Penerapan Kompleksitas Algoritma pada *Real-Time Operating System*

Daniel Riyanto - 13518075[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13518075@std.stei.itb.ac.id

***Abstract*—Pada era modern sekarang, setiap *operating system* sudah berjalan dengan sangat cepat dan memerlukan memori yang kecil dibandingkan dengan tahun-tahun sebelumnya. Proses yang berada di dalam *operating system* tersebut bisa berjalan dengan cepat karena adanya algoritma yang efisien di dalamnya. Keefisien algoritma diukur dari waktu eksekusi algoritma dan kebutuhan ruang memori. Algoritma yang efisien adalah algoritma yang meminimumkan kebutuhan waktu dan ruang. Mekanisme dalam *operating system* dikendalikan secara ketat yang memungkinkan beberapa proses untuk saling berkomunikasi, berbagi sumber daya, dan sinkronisasi aktivitas.**

Keywords*—proses, *operating system*, tugas, *buffer

I. PENDAHULUAN

Proses dalam sistem operasi adalah abstraksi dari program yang sedang berjalan dan merupakan unit kerja logis yang dijadwalkan oleh sistem operasi. Itu biasanya diwakili oleh struktur data yang berisi setidaknya ada kondisi eksekusi, identitas (*real-time*), atribut (mis. waktu eksekusi), dan sumber daya yang terkait dengannya. *Thread* adalah proses yang ringan yang membagi sumber daya dengan beberapa proses atau *thread* lainnya. Setiap *thread* harus berada dalam beberapa proses dan memanfaatkan sumber daya dari proses itu. Beberapa *thread* yang berada dalam proses yang sama berbagi sumber daya proses tersebut.

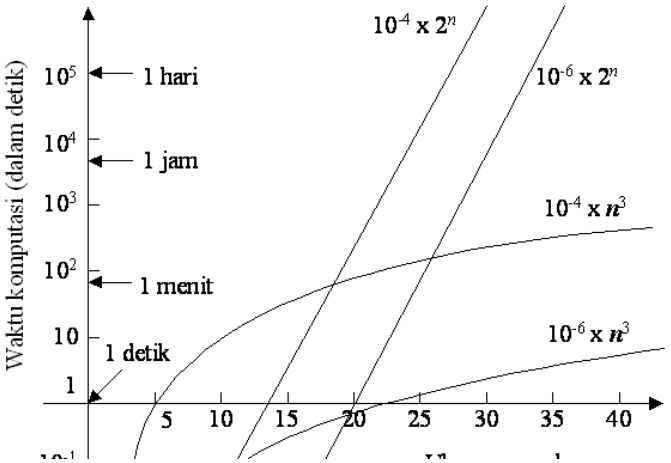
Semua *real-time operating system* harus menyediakan tiga fungsi spesifik untuk proses-proses, yaitu *scheduling*, *dispatching*, dan *intercommunication and synchronization*. Inti dari *operating system* adalah bagian terkecil yang menyediakan fungsi-fungsi ini. *Scheduler* menentukan proses mana yang akan dijalankan selanjutnya dalam *multitasking system*, sementara *dispatcher* melakukan pembukuan yang diperlukan untuk memulai proses itu. *Intertask communication and synchronization* memastikan bahwa proses-proses bekerja sama. Ketiga fungsi ini berhubungan erat dengan kompleksitas algoritma.[1]

II. TEORI DASAR

A. Kompleksitas Algoritma

Sebuah masalah dapat mempunyai banyak algoritma penyelesaian. Sebuah algoritma tidak saja harus benar tetapi juga harus efisien. Keefisien algoritma diukur dari waktu eksekusi algoritma dan kebutuhan ruang memori. Algoritma yang efisien adalah algoritma yang meminimumkan kebutuhan waktu dan ruang. Kebutuhan waktu dan ruang suatu algoritma bergantung pada ukuran masukan (n), yang menyatakan jumlah data yang diproses. Keefisien algoritma dapat digunakan untuk menilai algoritma yang bagus dari sejumlah algoritma penyelesaian masalah.[2]

Alasan mengapa dibutuhkan algoritam yang efisien ada di bawah ini.



Gambar 2.A.1 Grafik Hubungan Ukuran Masukan dengan Data Komputasi. Sumber: http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Kompleksitas%20Algoritma%20(2015).pdf. Diakses pada 5 Desember 2019, 11.20.

Kompleksitas algoritma terdiri dari dua macam yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu, dinyatakan oleh T(n), diukur dari jumlah komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n, di mana ukuran masukan (n) merupakan jumlah data yang diproses oleh sebuah algoritma. Sedangkan kompleksitas ruang, S(n), diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari masukan n. Dengan memenfaatkan kompleksitas waktu atau kompleksitas ruang, laju peningkatan waktu dan algoritma dapat ditentukan seiring dengan meningkatnya ukuran masukan (n).[3] Pada jurnal ini, akan dijelaskan lebih kea rah komplesitas waktu.

Kompleksitas waktu dibedakan atas tiga macam yaitu:

1. $T_{max}(n)$: kompleksitas waktu untuk kasus terburuk (*worst case*) → kebutuhan waktu maksimum.
2. $T_{min}(n)$: kompleksitas waktu untuk kasus terbaik (*best case*) → kebutuhan waktu minimum.
3. $T_{avg}(n)$: kompleksitas waktu untuk kasus rata-rata (*average case*) → kebutuhan waktu rata-rata.[2]

Kompleksitas algoritma dapat diukur dengan kasar melalui notasi asimptotik yaitu O(g) di mana g adalah sebuah fungsi dari ukuran data *input*. Contohnya:

- Kompleksitas linear O(n) – semua elemen diproses sekali.
- Kompleksitas kuadratik $O(n^2)$ – semua elemen diproses sebanyak n kali.[4]

T(n) = O(g(n)) jika terdapat konstanta C dan n_0 sedemikian sehingga:

$$T(n) \leq C(g(n)), \forall\ n \geq n_0$$

Contohnya: $3 * n^2 + n/2 + 12 \in O(n^2)$

Pengelompokan algoritma dengan O-Besar terurut membesar dari atas ke bawah berdasarkan kompleksitas algoritma ditampilkan pada tabel di bawah ini.

Kompleksitas	Notasi	Deskripsi
konstan	O(1)	Jumlah operasinya konstan dan tidak tergantung pada ukuran data *input*
logaritmik	$O(^2\log n)$ O(log n)	Jumlah operasinya sebanding dengan log(n) di mana n adalah ukuran data *input*
lanjar	O(n)	Jumlah operasinya sebandingan dengan ukuran data *input*
kuadratik	$O(n^2)$	Jumlah operasinya sebanding dengan kuadrat dari ukuran data *input*
kubik	$O(n^3)$	Jumlah operasinya sebanding dengan pangkat tiga dari ukuran data *input*
eksponensial	$O(2^n)$ $O(k^n)$	Jumlah operasinya itu eksponensial, berkembang dengan pesat
faktorial	O(n!)	Jumlah operasinya itu factorial, berkembang paling pesat

Tabel 2.A.1 Kelompok Algoritma berdasarkan Kompleksitas Waktu Asimptotik pada Kasus Terburuk

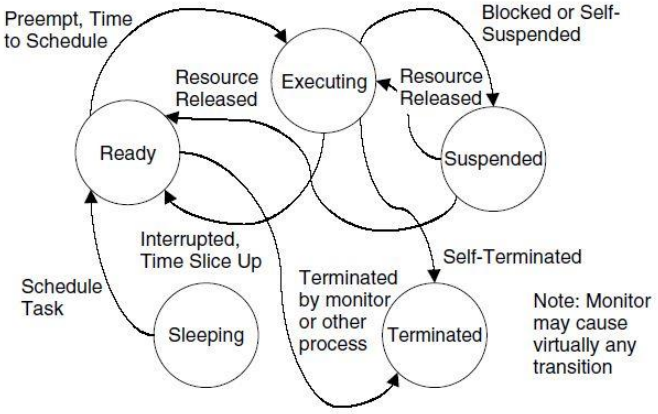
B. Real-Time Operating System

Ada banyak teori yang berhubungan dengan *real-time operating system* jadi diperlukan perhitungan yang teliti untuk mendapatkan banyak keuntungan dari teori-teori tersebut. Umumnya tiap *real-time system* berhubungan, yang artinya, interaksi alami mereka dengan peristiwa eksternal biasanya membutuhkan beberapa tugas yang serempak untuk mengatasi beberapa kontrol *thread*. Proses adalah objek aktif dari suatu sistem dan merupakan unit dasar pekerjaan yang ditangani oleh *scheduler*. Saat sebuah proses berjalan, proses itu bisa mengubah keadaannya dan kapan saja, dan mungkin salah satu dari keadaan-keadaan yang ada di bawah ini:

- *Dormant* (atau *sleeping*). Tugas telah dibuat dan diinisialisasi. Tugas ini belum siap dieksekusi, yang artinya proses dalam kondisi ini belum memnuhi syarat untuk dieksekusi.
- *Ready*. Proses-proses dalam kondisi ini sudah memenuhi syarat dan siap untuk dieksekusi tetapi belum melakukan eksekusi.
- *Executing*. Suatu proses menjalankan eksekusinya.
- *Suspended* (atau *blocked*). Proses-proses yang menunggu sumber daya tertentu dan belum dalam kondisi *ready* berada dalam kondisi *suspended* atau *blocked*.
- *Terminated*. Proses telah selesai dieksekusi, diakhiri sendiri dan tidak lagi dibutuhkan.

Sama dengan setiap proses, setiap *thread* hanya dapat berada di dalam salah satu dari lima kondisi ini kapan saja.

Diagram dari keadaan parsial yang sesuai dengan keadaan-keadaan proses atau *thread* digambarkan dalam Gambar 2.B.1.. Perlu dicatat bahwa sistem operasi yang berbeda memiliki aturan penamaan yang berbeda juga tetapi kondisi-kondisi yang ada di dalam beberapa tata nama yang bebas ini ada dalam satu bentuk atau lainnya di semua *real-time operating system*. Banyak sistem operasi modern memungkinkan beberapa proses



Gambar 2.B.1 Diagram kondisi-kondisi proses atau thread. Sumber: Laplante, Philip A., Real-Time Systems Design and Analysis, 3rd Edition, *bab 3.*

yang dibuat dalam program yang sama untuk memiliki akses tidak terbatas ke memori yang melalui fasilitas *thread*.

III. PEMBAHASAN

Model tugas sejauh ini dipertimbangkan berasumsi bahwa semua tugas bersifat independen dan bahwa semua tugas sapat dikesampingkan pada setiap titik pelaksanaannya. Namun, dari sudut pandang yang praktis, asumsi ini tidak logis karena interaksi tugas diperlukan dalam aplikasi yang paling umum. Dalam praktik, diperlukan mekanisme yang dikendalikan secara ketat yang memungkinkan beberapa tugas untuk berkomunikasi, berbagi sumber daya, dan sinkronisasi aktivitas.

A. Buffering Data

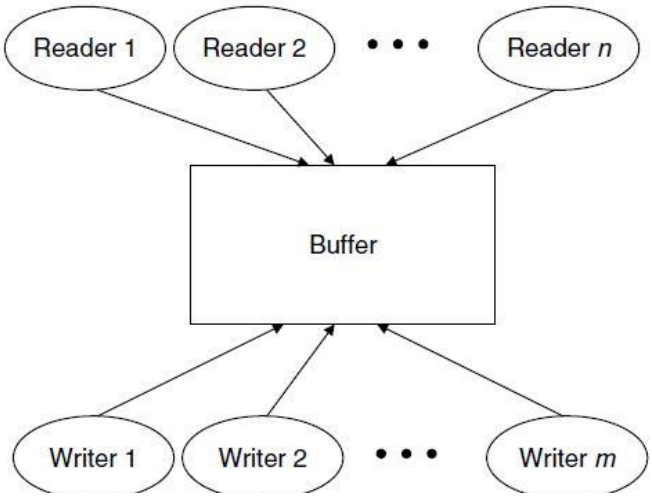
Beberapa mekanisme dapat digunakan untuk meneruskan

data antar tugas dalam *multitasking system*. Yang paling sederhana dan tercepat di antara beberapa mekanisme ini adalah penggunaan variable global. Meskipun variable global dianggap bertentangan dengan praktik *software engineering* yang baik, variabel global sering digunakan dalam operasi kecepatan tinggi.

Salah satu masalah yang terkait dengan penggunaan variabel global adalah tugas-tugas yang prioritasnya lebih tinggi dapat mendahului rutinitas yang prioritasnya rendah pada waktu yang tidak tepat dan merusak data global. Misalnya, satu tugas dapat menghasilkan data sebesar 100 unit per detik secara konstan sedangkan yang lain dapat mengkonsumsi data-data ini dengan laju kurang dari 100 unit per detik. Dengan asumsi bahwa interval produksi itu terbatas (dan relatif pendek), tingkat konsumsi yang lebih lambat dapat ditampung jika produsen mengisi penyimpanan *buffer* dengan data. *Buffer* menampung kelebihan data sampai tugas konsumen bisa menyusul. *Buffer* dapat berupa *queue* atau struktur data lainnya termasuk variabel yang tidak terorganisir. Tentu saja jika tugas konsumen mengkonsumsi informasi ini lebih cepat daripada yang dihasilkan, atau jika konsumen tidak dapat mengikuti dengan produsen, masalah pun terjadi. Pemilihan *buffer* yang ukurannya sesuai adalah kritis dalam mengurangi atau menghilangkan masalah ini.[1]

B. Time-Relative Buffering

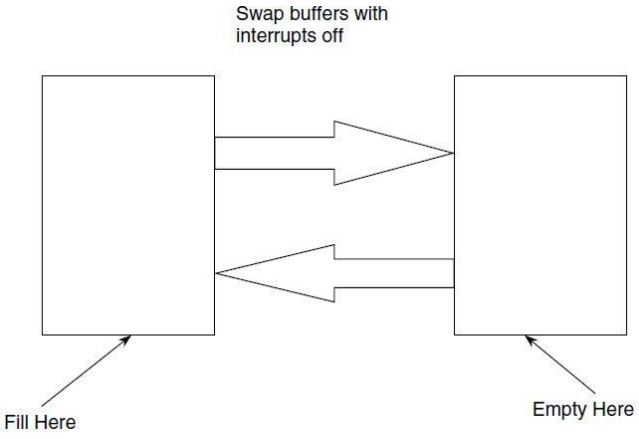
Umumnya, penggunaan global variabel digunakan dalam teknik *double buffering* atau *Ping-Pong buffering*. Teknik ini digunakan katika data *time-relative* perlu dipindahkan antara siklus-siklus yang berbeda tingkat, atau ketika satu set data lengkap diperlukan oleh satu proses tetapi hanya dapat disediakan secara lambat oleh proses lain. Situasi ini hanyalah sebuah versi dari masalah *bounded-buffer* yang klasik di mana satu blok memori digunakan sebagai repositori untuk data yang dihasilkan "*writers*" dan dikonsumsi oleh "*readers*." Generalisasi selanjutnya adalah masalah *readers* dan *writers* di mana ada banyak *reader* dan *writer* yang sumber dayanya sama yang ditampilkan pada Gambar 3.B.1.[1]



Gambar 3.B.1 Masalah readers dan writers dengan n reader dan m writer. Sumber dayanya adalah bounded buffer. Buffer hanya bisa ditulis atau dibaca oleh satu reader atau writer. Sumber: Laplante, Philip A., Real-Time Systems Design and Analysis, 3rd Edition, *bab 3.*

Banyak telemetri sistem, yang mengirimkan blok data dari satu perangkat ke lainnya, menggunakan skema *double-buffering* dengan perangkat keras atau perangkat lunak untuk beralih mengganti *buffer*. Strategi ini juga digunakan dalam *disk controllers*, *graphical interfaces*, peralatan navigasi, control robot, dan banyak tempat lainnya. Contohnya, di layer operator untuk pabrik saus pasta, misalkan garis digambar pada layer secara satu per satu sampai gambar selesai. Dalam sistem animasi, itu tidak diinginkan untuk melihat proses menggambar ini. Namun, jika perangkat lunak menggambar satu gambar layar sambil menampilkan yang lain dan kemudian membalik layar ketika gambar yang baru sudah selesai digambar, beberapa perintah menggambar garis individu tidak akan terlihat. (Gambar 3.B.2)[1]

Contoh dari *time-correlated buffering* adalah unit pengukuran



Gambar 3.B.2 Konfigurasi double-buffering. Dua buffer yang identik diisi dan dikosongkan oleh tugas-tugas yang terus bergantian. Switching dilakukan baik oleh pointer perangkat lunak atau diskrit perangkat keras. Sumber: Laplante, Philip A., Real-Time Systems Design and Analysis, 3rd Edition, *bab 3.*

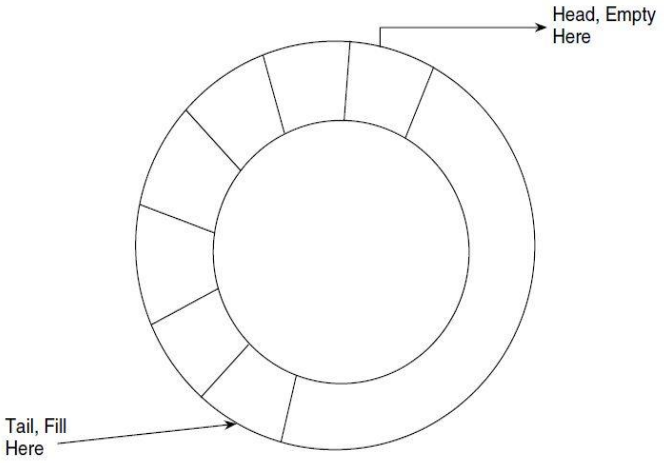
inersia. Unit ini membaca pulsa akselerometer dengan arah *x*, *y*, dan *z* dalam siklus 10 milidetik. Data ini akan diproses dalam siklus 40 milidetik, yang memiliki prioritas yang lebih rendah daripada siklus 10 milidetik (mis. itu dapat dikesampingkan). Data akselerometer diproses dalam siklus 40 milidetik haruslah *time-relative*, yang artinya, tidak diinginkan terjadinya memproses pulsa akselerometer *x* dan *y* dari waktu *t* tetapi pulsa akselerometer dari waktu *t* + 1. Skenario ini dapat terjadi jika siklus 40 milidetik telah selesai memproses data *x* dan *y* tetapi mendapatkan *interrupt* oleh siklus 10 milidetik. Untuk menghindari masalah ini, gunakan variabel *buffer xb*, *yb*, dan *zb* di siklus 40 milidetik dan *buffer* mereka, dengan *interrupt* dinonaktifkan. Siklus 40 milidetik mungkin berisi kode C berikut untuk menangani *buffering*:

```
ntrof();                /* menonaktifkan interrupts */
xb=x;                   /* buffer data */
yb=y;
zb=z;
intron();               /* mengaktifkan interrupts */
process(xb,yb,zb);      /* menggunakan buffered data */
```

Dalam praktik, prosedur pertama dalam setiap siklus akan menjadi rutinitas *buffering* ke *buffer* semua data dari tugas dengan prioritas lebih tinggi ke tugas saat ini (siklus "buffer in"). Prosedur terakhir dalam siklus adalah rutinitas untuk *buffer out* datake semua tugas yang prioritasnya rendah (siklus "buffer out").[1]

C. Ring Buffers

Struktur data spesial yang disebut *circular queue* atau *ring buffer* yang digunakan dalam cara yang sama sebagai *queue* dan dapat digunakan untuk memecahkan masalah sinkronisasi beberapa tugas *reader* dan *writer*. Namun, *ring buffer* lebih mudah dikelola daripada *double buffer* atau *queue* ketika ada ada lebih dari dua *reader* dan *writer*.



Gambar 3.C.1 Sebuah ring buffer. Melakukan penulisan ke buffer di indeks tail dan membaca data dari indeks head. Sumber: Laplante, Philip A., Real-Time Systems Design and Analysis, 3rd Edition, *bab 3.*

Di dalam *ring buffer*, *input* dan *output* di *list* secara bersama tercapai dengan menjaga indeks *head* dan *tail*. Data-data dimuat di bagian *tail* dan dibaca dari *head*. Gambar 3.C.1 menggambarkan hal ini. Misalkan *ring buffer* adalah struktur tipe *ring_buffer* yang mencakup *array integer* yang berukuran *N* yang disebut *content*, yaitu,

```
typedef struct ring_buffer
{
    int contents[N];
    int head;
    int tail;
}
```

Ini lebih lanjut diasumsikan bahwa indeks *head* dan *tail* telah dinisialisasi ke 0, yaitu mulai dari *buffer*.

Implementasi dari operasi *read* (data, S) dan *write* (data, S) yang membaca dan menulis ke *ring buffer* S, masing-masing diberikan di bawah kode C.

```
void read (int data, ring_buffer *s)
{
    if (s->head==s->tail)
     data=NULL; /* underflow */
    else
     {
       data=s->contents +head; /* ambil data dari buffer */
       s->head=(s->head+1) % N; /* decrement indeks head */
     }
}

void write (int data, ring_buffer *s)
{
    if ((s->tail+1) %N==head)
     error(); /* overflow, aktifkan penangan kesalahan */
    else
     {
       s->contents+tail=data;
       tail=(tail+1) % N; /* mengurus wrap-around */
     }
}
```

Kode tambahan diperlukan untuk menguji kondisi *overflow* di *ring buffer*, dan tugas yang menggunakan *ring buffer* perlu menguji data untuk nilai *underflow* (NULL). *Overflow* terjadi ketika upaya dilakukan untuk menulis data ke *queue* yang penuh. *Underflow* adalah kondisi ketika tugas mencoba untuk mengambil data dari *buffer* yang kosong.

D. Mailboxes

Maibox atau *message exchange* adalah alat *intertask communication* yang tersedia dalam banyak komersial, *full-featured operating systems*. *Mailbox* adalah lokasi memori yang dapat digunakan satu atau beberapa tugas untuk mengirim data atau untuk sinkronisasi. Tugas bergantung pada kernel untuk memungkinkannya untuk menulis ke lokasi melalui operasi pos atau membaca darinya melalui operasi *pend*.

Operasi-operasi *mailbox*, *pend* dan *post* dapat dijelaskan dengan antarmuka yang di bawah ini:

```
void pend (int data, s);

void post (int data, s);
```

Perbedaan antara operasi *pend* dan hanya *polling mailbox* adalah tugas yang tertunda ditangguhkan sambal menunggu data muncul. Jadi, tidak ada waktu yang terbuang terus-menerus untuk memeriksa *mailbox*; yaitu kondisi menunggu yang sibuk dihilangkan.

Data-data yang dilewatkan dapat menjadi *flag* yang digunakan untuk melindungi sumber daya kritis (disebut *key*), sepotong data, atau *pointer* ke struktur data. Dalam sebagian besar implementasi, ketika *key* diambil dari *mailbox*, *mailbox* dikosongkan. Demikian, meskipun beberapa tugas dapat tertunda di *mailbox* yang sama, hanya satu tugas yang dapat menerima *key*. Sejak *key* mewakili akses ke sumber daya kritis, akses yang serempak dihalangi.

Mailbox paling baik diimplementasikan dalam sistem berdasarkan pada model tugas blok control dengan tugas *supervisor*. Sebuah tabel yang berisi daftar tugas dan sumber daya yang diperlukan (mis., *mailbox*, *printer*, dll.) disimpan

dengan tabel kedua yang berisi daftar beberapa sumber daya dan status-status mereka. Contohnya, pada Tabel 3.D.1 dan 3.D.2, tiga sumber daya saat ini ada, yaitu sebuah *printer* dan dua *mailbox*. Di sini *printer* sedang digunakan oleh tugas #100, sedangkan *mailbox* #1 sedang digunakan (saat ini sedang dibaca dari atau ditulis ke) oleh tugas #102. Tugas #104 sedang tertunda di *mailbox* #1 dan ditangguhkan karena tidak tersedia. *Mailbox* #2 saat ini tidak sedang digunakan atau ditunda oleh tugas apapun.

ID Tugas #	Sumber Daya	Status
100	*Printer*	Mempunyai tugas
102	*Mailbox* 1	Mempunyai tugas
104	*Mailbox* 1	Tertunda

Tabel 3.D.1 Tabel Permintaan Sumber Daya Tugas. Sumber: : Laplante, Philip A., Real-Time Systems Design and Analysis, 3rd Edition, *bab 3.*

Sumber Daya	Status	Pemilik
Printer 1	Sibuk	100
Mailbox 1	Sibuk	102
Mailbox 2	Kosong	Tidak ada

Tabel 3.D.2 Tabel Sumber Daya yang Digunakan Bersama Dengan Tabel Permintaan Sumber Daya Tugas. Sumber: Laplante, Philip A., Real-Time Systems Design and Analysis, 3rd Edition, *bab 3.*

Ketika *supervisor* dipanggil oleh *system call* atau *hardware interrupt*, itu memeriksa tabel untuk memeriksa apakah ada tugas yang tertunda di *mailbox*. Jika *key* tersedia (status *key* adalah "full") maka tugas itu harus dimulai kembali. Demikian pula jika tugas dikirimkan ke *mailbox* maka sistem operasi harus memastikan bahwa *key* ditempatkan di *mailbox* dan statusnya diperbarui ke "full".

Sering ada operasi lain di *mailbox*. Contohnya, dalam beberapa implementasi, operasi *accept* diizinkan. *Accept* mengizinkan tugas membaca *key* jika tersedia, atau segera mengembalikan kode kesalahan jika *key* tidak tersedia. Dalam implementasi lain, operasi *pend* dilengkapi dengan batas waktu untuk mencegah kebuntuan.

E. Queues

Beberapa sistem operasi mendukung satu tipe *mailbox* yang dapat mengantri banyak *pend* permintaan. Sistem-sistem ini menyediakan operasi *qpost, qpend, dan qaccept* untuk *post, pend,* dan *accept* data ke/dari *queue*. Dalam hal ini, *queue* bisa dianggap sebagai *array mailbox* apapun, dan implementainya difalitisasi melalui tabel sumber daya sama yang sudah dibahas.

Queue tidak boleh digunakan untuk melewati *array* data, melainkan *pointer* yang seharusnya digunakan sebagai gantinya. *Queue* berguna dalam mengimplementasikan *server* perangkat tempat kumpulan pangkat terlibat. Di sini *ring buffer* menyimpan permintaan untuk sebuah perangkat, dan *queue* bisa digunakan di *head* dan *tail* untuk mengontrol akses ke *ring buffer*. Skema ini berguna dalam pembangunan perangkat lunak pengontrol perangkat.

F. Critical Regions

Multitasking system berkaitan dengan *resource sharing*. Dalam kebanyakan kasus, sumber data ini hanya dapat digunakan oleh satu tugas dan saat penggunaannya tidak boleh mendapatkan *interrupt*. Sumber daya semacam itu dapat digunakan kembali secara berturut-turut dan mereka termasuk peripheral, *shared memory*, dan CPU tertentu. Saat CPU melindungi dirinya sendiri terhadap penggunaan simultan, kode yang berinteraksi dengan sumber daya lainnya tidak bisa melindungi dirinya sendiri. Kode semacam ini dinamakan *critical region*. Jika dua tugas masuk ke *critical region* yang sama secara bersamaan maka *error* yang berbahaya dapat terjadi. Sebagai contoh, dalam dua program bahasa C, Task_A dan Task_B berjalan dalam *round-robin system*. Task_B menampilkan pesan, "Saya adalah Task_B" dan Task_A menampilkan pesan, "Saya adalah Task_A". Di tengah-tengah pencetakan ke layar monitor, Task_B mendapatkan *interrupt* oleh Task_A yang memulai menampilkan pesannya sendiri. Hasilnya adalah *output* yang salah:

```
Saya adalah Saya adalah Task_A Task_B
```

Perhatian ditempatkan di tengah teks untuk menunjukkan bahwa itu mengakibatkan *interrupt* pada keluaran dari Task_B. Komplikasi yang lebih serius dapat terjadi jika kedua tugas tadi mengendalikan perangkat pada sebuah *embedded system*. Penggunaan sumber daya yang dapat digunakan kembali secara berturut-turut dapat menghasilkan sebuah *collision*. Fokus sekarang adalah menyediakan sebuah mekanisme untuk mencegah *collision.*

IV. KESIMPULAN

Dari beberapa analisa yang sudah dijelaskan, bisa dilihat bahwa setiap proses dalam sistem operasi memerlukan algoritma yang efisien sehingga sistem operasi bisa berjalan dengan cepat. Mekanisme dalam *operating system* umumnya dikontrol secara ketat sehingga dapat memungkinkan beberapa proses yang ada di dalamnya bisa saling berkomunikasi, berbagi sumber daya dan sinkronisasi aktivitas. Jika pengetahuan tentang *buffer* yang dimiliki itu tinggi maka bisa membuat sistem operasi yang berjalan dengan cepat dan hanya memerlukan memori yang kecil.

V. UCAPAN TERIMA KASIH

Penulis bersyukur kepada kepada Tuhan Yang Maha Esa karena telah memberikan kesehatan dan waktu di tengah kesibukan ini sehingga makalah ini dapat selesai walaupun tidak begitu bagus. Penyusunan makalah ini tentu tidak lepas dari dukungan dan partisipasi semua pihak. Untuk itu penulis menyampaikan terima kasih kepada Bu Fariska Zakhralativa Ruskanda, S.T., M.T. sebagai dosen Matematika Diskrit Kelas 03 dan orang tua yang selalu mendoakan penulis.

REFERENSI

[1] Laplante, Philip A., *Real-Time Systems Design and Analysis, 3rd Edition,* bab 3.

[2] http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Kompleksitas%20Algoritma%20(2015).pdf. Diakses pada 5 Desember 2019, 11.20.

[3] Azizah, Ulfah Nur, *Perbandingan Detektor Tepi Prewit Dan Detektor Tepi Laplacian Berdasarkan Kompleksitas Waktu Dan Citra Hasil*, hlm. 31.
[4] Nakov, Svetlin, Veselin Kolev dan rekan-rekannya, *Fundamentals of Computer Programming with C#*, bab 19.

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, 5 Desember 2019

Daniel Riyanto
13518075