Implementasi CPU *Scheduling* dalam *Multiprogramming* dengan Pendekatan *Greedy*

Amar Fadil - 13520103
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13520103@std.stei.itb.ac.id

***Abstract*—Dalam *multiprogramming*, utilisasi CPU diharapkan selalu tinggi. Dibutuhkan sebuah algoritma penjadwalan *program* sehingga CPU memiliki utilisasi penuh setiap saat. Strategi *greedy* dapat digunakan dalam menjadwalkan *program* secara efisien yang biasa disebut sebagai *priority scheduling*. *Shortest first job* merupakan salah satu algoritma *priority scheduling non-preemptive*. Diharapkan dengan *priority scheduling*, kriteria CPU *scheduler* yang bagus terpenuhi dengan baik.**

Keywords—multiprogramming; greedy; cpu scheduling; shortest job first

I. PENDAHULUAN

Multiprogramming mengharuskan utilisasi CPU yang tinggi setiap saat sehingga tidak terdapat waktu CPU untuk mengganggur. Dibutuhkan sebuah penjadwalan yang disebut sebagai CPU *scheduling*. Beberapa kriteria dalam menilai CPU *scheduling* yang bagus, diantaranya utilisasi CPU maksimum, *throughput* maksimum, *turnaround time* minimum, *waiting time* minimum, dan *response time* minimum [1]. Sebuah *job* seringkali disebut sebagai proses. Penjadwalan proses dapat dijalankan secara *preemptive*, yakni penyerobotan paksa proses sedang berjalan oleh proses yang baru muncul, atau secara *non-preemptive*, yakni proses yang berjalan harus diselesaikan terlebih dahulu sebelum memproses proses baru.

Secara sederhana, pembagian penjadwalan proses dapat dilaksanakan dengan FCFS (First Come, First Served), yakni proses yang pertama kali masuk akan diproses pertama kali juga. Pembagian penjadwalan dengan FCFS ini bersifat *non-preemptive*, sehingga dapat terjadi *convoy effect* berupa fenomena proses baru menunggu proses yang sedang dikerjakan dalam waktu sangat lama. Hal ini dapat diselesaikan dengan *round robin*, yakni algoritma penjadwalan yang bersifat pre-emptif dalam satuan waktu yang disebut *time quantum*. *Time quantum* harus dibuat sedemikian rupa sehingga tidak terlalu kecil (banyak overhead akibat *context switching*) dan tidak terlalu besar (bekerja layaknya FCFS) yang biasanya berkisar antara 10-100ms.

Selain FCFS dan *round robin*, terdapat algoritma yang memakai strategi greedy, seringkali disebut sebagai *priority scheduling* karena mengurutkan proses pengerjaan berdasarkan prioritas setiap proses. Salah satu algoritma *prority scheduling* adalah *Shortest Job First* (SJF) yang mengurutkan proses pengerjaan berdasarkan prediksi CPU *burst* selanjutnya dari setiap proses. Pada penjadwalan *real-time*, penjadwalan dengan *priority scheduling* dapat dilakukan dengan *rate monotonic shceduling* untuk *soft real-time system* yang tidak memiliki deadline pengerjaan proses dan *earliest deadline first* (EDF) untuk *hard real-time system* yang mewajibkan proses harus diproses sehingga tidak boleh melewati deadline [1].

Dalam makalah ini, akan dibahas penggunaan strategi *greedy* dalam mengimplementasikan *priority scheduling* seperti SJF, terutama pada kasus *non-preemptive*. Implementasi ini dapat digunakan dalam level sistem operasi. Selain itu, juga akan dibahas perbandingan antara *shortest job first* pada kasus *non-preemptive* dengan penjadwalan *non-preemptive* lainnya seperti FCFS.

II. TEORI DASAR

A. Algoritma Greedy

Algoritma greedy merupakan salah satu cara untuk memecahkan masalah pada suatu permasalahan. Algoritma ini cukup populer dan mengandalkan pendekatan yang sederhana untuk persoalan optimalisasi, yakni masalah mengenai pencarian maksimasi dan minimasi. Banyak sekali contoh permasalahan yang dapat diselesaikan dengan algoritma *greedy*, akan tetapi *greedy* mempunyai kelemahan yakni hanya mendapatkan *local* optimum alih-alih *global* optimum yang sebenarnya diinginkan.

Prinsip kerja dari algoritma ini adalah "*take what you can get now*", yaitu mencari solusi optimal pada setiap langkah. Jika solusi yang ingin dicari adalah nilai maksimal, algoritma greedy akan mengambil nilai maksimal saat itu tanpa memikirkan konsekuensi pada langkah berikutnya. Sebaliknya, jika solusi yang diinginkan adalah solusi dengan nilai minimal, nilai minimal akan diambil. Nilai maksimal atau minimal sementara ini disebut *local* optimum. Pada langkah berikutnya, algoritma greedy akan kembali mencari *local* optimum. Pencarian *local* optimum ini terus diulang hingga langkah terakhir. Pada langkah terakhir, algoritma greedy diharapkan dapat menemukan solusi optimal dari permasalahan dengan solusinya disebut sebagai *global* optimum. Meskipun begitu, sebagian permasalahan yang ada tidak dijamin ditemukan solusi optimalnya dengan algoritma greedy, atau dengan kata lain *local* optimumnya berbeda dengan *global* optimum.

Supaya strategi *greedy* yang dipakai efektif dan efisien dalam menyelesaikan masalah secara akurat, diperlukan perencanaan fungsi seleksi yang baik. Akan tetapi, sulit untuk membuktikan bahwa sebuah strategi greedy dengan fungsi seleksi yang dipilih pasti merupakan *global* optimum. Untuk membuktikan fungsi seleksi konvergen menuju *global* optimum, harus dilakukan pembuktian dengan induksi matematika, yang tentunya sulit untuk dilakukan. Meski begitu, kita juga dapat membuktikan bahwa suatu strategi *greedy* bukan merupakan *global* optimum hanya dengan memberikan instansi permasalahan yang menunjukkan hasil *local* optimum berbeda dengan *global* optimum.

Terdapat beberapa elemen *greedy*, yakni sebagai berikut: [2]

1) *Himpunan Kandidat, C*: berisi kandidat yang akan dipilih pada setiap langkah (misalnya koin, job, benda, dsb).

2) *Himpunan Solusi, S*: berisi kandidat yang sudah dipillih dan akan menjadi solusi permasalahan.

3) *Fungsi Solusi*: menentukan apakah himpunan kandidat yang dipilih sudah menjadi solusi permasalahan.

4) *Fungsi Seleksi (selection function)*: memilih kandidat berdasarkan strategi *greedy* tertentu. Strategi *greedy* ini bersifat heuristik.

5) *Fungsi Kelayakan (feasible)*: memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi (layak atau tidak).

6) *Fungsi Obyektif*: memaksimumkan atau meminimumkan.

Skema umum yang digunakan dalam strategi greedy adalah sebagai berikut: [2]

```
function greedy(C: himpunan_kandidat) → himpunan_solusi
{ mengembalikan solusi dari persoalan optimasi dengan
algoritma greedy }

Deklarasi
  x: kandidat
  S: himpunan_solusi

Algoritma
  S ← {}    { inisialisasi S kosong }
  while ((not SOLUSI(S)) and (Q ≠ {})) do
    x ← SELEKSI(C)   { pilih sebuah kandidat dari C }
    C ← C – {i}      { buang x dari C }
    if (LAYAK(S ∪ {x})) then { x memenuhi kelayakan }
      S ← S ∪ {x}    { masukkan x ke himp. solusi }
    endif
  endwhile
  { SOLUSI(S) or C = {} }

  if (SOLUSI(S)) then  { solusi sudah lengkap }
    return S
  else
    write('tidak ada solusi')
  endif
```

B. Multiprogramming

Multiprogramming merupakan konsep yang menekankan penggunaan CPU semaksimal mungkin sehingga tidak menganggur. Selama eksekusi proses, akan terjadi siklus eksekusi CPU dengan I/O *wait* yang saling bergantian. Untuk mencapai *multiprogramming*, dibutuhkan sebuah *CPU Scheduling* yang efektif dan efisien, sehingga proses yang menunggu I/O *wait* memberikan bagiannya kepada proses lain yang membutuhkan.

C. CPU Scheduling

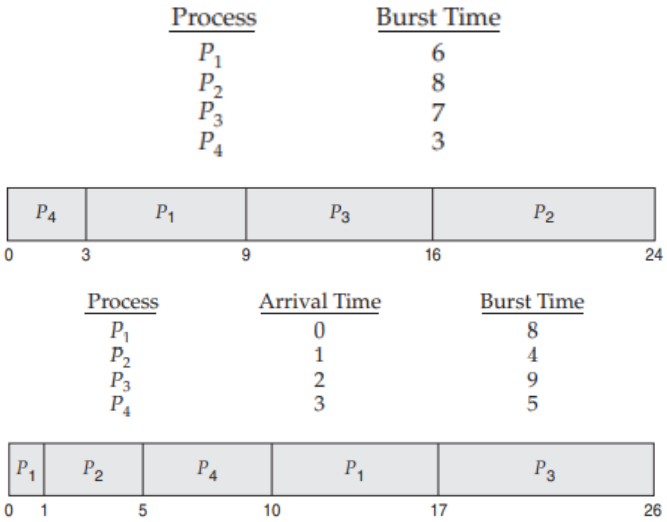
CPU scheduling merupakan sebuah algoritma yang dapat menjadwalkan proses sedemikian rupa untuk dijalankan secara berurutan. Hal ini biasanya diimplementasikan dalam level sistem operasi. Tujuan utama *CPU Scheduling* adalah tercapainya *multiprogramming*. Dalam mencapai *multiprogramming*, CPU *Scheduling* memiliki kriteria sebagai berikut [1]*:*

- Utilisasi CPU maksimum: penggunaan CPU secara maksimum setiap waktu, dimana CPU akan selalu sibuk dalam mengerjakan proses
- *Throughput* maksimum: banyaknya proses yang diekseskusi dalam satuan waktu bernilai maksimum, sehingga banyak proses yang dapat diproses dalam suatu waktu.
- *Turnaround time* minimum: waktu total yang dibutuhkan untuk mengeksekusi proses, dimulai dari waktu kedatangan hingga *burst time* proses, seminimum mungkin sehingga proses dapat diselesaikan dengan cepat.
- *Waiting time* minimum: waktu yang dibutuhkan proses dalam menunggu di *ready queue* seminimum mungkin.

Sebuah penjadwalan dapat memiliki sifat *preemptive*, yakni penyerobotan paksa proses sedang berjalan oleh proses yang baru muncul, atau secara *non-preemptive*, yakni proses yang berjalan harus diselesaikan terlebih dahulu sebelum memproses proses baru. CPU *scheduling* dapat dilakukan dalam beberapa tempat, yaitu ketika proses dalam keadaan *running* menjadi *waiting* (*non-preemptive*) atau *ready* (*preemptive*), ketika proses dalam keadaan *waiting* menjadi *ready* (*preemptive*), dan ketika proses diterminasi (*non-preemptive*). *Context switching* merupakan tahap penggantian proses yang dikerjakan dengan proses lainnya yang akan dikerjakan. Terjadinya *context switching* tentunya mengakibatkan overhead dalam membaca *process control block* (PCB) dari proses yang akan dijalankan dan menuliskan PCB dari proses yang sedang dikerjakan jika masih memiliki waktu *burst.*

Beberapa contoh CPU *scheduler* adalah *first-come first-served* (FCFS), *round-robin* (RR), dan *shortest job first* (SJF). Dalam kasus sistem *real-time*, terdapat pula *rate monotonic scheduling* dan *earliest deadline first* (EDF). SJF dan CPU *scheduler* untuk sistem *real-time* merupakan jenis *priority scheduler*, dimana penjadwalan dilakukan berdasarkan suatu prioritas. Untuk SJF *non-preemptive*, proses diurutkan berdasarkan prioritas prediksi CPU *burst time* selanjutnya (biasanya prediksi dilakukan dengan *exponential average*), dan untuk SJF *preemptive*, juga disebut sebagai *shortest remaining time first* (SRTF), proses diurutkan berdasarkan waktu CPU *burst* yang tersisa. Sedangkan untuk *rate-monotonic scheduling*, proses diurutkan berdasarkan oleh *inverse* dari periodik, dengan

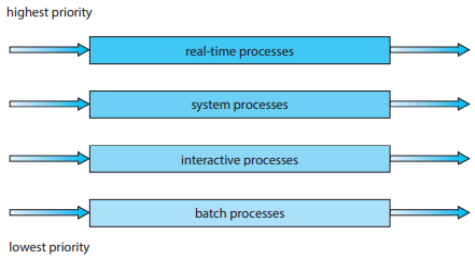
kata lain, periodik paling singkat mendapatkan prioritas paling tinggi, dan periodik paling lama mendapatkan prioritas paling rendah. Terakhir untuk EDF, proses diurutkan berdasarkan *deadline* proses, sehingga tidak memerlukan periodik proses layaknya *rate monotonic scheduling*.



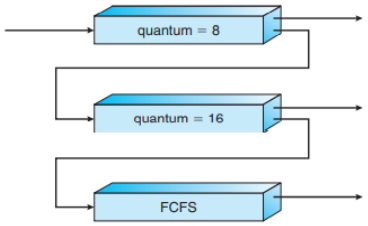
Gambar II.1 Contoh penjadwalan *non-preemptive SJF dan preemptive SJF (SRTF).*
Sumber: S. Abraham, *Operating System Concepts 10th ed., pp. 207-209*

Terdapat metode lain dalam menjadwalkan proses, yakni *multilevel queue scheduling* yang membagi *ready queue* menjadi beberapa *queue*, dimana tiap *queue* memiliki algoritma penjadwalan yang berbeda-beda (FCFS, RR, SJF, atau SRTF). Pada *scheduling* tersebut, *queue* dapat di-*schedule* kembali, biasanya dalam bentuk *fixed priority scheduling* dengan prioritas absolut yang memungkinkan *starvation*. Untuk mengatasi ini, *queue* dapat memiliki *time-slice* dengan cara dibagi menjadi dua jenis: *queue* untuk *proses background* dan untuk proses *foreground* dengan pembagian prioritas 20% dan 80% terurut.

Selain *multilevel queue scheduling*, juga terdapat *multilevel feedback queue scheduling* dimana proses dapat dipindahkan dalam beberapa *queue*. Hal ini mempermudahkan dalam mengimplementasikan *aging* untuk menghilangkan *starvation*. Parameter yang digunakan dalam penjadwalan ini adalah banyaknya queue, algoritma yang dipakai untuk setiap queue, metode untuk upgrade/demote suatu proses, dan metode untuk menentukan *queue* yang dipakai untuk proses baru.



(a)



(b)

Gambar II.2 (a) Multilevel queue scheduling dan (b) Multi-feedback queue scheduling.
Sumber: S. Abraham, *Operating System Concepts 10th ed., pp. 215-216*

III. Aplikasi Strategi Greedy

A. *Mapping Elemen Greedy*

Dalam penjadwalan menggunakan *shortest job first*, terdapat beberapa identifikasi elemen-elemen dari suatu masalah yang dapat diselesaikan secara sistematis menggunakan strategi *greedy*:

1) Himpunan Kandidat, C: merupakan beberapa proses yang tersedia dalam *ready queue*. Dalam kasus *non*-preemptive, setiap proses hanya dapat dijalankan sekali hingga selesai. Meski begitu, pada kasus *preemptive*, setiap proses mungkin dijalankan beberapa kali hingga selesai. Proses biasanya diidentifikasi dengan P_{i+1}, dengan *subscript i* merupakan indeks proses pada *ready queue* dengan panjang *n* (P_1, P_2, …, P_n).

2) Himpunan Solusi, S: merupakan himpunan yang berisikan anggota himpunan kandidat terurut berdasarkan proses mana yang dijalankan terlebih dahulu di CPU. Misalnya untuk kasus *non-preemptive*, himpunan solusi S = {P_2, P_1, P_3} berarti CPU akan menjalankan proses kedua, dilanjutkan dengan proses pertama, dan diakhiri dengan proses ketiga. Untuk kasus *preemptive*, misalnya himpunan solusi S = {P_3, P_1, P_3, P_2, P_1}

3) Fungsi Solusi: merupakan fungsi yang mengembalikan benar jika semua proses pada *ready queue* telah dijalankan (*ready queue* kosong), dan salah jika sebaliknya.

4) Fungsi Seleksi: merupakan strategi *greedy* yang digunakan dalam memilih proses pada *ready queue* (himpunan kandidat). Hasil dari fungsi seleksi adalah proses terpilih dari himpunan kandidat yang kemudian menjadi anggota himpunan solusi. Pada *shortest job first non-preemptive*, strategi *greedy* yang digunakan adalah mengambil proses dengan *next burst time* paling kecil.

5) Fungsi Kelayakan: merupakan fungsi yang mengembalikan benar jika proses yang dipilih, sebelumnya berada pada *ready queue*, dan salah jika sebaliknya. Secara teknis, fungsi ini selalu mengembalikan nilai benar, sehingga dalam skema algoritma SJF, hasil fungsi seleksi tidak perlu diperiksa (langsung dimasukkan kedalam himpunan solusi).

6) Fungsi Objektif: terdiri dari beberapa kriteria yang harus diminimumkan atau dimaksimumkan, seperti pada kriteria CPU *scheduling* di teori dasar (memaksimumkan utilisasi CPU, memaksimumkan *throughput*, meminimumkan *turnaround*

time, meminimumkan *waiting time*, dan memnimumkan *response time*).

B. *Algoritma Penjadwalan SJF Non-preemptive*

Untuk menjadwalkan proses dengan *shortest job first scheduling* pada kasus *non-preemptive*, dapat dilakukan dengan tahapan berikut:

1) Inisialisasi himpunan solusi kosong.
2) Selama *ready queue* belum kosong, lakukan tahapan berikut:
   a.) Cari proses yang memiliki *next burst time* terkecil.
   b.) Hapus proses hasil tahapan 2.a. pada *ready queue*.
   c.) Tambahkan proses hasil tahapan 2.a. ke dalam himpunan solusi.
3) Kembalikan himpunan solusi.

Secara algoritmik, *shortest job first scheduling* untuk kasus *non-preemptive* dapat dilakukan sebagai berikut:

```
function SJFNonPreemptive(Q: himpunan_kandidat) → himpunan_solusi
{ mengembalikan urutan penjadwalan proses dengan algoritma SJF untuk kasus non-preemptive }

Deklarasi
   S: himpunan_solusi
   p: Process
   {- type Process {
        burst: integer
   } -}

Algoritma
   S ← {}
   while (Q ≠ {}) do
        p ← proses dengan next burst time terkecil
        Q ← Q - {i}
        S ← S ∪ {i}
   endwhile
   return S
```

Analisis kompleksitas algoritma tersebut membutuhkan satu loop utama, yakni ketika Q tidak kosong, dan membutuhkan loop untuk mencari proses dengan *next burst time* terkecil. Jika diketahui panjang Q adalah N, maka total operasi perbandingan yang dilakukan adalah (N+1) + (N) + (N-1) + … + 3 + 2 = $N(N+1)/2 + N = O(N^2)$. Kompleksitas ini dapat dioptimasi dengan mengurutkan proses dari *next burst time* paling kecil terlebih dahulu, sehingga untuk mencari proses dengan *next burst time* terkecil cukup hanya mengambil elemen pertama dari *ready queue* yang telah terurut. Jika proses diurutkan terlebih dahulu, maka total operasi perbandingan yang dilakukan adalah sebanyak panjang elemen Q, yaitu O(N).

Secara umum, jika diketahui *arrival time* dari masing-masing proses (proses datang tidak secara bersamaan), maka proses terlebih dahulu diurutkan mulai dari kedatangan paling cepat hingga paling lama (bisa menggunakan *priority queue*). Setelah diurutkan, penjadwalan dapat dilakukan dengan tahapan berikut:

1) Inisialisasi himpunan solusi kosong.
2) Inisialisasi t = waktu kedatangan proses pertama (paling awal muncul). Variabel ini digunakan untuk mencari proses yang tersedia hingga pada saat t.
3) Selama *ready queue* belum kosong, lakukan tahapan berikut:
   a.) Cari proses yang memiliki *next burst time* terkecil dengan syarat waktu kedatangan sama atau lebih kecil daripada t.
   b.) Hapus proses hasil tahapan 2.a. pada *ready queue*.
   c.) Tambahkan proses hasil tahapan 2.a. ke dalam himpunan solusi.
   d.) Jika *ready queue* masih belum kosong, assign *t* dengan nilai maksimum (mana yang paling besar) antara t + *next burst time* dari proses hasil tahapan 2.a dan *arrival time* proses pertama di *ready queue* (setelah proses hasil tahapan 2.a. dihapus dari *ready queue*).
4) Kembalikan himpunan solusi.

Secara algoritmik, generalisasi *shortest job first scheduling* untuk kasus *non-preemptive*, yakni waktu kedatangan proses diketahui, dapat dilakukan sebagai berikut:

```
function SJFNonPreemptive2(Q: himpunan_kandidat) → himpunan_solusi
{ mengembalikan urutan penjadwalan proses dengan algoritma SJF untuk kasus non-preemptive dan arrival time diketahui (proses dianggap datang tidak bersamaan). Asumsi Q tidak kosong dan dimulai dari indeks 0. }

Deklarasi
  S: himpunan_solusi
  p: Process
  {- type Process {
    arrival: integer,
    burst: integer
  } -}
  t: integer

Algoritma
  S ← {}
  t ← Q[0].arrival
  while (Q ≠ {}) do
    p ← proses dengan next burst time terkecil dan waktu arrival ≤ t
    Q ← Q - {i}
    S ← S ∪ {i}
    if (Q ≠ {}) then
      t ← max(t + p.burst, Q[0].arrival)
    endif
  endwhile
  return S
```

Analisis kompleksitas algoritma tersebut mirip seperti algoritma *SJFNonPreemptive*, dimana algoritma ini membutuhkan satu loop utama, yakni ketika Q tidak kosong, dan

satu loop untuk mencari proses dengan *next burst time* terkecil dengan syarat *arrival time* ≤ t. Pada kasus terburuk dimana saat syarat loop untuk mencari proses selalu dipenuhi oleh semua proses pada *ready queue*, jika diketahui panjang Q adalah N, maka kompleksitas waktu totalnya adalah $O(N^2)$. Pada kasus terbaik, dimana syarat loop untuk mencari proses hanya dipenuhi oleh proses pertama pada *ready queue*, maka kompleksitas waktu totalnya adalah O(N).

C. Analisis Hasil Pengujian

Contoh instansi permasalahan penjadwalan adalah terdapat 4 proses yang ada pada *ready queue*, P_1, P_2, P_3, dan P_4, masing-masing memiliki *next burst time* secara terurut 9, 2, 6, 5. Untuk menentukan penjadwalan, diketahui C = Q = { P_1, P_2, P_3, P_4 }. Langkah-langkah penyelesaian instansi permasalahan tersebut sebagai berikut:

1) Langkah 1: Inisialisasi himpunan solusi kosong.

2) Langkah 2: Q masih tidak kosong, cari proses yang memiliki *next burst terkecil*, yakni P_2. Hapus P_2 pada Q dan tambahkan P_2 pada S. Maka Q = { P_1, P_3, P_4 } dan S = { P_2 }.

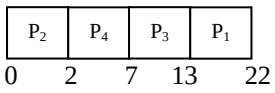
3) Langkah 3: Q masih tidak kosong, cari proses selanjutnya yang memiliki *next burst terkecil*, yakni P_4. Hapus P_4 pada Q dan tambahkan P_4 pada S. Maka Q = { P_1, P_3 } dan S = { P_2, P_4 }.

4) Langkah 4: Q masih tidak kosong, cari proses selanjutnya yang memiliki *next burst terkecil*, yakni P_3. Hapus P_3 pada Q dan tambahkan P_3 pada S. Maka Q = { P_1 } dan S = { P_2, P_4, P_3 }.

5) Langkah 5: Q masih tidak kosong, cari proses selanjutnya yang memiliki *next burst terkecil*, yakni P_1. Hapus P_1 pada Q dan tambahkan P_1 pada S. Maka Q = { } dan S = { P_2, P_4, P_3, P_1 }.

6) Langkah 6: Q sudah kosong, kembalikan S = { P_2, P_4, P_3, P_1 }.

Gantt chart dari permasalahan diatas adalah sebagai berikut (*chart* tidak sesuai skala):



Di awal, CPU akan memproses P_2 terlebih dahulu dengan burst time 2 pada waktu 0 hingga *completion time* 2. Kemudian, CPU memproses P_4 dengan burst time 5 pada waktu 2 hingga *completion time* 7. Selanjutnya, CPU memproses P_3 dengan burst time 6 pada waktu 7 hingga *completion time* 13. Terakhir, CPU memproses P_1 dengan burst time 9 pada waktu 13 hingga *completion time* 22.

Turnaround time yang dibutuhkan masing-masing proses sama dengan *completion time* dengan rata-ratanya 11. *Waiting time* yang dibutuhkan setiap proses secara terurut dari P_1 hingga P_4 adalah 13, 0, 7, dan 2, dengan rata-ratanya adalah 5,5. Karena CPU tidak memiliki waktu *idle* (tidak mengerjakan proses apapun), maka utilisasi CPU adalah 100%. Jika dikerjakan menggunakan FCFS dengan P_1, P_2, P_3, dan P_4 datang secara berurutan, rata-rata *turnaround time* yang dibutuhkan adalah (9 + 11 + 17 + 22) / 4 = 14,75 dan rata-rata *waiting time* yang dibutuhkan adalah (0 + 9 + 11 + 17) / 4 = 9,25. Dapat dilihat bahwa *turnaround time* dan *waiting time* yang dikerjakan dengan SJF lebih kecil daripada dikerjakan secara FCFS, sehingga SJF memiliki kriteria CPU *scheduling* yang lebih baik daripada FCFS.

Pada contoh sebelumnya, *arrival time* masing-masing proses dianggap sama. Kasus instansi permasalahan penjadwalan lain yang memiliki proses *arrival time* yang mungkin berbeda adalah terdapat 4 proses yang ada pada *ready queue*, P_1, P_2, P_3, dan P_4, masing-masing memiliki *arrival time* secara terurut 9, 0, 4, 4 dan *burst time* secara terurut 4, 3, 9, 5. Untuk menentukan penjadwalan, diketahui C = Q = { P_1, P_2, P_3, P_4 }. Karena *arrival time* diketahui, Q harus diurutkan berdasarkan *arrival time* dari paling kecil hingga paling besar, sehingga Q = { P_2, P_3, P_4, P_1 }. Langkah-langkah penyelesaian instansi permasalahan tersebut sebagai berikut:

1) Langkah 1: Inisialisasi himpunan solusi kosong.

2) Langkah 2: Inisialisasi t = Q[0].arrival = waktu kedatangan P_2 = 0.

3) Langkah 3: Q masih tidak kosong, cari proses yang memiliki *next burst terkecil* dan *arrival time* ≤ t. Hanya P_2 yang memenuhi syarat. Hapus P_2 pada Q dan tambahkan P_2 pada S. Maka Q = { P_3, P_4, P_1 } dan S = { P_2 }. Karena Q masih tidak kosong, t = max(t + p.burst, Q[0].arrival) = max (t + *burst time* P_2, *arrival time* P_3) = max(0 + 3, 4) = max(3, 4) = 4.

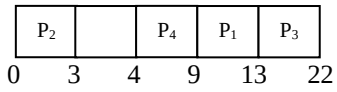
4) Langkah 4: Q masih tidak kosong, cari proses selanjutnya yang memiliki *next burst terkecil* dan *arrival time* ≤ t, yakni P_4. Hapus P_4 pada Q dan tambahkan P_4 pada S. Maka Q = { P_3, P_1 } dan S = { P_2, P_4 }. Karena Q masih tidak kosong, t = max(4 + 5, 4) = max(9, 4) = 9.

5) Langkah 5: Q masih tidak kosong, cari proses selanjutnya yang memiliki *next burst terkecil* dan *arrival time* ≤ t, yakni P_1. Hapus P_1 pada Q dan tambahkan P_1 pada S. Maka Q = { P_3 } dan S = { P_2, P_4, P_1 }. Karena Q masih tidak kosong, t = max(9 + 4, 4) = max(13, 4) = 13.

6) Langkah 6: Q masih tidak kosong, cari proses selanjutnya yang memiliki *next burst terkecil* dan *arrival time* ≤ t, yakni P_3. Hapus P_3 pada Q dan tambahkan P_3 pada S. Maka Q = { } dan S = { P_2, P_4, P_1, P_3 }. Karena Q sudah kosong, t tidak diubah.

7) Langkah 7: Q sudah kosong, kembalikan S = { P_2, P_4, P_1, P_3 }.

Gantt chart dari permasalahan diatas adalah sebagai berikut (*chart* tidak sesuai skala):



Di awal, CPU akan memproses P_2 terlebih dahulu dengan burst time 3 pada waktu 0 hingga *completion time* 3. Pada t = 3, P_2 selesai dikerjakan, tetapi tidak ada proses baru yang datang,

sehingga *ready queue* kosong dan CPU dalam keadaan *idle* (ditandai dengan kotak kosong). Kemudian, pada t = 4, CPU mendapatkan dua proses baru, yakni P_3 dan P_4, sehingga *ready queue* sekarang adalah P_3 dan P_4. CPU akan memproses P_4 terlebih dahulu karena *burst time*-nya lebih kecil, yakni 5, pada waktu 4 hingga *completion time* 9. Pada t = 8, CPU mendapatkan satu proses baru, yakni P_1, sehingga ada dua proses pada *ready queue*, yaitu P_3 dan P_1. Karena penjadwalan bersifat *non-preemptive* dan P_4 masih dijalankan, maka CPU akan memproses P_4 hingga selesai terlebih dahulu. Saat t = 9, CPU selesai memproses P_4, kemudian CPU akan memproses P_1 terlebih dahulu dengan *burst time* 6 hingga *completion time* 13. Terakhir, pada t = 13, P_1 selesai diproses dan CPU memproses P_3 dengan burst time 9 hingga *completion time* 22.

Turnaround time yang dibutuhkan masing-masing proses tersebut adalah *completion time* dikurangi *arrival time*, yakni 5, 3, 18, 5 secara terurut dari P_1 hingga P_4, dengan rata-ratanya 7,75. *Waiting time* yang dibutuhkan setiap proses secara terurut dari P_1 hingga P_4 adalah 1, 0, 9, dan 0, dengan rata-ratanya adalah 2,5. Karena CPU memiliki waktu *idle* sebesar 1, maka utilisasi CPU adalah ((22-1)/22) x 100% = 95,45%. Jika dikerjakan menggunakan FCFS, rata-rata *turnaround time* yang dibutuhkan adalah (14 + 3 + 9 + 14) / 4 = 10 dan rata-rata *waiting time* yang dibutuhkan adalah (10 + 0 + 0 + 9) / 4 = 4,75. Dapat dilihat bahwa *turnaround time* dan *waiting time* yang dikerjakan dengan SJF pada kasus ini juga lebih kecil daripada dikerjakan secara FCFS, sehingga SJF pada kasus ini juga memiliki kriteria CPU *scheduling* yang lebih baik daripada FCFS.

IV. Kesimpulan

Banyak algoritma penjadwalan yang dapat digunakan sebagai CPU *scheduler*, tetapi algoritma penjadwalan yang baik harus dapat memenuhi kriteria CPU *scheduler* yakni utilisasi CPU maksimum, *throughput* maksimum, *turnaround time* minimum, *waiting time* minimum, dan *response time* minimum. Penjadwalan sederhana *non-preemptive* dapat menggunakan *first-come first-served* (FCFS) dan *preemptive* dengan *round-robin* (RR). Alternatif penjadwalan adalah menggunakan sebuah prioritas dalam mengurutkan proses yang akan dijalankan, disebut juga sebagai *priority scheduling*. Salah satu *priority scheduling* yang sederhana adalah *shortest job first* (SJF) untuk *non-preemptive* dan *shortest remaining time first* (SRTF) untuk *preemptive*. Pada sistem *real-time*, terdapat *priority scheduling* berupa *rate monotonic scheduling* dan *earliest deadline first* (EDF).

Pada kasus *non-preemptive*, SJF memiliki kelebihan dalam memenuhi kriteria CPU *sheduling* yang lebih bagus dibandingkan FCFS, baik jika *arrival time* diketahui maupun ketika proses datang secara bersamaan. *Turnaround time* dan *waiting time* yang dihasilkan SJF lebih kecil daripada FCFS, sehingga sistem dapat lebih bekerja secara interaktif dan responsif tanpa waktu interupsi atau jeda yang lama. Hal ini tentunya sangat krusial dalam keberjalanan sistem *multiprogramming* yang membutuhkan responsitivitas penjadwalan, sehingga SJF sebagai algoritma yang memakai strategi *greedy* cocok dipakai dalam kasus penjadwalan *non-preemptive*.

Penulis tentunya menyadari dengan waktu yang cukup singkat ditambah dengan beban untuk dapat mengerjakan UAS dengan baik sehingga banyak hal yang bisa ditambahkan namun tidak sempat untuk dimasukkan, termasuk namun tak terbatas pada:pembahasan mengenai *priority scheduling* seperti implementasi dan analisis penjadwalan *preemptive* dengan SRTF, perbandingannya dengan *round robin*, serta *priority scheduling* pada sistem *real-time* seperti *rate monotonic scheduling* dan *earliest deadline first* (EDF).

Video Link at Youtube

Berikut penulis lampirkan video link penjelasan verbal makalah ini pada pranala berikut: https://youtu.be/x2aJln8_RFw

Ucapan Terima Kasih

Dengan selesainya makalah ini, saya mengucapkan terima kasih yang sebesar-besarnya kepada Allah SWT terhadap karunia-Nya sehingga saya bisa membuat makalah ini. Saya juga berterima kasih kepada orang tua dan teman-teman yang telah membantu saya dalam bantuan mental sehingga membuat saya semangat dalam menyelesaikan makalah ini. Terakhir, saya berterima kasih kepada ibu Dr. Masayu Leylia Khodra, S.T., M.T. dan ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen K1 dan K2 yang telah mengajar saya selama kuliah online IF2211, serta bapak Dr. Ir. Rinaldi Munir, MT. selaku dosen yang telah membimbing saya dalam mata kuliah IF2211 Strategi Algoritma.

Referensi

[1] A. Silberschatz, P. Baer Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ: Wiley, 2018.

[2] R. Munir, "Algoritma Greedy (Bagian 1)." Program Studi Teknik Informatika STEI ITB, pp. 1–68, 2021.

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, 15 Mei 2022

Amar Fadil - 13520103