **Politiques d'ordonnancement temps réel :**
   - `SCHED_FIFO` **:** Pour les tâches critiques principales de la boucle de contrôle. Les priorités RT (1-99) doivent être soigneusement assignées en fonction de la criticité et des dépendances des tâches (e.g., en utilisant une approche Rate Monotonic Analysis si les tâches sont périodiques).
   - `SCHED_DEADLINE` **:** Peut être une meilleure option si les tâches ont des échéances explicites et variées, ou si l'on souhaite une meilleure isolation grâce au contrôle d'admission et au CBS. Nécessite une bonne modélisation des paramètres (runtime, period, deadline) de chaque tâche.
3. **Isolation de cœurs CPU (CPU Shielding/Partitioning) :** Dédier un ou plusieurs cœurs CPU exclusivement aux tâches temps réel critiques. Les tâches non-RT, les interruptions non critiques, et les softirqs sont confinés aux autres cœurs.
   - Utiliser `isolcpus=<liste_cpus>` comme paramètre de démarrage du noyau pour isoler des cœurs de l'ordonnanceur général.
   - Utiliser `taskset` ou les `cpusets` de `cgroups` pour lier les tâches RT aux cœurs isolés et les tâches non-RT aux cœurs restants.
   - Configurer l'affinité des interruptions (`/proc/irq/.../smp_affinity`) pour que les interruptions critiques (liées aux capteurs/actionneurs) soient sur les cœurs RT, et les autres ailleurs.

4. **Verrouillage mémoire (`mlockall()`) :** Empêcher la pagination des espaces mémoire des tâches RT critiques en utilisant `mlockall(MCL_CURRENT | MCL_FUTURE)` pour éviter les latences dues aux défauts de page.
5. **Minimisation des sources de latence applicative :**
   — Éviter les allocations dynamiques de mémoire dans les boucles temps réel.
   — Utiliser des algorithmes sans verrou (lock-free) ou avec des verrous RT-safe (PI-mutex) pour la communication entre tâches RT.
   — Optimiser l'accès aux périphériques.
6. **Tests et validation rigoureux :** Utiliser `cyclictest` intensivement pour mesurer les pires latences sous différentes charges. Utiliser `ftrace` et `perf` pour traquer les sources de latence inattendues. Effectuer des tests de stress et de longue durée.

[!] Important / Attention

La conception d'un système temps réel strict est une discipline exigeante qui va au-delà de la simple configuration de l'ordonnanceur. Elle implique une analyse minutieuse de l'ensemble du système, du matériel au logiciel applicatif.

§7.3 Cas 3 : Ordonnancement d'un Grand Nombre de Tâches de Calcul sur un Cluster HPC

[i] Note

Scénario : Un centre de calcul universitaire gère un cluster HPC utilisé par de nombreux chercheurs pour des simulations scientifiques parallèles (MPI, OpenMP) qui peuvent durer des heures ou des jours et consommer des centaines de cœurs.

• 7.3.1 Défis d'Ordonnancement

— **Partage équitable des ressources entre utilisateurs et projets.**
— **Maximisation de l'utilisation du cluster (throughput).**
— **Minimisation du temps d'attente des jobs.**
— **Gestion des jobs avec des besoins en ressources hétérogènes** (nombre de nœuds, cœurs, mémoire, GPUs, licences logicielles).
— **Prise en compte des priorités** (projets urgents, utilisateurs avec des quotas différents).

• 7.3.2 Approches d'Ordonnancement (via un Ordonnanceur de Batch comme Slurm)

1. **Configuration des files d'attente (Partitions dans Slurm) :** Définir des partitions avec des limites de temps, des tailles de jobs maximales/minimales, des accès res-

treints à certains groupes d'utilisateurs ou types de matériel (e.g., nœuds GPU).

2. **Politiques de priorité multifactorielles :** Slurm utilise un algorithme de priorité multifactoriel qui peut prendre en compte :
   — L'âge du job (temps passé en attente).
   — La taille du job (nombre de nœuds/cœurs demandés).
   — Le "fairshare" : une mesure de l'utilisation passée des ressources par l'utilisateur ou le projet, pour favoriser ceux qui ont moins utilisé le cluster récemment.
   — La qualité de service (QoS) associée au job, qui peut accorder des priorités plus élevées ou des accès privilégiés.
3. **Backfilling :** Configurer Slurm pour utiliser le backfilling (souvent activé par défaut) pour permettre à des jobs plus petits de s'exécuter s'ils ne retardent pas le démarrage prévu des jobs plus gros et plus prioritaires.
4. **Réservations :** Permettre de réserver des ressources pour des utilisations spécifiques (démonstrations, cours, maintenance).
5. **Gestion des dépendances de jobs :** Permettre aux utilisateurs de soumettre des chaînes de jobs où un job ne démarre qu'après la complétion réussie d'un autre.
6. **Monitoring et Comptabilité (Accounting) :** Utiliser les outils de Slurm (`squeue`, `sinfo`, `scontrol`, `sacct`) pour surveiller l'état du cluster, des jobs, et pour analyser l'utilisation des ressources à des fins de reporting et de planification.
7. **Ordonnancement local sur les nœuds de calcul :** Une fois qu'un job est alloué à un ensemble de nœuds, l'ordonnanceur local du noyau Linux (CFS) sur chaque nœud gère les processus du job. Pour les applications MPI, chaque processus MPI s'exécute généralement sur un cœur dédié. Les `cgroups` (gérés par Slurm via des plugins) sont utilisés pour confiner les processus du job aux cœurs alloués et pour contrôler la mémoire.

[?] Question / Pour aller plus loin Comment les politiques de "fairshare" dans les ordonnanceurs HPC peuvent-elles parfois entrer en conflit avec l'objectif de maximiser le débit global du cluster, et comment trouver un équilibre ?

§7.4 Cas 4 : Application Multimédia (e.g., Traitement Vidéo en Direct)

[i] Note

Scénario : Une application de visioconférence ou de streaming vidéo en direct doit capturer, encoder, transmettre et décoder des flux vidéo et audio avec une faible latence et sans "glitches" (saccades, pertes d'images).

7.4.1 Défis d'Ordonnancement

— **Latence de bout en bout faible et prévisible.**
— **Débit soutenu pour l'encodage/décodage.**
— **Synchronisation audio/vidéo.**
— **Coexistence avec d'autres applications sur un poste de travail ou un appareil mobile.**

7.4.2 Stratégies d'Ordonnancement

1. **Utilisation judicieuse des priorités temps réel (souple) :** Les threads critiques pour le pipeline multimédia (capture, encodage/décodage audio/vidéo, rendu) peuvent se voir assigner des priorités `SCHED_RR` ou `SCHED_FIFO` modérées (e.g., entre 10 et 50).
   — **Avantage :** Assure que ces threads obtiennent le CPU rapidement lorsqu'ils ont du travail.
   — **Précaution :** Éviter des priorités trop élevées qui pourraient affamer d'autres parties du système (interface utilisateur, réseau). Les tâches doivent être bien conçues pour se bloquer ou céder le CPU lorsqu'elles n'ont pas de travail immédiat.
2. **Alternative :** `SCHED_DEADLINE` **(plus avancé) :** Si les composants du pipeline ont des échéances et des besoins en calcul bien définis, `SCHED_DEADLINE` pourrait offrir un meilleur contrôle et une meilleure isolation. Plus complexe à mettre en œuvre pour l'application.
3. **Utilisation de** `nice values` **pour les threads moins critiques :** Les threads de l'interface utilisateur ou les tâches de fond de l'application peuvent utiliser des `nice values` standards ou légèrement augmentées.
4. **Affinité CPU :** Peut être utile pour dédier certains cœurs aux tâches d'encodage/décodage intensives, surtout sur des systèmes multi-cœurs.
5. **Optimisation du pipeline de données :** Minimiser les copies de données, utiliser des E/S asynchrones, et des tampons (buffers) de taille appropriée pour lisser les variations de charge.
6. **PipeWire / PulseAudio / JACK :** Sur les systèmes de bureau Linux, les serveurs de son comme PipeWire ou PulseAudio, et JACK pour l'audio professionnel, implémentent leurs propres mécanismes d'ordonnancement et de priorisation pour les flux audio, souvent en utilisant les politiques RT du noyau pour leurs threads critiques. Les applications s'interfacent avec ces serveurs.
7. **Sur les systèmes embarqués/mobiles :** Des mécanismes spécifiques comme EAS (Energy-Aware Scheduling) et des optimisations pour les architectures big.LITTLE sont cruciaux pour équilibrer performance et consommation d'énergie. Les threads multimédia peuvent être poussés vers les "gros" cœurs performants lorsque c'est nécessaire.

§7.5 Bonnes Pratiques et Erreurs Courantes

7.5.1 Bonnes Pratiques

- **Comprendre la charge de travail :** Avant toute optimisation, caractériser la nature des tâches (CPU-bound, I/O-bound, interactif, batch, temps réel), leurs dépendances, et les objectifs de performance.
- **Mesurer avant et après chaque changement :** Utiliser des outils de monitoring et de profilage pour quantifier l'impact des modifications.
- **Commencer par les optimisations les plus simples :** 'nice values', affinité CPU de base, configuration de l'application elle-même.
- **Utiliser les politiques RT avec parcimonie et prudence :** Réserver `SCHED_FIFO/RR/DEADLINE` aux tâches qui en ont réellement besoin et s'assurer qu'elles sont bien écrites (se bloquent, ne bouclent pas indéfiniment).
- **Privilégier les `cgroups` pour la gestion des ressources entre groupes d'applications** plutôt que de jouer excessivement avec les 'nice values' de processus individuels de manière globale.
- **Tester sur un matériel représentatif de la production.**
- **Documenter les changements et leurs justifications.**

7.5.2 Erreurs Courantes à Éviter

- **Optimisation prématurée :** Changer des paramètres sans avoir identifié un réel problème de performance.
- **Abus des priorités temps réel :** Mettre trop de tâches en RT ou avec des priorités trop élevées peut déstabiliser le système.
- **Ignorer les coûts de migration et de commutation de contexte :** Une affinité CPU trop stricte ou des quanta trop petits peuvent être contre-productifs.
- **Se concentrer uniquement sur le CPU :** Les goulots d'étranglement peuvent provenir des E/S (disque, réseau), de la mémoire, ou de verrous applicatifs.
- **Modifier trop de paramètres à la fois :** Rend difficile l'identification de l'effet de chaque changement.
- **Négliger l'impact sur la consommation d'énergie**, surtout sur les appareils mobiles ou embarqués.
- **Ne pas considérer les interactions avec d'autres sous-systèmes** (mémoire, réseau, E/S).

Ce chapitre a montré, à travers des exemples, que l'optimisation de l'ordonnancement est un art subtil qui nécessite une bonne compréhension du système, des outils, et de la charge de travail. Il n'y a pas de solution universelle, mais une approche méthodique et informée mène généralement à de bons résultats.

8

Conclusion et Perspectives

Au terme de ce cours complet sur l'ordonnancement sous Linux, incluant les systèmes temps réel et distribués, nous avons parcouru un vaste paysage, allant des concepts théoriques fondamentaux aux mécanismes internes complexes du noyau, en passant par les défis spécifiques des environnements temps réel et distribués, et les outils pratiques d'analyse et d'optimisation.

L'ordonnancement est véritablement au cœur de la performance, de la réactivité et de la fiabilité des systèmes d'exploitation modernes. Sa maîtrise est une compétence essentielle pour tout ingénieur système, développeur noyau, ou administrateur cherchant à exploiter au mieux les capacités des infrastructures informatiques contemporaines.

§8.1 Résumé des Concepts Clés

Récapitulons les points essentiels abordés tout au long de ce cours :

- **Fondamentaux de l'ordonnancement :** Nous avons défini les objectifs (équité, débit, temps de réponse, etc.), les critères d'évaluation, et exploré les algorithmes classiques (FCFS, SJF, Priorités, RR, Files Multiniveaux) qui constituent la base théorique.
- **Ordonnanceur Linux :** Nous avons disséqué l'architecture modulaire de l'ordonnanceur Linux avec ses classes d'ordonnancement.
  - **CFS (`fair_sched_class`) :** L'ordonnanceur par défaut, visant une équité proportionnelle grâce au `vruntime` et aux arbres rouge-noir, influencé par les `nice values` et les `cgroups`.
  - **Ordonnanceurs RT (`rt_sched_class`) :** `SCHED_FIFO` et `SCHED_RR` pour les tâches temps réel avec priorités statiques.
  - **Ordonnanceur Deadline (`dl_sched_class`) :** `SCHED_DEADLINE` basé sur EDF et CBS pour des garanties temps réel plus fortes avec contrôle d'admission.
- **Systèmes Temps Réel :** Nous avons identifié les défis de la latence dans les OS généralistes et comment le patch `PREEMPT_RT` transforme Linux pour le rendre apte aux applications temps réel strictes (préemption du noyau, IRQs threadées, PI-mutex). Les concepts d'inversion de priorité et d'analyse d'ordonnançabilité (RMS, EDF) ont été introduits.
- **Systèmes Distribués :** L'ordonnancement a été étendu à des ensembles de nœuds,

avec des objectifs de répartition de charge, de parallélisme et de tolérance aux pannes. Nous avons exploré les taxonomies des ordonnanceurs distribués, la migration de tâches, et des exemples concrets dans les domaines du HPC (Slurm) et du Cloud/Conteneurs (Kubernetes, Mesos).
- **Outils et Optimisation :** Une panoplie d'outils (`top`, `ps`, `perf`, `ftrace`, eBPF), d'interfaces (`/proc`), et de techniques (`cgroups`, affinité CPU, paramètres noyau) a été présentée pour monitorer, analyser et optimiser le comportement de l'ordonnancement.
- **Études de Cas :** Des scénarios pratiques ont illustré l'application des connaissances acquises pour optimiser des serveurs web, des systèmes temps réel, des clusters HPC et des applications multimédia.

[!] Important / Attention

L'un des messages clés de ce cours est que l'ordonnancement est un domaine de compromis. Il n'existe pas d'algorithme ou de configuration "parfait" pour toutes les situations. Le choix optimal dépend toujours de la nature de la charge de travail, des contraintes matérielles, et des objectifs de performance spécifiques.

§8.2 Évolutions Futures de l'Ordonnancement sous Linux et au-delà

Le domaine de l'ordonnancement est loin d'être statique. Il continue d'évoluer pour s'adapter aux nouvelles architectures matérielles, aux nouveaux paradigmes applicatifs, et aux exigences changeantes des utilisateurs. Voici quelques tendances et perspectives :
- **Prise en compte des Architectures Hétérogènes :** Les processeurs modernes incluent de plus en plus souvent des cœurs hétérogènes (e.g., ARM big.LITTLE, futurs designs x86 avec cœurs performants et cœurs efficients). L'ordonnancement doit être capable de placer intelligemment les tâches sur le type de cœur le plus approprié en fonction de leurs besoins (performance vs. efficacité énergétique). Des mécanismes comme HMP (Heterogeneous Multi-Processing) et EAS (Energy-Aware Scheduling) dans Linux tentent de relever ce défi. La complexité augmente avec le nombre de types de cœurs et leurs caractéristiques.
- **Ordonnancement Conscient de l'Énergie (Energy-Aware Scheduling - EAS) :** Avec l'importance croissante de l'efficacité énergétique, surtout pour les appareils mobiles et les grands datacenters, EAS est devenu un domaine de recherche et développement majeur. L'objectif est de prendre des décisions d'ordonnancement (choix du CPU, fréquence, état d'inactivité) qui minimisent la consommation d'énergie tout en respectant les contraintes de performance [13]. Cela implique une modélisation fine de la puissance des CPUs et des besoins des tâches.
- **Améliorations continues pour `PREEMPT_RT` et son intégration Mainline :** Le travail pour réduire les latences et améliorer le déterminisme de Linux se poursuit. L'objectif à long terme de la communauté `PREEMPT_RT` est d'intégrer la totalité de ses fonctionna-

lités essentielles dans le noyau Linux principal, rendant ainsi le temps réel une option standard plus accessible.

- **Ordonnancement sensible à la topologie et aux interférences des ressources partagées :** Sur les systèmes multi-cœurs complexes, les ressources partagées (caches L2/L3, bande passante mémoire, interconnexions) peuvent devenir des points de contention. Des recherches sont en cours pour développer des ordonnanceurs qui tiennent compte de ces interférences pour mieux isoler les tâches ou co-localiser intelligemment celles qui peuvent partager des ressources sans se nuire.
- **Utilisation accrue de eBPF pour l'ordonnancement et le monitoring :** eBPF offre des possibilités fascinantes pour observer et potentiellement influencer l'ordonnanceur de manière dynamique et personnalisée, sans modifier le code source du noyau. On peut imaginer des politiques d'ordonnancement spécifiques à une application, implémentées en partie via eBPF.
- **Ordonnancement dans les environnements virtualisés et conteneurisés :** Assurer l'équité et l'isolation des performances entre machines virtuelles ou conteneurs, tout en optimisant l'utilisation des ressources physiques sous-jacentes, reste un défi complexe, surtout avec des niveaux d'imbrication (VM dans VM, conteneurs dans VM).
- **Ordonnancement pour les accélérateurs spécialisés (GPU, FPGA, TPU) :** Avec la prolifération des accélérateurs matériels, l'ordonnancement doit s'étendre pour gérer l'accès et l'utilisation efficace de ces ressources, souvent en coordination avec l'ordonnanceur CPU.
- **Ordonnancement conscient de la sécurité :** Comment les décisions d'ordonnancement peuvent-elles impacter la sécurité ? Par exemple, éviter que des tâches de sensibilités différentes ne partagent des ressources d'une manière qui pourrait faciliter des attaques par canaux auxiliaires (side-channel attacks).

§8.3 L'Intelligence Artificielle et l'Ordonnancement ?

Une perspective intrigante est l'application des techniques d'Intelligence Artificielle (IA), et en particulier de l'Apprentissage Automatique (Machine Learning - ML), à l'ordonnancement.

- **Prédiction du comportement des tâches :** Des modèles de ML pourraient apprendre à prédire la durée des CPU bursts, les besoins en E/S, ou les schémas d'accès mémoire des tâches, améliorant ainsi la précision des décisions d'ordonnanceurs comme SJF ou ceux basés sur des estimations.
- **Optimisation adaptative des paramètres :** Un agent ML pourrait apprendre à ajuster dynamiquement les paramètres de l'ordonnanceur (quanta, seuils de préemption, poids) en fonction de la charge de travail observée pour optimiser un objectif donné (e.g., latence, débit, consommation d'énergie).
- **Ordonnancement dans les systèmes distribués complexes :** Pour des systèmes à grande échelle avec de nombreuses contraintes hétérogènes, des approches basées sur l'apprentissage par renforcement (Reinforcement Learning) pourraient découvrir des politiques d'ordonnancement efficaces là où les heuristiques traditionnelles peinent.

Cependant, l'utilisation de l'IA pour l'ordonnancement au niveau du noyau pose des défis significatifs :

- **Surcharge (Overhead) :** Les modèles de ML, surtout les réseaux de neurones profonds, peuvent être coûteux en termes de calcul et de mémoire, ce qui peut être rédhibitoire pour des décisions d'ordonnancement qui doivent être prises en quelques microsecondes.
- **Déterminisme et Explicabilité :** Le comportement des modèles de ML peut être difficile à prédire et à expliquer, ce qui est problématique pour les systèmes critiques ou temps réel.
- **Collecte de données et entraînement :** Nécessité de grandes quantités de données représentatives pour entraîner les modèles efficacement.

Malgré ces défis, c'est un domaine de recherche actif, et des solutions hybrides ou des modèles plus légers pourraient émerger.

§8.4 Ressources pour Aller Plus Loin

Ce cours a fourni une base solide, mais le domaine de l'ordonnancement est vaste et en constante évolution. Pour ceux qui souhaitent approfondir leurs connaissances, voici quelques pistes :

- **Documentation du Noyau Linux :** La documentation officielle, accessible via le code source du noyau ou sur `https://www.kernel.org/doc/html/latest/`, notamment la section sur l'ordonnanceur ('scheduler/'). C'est la source la plus à jour.
- **Livres de Référence :**
  - "Understanding the Linux Kernel" par Daniel P. Bovet et Marco Cesati (bien que datant un peu, les concepts fondamentaux restent valides).
  - "Linux Kernel Development" par Robert Love (excellente introduction au développement noyau, avec des chapitres sur l'ordonnancement).
  - "Hard Real-Time Computing Systems" par Giorgio C. Buttazzo (référence pour l'ordonnancement temps réel).
  - "Distributed Systems : Concepts and Design" par George Coulouris et al. (pour les systèmes distribués).
  - "BPF Performance Tools" par Brendan Gregg (pour eBPF et le profilage avancé).
- **Sites Web et Listes de Diffusion :**
  - **LWN.net (`https://lwn.net/`)** : Une ressource inestimable pour suivre les développements du noyau Linux, avec des articles de fond sur l'ordonnancement et d'autres sujets.
  - **Linux Kernel Mailing List (LKML)** : Pour les discussions de développement les plus pointues (très haut volume).
  - **Wiki du projet Real-Time Linux :** `https://wiki.linuxfoundation.org/realtime/start`
- **Conférences et Articles de Recherche :** Les actes de conférences comme OSPM (Operating Systems Platforms for Embedded Real-Time applications), ECRTS (Euromicro Conference on Real-Time Systems), RTSS (Real-Time Systems Symposium), USENIX

ATC, SOSP/OSDI, EuroSys, ainsi que les journaux spécialisés (e.g., ACM Transactions on Computer Systems, IEEE Transactions on Parallel and Distributed Systems) publient les dernières avancées.

— **Expérimentation Pratique :** La meilleure façon d'apprendre est de pratiquer. Mettre en place des environnements de test, utiliser les outils, lire le code source du noyau (même si c'est intimidant au début), et contribuer à des projets open source.

[i] Note

L'ordonnancement est un voyage fascinant au cœur des systèmes informatiques. Nous espérons que ce cours vous a donné les outils et la curiosité nécessaires pour continuer à explorer ce domaine dynamique et essentiel.

Références Bibliographiques

[1] Daniel P. Bovet et Marco Cesati. *Understanding the Linux Kernel*. 3rd. O'Reilly Media, 2005.

[2] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications*. 3rd. Springer, 2011.

[3] George Coulouris et al. *Distributed Systems : Concepts and Design*. 5th. Pearson Education, 2012.

[4] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[5] Linux perf community et autres contributeurs. *perf wiki*. 2023. URL : https://perf.wiki.kernel.org/index.php/Main_Page (visité le 27/10/2023).

[6] C. L. Liu et James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In : *Journal of the ACM (JACM)* 20.1 (1973), p. 46-61. DOI : 10.1145/321738.321743.

[7] Robert Love. *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010.

[8] Ingo Molnár. "CFS scheduler". In : *Linux Kernel Mailing List post*. Annonce initiale de CFS. Peut être difficile à trouver comme citation formelle, utiliser avec prudence ou chercher un article plus formel s'il existe. Avr. 2007. URL : https://lwn.net/Articles/230574/.

[9] OSADL (Open Source Automation Development Lab). *PREEMPT$_R$TWiki*. 2023. URL : https://wiki.linuxfoundation.org/realtime/start.

[10] Abraham Silberschatz, Peter B. Galvin et Greg Gagne. *Operating System Concepts*. 10th. Wiley, 2018.

[11] Andrew S. Tanenbaum et Herbert Bos. *Modern Operating Systems*. 4th. Pearson Education, 2015.

[12] The Linux Kernel community. *ftrace - Function Tracer*. 2023. URL : https://www.kernel.org/doc/Documentation/trace/ftrace.rst (visité le 27/10/2023).

[13] The Linux Kernel community. *Kernel Scheduler Documentation*. 2023. URL : https://www.kernel.org/doc/html/latest/scheduler/index.html.