Ordonnancement dans la famille Linux : État de l'art

Youri Hubaut

Résumé

Dans ce travail, nous tenterons de retracer l'évolution des ordonnanceurs de processus dans la famille Linux. Des premières versions jusqu'aux dernières avancées.

Introduction

Mais qu'est-ce que l'ordonnancement ? Eswaran, Gray, Lorie et Traiger (Eswaran et al., 1976) tentent de définir ce concept comme étant le processus de répartition d'un ensemble de transactions entre différents acteurs. Ces transactions se décomposent en séquences d'étapes visant à accomplir divers travaux. De manière plus générale, on qualifie d'ordonnanceur tout programme visant à partager le travail entre les différentes ressources et acteurs tout en déterminant l'ordre dans lequel celles-ci seront attribuées.

Ce terme est relativement large et englobe plusieurs paradigmes. Généralement, il tend à désigner la gestion des différents processus dans des systèmes d'exploitations multitâches. On qualifie un système d'exploitation de multitâche, si celui-ci peut interlacer "simultanément" l'exécution de plusieurs processus. Cela donne, sur des machines avec un seul processeur, l'illusion que de multiples programmes s'exécutent simultanément. On utilise également la notion de préemption lorsque les processus ne choisissent pas activement de passer la main (Bach, 1986). À noter qu'une terminologie (Casavant and Kuhl, 1988) et des classifications (Wang and Morris, 1985) ont été proposées afin d'y voir un peu plus clair dans cette grande famille.

Les ordonnanceurs peuvent avoir différents objectifs : maximiser la quantité de travail effectué par unité de temps, minimiser le temps de réponse (le délai qui s'écoule entre le début du travail et son exécution), minimiser la latence (le délai entre son début et sa fin), économiser la batterie ou encore maximiser l'équité (*fairness*) en répartissant les ressources au mieux entre les différents acteurs tout en prenant en compte leurs caractéristiques intrinsèques. Ils dépendent égalements des buts visés par l'environnement : job, interactive, ou temps réel (Hansen, 1973).

En plus, on demande que les ordonnanceurs soient performants et efficaces. Une gestion optimale des ressources afin de faire bénéficier au maximum les utilisateurs sans subir le surcoût associé au système d'ordonnancement lui-même. Généralement, le choix d'ordonnancement est effectué dans quatre cas : lorsqu'un programme se termine, lors d'une interruption E/S, lors d'une barrière implicite ($mutex$) ou encore lorsque le quantum de temps associé expire (Bovet and Cesati, 2005).

On distingue trois grandes familles : les jobs, interactifs et temps réel. Toutes tentent d'assurer une certaine équité et d'équilibrer les ressources au maximum. Les jobs se concentrent sur le traitement des calculs intensifs et visent à maximiser le nombre de travaux effectués, à réduire le délai entre le début de la tâche et sa fin tout en faisant tourner le CPU au maximum de ses capacités. Les interactifs correspondent davantage à la vision conventionnelle d'un OS ; ils répondent rapidement aux demandes de l'utilisateur et essayent de contenter ses attentes au niveau des performances. Enfin, les temps réels qui visent à réaliser des objectifs dans des limites imparties et ainsi assurer une meilleure prédictabilité (Tanenbaum and Woodhull, 2005).

Les types Job

Les ordonnanceurs de types Job ont un statut un peu particulier. En effet, il existe de très nombreux algorithmes conçus spécialement pour eux (Méndez et al., 2006). Toutefois, dans le cas de Linux, l'habitude consiste à créer sa propre grille horaire au travers des logiciels hérités de Unix : *Crontab* pour les tâches récurrentes et les commandes simplifiées *At* pour les ponctuelles et *Batch* pour démarrer le travail directement (X/Open, 1994).

Il faut signaler que Linux n'est pas un système d'exploitation spécialement conçu pour ce genre de tâches et qu'il est préférable alors d'utiliser des technologies plus adaptées comme celles des *mainframes* ou des *supercomputers* qui peuvent potentiellement utiliser des variantes de Linux (Padua, 2011).

Les types interactifs

Débuts

Au début, les ordonnanceurs interactifs de Linux se limitaient au respect de la norme *POSIX* qui définit : SCHED_FIFO, SCHED_RR, SCHED_SPORADIC (apparu en 2008, sorte de temps réel) et SCHED_OTHER (libre à l'implémentation) (IEEE, 2013). Et plus précisément, les premières versions de Linux (1.2) semblent utiliser par défaut l'un des algorithmes les plus classiques d'ordonnancement qu'est le *round robin with circular runqueue* (Maxwell, 1999; Beck et al., 1996). Les classes d'ordonnancements de *POSIX* ne seront introduites réellement que dans le noyau 2.2 en même temps que le support du SMP (Molloy and Honeyman, 2001).

L'algorithme consiste à attribuer un quantum de temps fixe à chaque processus. Celui-ci peut être entièrement consommé ou non, s'il s'arrête ou se bloque. L'ordonnanceur sélectionne alors le processus suivant dans ceux étant prêts à être exécuté, chargés en mémoire et préemptés dans la file. Si aucun n'est éligible, le processeur attend jusqu'à la prochaine interruption (Corbató et al., 1962). Le problème est de déterminer un quantum de temps idéal afin que le *context switch* ne soit pas trop pénalisant si le délai est trop court ou pas assez réactif si trop long. En outre, il répond difficilement à la notion de *burst* accordé à des processus qui ne font qu'attendre (Bach, 1986). Il a l'avantage d'être rapide et relativement simple à implémenter.

$O(n)$

Utilisé dans les versions 2.4 à 2.6 du noyau Linux, cet ordonnanceur divise le temps processeur en "époque". Dans chacune des ces époques, chaque tâche peut être exécutée jusqu'à la fin de son quantum de temps. Si elle ne l'utilise pas complètement, la moitié du temps restant est reporté pour l'époque suivante. Les processus sont sélectionnés sur base d'une métrique déterminée par l'implémentation et la priorité de chaque tâche exécutable est déterminée lors d'un changement de contexte (Bovet and Cesati, 2005). Cet ordonnanceur doit donc l'origine de son nom au fait que chaque priorité associée à un processus est réévaluée lors de l'appel à l'algorithme. Il était donc peu efficace et surtout passait difficilement à l'échelle. Il éprouve des difficultés au-delà de 100 threads actifs (Nieh et al., 2001).

$O(1)$

Afin de régler les problèmes associés à l'ordonnanceur $O(n)$, Ingo Molnár a introduit son $O(1)$ au noyau 2.6, visant à minimiser le surcoût lié au changement de contexte. À cette époque, beaucoup de changements ont été apportés au noyau. Plusieurs travaux ont été effectués afin de modifier l'architecture du système d'ordonnancement et de certains problèmes liés au *Big Kernel Lock* (Bryant and Hawkes, 2003) qui empêchaient la préemption dans certaines parties du noyau. À noter également que dans la version 2.6.16, a été rajouté SCHED_BATCH et dans la 2.6.23, SCHED_IDLE qui sont respectivement des politiques intensives et modérées appliquées au SCHED_OTHER (Pabla, 2009).

L'algorithme en lui-même consistait en une série de *runqueue* constituée de deux listes : l'une contenant les processus actifs et l'autre, ceux étant expirés. Chaque processus reçoit un quantum fixe de temps, déterminé par la priorité associée à ce genre de tâches, après lequel il est transféré vers l'autre liste. Lorsque la liste est vidée de tous ceux en attentes d'exécution, les pointeurs vers listes sont échangés et le travail reprend (Love, 2010).

L'algorithme portera mieux son nom $O(1)$ après le noyau 2.6.8.1 qui utilisera les fameuses *runqueue* de 0 à 140 et des *priority arrays*. Son interactivité sera également amélioré par Con Kolivas (Aas, 2005).

Le problème de cet ordonnanceur est qu'il est souvent difficile de différencier les tâches interactives des non-interactives. En effet, on peut essayer d'identifier les processus interactifs en analysant leur temps moyen d'inactivité (sommeil, attente, ...) ; ceux ayant un grand temps étant probablement interactifs. Il est alors possible de leur accorder un bonus de priorité, ce qui peut potentiellement déséquilibrer l'équité (Wong et al., 2008a).

CFS

Enfin, dans le noyau 2.6.23, est incorporé l'implémentation, réalisée par Con Kolivas, de ce qui s'appelait le *Fair-share scheduling* au doux nom de *Rotating Staircase Deadline* et *Staircase Process*, inspiré du *Completely Fair Scheduler* de Molnár (Torvald, 2007).

L'algorithme utilise un arbre rouge-noir pour répartir le temps entre les différentes tâches et stocke le temps effectué par chacun dans les *sched_entities*, qui sont héritées de *task_struct* sorte de descripteur de processus, apparu avec le patch pour le nouvel ordonanceur (Pabla, 2009). Ces nœuds sont indexés par le temps d'exécution processeur en nanosecondes. Un temps maximum est également calculé pour chacun des processus et qui est déterminé comme étant celui correspondant à la répartition la plus équitable entre tous les processus. Ce temps est défini comme étant égal au temps attendu avant d'être exécuté divisé par le nombre de processus. En effet, si il y a N processus, chacun devrait obtenir un N^e du temps (Pawar and Patil, 2014).

Lorsque l'algorithme d'ordonnancement est appelé, ces actions sont effectuées :

1. Le nœud le plus à gauche est choisi (le plus petit temps d'exécution) et est exécuté.
2. Si le processus s'achève, il est retiré du système et de l'arbre.
3. Si le processus fini son quantum de temps ou est interrompu (volontairement ou à cause d'une interruption), il est réintroduit dans l'arbre en fonction de son nouveau temps d'exécution.

4. Enfin, on recommence la procédure.

Si le processus passe beaucoup de temps à dormir, alors son temps passé est faible et recevra donc automatiquement une augmentation en priorité lorsqu'il en aura besoin. Par conséquent, de telles tâches n'obtiennent pas moins de temps processeur que celles qui fonctionnent continuellement (Wong et al., 2008b).

L'idée du CFS est dérivée du *Weighted Fair Queueing* utilisé par les ordonnanceurs réseau qui s'occupent de la distribution des paquets de données (Demers et al., 1989). La conversion vers la gestion des processus s'est effectuée au travers du *stride scheduling*. En effet, celui-ci propose un mécanisme d'ordonnancement des ressources déterministes qui calcule un intervalle de temps ou *stride* pendant lequel chaque client attend et qui est également proportionnel à sa propre consommation des ressources (Waldspurger, 1995).

Le CFS a reçu un patch en novembre 2010 (2.6.38) pour le rendre plus équitable sur les stations de travail et bureaux, appuyé par des tests de performances effectués plus tôt (Jose et al., 2014). Sur une idée de Torvalds et les travaux de Galbraith, il regroupe des processus afin de mieux servir les applications multi-thread (Wong et al., 2008b). Depuis, il semble être plus équitable et plus efficace que l'ordonnanceur $O(1)$ (Wong et al., 2008a).

BFS : Le CFS a eu un petit frère également développé par Con Kolivas et dénommé *Brain Fuck Scheduler* parce que, selon les mots de son créateur, il brise les codes actuels de design en matière de mise à l'échelle. Il a pour but d'être le concurrent du CFS sur des modèles multicœurs (Kolivas and Torvald, 2010). Il n'est pas un ordonnanceur officiel, en ce sens qu'il n'est pas incorporé au noyau Linux (Kolivas, 2009). Malgré le fait qu'il offre des meilleurs temps de réaction que le CFS et est parfois plus rapide que celui-ci pour accomplir certaines tâches, il a dû mal à passer à l'échelle ou à fonctionner sur des systèmes à accès mémoire non-uniforme (NUMA). Il est surtout destiné à un usage de type bureautique sur des architectures actuelles (Taylor Groves, 2009).

Les types temps réel

Linux n'est pas un système d'exploitation conçu pour les tâches temps réelles mais propose quand même certaines notions notamment au niveau des ordonnanceurs qualifiés de *Soft real-time* où une tolérance est permise au niveau des échéances. L'idée du temps réel est de diviser le travail en petites tâches prédictibles et connues à l'avance afin de pouvoir quantifier le temps nécessaire à leur réalisation et donc permettre une meilleure prédictabilité. Le but des ordonnanceurs est dès lors de s'assurer que chaque processus puisse respecter ses propres limites temporelles (Stankovic et al., 2012).

Il y a également une distinction effectuée entre les tâches dites périodiques et apériodiques qui représentent respectivement des actions étant réalisées de manière récurrente et celles surgissant de manière non prédictible, sporadiques. Dès lors, on peut se demander si les objectifs peuvent être respectés et on établit une condition de faisabilité afin de prouver l'efficacité de cet ordonnanceur (Baruah et al., 1990). Énormément de travaux ont été effectués sur ce sujet (Sha et al., 2004).

Début FIFO et RR

Linux se conforme globalement à norme *POSIX*, nottament avec ces politiques temps réel à priorité fixe (SCHED_FIFO & SCHED_RR). Ces algorithmes sont relativement efficaces lorsqu'il y a peu de tâches mais ont du mal à passer à l'échelle. Le FIFO minimise le temps de réponse de l'ensemble des tâches et toutes les tâches partagent le même temps maximal. Il n'est pas optimal pour des tâches ayant des échéances différentes et peut créer des inversions de priorités (Klein et al., 1993). Le Round Robin est théoriquement plus équitable mais il risque d'être déséquilibré si les tâches ont des délais différents, de plus il n'est pas optimal même si des améliorations ont été proposées (Shreedhar and Varghese, 1995).

Les premières grandes avancées sur cette problématique dans le noyau Linux débutèrent avec la version 2.4.18, et ses algorithmes *Constant Bandwidth Server* (Abeni and Buttazzo, 1998) et *Greedy Reclamation of Unused Bandwidth* (Lipari and Baruah, 2000) qui réclament les ressources non utilisées du processeur. Le problème à cette époque était le manque de support multicœur et la difficulté de porter vers des versions différentes du noyau. Il fallait également se défaire du *Big Kernel Lock*. Notons, que des politiques spéciales *SCHED_SOFTRR* et *SCHED_ISO* sont apparues et permettent aux utilisateurs privilégiés de faire tourner les tâches en temps réel jusqu'à usage complet du processeur (Scordino and Lipari, 2006).

Earliest deadline first

Le *SCHED_DEADLINE* apparaît officiellement avec la version 3.14 du noyau. Il est issu d'un travail de plusieurs années et est davantage connu comme une implémentation du Earliest Deadline First (Faggioli et al., 2009a). Il vise à remplacer les poliques *RR et FIFO* qui ne correspondent pas tout à fait aux attentes des tâches temps réelles (Buttazzo, 2011). L'idée sous-jacente à cet algorithme est de travailler avec une file à priorité ordonnée par rapport au temps avant l'échéance. Des résultats théoriques ayant été obtenus auparavant suite aux travaux de Liu et Layland (Liu and Layland, 1973).

L'algorithme en lui-même : On appelle q_i le temps restant d'éxécution (le temps processeur que la tâche peut utilisée avec sa période T_i et Q_i sa fin), et d_i une échéance qui permet d'assigner dynamiquement la priorité (Lelli and Faggioli, 2015).

Au début, q_i et d_i valent 0. Quand une tâche se réveille à l'instant t, l'ordonnanceur regarde si son d_i est toujours valable (soit si $q_i < (d_i - t) * Q_i/T_i$), sinon il redéfinit $d_i = t + T_i$ et $Q_i \mathrel{+}= q_i$. Quand une tâche s'exécute, son q_i est diminué du temps effectué. Si q_i arrive à 0, la tâche est suspendue et ne peut être sélectionnée par l'ordonnanceur pour exécution (parce qu'il ne respecterait pas alors le Q_i sur une période T_i). La tâche est reprise seulement au temps $t = d_i$, son q_i sera alors égale à Q_i et son échéance postposée à $d_i \mathrel{+}= T_i$.

Cependant ce genre d'algorithmes est généralement conçu pour des processeurs monocœurs. Lorsque plusieurs processus peuvent être exécutés en même temps, on distingue deux cas d'ordonnanceur (Faggioli et al., 2009b) : les globaux et les partitionnés (et des hybrides). Dans le global, toutes les tâches sont fournies à la même structure de donnée et tournent sur les différents cœurs alors que dans le partionné, différentes listes sont maintenues pour chaque processeur. Ils ont chacun leurs avantages et inconvénients et il y a beaucoup de littérature sur ce sujet (Bastoni et al., 2010; Lelli et al., 2012). En l'occurrence, Linux utilise une file de tâches distribuée, c'est à dire que chaque processeur à sa propre file mais que des tâches peuvent être transférées de l'une vers l'autre.

L'un des plus gros problèmes du temps réel est lié aux ressources partagées (Buttazzo, 2011). En effet, les sections critiques peuvent entraîner une grande surcharge de travail, ou des risques d'inversions de priorité. On améliore généralement ces algorithmes avec ce qui s'appelle *deadline inheritance over shared resources* ou *DFP* (Jansen et al., 2003). Il existe également des *serveurs* qui visent à améliorer l'utilisation sur les différents cœurs en utilisant les ressources au maximum, en équilibrant au mieux la charge. Notamment, le *Bandwidth Inheritance algorithm*, extension du *Constant Bandwidth Server* (Abeni and Buttazzo, 1998) intégré au noyau Linux en même temps que le *EDF*. Ou encore un de ses enfants, le *Greedy Reclamation of Unused Bandwidth* (Lipari and Baruah, 2000). Citons également, le *Barbershop Load Distribution algorithm* de Rakib Mullick qui tendrait à mieux répartir les tâches en fonction de la charge présente sur les cœurs (Brown, 2012).

Discussion

Nous avons vu la situation chez Linux mais qu'en est-il pour d'autres grands systèmes d'exploitation ? Ont-ils opté pour les mêmes choix d'ordonnancement ? Si non, quelles en sont les raisons ?

Microsoft

Chez Microsoft, depuis Windows NT, un ordonnanceur avec plusieurs files à priorités est employé. Il correspond plus ou moins à l'idée du $O(n)/O(1)$ mais avec 32 priorités dont 16 concernent les opérations relatives au *soft real-time* et la priorité 0 au système de nettoyage des pages mémoires inutilisées (Jones and Regehr, 1999). Nottons que cet ordonnanceur n'effectue pas de distinctions entre les tâches d'un même programme et les programmes. Si un exécutable A possède 10 *threads* et B en détient 2, chaque *thread* recevra théoriquement un douzième du temps imparti. À remarquer qu'il existe plus de status différents possibles pour les tâches comparativement à Linux et que depuis Vista, le registre qui comptabilise le nombre de cycles effectués par un programme est employé (Russinovich and Solomon, 2009). Il existe également, depuis Windows 7, la possibilité d'utiliser son propre ordonnanceur pour ses tâches appelé *User-Mode Scheduling* (Windows7, 2016).

Apple

Chez Apple, depuis Mac OS X, la situation est analogue à celle de Linux, suite à leur certification à la norme *POSIX*. À la différence près qu'un ordonnanceur à base de files à priorités est employé par défaut. La borne pour les priorités associées à chaque tâche est très élevée et permet donc plus de finesse. Il existe quatre subdivisions dans les priorités : les normales, les élevées, les noyaux et les temps réels (Singh, 2006).

BSD

Chez FreeBSD, depuis la version 7.1, un ordonnanceur dénomé *ULE* est employé. Il a remplacé un *Round-robin* qui se faisait vieillissant et qui ne supportait pas les nouveautés matérielles (SMT / SMP). Le nouveau, *ULE*, emploie trois files de tâches exécutables : deux sont réservés à tous les processus courants et une à ceux en attente. À la différence de bon nombre d'ordonnanceurs, il réagit au travers d'événements et non par le biais de quantums de temps (Roberson, 2003). L'équité et l'absence de famine sont assurées par le fait que les files sont parcourues entièrement en fonction de leur priorité. Cette priorité est calculée en fonction du temps réellement employé dans l'intervalle accordé. Ce système se rapproche des files à priorités mais avec des différences notables (McKusick et al., 2014).

Analyse

Nous voyons que, parmi ces exemples, les ordonnanceurs employés par ces systèmes d'exploitation sont basés sur les modèles des files à priorités et non celui d'un arbre. On peut donc se demander ce qui justifie une telle différence. Linux a abandonné les files au profit du CFS basé davantage sur sur l'équité et sur le modèle théorique du *stride scheduling*. Si on se base sur les résultats des articles de Wong, Tan, Kumari et Lam (Wong et al., 2008a,b), le CFS permet de faire diminuer la variance dans le temps mis pour accomplir une tâche. Le $O(1)$ donne un résultat plus étendu sur la plage des temps, il offre donc une moins bonne équité. La différence d'interactivité n'est presque pas perceptible mais donne un très léger avantage au $O(1)$. Wang, Chen, Jiang et Li (Wang

et al., 2009) et Pabla (Pabla, 2009) arrivent à peu près aux mêmes conclusions.

Si les performances de ce système sont analogues en permettant une meilleure équité, qu'en est-il de sa mise à l'échelle ? Si on se réfère à l'article Li, Baumberger et Hahn (Li et al., 2009) qui propose une comparaison de différents type d'ordonnancement, le CFS est analogue au WFQ par design et possède donc une complexité en $O(logN)$ avec une latence positive constante mais dont la latence négative est borné par $O(n)$ (Parekh and Gallager, 1993). A contrario, les ordonnanceurs basés sur des files à priorités possèdent une complexité en $O(1)$, mais ils ont une plus faible équité avec une latence bornée par $O(n)$ en général. Seulement, si les poids des tâches sont bornés par une constante, ce qui est une hypothèse qui peut être appliquée à la pratique, alors la latence tant positive que négative peut être bornée par une constante (Shreedhar and Varghese, 1995).

Ces résultats théoriques peuvent être altérés par des détails d'implémentation qui les rendraient plus lents ou, au contraire, plus rapides. Seulement, on doute que des géants tels que Microsoft optent pour un système moins performant. Signalons qu'il y a encore des problèmes de performances dans les dernières versions du noyau Linux, nottament vis à vis de NUMA (Lozi et al., 2016; Blagodurov et al., 2010) qui peuvent être expliqués par la difficulté d'étudier la régression chez les ordonnanceurs (Chen et al., 2007) et que la nécessité de développer des outils de tests approfondis est plus que cruciale (Erickson et al., 2010).

Conclusion

Implémenter un ordonnanceur n'est pas tâche aisée, et le rendre correct d'un point de vue formel est encore plus difficile. Nous avons vu que la route fut longue pour arriver à l'état actuel qui est encore loin d'être parfait. Des concepts pourtant simples peuvent se retrouver mis à mal face à la complexité et la diversité des matériels disponibles. Le problème, même si bon nombres de résultats théoriques ont déjà été obtenus, demeure ouvert et qu'il faut rester réactif face aux nouveautés matérielles apportées par les constructeurs de matériel.

Références

Aas, J. (2005). Understanding the linux 2.6.8.1 cpu scheduler. *SGI, 2005*, 22 :05.

Abeni, L. and Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–, Washington, DC, USA. IEEE Computer Society.

Bach, M. J. (1986). *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Baruah, S., Mok, A., and Rosier, L. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190.

Bastoni, A., Brandenburg, B. B., and Anderson, J. H. (2010). An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14–24. IEEE.

Beck, M., Bohme, H., Kunitz, U., Magnus, R., Dziadzka, M., and Verworner, D. (1996). *Linux Kernel Internals*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Blagodurov, S., Zhuravlev, S., Fedorova, A., and Kamali, A. (2010). A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 557–558. ACM.

Bovet, D. and Cesati, M. (2005). *Understanding The Linux Kernel*. Oreilly & Associates Inc.

Brown, Z. (2012). Zack's kernel news. *Linux Magazine*.

Bryant, R. and Hawkes, J. (2003). Linux scalability for large numa systems. In *Linux Symposium*, page 76.

Buttazzo, G. C. (2011). *Hard real-time computing systems : predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media.

Casavant, T. L. and Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2) :141–154.

Chen, T., Ananiev, L. I., and Tikhonov, A. V. (2007). Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102.

Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. (1962). An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 335–344. ACM.

Demers, A., Keshav, S., and Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4) :1–12.

Erickson, J., Musuvathi, M., Burckhardt, S., and Olynyk, K. (2010). Effective data-race detection for the kernel. In *OSDI*, volume 10, pages 1–16.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11) :624–633.

Faggioli, D., Checconi, F., Trimarchi, M., and Scordino, C. (2009a). An edf scheduling class for the linux kernel. In *Proc. of the Real-Time Linux Workshop*.

Faggioli, D., Trimarchi, M., Checconi, F., Bertogna, M., and Mancina, A. (2009b). An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1984–1989. ACM.

Hansen, P. B. (1973). *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

IEEE (2013). Standard for information technologyportable operating system interface (posix(r)) base specifications, issue 7. *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)*, pages 1–3906.

Jansen, P. G., Mullender, S. J., Havinga, P. J., and Scholten, H. (2003). Lightweight edf scheduling with deadline inheritance.

Jones, M. B. and Regehr, J. (1999). Cpu reservations and time constraints : Implementation experience on windows nt. In *Proc. of the 3rd USENIX Windows NT Symposium*, pages 93–102. Citeseer.

Jose, J., Sujisha, O., Gilesh, M., and Bindima, T. (2014). On the fairness of linux o(1) scheduler. In *Intelligent Systems, Modelling and Simulation (ISMS), 2014 5th International Conference on*, pages 668–674.

Klein, M. H., Ralya, T., Pollak, B., Obenza, R., and Harbour, M. G. (1993). *A Practitioner's Handbook for Real-time Analysis*. Kluwer Academic Publishers, Norwell, MA, USA.

Kolivas, C. (2009). Con kolivas introduces new bfs scheduler. *Linux Magazine*, 109.

Kolivas, C. and Torvald, L. (2010). Faqs about bfs. v0.330.

Lelli, J., Faggioli, D., Cucinotta, T., and Lipari, G. (2012). An experimental comparison of different real-time schedulers on multicore systems. *Journal of Systems and Software*, 85(10) :2405–2416.

Lelli, J., S. C. A. L. and Faggioli, D. (2015). Deadline scheduling in the linux kernel. *Softw. Pract. Exper.*

Li, T., Baumberger, D., and Hahn, S. (2009). Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *ACM Sigplan Notices*, volume 44, pages 65–74. ACM.

Lipari, G. and Baruah, S. (2000). Greedy reclamation of unused bandwidth constant-bandwidth servers. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, Euromicro-RTS'00, pages 193–200, Washington, DC, USA. IEEE Computer Society.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61.

Love, R. (2010). *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition.

Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. (2016). The linux scheduler : a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems to appear, (EuroSys 2016), London, United Kingdom*.

Maxwell, S. (1999). *Linux Core Kernel Commentary*. Coriolis Group Books, Scottsdale, AZ, USA.

McKusick, M. K., Neville-Neil, G. V., and Watson, R. N. (2014). *The design and implementation of the FreeBSD operating system*. Pearson Education.

Molloy, S. P. and Honeyman, P. (2001). Scalable linux scheduling. In *Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference*, pages 199–212, Berkeley, CA, USA. USENIX Association.

Méndez, C. A., Cerdá, J., Grossmann, I. E., Harjunkoski, I., and Fahl, M. (2006). State-of-the-art review of optimization methods for short-term scheduling of batch processes. *Computers & Chemical Engineering*, 30(6-7) :913–946.

Nieh, J., Vaill, C., and Zhong, H. (2001). Virtual-time round-robin : An o(1) proportional share scheduler. In *Proceedings of the General Track : 2001 USENIX Annual Technical Conference*, pages 245–259, Berkeley, CA, USA. USENIX Association.

Pabla, C. S. (2009). Completely fair scheduler. *Linux Journal*, 2009(184).

Padua, D., editor (2011). *Encyclopedia of Parallel Computing*. Springer, 1 edition.

Parekh, A. K. and Gallager, R. G. (1993). A generalized processor sharing approach to flow control in integrated services networks : the single-node case. *IEEE/ACM Transactions on Networking*, 1(3) :344–357.

Pawar, Prajakta, D. S. and Patil, S. (2014). Illustration of completely fair scheduler for task scheduling using aa tree. *International Journal of Scientif Engineering and Technology Research*, 03(44) :8911–8914.

Roberson, J. (2003). Ule : A modern scheduler for freebsd. In *BSDCon*, pages 17–28.

Russinovich, M. and Solomon, D. A. (2009). *Windows Internals : Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th edition.

Scordino, C. and Lipari, G. (2006). Linux and real-time : Current approaches and future opportunities. In *ANIPLA International Congress, Rome, 2006*.

Sha, L., Abdelzaher, T., årzén, K.-E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., and Mok, A. K. (2004). Real time scheduling theory : A historical perspective. *Real-Time Syst.*, 28(2-3) :101–155.

Shreedhar, M. and Varghese, G. (1995). Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev.*, 25(4) :231–242.

Singh, A. (2006). *Mac OS X internals : a systems approach*. Addison-Wesley Professional.

Stankovic, J. A., Spuri, M., Ramamritham, K., and Buttazzo, G. C. (2012). *Deadline scheduling for real-time systems : EDF and related algorithms*, volume 460. Springer Science & Business Media.

Tanenbaum, A. S. and Woodhull, A. S. (2005). *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Taylor Groves, Jeff Knockel, E. S. (2009). Bfs vs cfs – scheduler comparison.

Torvald, L. (2007). Linux [patch] modular scheduler core and completely fair scheduler [cfs].

Waldspurger, C. A. (1995). Lottery and stride scheduling : Flexible proportional-share resource management. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA.

Wang, S., Chen, Y., Jiang, W., Li, P., Dai, T., and Cui, Y. (2009). Fairness and interactivity of three cpu schedulers in linux. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 172–177.

Wang, Y. and Morris, R. J. T. (1985). Load sharing in distributed systems. *IEEE Trans. Computers*, 34(3) :204–217.

Windows7 (2016). User-mode scheduling.

Wong, C., Tan, I., Kumari, R., Lam, J., and Fun, W. (2008a). Fairness and interactive performance of o(1) and cfs linux kernel schedulers. In *Information Technology, 2008. ITSim 2008. International Symposium on*, volume 4, pages 1–8.

Wong, C. S., Tan, I., Kumari, R. D., and Wey, F. (2008b). Towards achieving fairness in the linux scheduler. *SIGOPS Oper. Syst. Rev.*, 42(5) :34–43.

X/Open, C. (1994). *X/Open CAE Specification : Commands and utilities, issue 4, version 2*. Number n.4 in 1. X Open Company.