Algoritma *Greedy* pada Penjadwalan *Real-Time* untuk *Earliest Deadline First Scheduling* dan *Rate Monotonic Scheduling* serta Perbandingannya

Iftitakhul Zakiah/13515114
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515114@std.stei.itb.ac.id
iftitakhulzakiyah@gmail.com

***Abstrak*—*Greedy* adalah salah satu algoritma yang sering digunakan di bawah alam sadar manusia karena cenderung lebih mudah diaplikasikan dalam kehidupan sehari-hari dan juga karena relatif sederhana dibanding algoritma lainnya. Tidak hanya pada manusia, algoritma yang satu ini juga dapat digunakan pada sistem operasi untuk *scheduling* atau penjadwalan, contohnya ialah pada *Earliest Deadline First Scheduling* (EDF) dan *Rate-monotonic Scheduling.* EDF merupakan contoh algoritma yang digunakan dalam penjadwalan *real-time* dimana proses dengan periode atau *deadline* tercepat akan diprioritaskan dalam penggunaan sumber daya. Sedangkan *rate-monotonic* merupkan penjadwalan dengan memprioritaskan proses dengan *deadline* yang telah terlewati dengan harapan proses tersebut memiliki kebutuhan CPU lebih tinggi daripada proses lainnya. Penjadwalan *real-time* ini sangat dibutuhkan dalam *multiprogramming*, misalnya pada sitem operasi Linux yang mengimplementasikan penjadwalan *real-time* menggunakan POSIX *(Portable Operating System Interface)*.**

Kata kunci*—*EDF; rate-monotonic; Greedy; Penjadwalan; Periode; Proses

I. PENDAHULUAN

Algoritma *Greedy* ialah salah satu algoritma yang digunakan untuk persoalan optimasi yang menuntut pencarian solusi optimum. Algoritma ini akan memberikan solusi yang layak dengan menggunakan prinsip optimum lokal yang diharapkan dapat memberikan optimum global untuk pemecahan suatu masalah yang otomatis memberikan solusi terbaik.

Earliest Deadline First dan *Rate-monotonic* adalah algoritma penjadwalan dengan sistem *real-time* pada sistem operasi. EDF memilih suatu proses yang akan dieksekusi berdasarkan urutan *deadline*/periode yang akan datang paling cepat. Sedangkan *Rate-monotonic* memilih proses dimana jika dalam keberjalanannya sistem melewati suatu *deadline* maka proses yang akan langsung memotong proses yang sedang dijalankan CPU dan memberikan jatah CPU kepada proses yang memiliki *deadline* tersebut. Kedua sistem penjadwalan ini menggunakan algoritma *greedy* dalam memilih proses akan tetapi *greedy* yang dipilih berbeda. Jika EDF memilih *greedy by upcoming deadline*, maka *rate-monotonic* adalah *greedy by previous deadline.*

Dari segi sistem, ada beberapa jenis penjadwalan proses yaitu untuk semua, sistem *batch*, sistem interaktif, dan sistem *real-time.* Penjadwalan dalam sistem *real-time* berbeda dengan sistem *batch* maupun sistem interaktif. Sisi *real-time* lebih dapat diprediksi karena periode dan *deadline* yang diketahui oleh prosessor juga dapat memenuhi tenggat.

II. DASAR TEORI

A. ALGORITMA GREEDY

Greedy secara bahasa berarti rakus, tamak, loba, atau lainnya. Algoritma ini merupakan metode yang paling populer untuk memecahkan persoalan optimasi, yaitu persoalan yang menuntut pencarian solusi optimum dengan prinsip *"Take what you can get now!".*

Solusi optimum adalah solusi yang dapat bernilai maksimum/minimum (tergantung jenis persoalan optimasinya) dari sekumpulan alternatif solusi lain yang mungkin. Elemen yang diperhatikan untuk persoalan optimasi ada dua yaitu kendala dan fungsi objektif atau fungsi optimasi. Solusi yang memenuhi semua kendala disebut solusi layak, sedangkan solusi layak yang mengoptimumkan fungsi optimasi disebut solusi optimum.

Algoritma *greedy* membentuk solusi *step by step* atau tiap tahapnya memutuskan solusi yang akan diambil. Keputusan tersebut tidak dapat diubah lagi untuk tahap selanjutnya sehingga dapat langsung mengurangi/menyeleksi kemungkinan solusi yang dapat diambil. Pendekatan yang digunakan di dalam algoritma *greedy* adalah membuat pilihan yang terlihat memberikan hasil terbaik, yaitu dengan memberikan optimum lokal pada setiap langkah dengan harapan akan mengantarkan menuju solusi optimum global.

Algoritma *greedy* dilakukan per tahap dimana tiap tahap :

1. mengambil pilihan terbaik yang dapat diperoleh pada saat itu tanpa melihat dampak untuk ke depan (sesuai dengan prinsipnya).

2. berharap memperoleh solusi optimum global dari solusi-solusi optimum lokal yang diambil.

Algoritma *greedy* disusun dari lima elemen, yaitu himpunan kandidat, himpunan solusi, fungsi seleksi, fungsi kelayakan, dan fungsi obyektif.

1. Himpunan kandidat, ialah himpunan yang berisi elemen-elemen pembentuk solusi.
2. Himpunan solusi, ialah himpunan yang berisi kandidat-kandidat yang terpilih sebagai solusi persoalan.
3. Fungsi seleksi, fungsi yang digunakan unruk memilih kandidat yang paling memungkinkan mencapai solusi optimal. Kandidat yang telah terpilih pada suatu tahap, tidak dipilih lagi pada tahap selanjutnya.
4. Fungsi kelayakan, yaitu fungsi yang digunakan untuk memeriksa apakah suatu kandidat yang telah terpilih dapat memberikan solusi yang layak. Jika kandidat tersebut kayak, maka akan dimasukkan ke dalam himpunan solusi sedangkan kandidat yang tidak layak tidak akan dipertimbangkan lagi.
5. Fungsi obyektif, yaitu fungsi yang digunakan untuk mengoptimumkan nilai solusi.

Adakalanya optimum global merupakan solusi *pseudo-optimum* karena terdapat beberapa alasan. Pertama, algoritma *greedy* yang tidak beroperasi secara menyeluruh terhadap semua alternatif solusi yang ada. Kedua, pemilihan fungsi seleksi yang bisa berbeda sehingga harus memilih fungsi yang tepat jika ingin menghasilkan solusi yang benar-benar optimum. Jika jawaban terbaik mutlak untuk tidak ditemukan, maka algoritma *greedy* sering digunakan untuk menghampiri/ aproksimasi optimum daripada menggunakan algortima yang lebih rumit untuk menghasilkan solusi yang eksak. Algoritma *greedy* yang optimum dapat dibuktikan secara matematis.

Secara umum, skema algoritma *greedy* sebagai berikut :

```
function greedy (input C : himpunan_kandidat) → himpunan_kandidat
{Mengembalikan solusi dari persoalan optimasi dengan algoritma greedy
Masukan : himpunan kandidat C
Keluaran : himpunan solusi yang bertipe himpunan_kandidat
}
Deklarasi
   x : kandidat
   S : himpunan_kandidat
Algoritma
 S ← {} {inisiasi S dengan himpunan kosong}
 while (not SOLUSI(S) and C ≠{}) do
    x ← SELEKSI(C)    {pilih sebuah kandidat dari C}
    C ← C – {x} {elemen himpunan kandidat berkurang satu}
    if LAYAK (S U {x}) then
       S ← S U {x}
    endif
 endwhile
 {SOLUSI (S) or C = {}}
 if (SOLUSI (S)) then
    return S
 else
    output ('tidak ada solusi')
 endif
```

B. SISTEM PENJADWALAN REAL-TIME

Penjadwalan CPU merupakan basis dari sistem operasi yang menunjang *multiprogramming*. Dengan *switch* proses di CPU, sistem operasi dapat membuat komputer menjadi lebih produktif.

Dalam sistem yang memiliki satu prosesor, hanya ada satu proses yang dapat berjalan dalam satu waktu. Proses lainnya harus menunggu hingga CPU kosong dan dapat dijadwalkan ulang. Sedangkan pada sistem *multiprogramming*, beberapa proses dapat berjalan dalam satu waktu untuk meningkatkan kemanfaatan CPU. Idenya relatif sederhana, yaitu memberikan jatah CPU ke proses lain saat suatu proses sedang menunggu *I/O burst* untuk proses yang sedang dieksekusi. Setiap satu proses sedang menunggu, proses lainnya dapat menggunakan CPU. Penjadwalan semacam ini sangat fundamental dalam sistem operasi. Hampir semua sumber daya komputer telah dijadwalkan sebelum digunakan.

Skema penjadwalan sendiri ada yang *preemptive* dan *nonpreemptive*. Pada skema penjadwalan *preemptive*, proses yang sedang menggunakan CPU dapat dipotong oleh proses lain, sedangkan skema *nonpreemptive*, proses yang sedang menggunakan CPU tidak dapat dipotong oleh proses lain sehingga harus diselesaikan dari awal hingga akhir dalam sekali eksekusi.

Penjadwalan dapat dilihat dari berbagai sistem, yaitu sistem *batch*, sistem interaktif,dan sistem *real-time*. Sistem *batch* menekankan pada *troughput* dan penggunaan CPU yang maksimal juga waktu *turnaround* yang minimal. Sedangkan sistem interaktif meminimalkan waktu respon dan proporsionalitas. Untuk sistem *real-time* sendiri cenderung mudah diprediksi dan dapat memenuhi tenggat.

Secara umum, sistem *real-time* dapat dibedakan menjadi dua jenis yaitu sistem *soft real-time* dan sistem *hard real-time.* Pada *soft real-time*, tidak ada jaminan saat *critical real-time process* akan dijadwalkan. Jadi lebih menjamin *noncritical process.* Sedangkan *hard real-time system* memiliki permintaan yang kaku. Sebuah *task* harus dieksekusi sebelum *deadline*, jika *task* tersebut selesai diluar *deadline* maka sama saja tidak dieksekusi sama sekali.

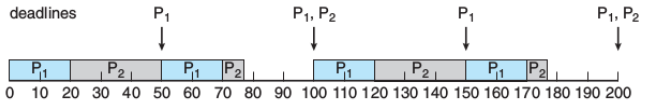
Ada beberapa jenis penjadwalan *real-time* yaitu *priority based scheduling, rate-monotonic scheduling, earliest deadline first scheduling, proportional share scheduling,* dan *POSIX real-time scheduling.*

C. RATE-MONOTONIC SCHEDULING

Algoritma *rate-monotonic scheduling* dijadwalkan secara periodik menggunakan aturan prioritas yang statik dengan *preemption.* Jika terdapat proses dengan prioritas rendah sedang menggunakan jatah CPU sedangkan ada proses lain yang berprioritas lebih tinggi sudah siap menggunakan CPU,

maka proses dengan proritas rendah akan di *switch* dengan proses berprioritas tinggi.

Pada penjadwalan ini, proses dengan periode yang lebih kecil berarti memiliki prioritas lebih tinggi dibanding proses yang memiliki periode yang lebih besar. Hal ini disebabkan untuk tiap periode proses terlewati maka prioritasnya menjadi tinggi, sedangkan jika proses tersebut telah dieksekusi dan belum melewati periode nya, maka prioritasnya menjadi rendah. Misalnya ada dua proses P1 dan P2. P1 memiliki waktu eksekusi 20 dengan periode 50, sedangkan P2 memiliki waktu eksekusi 35 dengan periode 100. Maka penggunaan CPU seperti *gant-chart* dibawah ini.



Gambar 1. Gant-chart kasus *rate-monotonic*

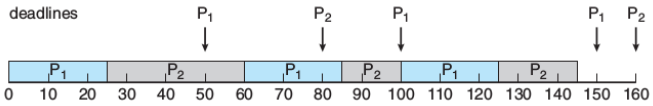
Sumber : Silberschatz et al."Operating System Concepts" 9th ed.

D. EARLIEST DEADLINE FIRST SCHEDULING

Earliest Deadline First (EDF) menjadwalkan secara dinamis membuat prioritas sesuai dengan *deadline*. *Task* atau prose yang memiliki *deadline* terdekat memiliki prioritas lebih besar daripada *task* yang memiliki *deadline* lebih lama.

EDF memiliki aturan saat *task* sedang dieksekusi, *task* tersebut harus memberi tahu kepada sistem permintaan *deadline*nya. Berbeda dengan *rate-monotonic* yang memiliki prioritas statis, EDF memiliki prioritas yang dinamis karena prioritas pada EDF selalu menyesuaikan dengan dengan *deadline task* baru yang sedang dijalankan. EDF juga tidak meminta proses secara periodik. Pada beberapa kasus, *rate-monotonic* tidak dapat memberikan hasil yang sesuai (*task* diselesaikan melewati *deadline*) dan dapat diatasi dengan penjadwalan EDF.

Contohnya terdapat dua proses P1 dan P2. P1 memiliki waktu eksekusi 25 dengan periode 50, sedangkan P2 memiliki waktu eksekusi 35 dengan periode 80. Berikut merupakan *gant-chart* untuk kasus ini



Gambar 2. *Gant-chart* kasus *earliest deadline first*

Sumber : Silberschatz et al. "Operating System Concepts" 9th ed.

III. PEMBAHASAN

Pada pembahasan kali ini diambil contoh terdapat tiga proses yang akan dieksekusi oleh CPU. Waktu eksekusi direpresentasikan t, periode p.

P1 memiliki t1 = 30 dan p1 = 80;

P2 memiliki t2 = 20 dan p2 = 50;

P3 memiliki t3 = 30 dan p3 = 90;

ketiga proses diatas akan dieksekusi dengan penjadwalan *rate-monotonic* dan *earliest deadline first.* Asumsi : proses dengan nomor lebih kecil memiliki prioritas *default* awal lebih tinggi.

Dari kasus diatas maka didapatkan :

Himpunan kandidat : himpunan proses yang merepresentasikan P1, P2, P3.

Himpunan solusi : kumpulan proses yang menghasilkan urutan pengerjaan P1, P2, dan P3 yang sudah selesai oleh CPU.

Fungsi seleksi :

Rate-monotonic : memilih proses dengan *deadline* yang sudah terlewati sebelumnya

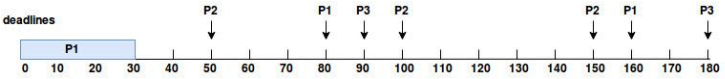
Earliest Deadline First : memilih proses dengan *deadline* paling dekat dengan waktu eksekusi saat tertentu.

Fungsi layak : memeriksa apakah suatu proses yang dieksekusi telah melewati *deadline* atau belum.

Fungsi obyektif : jumlah proses yang dieksekusi paling banyak

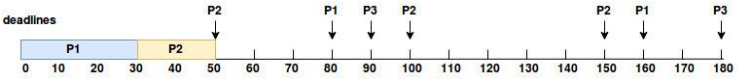
A. Rate-monotonic Scheduling

Pada *rate-monotonic scheduling*, proses yang pertama dikerjakan adalah proses P1 karena memiliki prioritas awal yang lebih tinggi. Maka diperoleh himpunan solusi {P1}.



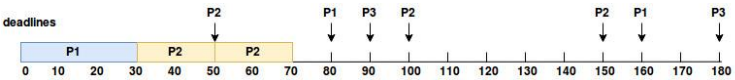
Gambar 3. *Rate-monotonic* langkah pertama

Proses selanjutnya yang akan dieksekusi adalah P2 karena P1 telah selesai diproses. Saat P2 selesai ternyata bersamaan dengan *deadline*/periode untuk P2, jadi P2 masih bisa dihitung sebagai proses yang berhasil dieksekusi. Dari langkah kedua ini, himpunan solusinya {P1, P2}.



Gambar 4. *Rate-monotonic* langkah kedua

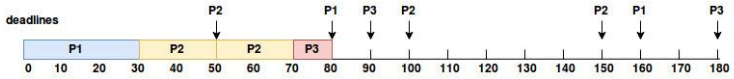
Sesuai dengan fungsi seleksinya, *rate-monotonic* akan mengeksekusi proses yang *deadline*nya telah terlewati sebelumnya, maka proses yang selanjutnya dieksekusi lagi adalah P2, walaupun proses P3 sama sekali belum dieksekusi oleh CPU. Sehingga himpunan solusi yang diperoleh adalah {P1, P2, P2}.



Gambar 5. *Rate-monotonic* langkah ketiga

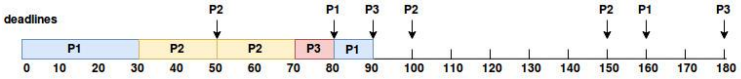
Setelah P2 selesai dikerjakan dan tidak ada periode proses yang telah terlewati lagi, maka CPU baru memberikan jatahnya kepada P3. Akan tetapi saat linimasa menunjukkan waktu ke-80, yaitu periode P1 maka proses P3 diberhentikan (*preemptive*) dan kemudian di *switch* dengan proses selanjutnya. P3 belum masuk ke dalam himpunan solusi karena

belum selesai dieksekusi, jadi himpunan solusi masih sama yaitu {P1, P2, P2}.



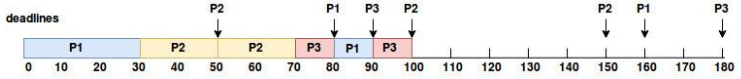
Gambar 6. *Rate-monotonic* langkah keempat

Karena periode P1 baru terlewati, maka CPU memberikan jatahnya kepada P1 meskipun P3 belum selesai dan periode P3 juga tidak jauh lagi dari linimasa saat ini. Jadi saat P1 baru dieksekusi selama 10 satuan waktu, CPU melakukan pemberhentian proses P1 (sebab periode P3 = 90). Saat penarikan proses P3 ternyata P3 belum selesai dieksekusi, masih kurang 20 satuan waktu lagi untuk mengeksekusinya, jadi P1 maupun P3 tidak dihitung dalam himpunan solusi (himpunan solusi masih sama yaitu {P1, P2, P2}).



Gambar 7. *Rate-monotonic* langkah kelima

Setelah melewati *deadline*, P3 seharusnya diberi jatah CPU lagi seperti *gant-chart* dibawah ini.



Gambar 8. Langkah keenam pada *Rate-monotonic* yang seharusnya

Himpunan solusi akhir : {P1, P2, P2}

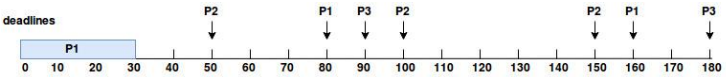
Pada kasus seperti ini, *rate-monotonic scheduling* sudah tidak dapat digunakan karena tidak memenuhi tenggat waktu/*deadline* yang dibutuhkan oleh suatu proses. Maka dari itu dibutuhkan suatu algoritma penjadwalan lain yang memperhatikan *deadline* mendatang. Walau begitu, *rate-monotonic scheduling* juga tetap bisa diaplikasikan pada proses-proses yang dapat ditangani dengan meng-*greedy*-kan *deadline-deadline* yang telah dilewati dengan harapan semakin kecil *deadline* tersebut maka suatu proses semakin membutuhkan eksekusi CPU dengan cepat.

Secara garis besar, *pseudocode rate-monotonic scheduling* dapat dituliskan sebagai berikut :

```
function rate_monotonic (input C : himpunan_proses, HP : himpunan_period) → himpunan_proses
{Mengembalikan urutan proses yang akan dikerjakan CPU dengan mengoptimumkan deadline yang sudah terlewati
Masukan : himpunan kandidat C
Keluaran : himpunan solusi yang bertipe himpunan_proses
}

Deklarasi
 P : Proses {Proses merupakan tipe bentukan yang terdiri dari time dan period}
 S : himpunan_proses {digunakan untuk himpunan solusi}
 stop : boolean

Algoritma
 S ← {}
 stop ← false
 P ← Proses yang pertama dikerjakan
 while (not stop) do
    if (timeline ∈ HP) then
    {timeline merupakan variabel static}
       P ← proses dengan period sama dengan timeline
    endif
    if (P.period ≥ timeline) then
       if (P.time+timeline <= period terdekat) then
          S ← S U {P}
          C ← C -{P}
          timeline ← timeline + P.time
        else
           C.P.time ← selisih waktu “saat ini” dengan saat assign
           timeline ← timeline + C.P.time
        endif
    else
       stop ← true
    endif
    P ← Proses dengan periode terdekat
 endwhile

 if (S ≠ {}) then
    return S
 else
    output (‘tidak ada proses yang dapat dikerjakan’)
 endif
```

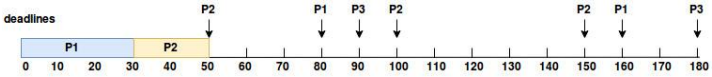
B. Earliest Deadline First Scheduling

Untuk kasus ini, pada *earliest deadline first scheduling*, proses pertama yang didahulukan sama dengan *rate-monotonic scheduling*, yaitu mengeksekusi P1 terlebih dahulu, sehingga diperoleh himpunan solusi {P1} dengan *gant-chart* sebagai berikut.



Gambar 9. *Earliest Deadline First* langkah pertama

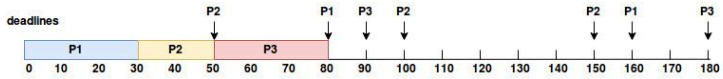
Setelah proses P1 selesai dikerjakan, maka P2 dikerjakan dahulu karena *deadline* terdekat yang akan dihadapi adalah *deadline* milik P2. Sehingga saat proses P2 selesai, bertepatan dengan waktu tenggatnya, diperoleh {P1, P2} sebagai himpunan solusi.



Gambar 10. *Earliest Deadline First* langkah kedua

Berbeda dengan *rate-monotonic scheduling*, walaupun *deadline* P2 sudah terlewati, CPU tidak langsung mengerjakan P2 lagi tetapi akan mengeksekusi P3 karena *deadline* P3 yang

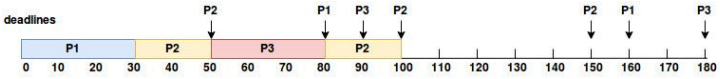
lebih dahulu dihadapi daripada P2. Maka himpunan solusi yang diperoleh {P1, P2, P3}.



Gambar 11. *Earliest Deadline First* langkah ketiga

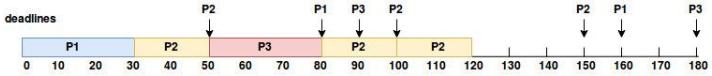
Begitu juga saat linimasa menunjukkan 80 satuan waktu yaitu saat *deadline* P1, CPU tidak mengerjakan P1 lagi karena ada *deadline* proses yang lebih dekat daripada *deadline* proses P1 yaitu *deadline* proses P2.

Sehingga P2 dieksekusi oleh CPU walaupun saat linimasa di 90 satuan waktu terdapat *deadline* proses P3 juga. Sejauh ini proses yang dikerjakan sesuai dengan periode yang telah diberikan dari masing-masing proses, termasuk saat di 100 satuan waktu. Sehingga himpunan solusi yang didapatkan {P1, P2, P3, P2}.



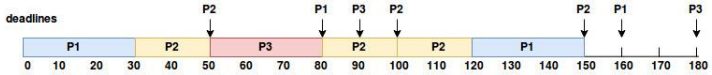
Gambar 12. *Earliest Deadline First* langkah keempat

Setelah keempat proses sebelumnya berhasil melewati tenggat waktu yang diberikan, maka proses yang selanjutnya dieksekusi adalah proses ke 2 (P2) karena dibanding kedua proses lainnya, *deadline* P2 lebih cepat walaupun P2 sudah dua kali dieksekusi sebelumnya. Sehingga diperoleh himpunan solusi {P1, P2, P3, P2, P2}.



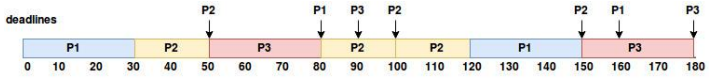
Gambar 13. *Earliest Deadline First* langkah kelima

Setelah P2 selesai, maka giliran P1 yang dieksekusi sebab *deadline*nya lebih cepat, maka didapat {P1, P2, P3, P2, P2, P1} sebagai himpunan solusi karena hingga saat ini saat linimasa menunjukkan tenggat waktu suatu proses, proses tersebut telah selesai dieksekusi sehingga proses yang dieksekusi pasi masuk ke dalam himpunan solusi.



Gambar 14. *Earliest Deadline First* langkah keenam

Kemudian proses yang dieksekusi adalah P3 karena semua *deadline* proses akan terlewati dalam waktu dekat sudah selesai dieksekusi prosesnya. Dan saat P3 selesai dieksekusi, bersamaan dengan periode P3 sehingga P3 masuk ke dalam himpunan solusi. Himpunan solusi yang diperoleh {P1, P2, P3, P2, P2, P1, P3}.



Gambar 15. *Earliest Deadline First* langkah ketujuh

Dari uraian kasus diatas yang menggunakan *earliest deadline first* sebagai algoritma penjadwalan proses, pemilihan proses dilakukan secara *greedy by upcoming deadline* atau memilih proses yang *deadline*nya paling dekat dengan "saat ini" sehingga semua proses berhasil dieksekusi saat periodenya. Otomatis dapat membuat hasil yang lebih optimum karena proses tidak ada yang *miss deadline*.

Secara umum, *pseudocode earliest deadline first scheduling* ialah sebagai berikut :

```
function earliest_deadline_first (input C : himpunan_proses, HP : himpunan_period) → himpunan_proses
{Mengembalikan urutan proses yang akan dikerjakan CPU dengan mengoptimumkan deadline terdekat yang akan dihadapi
}

Deklarasi
 S : himpunan_proses
 P : Proses {Proses merupakan tipe bentukan yang terdiri dari  dua integer time dan period}
 stop : boolean

Algoritma
 S ← {}
 stop ← false
 P ← Proses yang pertama dikerjakan
 while (not stop) do
    if (P.period ≥ timeline) then
    {timeline merupakan variabel static}
       S ← S U {P}
       C ← C -{P}
       timeline ← timeline + P.time
    else
       stop ← true
    endif
    P ← Proses dengan periode terdekat yang dilihat dari HP
 endwhile

 if (S ≠ {}) then
    return S
 else
    output ('tidak ada proses yang dapat dikerjakan')
 endif
```

IV. KESIMPULAN

Berdasarkan hasil pengujian dengan menggunakan *rate-monotonic scheduling* atau bisa disebut *greedy by previous deadline*, beberapa proses tidak dapat selesai tepat waktu karena proses belum selesai dieksekusi saat timeline sama dengan *deadline*/periodenya sehingga algoritma penjadwalan ini kurang optimum.

Berbeda dengan *rate-monotonic scheduling*, *earliest deadline first scheduling* menghasilkan solusi yang lebih optimum dilihat dari jumlah proses yang diselesaikan tepat waktu karena algoritma penjadwalan ini menggunakan *greedy by upcoming deadline* sehingga untuk setiap *deadline* terdekat, proses tersebut memiliki prioritas yang lebih tinggi daripada proses lainnya.

Jika dilihat secara garis besar, algoritma *greedy* ini dapat dijadikan dasar untuk penggunaan algoritma lain yang lebih kompleks karena algoritma *greedy* relatif lebih sederhana dibandingkan algoritma lain, misalnya algoritma penjadwalan proses pada sistem *real-time* maupun sistem penjadawalan lainnya.

V. UCAPAN TERIMA KASIH

Dengan terbentuknya makalah ini, penulis mengucapkan terima kasih kepada Allah SWT yang memberikan nikmat dan izin sehingga penulis dapat menyelesaikan makalah. Tidak lupa penulis mengucapkan terima kasih juga kepada keluarga penulis, terutama kedua orangtua yang telah memberikan dukungan moral dan juga berbagai fasilitas yang menunjang untuk penulisan makalah. Kemudian terima kasih untukBapak Rinaldi Munir, Ibu Nur Ulfa Maulidevi, dan Ibu Masayu Leylia Khodra yang telah bersedia menjadi dosen mata kuliah Strategi Algoritma ini selama satu semester dengan sabar dan selalu mencurahkan ilmu-ilmu yang dimiliki. Begitu juga untuk teman-teman dari Teknik Informatika 2015 yang saling mendukung dalam pembuatan makalah.

Dan terima kasih untuk para pembaca yang telah bersedia meluangkan waktunya untuk membaca makalah ini. Karena kesempurnaan hanya datang dari Allah SWT dan kesalahan berasal dari penulis, maka penulis bersedia menerima kritik dan saran untuk perbaikan makalah. Semoga ilmu yang terdapat dari makalah dapat dikembangkan lagi dan diimplementasikan kepada masyarakat. Terima kasih.

REFERENSI

[1] Munir, Rinaldi. "Diktat Kuliah IF2211 Strategi Algoritma", Informatika. Bandung : 2009

[2] Silberschatz, Abraham et al. "Operating System Concepts" 9th ed. Courier press. United States of America : 2013.

[3] Anonymous. "Real Time Scheduling" dari : http://web.cs.ucla.edu/classes/winter17/cs111/readings/realtime.html, diakses pada 16 Mei 2017, pukul 14:48

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.

Purwokerto, 18 Mei 2017

Iftitakhul Zakiah
13515114