Implementasi *Completely Fair Scheduler* dengan *Red-Black Tree* pada Kernel Linux

Agung Baptiso Sorlawan, 13513043[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13513043@std.stei.itb.ac.id

Abstak—Impementasi pohon pencarian biner sebagai sturktur data sangatlah penting dalam *computer science*. Salah satu aplikasinya adalah *scheduler* khususnya *scheduler* pada kernel Linux, yang menggunakan *red-black tree*. Alasan penggunaan *red-black tree* adalah karena *red-black tree* memiliki *worst case* kompleksitas waktu asimptotik yang sangat cepat yakni O(log n). Bagaimana ?

Kata Kunci—scheduler, kernel, red-black tree, completely fair scheduler.

I. PENDAHULUAN

Completely Fair Scheduler (CFS) adalah sebuah penjadwal (*scheduler*) proses yang pertama kali dikenalkan pada Linux 2.6.23 dan menjadi *default scheduler* untuk Linux. CFS menangani alokasi *resource* untuk melakukan eksekusi proses, dan bertujuan untuk meningkatkan efektifitas penggunaan CPU secara keseluruhan.

Pada Linux, *task* adalah sebuah proses atau *thread*. Jadi secara umum tugas CFS menentukan kapan sebuah *task* akan dieksekusi pada CPU.

Sebelum CFS dikenalkan pada Linux, *tasks* disimpan pada sebuah struktur data *queue.* CFS menggunakan *Red-Black Tree (rbtree)* atau Pohon Merah-Hitam yang diurutkan berdasarkan waktu eksekusi. Pohon Merah-Hitam memiliki *worst case* kompleksitas waktu asimptotik O(log n). Sehingga cocok untuk perangkat lunak yang *time-sensitive* atau *real-time*. *Scheduler* pada Linux menggunakan unit waktu *nanosecond* sehingga, Pohon Merah-Hitam menjadi pilihan struktur data yang tepat.

II. LANDASAN TEORI

A. *Process Scheduler*

Process Scheduler (penjadwal proses) adalah bagian dari sistem operasi yang menentukan proses mana yang akan dieksekusi pada waktu tertentu. Berdasarkan tingkat kooperasinya, penjadwal proses dibagi menjadi *preemptive* dan *cooperative scheduler*. Penjadwal yang *preemptive,* apabila sebuah proses terjadwal akan berjalan, dan harus menghentikan sementara proses yang sedang berjalan, maka proses yang sedang berjalan tersebut akan dihentikan tanpa persetujuan/kooperasi dari proses tersebut. Sementarai pada penjadwal yang *cooperative*, sistem operasi tidak pernah menginisiasi terjadinya pergantian eksekusi proses (*context switch*), tetapi sedang berjalan memberikan kontrol yang memungkinkan terjadinya *context switch.* Linux menggunakan penjadwal yang *preemptive.*

B. Pohon Merah-Hitam

Pohon Merah-Hitam (PHM) adalah sebuah jenis pohon pencarian biner yang *self-balanced*. *Self-balanced* berarti pada proses penambahan atatu penghapusan *node,* pohon secara otomatis melakukan *balancing* untuk menjaga tinggi pohon sekecil mungkin.

Setiap *node* pada PHM memiliki sebuah bit tambahan. Bit tersebut merepresentasikan warna dari *node* terebut (merah atau biru). Warna - warna ini juga akan dijaga keseimbangannya pada saat *balancing.*

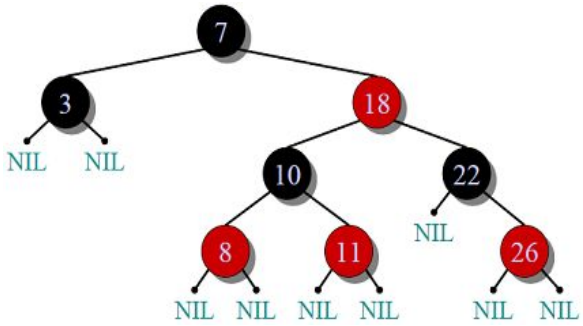
Balancing memastikan pencarian pada PHM memiliki kompleksitas asimptotik waktu rata - rata O(log n), dan *worst case* yang sama yakni O(log n). Operasi penambahan dan penghapusan *node* juga memiliki kompleksitas asimptotik waktu rata - rata dan *worst case* O(log n).

Pada saat modifikasi pohon, PHM melakukan proses *balancing* untuk memastikan beberapa kondisi tetap terjaga. Adapun kondisi - kondisi tersebut adalah:

1. Setiap *node* dapat berwarna merah atau hitam
2. Akar berwarna hitam. Kondisi ini dapat diabaikan, karena tidak berpengaruh pada analisis.
3. Semua daun berwarna hitam dan nilainya adalah NIL.
4. Apabila sebuah *node* berwarna merah, maka kedua anaknya harus berwarna hitam
5. Setiap lintasan dari *node* manapun ke keturunannya yang merupakan daun (bernilai

NIL), harus melewati jumlah *node* berwarna hitam yang sama.

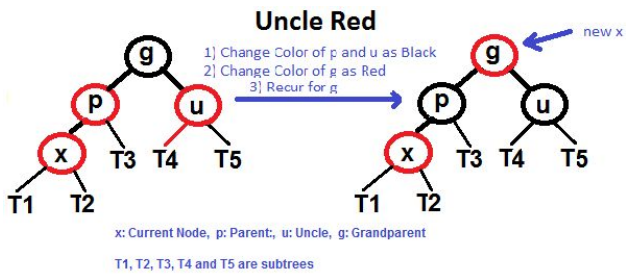
Kondisi - kondisi ini memastikan sebuah batasan pada PHM: panjang lintasan dari akar ke daun terjauh tidak lebih dari dua kali panjang lintasan dari akar ke daun terpendek. Karena operasi pencarian, penambahan, dan penghapusan memerlukan waktu *worst case* yang proporsional terhadap tinggi pohon, maka dapat disimpulkan bahwa operasi pada PHM efektif pada skenario *worst case.*



Gambar 2.1. Contoh Pohon Merah-Hitam
Sumber:www.geeksforgeeks.org/red-black-tree-set-1-introduction
(akses: 7 Desember 2016)

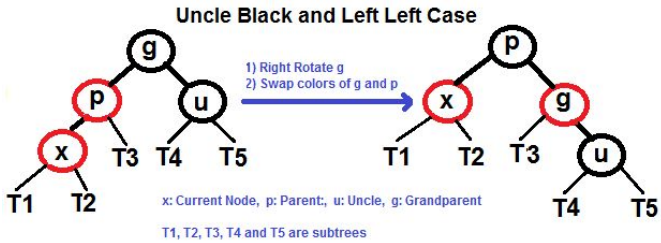
Berikut ini adalah operasi - operasi yang dapat dilakukan pada PHM, dan algoritma *balancing*nya:

1. Penambahan (*insertion*)
   Algoritma penambahan node pada PHM:
   1) Lakukan proses penambahan node seperti pada pohon pencarian biner biasa (dengan membandingkan nilai node yang ditambahkan pada pohon). Warnai node x yang baru dengan warna **merah**.
   2) Apabila x merupakan node pertama/akar, ganti warna x menjadi **hitam.**
   3) Langkah - langkah selanjutnya, tergantung warna dari *uncle node* (*sibling* dari *parent*).
      a) uncle node berwarna merah
         i) ganti warna *parent* dan *uncle* menjadi **hitam**
         ii) ganti warna *grandparent* menjadi **merah**
         iii) x = *grandparent*. Iterasi ulang dari langkah 2) untuk x yang baru.



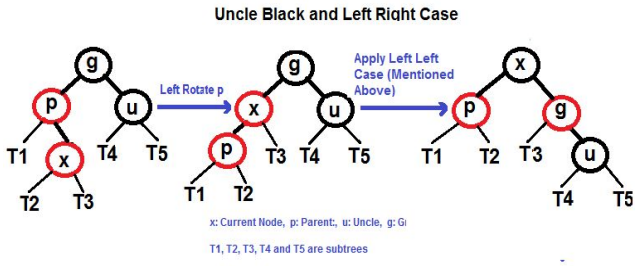
Gambar 2.2. Penambahan node untuk *uncle node* yang berwarna merah
Sumber:www.geeksforgeeks.org/red-black-tree-set-2-insert/
(akses: 7 Desember 2016)

b) uncle node berwarna hitam. Algoritma untuk kasus *uncle node* berwarna hitam, bergantung pada *parent node(p)*, *grandparent node(g)*.
   i) p adalah *left child* g, dan x adalah *left child* p (kasus *left left*)
      (1) right rotate g
      (2) tukar warna g dengan p



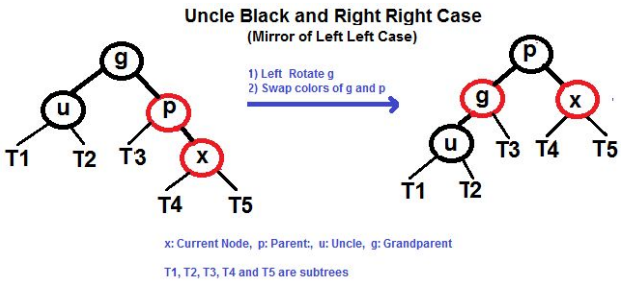
Gambar 2.3. Penambahan node untuk *uncle node* berwarna hitam dan kasus *left left*
Sumber:www.geeksforgeeks.org/red-black-tree-set-2-insert/
(akses: 7 Desember 2016)

   ii) p adalah *left child* g, dan x adalah *right child* dari p (kasus *left right*)
      (1) left rotate p
      (2) terapkan kasus *left left* pada x



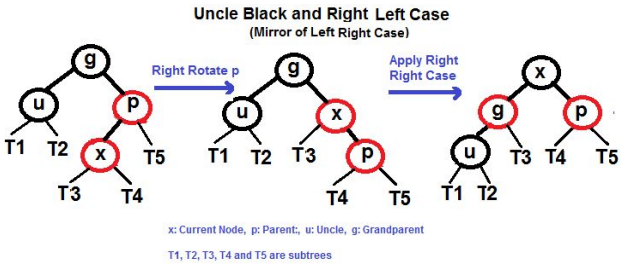
Gambar 2.4. Penambahan node untuk *uncle node* berwarna hitam dan kasus *left right*
Sumber:www.geeksforgeeks.org/red-black-tree-set-2-insert/
(akses: 7 Desember 2016)

   iii) p adalah *right child* g, dan x adalah *right child* p (kasus *right right*)
      (1) left rotate g
      (2) tukar warna g dengan p



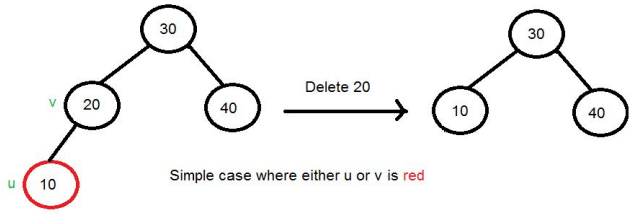
Gambar 2.5. Penambahan node untuk *uncle node* berwarna hitam dan kasus *left right*
Sumber:www.geeksforgeeks.org/red-black-tree-set-2-insert/
(akses: 7 Desember 2016)

iv) p adalah *right child* g, dan x adalah *left child* dari p (kasus *right left*)
(1) right rotate p
(2) terapkan kasus *right right* pada x



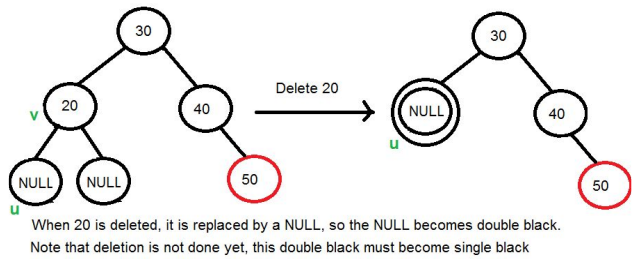
Gambar 2.6. Penambahan node untuk *uncle node* berwarna hitam dan kasus *left right*
Sumber:www.geeksforgeeks.org/red-black-tree-set-2-insert/
(akses: 7 Desember 2016)

2. Penghapusan (*deletion*)
Algoritma penghapusan node pada PHM:
1) Lakukan penghapusan node seperti pada pohon pencarian biner biasa. Untuk penghapusan terebut kita selalu berakhir menghapus sebuah daun atau node dengan satu anak. Untuk node internal, kita men*copy node successor* dan secara rekursif menghapus *node successor,* sehingga kita selalu berakhir pada daun atau node dengan satu anak.
Misal **v** adalah *node* yang dihapus dan **u** adalah *node* yang menggantikan.
2) Apabila u dan v salah satu berwarna merah, warnai u warna hitam. Penghapusan selesai (hentikan algoritma)



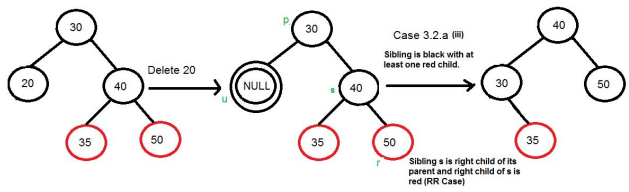
Gambar 2.7. Penghapusan node untuk untuk u atau v berwarna merah
Sumber:www.geeksforgeeks.org/red-black-tree-set-3-delete-2/
(akses: 7 Desember 2016)

3) Apabila u dan v berwarna hitam. Akan terjadi kondisi *double-black*.



Gambar 2.8. Kondisi *double-black*
Sumber:www.geeksforgeeks.org/red-black-tree-set-3-delete-2/
(akses: 7 Desember 2016)

4) Ubah kondisi *double-black* menjadi *single-black.* Misal **s** adalah sibling dari node yang dihapus. Apabila salah satu anak dari s berwarna merah, lakukan rotasi. Misal **r** adalah anak s yang berwarna merah. Maka kasus ini dapat dibagi menjadi 4 kasus:
i) Kasus *left left* (mirror dari iii dibawah)
ii) Kasus *left right* (mirror dari iv dibawah)
iii) Kasus *right right*



Gambar 2.8. Kondisi *double-black*
Sumber:www.geeksforgeeks.org/red-black-tree-set-3-delete-2/ (akses: 7 Desember 2016)

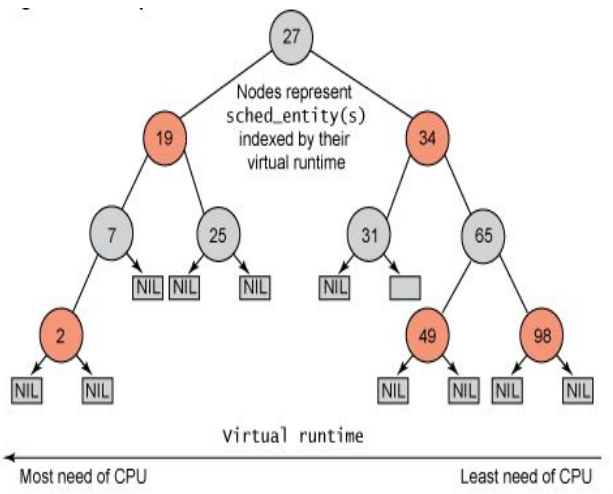
iv) Kasus *right left*

III. IMPLEMENTASI *COMPLETELY FAIR SCHEDULER* DENGAN POHON MERAH-HITAM PADA LINUX KERNEL

Seperti kita tahu, PHM cocok digunakan untuk aplikasi dengan operasi penambahan dan penghapusan node yang dilakukan secara intensif. Salah satu contohnya adalah *scheduler*, dimana *scheduler* berfungsi menentukan proses mana yang akan dieksekusi pada waktu tertentu. Selain itu *scheduler* juga menjalankan fungsi *context switch*, yakni mengganti proses yang sedang dieksekusi dengan proses yang baru.

Pada kernel Linux, *schduler* yang digunakan menggunakan algoritma *Completely Fair Scheduler* (CFS). CFS mencoba untuk adil kepada semua proses(*task*) dengan memberikan kepada mereka kesempatan yang sama untuk dieksekusi di prosesor. Untuk itu, CFS menyimpan jumlah waktu yang disediakan untuk sebuah *task* dalam bentuk *virtual runtime.* Semakin kecil *virtual runtime* sebuah *task,* semakin sedikit waktu yang diizinkan untuk *task* mengakses prosesor.

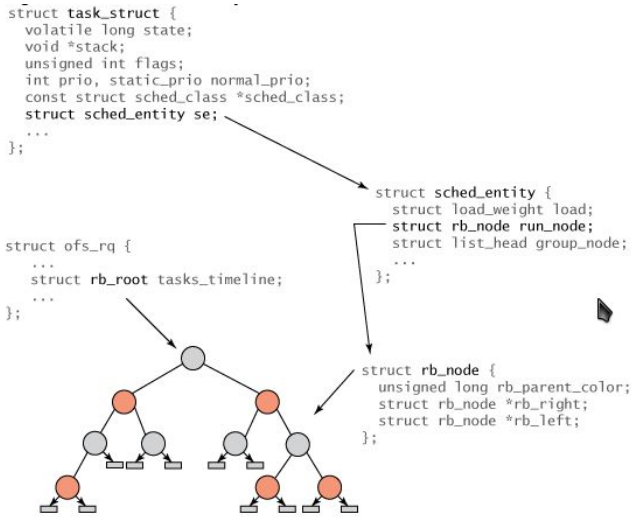
Tasks disimpan dalam Pohon Merah-Hitam yang diurutkan berdasarkan *virtual runtime* dari masing - masing *task.*



Gambar 3.1.Representasi PHM pada CFS
Sumber:www.ibm.com/developerworks/library/l-completely-fair-scheduler/
(akses: 7 Desember 2016)

Proses pada Linux kernel direpresentasikan dengan `task_struct`. *Scheduler* tidak menggunakan *struct* ini, melainkan `sched_entity` yang kemudian digunakan di dalam `task_struct`.

Pada `sched_entity` terdapat *pointer* ke *node* di PHM, dimana setiap node mengandung *pointer* ke *left child, right child,* dan warna dari *parent node*.



Gambar 3.2. Implementasi internal CFS dan hubungannya dengan PHM
Sumber:www.ibm.com/developerworks/library/l-completely-fair-scheduler/
(akses: 7 Desember 2016)

Setiap kali *scheduler* memilih *task*, ada beberapa langkah yang dilakukan:

1. Memilih *task* dengan *virtual runtime* terkecil yakni yang terletak paling bawah dan paling kiri pada PHM.
2. *Task* tersebut dihapus dari PHM.
3. Apabila terjadi *context switch*, yakni *scheduler* menjadwalkan task baru untuk menggantikan *task* yang sedang berjalan, maka nilai *virtual runtime* dari task yang sedang berjalan dinaikkan nilainya. Jumlah penambahan nilai *virtual runtime* sama dengan jumlah waktu *task* dieksekusi.
4. Task ini masuk kondisi *sleep* dan dimasukkan kembali ke PHM dengan nilai *virtual runtime* yang baru
5. Task baru yang dipilih dieksekusi.

Siklus diatas dilakukan secara terus-menerus.

IV. SIMPULAN

Implementasi *Completely Fair Scheduler* (CFS) dengan Pohon Hitam-Merah memang efisien, karena setiap operasi penambahan dan penghapusan *node* pada saat *context switch* dapat dilakukan dengan cepat.

V. UCAPAN TERIMA KASIH

Pertama - tama, penulis berterima kasih dan bersyukur kepada Tuhan Yang Maha Esa karena berkat-Nya sehingga penulis dapat menyelesaikan makalah ini dengan baik. Penulis juga mengucapkan terima kasih kepada Ibu Harlili, selaku dosen dari mata kuliah Matematika Diskrit dalam mengajarkan pokok bahasan Pohon.

DAFTAR PUSTAKA

[1] Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

[2] https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf

[3] Peter Brucker, Sigrid Knust. Complexity results for scheduling problems

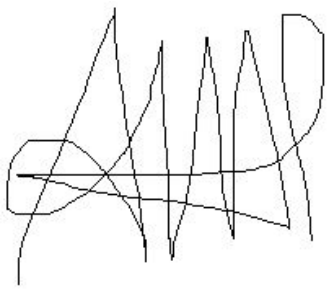
[4] Rudolf Bayer (1972). "Symmetric binary B-Trees: Data structure and maintenance algorithms". *Acta Informatica*. **1** (4): 290–306. doi:10.1007/BF00289509

[5] Leonidas J. Guibas and Robert Sedgewick (1978). "A Dichromatic Framework for Balanced Trees". *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. pp. 8–21. doi:10.1109/SFCS.1978.3.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Desember 2016



Agung Baptiso, 13513043