B a b 2

Proses dan Thread

Tujuan Pelajaran

Setelah mempelajari bab ini, Anda diharapkan :

- Memahami konsep dasar dan definisi dari proses
- Menjelaskan keadaan/status proses
- Memahami Process Control Block
- Memahami operasi-operasi Proses
- Memahami Hubungan antar Proses
- Memahami konsep Thread
- Memahami konsep dasar Penjadualan CPU
- Memahami algoritma-algoritma Penjadualan CPU, meliputi FCFS, SJF, Prioritas, Round Robin, Multiprocessor

2.1. Proses

Satu diskusi mengenai sistem operasi yaitu bahwa ada sebuah pertanyaan mengenai untuk apa menyebut semua aktivitas CPU. Sistem batch mengeksekusi jobs, sebagaimana suatu sistem *time-shared* telah menggunakan program pengguna, atau tugas-tugas/pekerjaan-pekerjaan. Bahkan pada sistem tunggal, seperti *Microsoft Windows* dan *Macintosh OS*, seorang pengguna mampu untuk menjalankan beberapa program pada saat yang sama: sebuah *Word Processor*, *Web Browser*, dan paket *e-mail*. Bahkan jika pengguna dapat melakukan hanya satu program pada satu waktu, sistem operasi perlu untuk mendukung aktivitas program internalnya sendiri, seperti managemen memori. Dalam banyak hal, seluruh aktivitas ini adalah serupa, maka kita menyebut seluruh program itu proses-proses (*processes*).

Istilah job dan proses digunakan hampir dapat dipertukarkan pada tulisan ini. Walau kami pribadi lebih menyukai istilah proses, banyak teori dan terminologi sistem-operasi dikembangkan selama suatu waktu ketika aktivitas utama sistem operasi adalah job processing. Akan menyesatkan untuk menghindari penggunaan istilah umum yang telah diterima bahwa memasukkn kata job (seperti penjadualan job) hanya karena proses memiliki job pengganti/ pendahulu.

2.1.1. Konsep Dasar dan Definisi Proses

Secara informal; proses adalah program dalam eksekusi. Suatu proses adalah lebih dari kode program, dimana kadang kala dikenal sebagai bagian tulisan. Proses juga termasuk aktivitas yang sedang terjadi, sebagaimana digambarkan oleh nilai pada program counter dan isi dari daftar prosesor/ processor's register. Suatu proses umumnya juga termasuk process stack, yang berisikan data temporer (seperti parameter metoda, address yang kembali, dan variabel lokal) dan sebuah data section, yang berisikanvariabel global.

Kami tekankan bahwa program itu sendiri bukanlah sebuah proses; suatu program adalah satu entitas pasif; seperti isi dari sebuah berkas yang disimpan didalam disket, sebagaimana sebuah proses dalam suatu entitas aktif, dengan sebuah *program counter* yang mengkhususkan pada instruksi selanjutnya untuk dijalankan dan seperangkat sumber daya/ *resource* yang berkenaan dengannya.

Walau dua proses dapat dihubungkan dengan program yang sama, program tersebut dianggap dua urutan eksekusi yang berbeda. Sebagai contoh, beberapa pengguna dapat menjalankan copy yang berbeda pada mail program, atau pengguna yang sama dapat meminta banyak copy dari program editor. Tiap-tiap proses ini adakah proses yang berbeda dan walau bagian tulisan-text adalah sama, data section bervariasi. Juga adalah umum untuk memiliki proses yang menghasilkan banyak proses begitu ia bekerja. Kami mendiskusikan masalah tersebut pada Bagian 2.4.

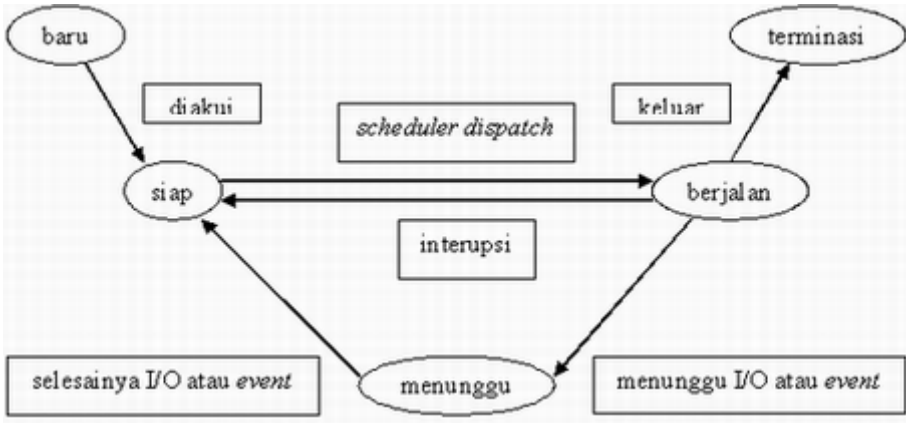
2.1.2. Keadaan Proses

Sebagaimana proses bekerja, maka proses tersebut merubah state (keadaan statis/ asal). Status dari sebuah proses didefinisikan dalam bagian oleh aktivitas yang ada dari proses tersebut. Tiap proses mungkin adalah satu dari keadaan berikut ini:

- *New*: Proses sedang dikerjakan/ dibuat.
- *Running*: Instruksi sedang dikerjakan.
- *Waiting*: Proses sedang menunggu sejumlah kejadian untuk terjadi (seperti sebuah penyelesaian I/Oatau penerimaan sebuah tanda/ signal).
- *Ready*: Proses sedang menunggu untuk ditugaskan pada sebuah prosesor.

- *Terminated*: Proses telah selsesai melaksanakan tugasnya/ mengeksekusi.

Nama-nama tersebut adalah arbitrer/ berdasar opini, istilah tersebut bervariasi disepanjang sistem operasi. Keadaan yang mereka gambarkan ditemukan pada seluruh sistem. Namun, sistem operasi tertentu juga lebih baik menggambarkan keadaan/ status proses. Adalah penting untuk menyadari bahwa hanya satu proses dapat berjalan pada prosesor mana pun pada waktu kapan pun. Namun, banyak proses yang dapat ready atau waiting. Keadaan diagram yang berkaitan dangan keadaan tersebut dijelaskan pada Gambar 2-1.



Gambar 2-1. Keadaan Proses

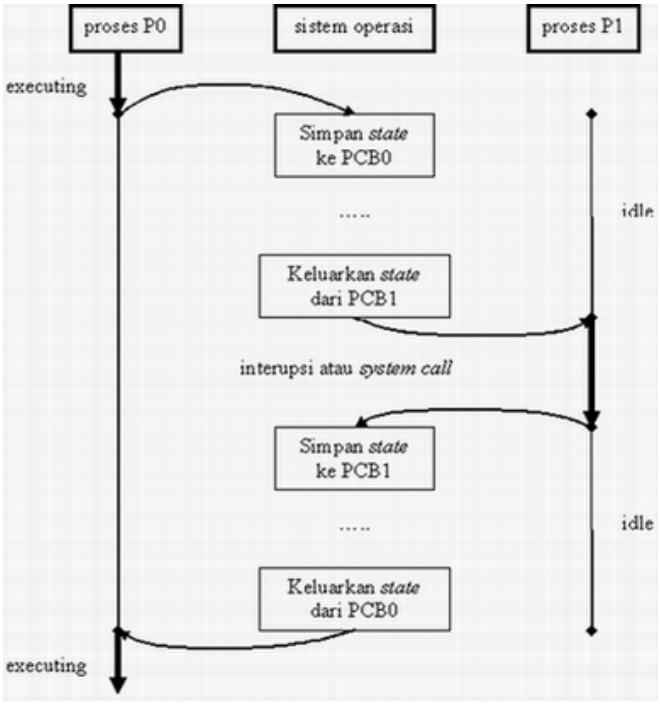
2.1.3. *Process Control Block*

Tiap proses digambarkan dalam sistem operasi oleh sebuah *process control block* (PCB) - juga disebut sebuah *control block*. Sebuah PCB ditunjukkan dalam Gambar 2-2. PCB berisikan banyak bagian dari informasi yang berhubungan dengan sebuah proses yang spesifik, termasuk ini:

- Keadaan proses: Keadaan mungkin, *new*, *ready*, *running*, *waiting*, *halted*, dan juga banyak lagi.
- *Program counter*: *Counter* mengindikasikan address dari perintah selanjutnya untuk dijalankan untuk proses ini.
- CPU register: Register bervariasi dalam jumlah dan jenis, tergantung pada rancangan komputer.
- Register tersebut termasuk accumulator, index register, stack pointer, general-puposes register, ditambah code information pada kondisi apa pun. Besertaan dengan program counter, keadaan/ status informasi harus disimpan ketika gangguan terjadi, untuk memungkinkan proses tersebut berjalan/bekerja dengan benar setelahnya (lihat Gambar 2-3).
- Informasi managemen memori: Informasi ini dapat termasuk suatu informasi sebagai nilai dari dasar dan batas register, tabel page/ halaman, atau tabel segmen tergantung pada sistem memori yangdigunakan oleh sistem operasi (lihat Bab 4).
- Informasi pencatatan: Informasi ini termasuk jumlah dari CPU dan waktu riil yang digunakan, batas waktu, jumlah akun, jumlah job atau proses, dan banyak lagi.
- Informasi status I/O: Informasi termasuk daftar dari perangkat I/O yang di gunakan pada proses ini, suatu daftar open berkas dan banyak lagi.

- PCB hanya berfungsi sebagai tempat menyimpan/ gudang untuk informasi apa pun yang dapat bervariasi dari prose ke proses.

Gambar 2-2. *Process Control Block*.



Gambar 2-3. *CPU Register*.

2.1.4. *Threads*

Model proses yang didiskusikan sejauh ini telah menunjukkan bahwa suatu proses adalah sebuah program yang menjalankan eksekusi thread tunggal. Sebagai contoh, jika sebuah proses menjalankan sebuah program Word Processor, ada sebuah thread tunggal dari instruksi-instruksi yang sedang dilaksanakan.

Kontrol thread tunggal ini hanya memungkinkan proses untuk menjalankan satu tugas pada satu waktu. Banyak sistem operasi modern telah memiliki konsep yang dikembangkan agar memungkinkan sebuah proses untuk memiliki eksekusi *multithreads*, agar dapat dapat secara terus menerus mengetik dalam karakter dan menjalankan pengecek ejaan didalam proses yang sama. Maka sistem operasi tersebut memungkinkan proses untuk menjalankan lebih dari satu tugas pada satu waktu. Pada Bagian 2.5 akan dibahas proses *multithreaded*.

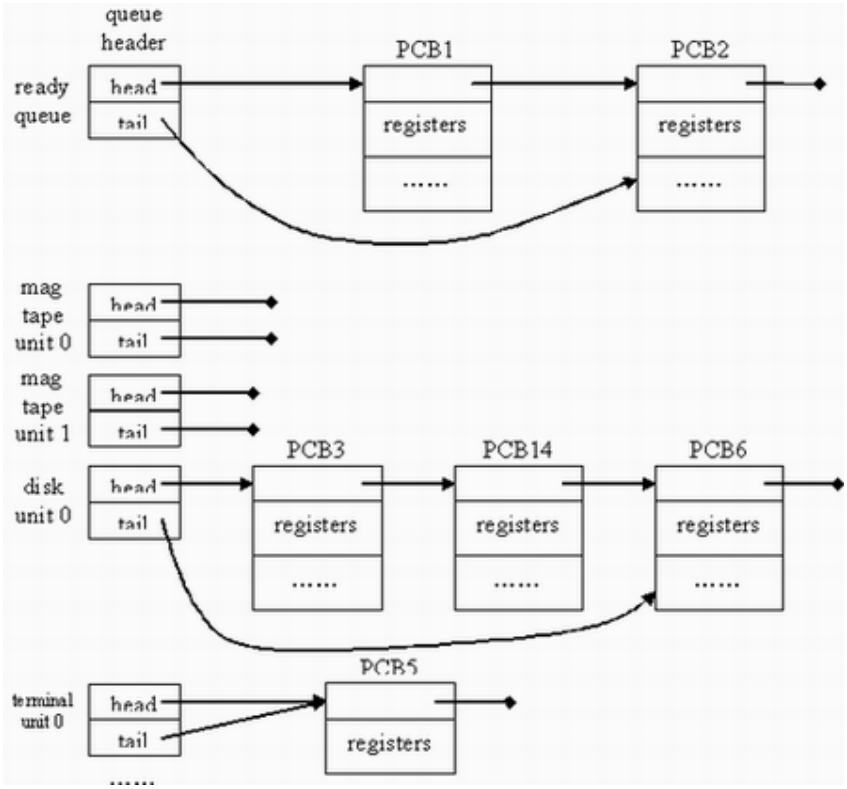
2.2. Penjadualan Proses

Tujuan dari multiprogramming adalah untuk memiliki sejumlah proses yang berjalan pada sepanjang waktu, untuk memaksimalkan penggunaan CPU. Tujuan dari pembagian waktu adalah untuk mengganti CPU diantara proses-proses yang begitu sering sehingga pengguna dapat berinteraksi dengan setiap program sambil CPU bekerja. Untuk sistem uniprosesor, tidak akan ada lebih dari satu proses berjalan. Jika ada proses yang lebih dari itu, yang lainnya akan harus menunggu sampai CPU bebas dan dapat dijadualkan kembali.

2.2.1. Penjadualan Antrian

Ketika proses memasuki sistem, mereka diletakkan dalam antrian job. Antrian ini terdiri dari seluruh proses dalam sistem. Proses yang hidup pada memori utama dan siap dan menunggu/ wait untuk mengeksekusi disimpan pada sebuah daftar bernama ready queue. Antrian ini biasanya disimpan sebagai daftar penghubung. Sebuah header ready queue berisikan penunjuk kepada PCB-PCB awal dan akhir. Setiap PCB memiliki pointer field yang menunjukkan proses selanjutnya dalam ready queue.

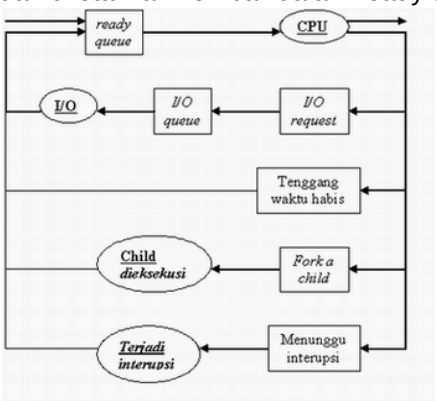
Juga ada antrian lain dalam sistem. Ketika sebuah proses mengalokasikan CPU, proses tersebut berjalan/bekerja sebentar lalu berhenti, di interupsi, atau menunggu suatu kejadian tertentu, seperti penyelesaian suatu permintaan I/O. Pada kasus ini sebuah permintaan I/O, permintaan seperti itu mungkin untuk sebuah tape drive yang telah diperuntukkan, atau alat yang berbagi, seperti disket. Karena ada banyak proses dalam sistem, disket bisa jadi sibuk dengan permintaan I/O untuk proses lainnya. Maka proses tersebut mungkin harus menunggu untuk disket tersebut. Daftar dari proses yang menunggu untuk peralatan I/O tertentu disebut sebuah device queue. Tiap peralatan memiliki device queuenya sendiri (Lihat Gambar 2-4).



Gambar 2-4. Device Queue.

Reprensentasi umum untuk suatu diskusi mengenai penjadualan proses adalah diagram antrian, seperti pada Gambar 2-5. Setiap kotak segi empat menunjukkan sebuah antrian. Dua tipe antrian menunjukan antrian yang siap dan suatu perangkat device queues. Lingkaran menunjukkan sumber-sumber yang melayani sistem. Sebuah proses baru pertama-tama ditaruh dalam ready queue. Lalu menunggu dalam ready queue sampai proses tersebut dipilih untuk dikerjakan/lakukan atau di dispatched. Begitu proses tersebut mengalokasikan CPU dan menjalankan/ mengeksekusi, satu dari beberapa kejadian dapat terjadi.

- Proses tersebut dapat mengeluarkan sebuah permintaan I/O, lalu di tempatkan dalam sebuah antrian I/O.
- Proses tersebut dapat membuat subproses yang baru dan menunggu terminasinya sendiri.
- Proses tersebut dapat digantikan secara paksa dari CPU, sebagai hasil dari suatu interupsi, dan diletakkan kembali dalam ready queue.



Gambar 2-5. Diagram Anrian.

Dalam dua kasus pertama, proses akhirnya berganti dari waiting state menjadi ready state, lalu diletakkan kembali dalam ready queue. Sebuah proses meneruskan siklus ini sampai berakhir, disaat dimana proses tersebut diganti dari seluruh queue dan memiliki PCB nya dan sumber-sumber/ resources dialokasikan kembali.

2.2.2. Penjadual / Scheduler

Sebuah proses berpindah antara berbagai penjadualan antrian selama umur hidupnya. Sistem operasi harus memilih, untuk keperluan penjadualan, memproses antrian-antrian ini dalam cara tertentu. Pemilihan proses dilaksanakan oleh penjadual yang tepat/ cocok. Dalam sistem batch, sering ada lebih banyak proses yang diserahkan daripada yang dapat dilaksanakan segera. Proses ini dipitakan/ disimpan pada suatu alat penyimpan masal (biasanya disket), dimana proses tersebut disimpan untuk eksekusi dilain waktu. Penjadualan long term, atau penjadual job, memilih proses dari pool ini dan mengisinya kedalam memori eksekusi.

Sebuah proses dapat mengeksekusi untuk hanya beberapa milidetik sebelum menunggu permintaan I/O. Seringkali, penjadualan shorterm mengeksekusi paling sedikit sekali setiap 100 milidetik. Karena durasi waktu yang pendek antara eksekusi, penjadualan shorterm haruslah cepat. Jika memerlukan 10 mili detik untuk menentukan suatu proses eksekusi selama 100 mili detik, maka 10/(100 + 10) = 9 persen CPU sedang digunakan (terbuang) hanya untuk pekerjaan penjadualan.

Penjadualan longterm pada sisi lain, mengeksekusi jauh lebih sedikit. Mungkin ada beberapa menit antara pembuatan proses baru dalam sistem. Penjadualan longterm mengkontrol derajat multiprogramming (jumlah proses dalam memori). Jika derajat multiprogramming stabil, lalu tingkat rata-rata dari penciptaan proses harus sama dengan tingkat kepergian rata rata dari proses yang meninggalkan sistem. Maka penjadualan longterm mungkin diperlukan untuk dipanggil hanya ketika suatu proses meninggalkan sistem. Karena interval yang lebih panjang antara eksekusi, penjadualan longterm dapat memakai waktu yang lebih lama untuk menentukan proses mana yang harus dipilih untuk dieksekusi.

Adalah penting bagi penjadualan longterm membuat seleksi yang hati-hati. Secara umum, kebanyakan proses dapat dijelaskan sebagai I/O bound atau CPU bound. Sebuah proses I/O bound adalah salah satu yang membuang waktunya untuk mengerjakan I/O dari pada melakukan perhitungan. Suatu proses CPU-bound, pada sisi lain, adalah salah satu yang jarang menghasilkan permintaan I/O, menggunakan lebih banyak waktunya melakukan banyak komputasi daripada yang digunakan oleh proses I/O bound. Penting untuk penjadualan longterm memilih campuran proses yang baik antara proses I/O bound dan CPU bound. Jika seluruh proses adalah I/O bound, ready queue akan hampir selalu kosong, dan penjadualan short term akan memiliki sedikit tugas. Jika seluruh proses adalah CPU bound, I/O waiting queue akan hampir selalu kosong, peralatan akan tidak terpakai, dan sistem akan menjadi tidak imbang. Sistem dengan kinerja yang terbaik akan memiliki kombinasi proses CPU bound dan I/O bound.

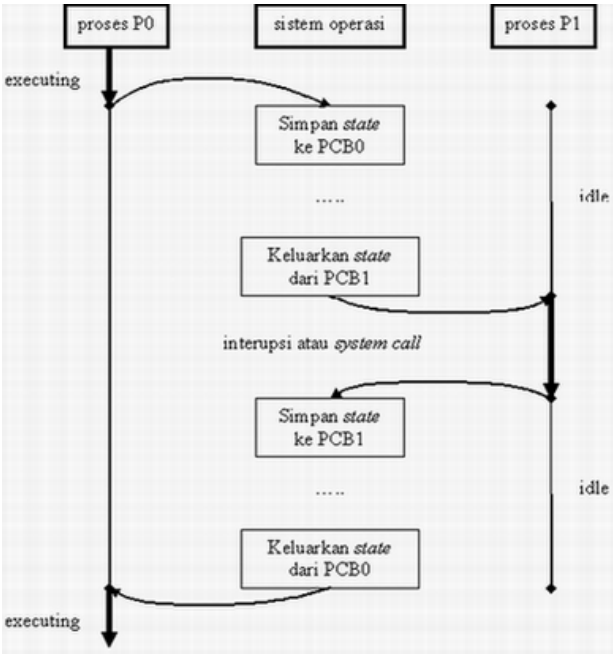
Gambar 2-6. Penjadual Medium-term.

Pada sebagian sistem, penjadual long term dapat tidak turut serta atau minimal. Sebagai contoh, sistem time-sharing seperti UNIX sering kali tidak memiliki penjadual long term. Stabilitas sistem-sistem ini bergantung pada keterbatasan fisik (seperti jumlah terminal yang ada) atau pada penyesuaian sendiri secara alamiah oleh manusia sebagai pengguna. Jika kinerja menurun pada tingkat yang tidak dapat diterima, sebagian pengguna akan berhenti.

Sebagian sistem operasi, seperti sistem time sharing, dapat memperkenalkan sebuah tambahan, penjadualan tingkat menengah. Penjadual medium-term ini digambarkan pada Gambar 2-5. Ide utama/kunci dibelakang sebuah penjadual medium term adalah kadang kala akan menguntungkan untuk memindahkan proses dari memori (dan dari pengisian aktif dari CPU), dan maka untuk mengurangi derajat dari multiprogramming. Dikemudian waktu, proses dapat diperkenalkan kedalam memori dan eksekusinya dapat dilanjutkan dimana proses itu di tinggalkan/ diangkat. Skema ini disebut *swapping*. Proses di *swapped out*, dan lalu di *swapped in*, oleh penjadual jangka menengah. *Swapping* mungkin perlu untuk meningkatkan pencampuran proses, atau karena suatu perubahan dalam persyaratan memori untuk dibebaskan. *Swapping* dibahas dalam Bagian 4.2.

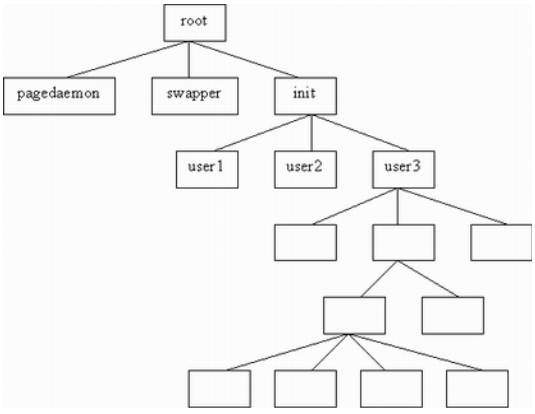
2.2.3. Alih Konteks / Switch Context

Mengganti CPU ke proses lain memerlukan penyimpanan suatu keadaan proses lama (*state of old process*) dan kemudian beralih ke proses yang baru. Tugas tersebut diketahui sebagai alih konteks (*context switch*). Alih konteks sebuah proses digambarkan dalam PCB suatu proses; termasuk nilai dari CPU register, status proses (lihat Gambar 2-7). dan informasi managemen memori. Ketika alih konteks terjadi, kernel menyimpan konteks dari proses lama kedalam PCB nya dan mengisi konteks yang telah disimpan dari process baru yang telah terjadual untuk berjalan. Pergantian waktu konteks adalah murni overhead, karena sistem melakukan pekerjaan yang tidak perlu. Kecepatannya bervariasi dari mesin ke mesin, bergantung pada kecepatan memori, jumlah register yang harus di copy, dan keberadaan instruksi khusus (seperti instruksi tunggal untuk mengisi atau menyimpan seluruh register). Tingkat kecepatan umumnya berkisar antara 1 sampai 1000 mikro detik



Gambar 2-7. Alih Konteks.

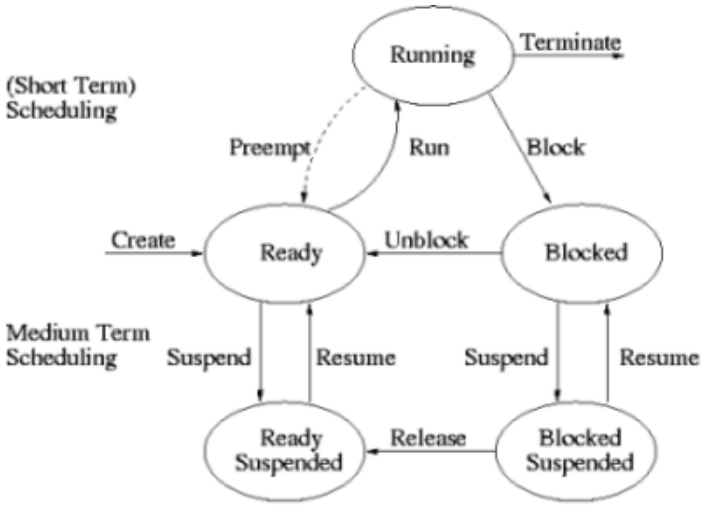
Waktu alih konteks sangat begantung pada dukungan perangkat keras. Sebagai contoh, prosesor seperti UltraSPARC menyediakan dua rangkap register. Sebuah alih konteks hanya memasukkan perubahan pointer ke perangkat register yang ada. Tentu saja, jika ada lebih proses-proses aktif yang ada dari pada yang ada di perangkat register, sistem menggunakan bantuan untuk meng-copy data register pada dan dari memori, sebagaimana sebelumnya. Semakin sistem operasi kompleks, makin banyak pekerjaan yang harus dilakukan selama alih konteks. Sebagaimana dilihat pada Bab 4, teknik managemen memori tingkat lanjut dapat mensyaratkan data tambahan untuk diganti dengan tiap konteks. Sebagai contoh, ruang alamat dari proses yang ada harus dijaga sebagai ruang pada pekerjaan berikutnya untuk digunakan. Bagaimana ruang alamat di jaga, berapa banyak pekerjaan dibutuhkan untuk menjaganya, tergantung pada metoda managemen memori dari sistem operasi. Sebagaimana akan kita lihat pada Bab 4, alih konteks telah menjadi suatu keharusan, bahwa programmer menggunakan struktur (*threads*) untuk menghindarinya kapan pun memungkinkan.



Gambar 2-8. Pohon Proses.

2.3. Operasi-Operasi Pada Proses

Proses dalam sistem dapat dieksekusi secara bersama-sama, proses tersebut harus dibuat dan dihapus secara dinamis. Maka, sistem operasi harus menyediakan suatu mekanisme umtuk pembuatan proses dan terminasi proses.



Gambar 2-9. Operasi pada Proses.

2.3.1. Pembuatan Proses

Suatu proses dapat membuat beberapa proses baru, melalui sistem pemanggilan pembuatan proses, selama jalur eksekusi. Pembuatan proses dinamakan induk proses, sebagaimana proses baru disebut anak dari proses tersbut. Tiap proses baru tersebut dapat membuat proses lainnya, membentuk suatu pohon proses (lihat Gambar 2-7).

Secara umum, suatu proses akan memerlukan sumber tertentu (waktu CPU, memori, berkas, perangkat I/O) untuk menyelesaikan tugasnya. Ketika suatu proses membuat sebuah subproses, sehingga subproses dapat mampu untuk memperoleh sumbernya secara langsung dari sistem operasi. Induk mungkin harus membatasi sumber diantara anaknya, atau induk dapat berbagi sebagian sumber (seperti memori berkas) diantara beberapa dari anaknya. Membatasi suatu anak proses menjadi subset sumber daya induknya mencegah proses apa pun dari pengisian sistem yang telalu banyak dengan menciptakan terlalu banyak subproses.

Sebagai tambahan pada berbagai sumber fisik dan logis bahwa suatu proses diperoleh ketika telah dibuat, data pemula (masukan) dapat turut lewat oleh induk proses sampai anak proses. Sebagai contoh, anggap suatu proses yang fungsinya untuk menunjukkan status sebuah berkas, katakan F1, pada layar terminal.

Ketika dibuat, akan menjadi sebagai sebuah masukan dari proses induknya, nama dari berkas F1, dan akan mengeksekusi menggunakan kumpulan data tersebut untuk memperoleh informasi yang diinginkan. Proses tersebut juga mendapat nama dari perangkat luar. Sebagian sistem operasi melewati sumber-sumber ke anak proses. Pada

sistem tersebut, proses baru bisa mendapat dua berkas terbuka yang baru, F1 dan perangkat terminal dan hanya perlu untuk mentransfer data antara kedua berkas tersebut.

Ketika suatu proses membuat proses baru, dua kemungkinan ada dalam term eksekusi:
1. Induk terus menerus untuk mengeksekusi secara bersama-sama dengan anaknya.
2. Induk menunggu sampai sebagian dari anaknya telah diakhiri/terminasi.

Juga ada dua kemungkinan dalam term dari *address space* pada proses baru:
1. Anak proses adalah duplikat dari induk proses.
2. Anak proses memiliki program yang terisikan didalamnya.

Untuk mengilustrasikan implementasi yang berbeda ini, mari kita mempelajari sistem operasi UNIX. Dalam UNIX, tiap proses diidentifikasi oleh pengidentifikasi proses, yang merupakan integer yang unik. Proses baru dibuat oleh sistem pemanggilan fork system call. Proses baru tersebut terdiri dari sebuah copy ruang alamat dari proses aslinya (original). Mekanisme tersebut memungkinkan induk proses untuk berkomunikasi dengan mudah dengan anak proses. Kedua proses (induk dan anak) meneruskan eksekusi pada instruksi setelah fork dengan satu perbedaan: Kode kembali untuk fork adalah nol untuk proses baru (anak), sebagaimana proses pengidentifikasi non nol (non zero) dari anak dikembalikan kepada induk.

Umumnya, sistem pemanggilan execlp digunakan setelah sistem pemanggilan fork. Oleh satu dari dua proses untuk menggantikan proses ruang memori dengan program baru. Sistem pemanggilan execlp mengisi suatu berkas binary kedalam memori (menghancurkan gambar memori pada program yang berisikan sistem pemanggilan execlp) dan memulai eksekusinya. Dengan cara ini, kedua proses mampu untuk berkomunikasi, dan lalu untuk pergi ke arah yang berbeda. Induk lalu dapat membuat anak yang lebh banyak atau jika induk tidak punya hal lain untuk dilakukan ketika anak bekerja, induk dapat mengeluarkan sistem pemanggilan wait untuk tidak menggerakkan dirinya sendiri pada suatu antrian yang siap sampai anak berhenti. Program C ditunjukkan pada Gambar 2-10 mengilustrasikan sistem pemanggilan pada UNIX yang sebelumnya dijelaskan. Induk membuat anak proses menggunakan sistem pemanggilan fork(). Kini kita mempunyai dua proses yang berbeda yang menjalankan sebuah copy pada program yang sama. Nilai dari pid untuk anak proses adalah nol (zero): maka untuk induk adalah nilai integer yang lebih besar dari nol. Anak proses meletakkan ruang alamatnya dengan UNIX command/bin/ls (digunakan untuk mendapatkan pendaftaran directory) menggunakan sistem pemanggilan execlp(). Ketika anak proses selesai, induk proses menyimpulkan dari pemanggilan untuk wait() dimana induk proses menyelesaikannya dengan menggunakan sistem pemanggilan exit().

Secara kontras, sistem operasi DEC VMS membuat sebuah proses baru dengan mengisi program tertentu kedalam proses tersebut, dan memulai pekerjaannya. Sistem operasi Microsoft Windows NT mendukung kedua model: Ruang alamat induk proses dapat di duplikasi, atau induk dapat menspesifikasi nama dari sebuah program untuk sistem operasi untuk diisikan kedalam ruang alamat pada proses baru.

2.3.2. Terminasi Proses

Sebuah proses berakhir ketika proses tersebut selesai mengeksekusi pernyataan akhirnya dan meminta sistem operasi untuk menghapusnya dengan menggunakan sistem pemanggilan exit. Pada titik itu, proses tersebut dapat mengembalikan data (keluaran) pada induk prosesnya (melalui sistem pemanggilan wait). Seluruh sumber-

sumber dari proses-termasuk memori fisik dan virtual, membuka berkas, dan penyimpanan I/O di tempatkan kembali oleh sistem operasi.

Ada situasi tambahan tertentu ketika terminasi terjadi. Sebuah proses dapat menyebabkan terminasi dari proses lain melalui sistem pemanggilan yang tepat (contoh abort). Biasanya, sistem seperti itu dapat dipanggil hanya oleh induk proses tersebut yang akan diterminasi. Bila tidak, pengguna dapat secara sewenang-wenang membunuh job antara satu sama lain. Catat bahwa induk perlu tahu identitas dari anaknya. Maka, ketika satu proses membuat proses baru, identitas dari proses yang baru diberikan kepada induknya.

Induk dapat menterminasi/ mengakhiri satu dari anaknya untuk beberapa alasan, seperti:

- Anak telah melampaui kegunaannya atas sebagaian sumber yang telah diperuntukkan untuknya.
- Pekerjaan yang ditugaskan kepada anak telah keluar, dan sistem operasi tidak memeperbolehkan sebuah anak untuk meneruskan jika induknya berakhir.

Untuk menentukan kasus pertama, induk harus memiliki mekanisme untuk memeriksa status anaknya. Banyak sistem, termasuk VMS, tidak memperbolehkan sebuah anak untuk ada jika induknya telah berakhir. Dalam sistem seperti ini, jika suatu proses berakhir (walau secara normal atau tidak normal), maka seluruh anaknya juga harus diterminasi. Fenomena ini, mengacu pada terminasi secara *cascading*, yang normalnya dimulai oleh sistem operasi.

Untuk mengilustrasikan proses eksekusi dan proses terminasi, kita menganggap bahwa, dalam UNIX, kami dapat mengakhiri suatu proses dengan sistem pemanggilan exit; proses induknya dapat menunggu untuk terminasi anak proses dengan menggunakan sistem pemanggilan wait. Sistem pemanggilan wait kembali ke pengidentifikasi proses dari anak yang telah diterminasi, maka induk dapat memberitahu kemungkinanan anak mana yang telah diterminasi. Jika induk menterminasi. Maka, anaknya masih punya sebuah induk untuk mengumpulkan status mereka dan mengumpulkan statistik eksekusinya.

2.4. Hubungan Antara Proses

Sebelumnya kita telah ketahui seluk beluk dari suatu proses mulai dari pengertiannya, cara kerjanya, sampai operasi-operasinya seperti proses pembentukannya dan proses pemberhentiannya setelah selesai melakukan eksekusi. Kali ini kita akan mengulas bagaimana hubungan antar proses dapat berlangsung, misal bagaimana beberapa proses dapat saling berkomunikasi dan bekerja-sama.

2.4.1. Proses yang Kooperatif

Proses yang bersifat simultan (*concurrent*) dijalankan pada sistem operasi dapat dibedakaan menjadi yaitu proses independent dan proses kooperatif. Suatu proses dikatakan independen apabila proses tersebut tidak dapat terpengaruh atau dipengaruhi oleh proses lain yang sedang dijalankan pada sistem.

Berarti, semua proses yang tidak membagi data apa pun (baik sementara/ tetap) dengan proses lain adalah independent. Sedangkan proses kooperatif adalah proses yang dapat dipengaruhi atau pun terpengaruhi oleh proses lain yang sedang dijalankan dalam sistem. Dengan kata lain, proses dikatakan kooperatif bila proses dapat membagi datanya dengan proses lain. Ada empat alasan untuk penyediaan sebuah lingkungan yang memperbolehkan terjadinya proses kooperatif:

1. Pembagian informasi: apabila beberapa pengguna dapat tertarik pada bagian informasi yang sama (sebagai contoh, sebuah berkas bersama), kita harus menyediakan sebuah lingkungan yang mengizinkan akses secara terus menerus ke tipe dari sumber-sumber tersebut.

2. Kecepatan penghitungan/ komputasi: jika kita menginginkan sebuah tugas khusus untuk menjalankan lebih cepat, kita harus membagi hal tersebut ke dalam subtask, setiap bagian dari subtask akan dijalankan secara parallel dengan yang lainnya. Peningkatan kecepatan dapat dilakukan hanya jika komputer tersebut memiliki elemen-elemen pemrosesan ganda (seperti CPU atau jalur I/O).

3. Modularitas: kita mungkin ingin untuk membangun sebuah sistem pada sebuah model modular-modular, membagi fungsi sistem menjadi beberapa proses atau threads.

4. Kenyamanan: bahkan seorang pengguna individu mungkin memiliki banyak tugas untuk dikerjakan secara bersamaan pada satu waktu. Sebagai contoh, seorang pengguna dapat mengedit, memcetak, dan meng-*compile* secara paralel.

```
import java.util.*;

public class BoundedBuffer {
      public BoundedBuffer() {
            // buffer diinisialisasikan kosong
            count = 0;
            in = 0;
            out = 0;
            buffer = new Object[BUFFER_SIZE];
 }

// produser memanggil method ini
     public void enter( Object item ) {
       while ( count == BUFFER_SIZE )
           ; // do nothing

        // menambahkan suatu item ke dalam buffer
       ++count;
       buffer[in] = item;
      in = ( in + 1 ) % BUFFER_SIZE;

      if ( count == BUFFER_SIZE )
            System.out.println( "Producer Entered " +
                  item + " Buffer FULL" );
      else
            System.out.println( "Producer Entered " +
                  item + " Buffer Size = " + count );
       }

// consumer memanggil method ini
      public Object remove() {
         Object item ;

        while ( count == 0 )
             ; // do nothing

         // menyingkirkan suatu item dari buffer
        --count;
       item = buffer[out];
       out = ( out + 1 ) % BUFFER_SIZE;
```

```
    if ( count == 0 )
        System.out.println( "Consumer consumed " +
                item + " Buffer EMPTY" );
    else
        System.out.println( "Consumer consumed " +
                item + " Buffer Size = " +count );
    return item;
  }

  public static final int NAP_TIME = 5;
  private static final int BUFFER_SIZE = 5;

  private volatile int count;
  private int in; // arahkan ke posisi kosong selanjutnya
  private int out; // arahkan ke posisi penuh selanjutnya
  private Object[] buffer;
}
```

Gambar 2-10. Program Produser Konsumer.

Sebuah proses produser membentuk informasi yang dapat digunakan oleh konsumen proses. Sebagai contoh sebuah cetakan program yang membuat banyak karakter yang diterima oleh driver pencetak. Untuk memperbolehkan produser dan konsumer proses agar dapat berjalan secara terus menerus, kita harus menyediakan sebuah item *buffer* yang dapat diisi dengan proses produser dan dikosongkan oleh proses konsumer. Proses produser dapat memproduksi sebuah item ketika konsumer sedang mengkonsumsi item yang lain. Produser dan konsumer harus dapat selaras. Konsumer harus menunggu hingga sebuah item diproduksi.

2.4.2. Komunikasi Proses Dalam Sistem

Cara lain untuk meningkatkan efek yang sama adalah untuk sistem operasi yaitu untuk menyediakan alat-alat proses kooperatif untuk berkomunikasi dengan yang lain lewat sebuah komunikasi dalam proses (IPC = Inter-Process Communication). IPC menyediakan sebuah mekanisme untuk mengizinkan proses-proses untuk berkomunikasi dan menyelaraskan aksi-aksi mereka tanpa berbagi ruang alamat yang sama. IPC adalah khusus digunakan dalam sebuah lingkungan yang terdistribusi dimana proses komunikasi tersebut mungkin saja tetap ada dalam komputer-komputer yang berbeda yang tersambung dalam sebuah jaringan. IPC adalah penyedia layanan terbaik dengan menggnakan sebuah sistem penyampaian pesan, dan sistem-sistem pesan dapat diberikan dalam banyak cara.

2.4.2.1. Sistem Penyampaian Pesan

Fungsi dari sebuah sistem pesan adalah untuk memperbolehkan komunikasi satu dengan yang lain tanpa perlu menggunakan pembagian data. Sebuah fasilitas IPC menyediakan paling sedikit dua operasi yaitu kirim (pesan) dan terima (pesan). Pesan dikirim dengan sebuah proses yang dapat dilakukan pada ukuran pasti atau variabel. Jika hanya pesan dengan ukuran pasti dapat dikirimkan, level sistem implementasi adalah sistem yang sederhana. Pesan berukuran variabel menyediakan sistem implementasi level yang lebih kompleks.

Berikut ini ada beberapa metode untuk mengimplementasikan sebuah jaringan dan operasi pengiriman/penerimaan secara logika:

- Komunikasi langsung atau tidak langsung.
- Komunikasi secara simetris/ asimetris.

- *Buffer* otomatis atau eksplisit.
- engiriman berdasarkan salinan atau referensi.
- Pesan berukuran pasti dan variabel.

2.4.2.2. Komunikasi Langsung

Proses-proses yang ingin dikomunikasikan harus memiliki sebuah cara untuk memilih satu dengan yang lain. Mereka dapat menggunakan komunikasi langsung/ tidak langsung.

Setiap proses yang ingin berkomunikasi harus memiliki nama yang bersifat eksplisit baik penerimaan atau pengirim dari komunikasi tersebut. Dalam konteks ini, pengiriman dan penerimaan pesan secara primitive dapat dijabarkan sebagai:

- Send (P, message) - mengirim sebuah pesan ke proses P.
- Receive (Q, message) - menerima sebuah pesan dari proses Q.

Sebuah jaringan komunikasi pada bahasan ini memiliki beberapa sifat, yaitu:

- Sebuah jaringan yang didirikan secara otomatis diantara setiap pasang dari proses yang ingin dikomunikasikan. Proses tersebut harus mengetahui identitas dari semua yang ingin dikomunikasikan.
- Sebuah jaringan adalah terdiri dari penggabungan dua proses.
- Diantara setiap pesan dari proses terdapat tepat sebuah jaringan.

Pembahasan ini memperlihatkan sebuah cara simetris dalam pemberian alamat. Oleh karena itu, baik keduanya yaitu pengirim dan penerima proses harus memberi nama bagi yang lain untuk berkomunikasi, hanya pengirim yang memberikan nama bagi penerima sedangkan penerima tidak menyediakan nama bagi pengirim. Dalam konteks ini, pengirim dan penerima secara sederhana dapat dijabarkan sebagai:

- Send (P, message) - mengirim sebuah pesan kepada proses P.
- Receive (id, message) - menerima sebuah pesan dari semua proses. Variabel id diatur sebagai nama dari proses dengan komunikasi.

2.4.2.3. Komunikasi Tidak Langsung

Dengan komunikasi tidak langsung, pesan akan dikirimkan pada dan diterima dari/ melalui *mailbox*(kotak surat) atau terminal-terminal, sebuah *mailbox* dapat dilihat secara abstrak sebagai sebuah objek didalam setiap pesan yang dapat ditempatkan dari proses dan dari setiap pesan yang bias dipindahkan. Setiap kotak surat memiliki sebuah identifikasi (identitas) yang unik, sebuah proses dapat berkomunikasi dengan beberapa proses lain melalui sebuah nomor dari *mailbox* yang berbeda. Dua proses dapat saling berkomunikasi apabila kedua proses tersebut sharing *mailbox*. Pengirim dan penerima dapat dijabarkan
sebagai:

- Send (A, message) - mengirim pesan ke *mailbox* A.
- Receive (A, message) - menerima pesan dari *mailbox* A.

Dalam masalah ini, link komunikasi mempunyai sifat sebagai berikut:

- Sebuah link dibangun diantara sepasang proses dimana kedua proses tersebut membagi *mailbox*.
- Sebuah link mungkin dapat berasosiasi dengan lebih dari dua proses.
- Diantara setiap pasang proses komunikasi, mungkin terdapat link yang berbeda-beda, dimana setiap link berhubungan pada satu *mailbox*.

Misalkan terdapat proses P1, P2 dan P3 yang semuanya share *mailbox*. Proses P1 mengirim pesan ke A, ketika P2 dan P3 masing-masing mengeksekusi sebuah kiriman

dari A. Proses mana yang akan menerima pesan yang dikirim P1? Jawabannya tergantung dari jalur yang kita pilih:

- Mengizinkan sebuah link berasosiasi dengan paling banyak 2 proses.
- Mengizinkan paling banyak satu proses pada suatu waktu untuk mengeksekusi hasil kiriman (*receive operation*).
- Mengizinkan sistem untuk memilih secara mutlak proses mana yang akan menerima pesan (apakah itu P2 atau P3 tetapi tidak keduanya, tidak akan menerima pesan). Sistem mungkin mengidentifikasi penerima kepada pengirim.

Mailbox mungkin dapat dimiliki oleh sebuah proses atau sistem operasi. Jika *mailbox* dimiliki oleh proses, maka kita mendefinisikan antara pemilik (yang hanya dapat menerima pesan melalui *mailbox*) dan pengguna dari *mailbox* (yang hanya dapat mengirim pesan ke *mailbox*). Selama setiap *mailbox* mempunyai kepemilikan yang unik, maka tidak akan ada kebingungan tentang siapa yang harus menerima pesan dari *mailbox*. Ketika proses yang memiliki *mailbox* tersebut diterminasi, *mailbox* akan hilang. Semua proses yang mengirim pesan ke *mailbox* ini diberi pesan bahwa *mailbox* tersebut tidak lagi ada.

Dengan kata lain, mempunyai *mailbox* sendiri yang independent, dan tidak melibatkan proses yang lain. Maka sistem operasi harus memiliki mekanisme yang mengizinkan proses untuk melakukan hal-hal dibawah ini:

- Membuat *mailbox* baru.
- Mengirim dan menerima pesan melalui *mailbox*.
- Menghapus *mailbox*.

Proses yang membuat *mailbox* pertama kali secara default akan memiliki *mailbox* tersebut. Untuk pertama kali, pemilik adalah satu-satunya proses yang dapat menerima pesan melalui *mailbox* ini. Bagaimana pun, kepemilikan dan hak menerima pesan mungkin dapat dialihkan ke proses lain melalui sistem pemanggilan.

2.4.2.4. Sinkronisasi

Komunikasi antara proses membutuhkan place by calls untuk mengirim dan menerima data primitive. Terdapat rancangan yang berbeda-beda dalam implementasi setiap primitive. Pengiriman pesan mungkin dapat diblok (*blocking*) atau tidak dapat dibloking (*nonblocking*) - juga dikenal dengan nama sinkron atau asinkron.

- Pengiriman yang diblok: Proses pengiriman di blok sampai pesan diterima oleh proses penerima (*receiving process*) atau oleh *mailbox*.
- Pengiriman yang tidak diblok: Proses pengiriman pesan dan mengkalkulasi operasi.
- Penerimaan yang diblok: Penerima mem blok samapai pesan tersedia.
- Penerimaan yang tidak diblok: Penerima mengembalikan pesan valid atau null.

2.4.2.5. *Buffering*

Baik komunikasi itu langsung atau tak langsung, penukaran pesan oleh proses memerlukan antrian sementara. Pada dasarnya, terdapat tiga jalan dimana antrian tersebut diimplementasikan:

- Kapasitas nol: antrian mempunyai panjang maksimum 0, maka link tidak dapat mempunyai penungguan pesan (*message waiting*). Dalam kasus ini, pengirim harus memblok sampai penerima menerima pesan.
- Kapasitas terbatas: antrian mempunyai panjang yang telah ditentukan, paling banyak n pesan dapat dimasukkan. Jika antrian tidak penuh ketika pesan

dikirimkan, pesan yang baru akan menimpa, dan pengirim pengirim dapat melanjutkan eksekusi tanpa menunggu. Link mempunyai kapasitas terbatas.

- Jika link penuh, pengirim harus memblok sampai terdapat ruang pada antrian.
- Kapasitas tak terbatas: antrian mempunyai panjang yang tak terhingga, maka, semua pesan dapat menunggu disini. Pengirim tidak akan pernah di blok.

2.4.2.6. Contoh Produser-Konsumer

Sekarang kita mempunyai solusi problem produser-konsumer yang menggunakan penyampaian pesan. Produser dan konsumer akan berkomunikasi secara tidak langsung menggunakan *mailbox* yang dibagi. Buffer menggunakan java.util.Vector class sehingga *buffer* mempunyai kapasitas tak terhingga. Dan send() dan read() method adalah nonblocking. Ketika produser memproduksi suatu item, item tersebut diletakkan ke *mailbox* melalui send() method. Konsumer menerima item dari *mailbox* menggunakan receive() method. Karena receive() nonblocking, consumer harus mengevaluasi nilai dari Object yang di-*return* dari receive(). Jika null, *mailbox* kosong.

```
import java.util.*;

public class Producer extends Thread {
    private MessageQueueueueue mbox;

    public Producer( MessageQueueueueue m ) {
        mbox = m;
    }

    public void run() {
        Date message;

        while ( true ) {
            int sleeptime = ( int ) ( Server.NAP_TIME * Math.random() );
            System.out.println( "Producer sleeping for " +
                sleeptime + " seconds" );
            try {
                Thread.sleep(sleeptime*1000);
            } catch( InterruptedException e ) {}

            message = new Date();
            System.out.println( "Producer produced " + message );
            mbox.send( message );
     }
    }
}
import java.util.*;

public class Consumer extends Thread {
       private MessageQueueueueue mbox;

       public Consumer( MessageQueueueueue m ) {
              mbox = m;
       }

public void run() {
       Date message;

       while ( true ) {
            int sleeptime = (int) (Server.NAP_TIME * Math.random());
            System.out.println("Consumer sleeping for " +
                sleeptime + " seconds" );
            try {
```

```
                Thread.sleep( sleeptime * 1000 );
            } catch( InterruptedException e ) {}

            message = ( Date ) mbox.receive();
            if ( message != null )
                System.out.println("Consumer consume " + message );
        }
    }
}
```

Gambar 2-11. Program Produser Konsumer Alternatif.

Kita memiliki dua aktor di sini, yaitu Produser dan Konsumer. Produser adalah thread yang menghasilkan waktu (Date) kemudian menyimpannya ke dalam antrian pesan. Produser juga mencetak waktu tersebut di layer (sebagai umpan balik bagi kita). Konsumer adalah thread yang akan mengakses antrian pesan untuk mendapatkan waktu (date) itu dan tak lupa mencetaknya di layer. Kita menginginkan supaya konsumer itu mendapatkan waktu sesuatu dengan urutan sebagaimana produser menyimpan waktu tersebut. Kita akan menghadapi salah satu dari dua kemungkinan situasi di bawah ini:

• Bila p1 lebih cepat dari c1, kita akan memperoleh output sebagai berikut:

```
…….
Consumer consume Wed May 07 14:11:12 ICT 2003
Consumer sleeping for 3 seconds
Producer produced Wed May 07 14:11:16 ICT 2003
Producer sleeping for 4 seconds
// p1 sudah mengupdate isi mailbox waktu dari Wed May 07
// 14:11:16 ICT 2003 ke Wed May 07 14:11:17 ICT 2003,
// padahal c1 belum lagi mengambil waktu Wed May 07 14:11:16
Producer produced Wed May 07 14:11:17 ICT 2003
Producer sleeping for 4 seconds
Consumer consume Wed May 07 14:11:17 ICT 2003
Consumer sleeping for 4 seconds
// Konsumer melewatkan waktu Wed May 07 14:11:16
. . .
```

Gambar 2-12. Keluaran Program Produser Konsumer.

• Bila p1 lebih lambat dari c1, kita akan memperoleh keluaran seperti berikut:

```
. . .
Producer produced Wed May 07 14:11:11 ICT 2003
Producer sleeping for 1 seconds
Consumer consume Wed May 07 14:11:11 ICT 2003
Consumer sleeping for 0 seconds
// c1 sudah mengambil isi dari mailbox, padahal p1 belum
// lagi megupdate isi dari mailbox dari May 07 14:11:11
// ICT 2003 ke May 07 14:11:12 ICT 2003, c1 mendapatkan
// waktu Wed May 07 14:11:11 ICT 2003 dua kali.
Consumer consume Wed May 07 14:11:11 ICT 2003
Consumer sleeping for 0 seconds
Producer sleeping for 0 seconds
Producer produced Wed May 07 14:11:12 ICT 2003
. . .
```

Gambar 2-13. Keluaran Program Produser Konsumer.

Situasi di atas dikenal dengan race conditions. Kita dapat menghindari situasi itu dengan mensinkronisasikan aktivitas p1 dan c1 (sehubungan dengan akses mereka ke *mailbox*). Proses tersebut akan didiskusikan pada Bagian 3.2.

2.4.2.7. *Mailbox*

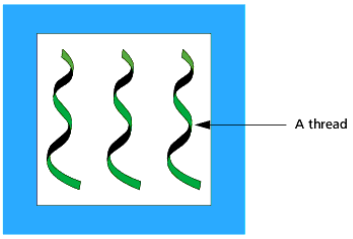
```
import java.util.*;

public class MessageQueue {
     private Vector q;

     public MessageQueue() {
          q = new Vector();
      }

// Mengimplementasikan pengiriman nonblocking
public void send( Object item ) {
      q.addElement( item );
}


// Mengimplementasikan penerimaan nonblocking
public Object receive() {
     Object item;
     if ( q.size() == 0 )
        return null;
     else {
          item = q.firstElement();
          q.removeElementAt(0);

     return item;
   }
  }
}
```

Gambar 2-14. Program Send/ Receive.

1. Menunggu sampai batas waktu yang tidak dapat ditentukan sampai terdapat ruang kosong pada *mailbox*.
2. Menunggu paling banyak n milidetik.
3. Tidak menunggu, tetapi kembali (*return*) secepatnya.
4. Satu pesan dapat diberikan kepada sistem operasi untuk disimpan, walau pun *mailbox* yang dituju penuh. Ketika pesan dapat disimpan pada *mailbox*, pesan akan dikembalikan kepada pengirim (*sender*). Hanya satu pesan kepada *mailbox* yang penuh yang dapat diundur (*pending*) pada suatu waktu untuk diberikan kepada thread pengirim.

2.5. *Thread*

Thread, atau kadang-kadang disebut proses ringan (*lightweight*), adalah unit dasar dari utilisasi CPU. Di dalamnya terdapat ID thread, program counter, register, dan stack. Dan saling berbagi dengan thread lain dalam proses yang sama.



Gambar 2-15. Thread.

2.5.1. Konsep Dasar

Secara informal, proses adalah program yang sedang dieksekusi. Ada dua jenis proses, proses berat (*heavyweight*) atau biasa dikenal dengan proses tradisional, dan proses ringan atau kadang disebut thread.

Thread saling berbagi bagian program, bagian data dan sumber daya sistem operasi dengan thread lain yang mengacu pada proses yang sama. Thread terdiri atas ID thread, program counter, himpunan register, dan stack. Dengan banyak kontrol thread proses dapat melakukan lebih dari satu pekerjaan pada waktu yang sama.

2.5.2. Keuntungan

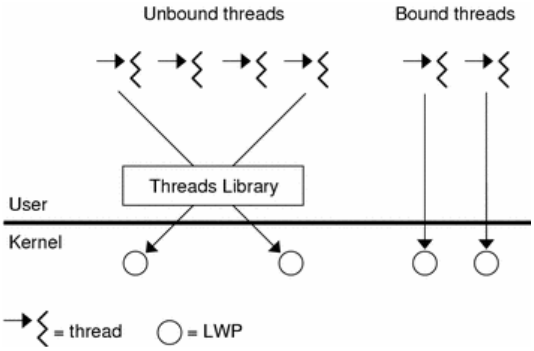
1. Tanggap: *Multithreading* mengizinkan program untuk berjalan terus walau pun pada bagian program tersebut di block atau sedang dalam keadaan menjalankan operasi yang lama/ panjang. Sebagai contoh, multithread web browser dapat mengizinkan pengguna berinteraksi dengan suatu thread ketika suatu gambar sedang diload oleh thread yang lain.

2. Pembagian sumber daya: Secara default, *thread* membagi memori dan sumber daya dari proses. Keuntungan dari pembagian kode adalah aplikasi mempunyai perbedaan aktifitas thread dengan alokasi memori yang sama.

3. Ekonomis: Mengalokasikan memori dan sumber daya untuk membuat proses adalah sangat mahal. Alternatifnya, karena thread membagi sumber daya dari proses, ini lebih ekonomis untuk membuat threads.

4. Pemberdayaan arsitektur multiprosesor: Keuntungann dari multithreading dapat ditingkatkan dengan arsitektur multiprosesor, dimana setiap thread dapat jalan secara parallel pada prosesor yang berbeda. Pada arsitektur prosesor tunggal, CPU biasanya berpindah-pindah antara setiap thread dengan cepat, sehingga terdapat ilusi paralelisme, tetapi pada kenyataannya hanya satu thread yang berjalan di setiap waktu.

2.5.3. *User Threads*



Gambar 2-16. User dan Kernel Thread.

User thread didukung oleh kernel dan diimplementasikan oleh thread library ditingkat pengguna. Library mendukung untuk pembentukan thread, penjadualan, dan managemen yang tidak didukung oleh kernel.

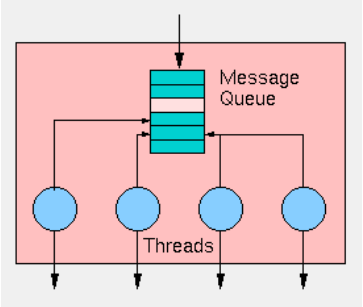
2.5.4. *Kernel Threads*

Kernel thread didukung secara langsung oleh sistem operasi: pembentukan thread, penjadualan, dan managemen dilakukan oleh kernel dalam ruang kernel. Karena managemen thread telah dilakukan oleh sistem operasi, kernel thread biasanya lebih lambat untuk membuat dan mengelola daripada pengguna thread. Bagaimana pun, selama kernel mengelola thread, jika suatu thread di block tehadap sistem pemanggilan, kernel dapat menjadualkan thread yang lain dalam aplikasi untuk dieksekusi. Juga, di dalam lingkungan multiprosesor, kernel dapat menjadualkan thread dalam prosesor yang berbeda. Windows NT, Solaris, dan Digital UNIX adalah sistem operasi yang mendukung kernel thread.

2.6. Model *Multithreading*

Dalam sub bab sebelumnya telah dibahas pengertian dari thread, keuntungannya, tingkatan atau levelnya seperti pengguna dan kernel. Maka dalam sub-bab ini pembahasan akan dilanjutkan dengan jenis-jenis thread tersebut dan contohnya baik pada Solaris mau pun Java.

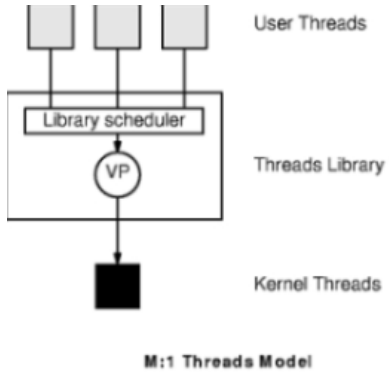
Sistem-sistem yang ada sekarang sudah banyak yang bisa mendukung untuk kedua pengguna dan kernel thread, sehingga model-model multithreading-nya pun menjadi beragam. Implementasi multithreading yang umum akan kita bahas ada tiga, yaitu model many-to-one, one-to-one, dan many-to-many.



Gambar 2-17. Model Multithreading.

2.6.1. Model *Many to One*

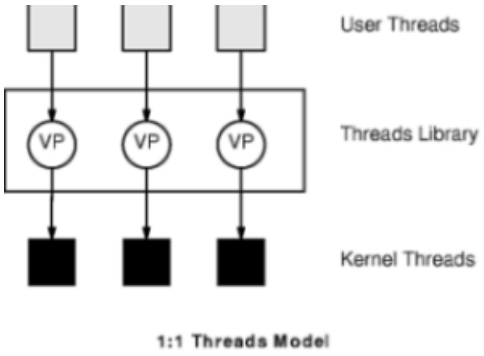
Model many-to-one ini memetakan beberapa tingkatan pengguna thread hanya ke satu buah kernel thread. Managemen proses thread dilakukan oleh (di ruang) pengguna, sehingga menjadi efisien, tetapi apabila sebuah thread melakukan sebuah pemblokingan terhadap sistem pemanggilan, maka seluruh proses akan berhenti (*blocked*). Kelemahan dari model ini adalah multihreads tidak dapat berjalan atau bekerja secara paralel di dalam multiprosesor dikarenakan hanya satu thread saja yang bisa mengakses kernel dalam suatu waktu.



Gambar 2-18. Model Many to One.

2.6.2. Model *One to One*

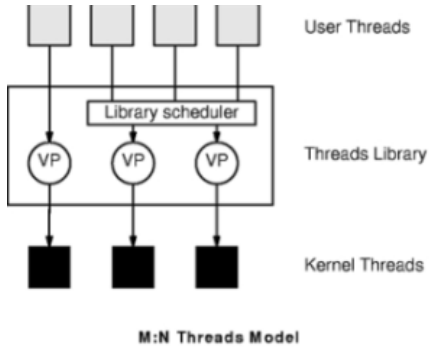
Model one-to-one memetakan setiap thread pengguna ke dalam satu kernel thread. Hal ini membuat model one-to-one lebih sinkron daripada model many-to-one dengan mengizinkan thread lain untuk berjalan ketika suatu thread membuat pemblokingan terhadap sistem pemanggilan; hal ini juga mengizinkan multiple thread untuk berjalan secara parallel dalam multiprosesor. Kelemahan model ini adalah dalam pembuatan thread pengguna dibutuhkan pembuatan korespondensi thread pengguna. Karena dalam proses pembuatan kernel thread dapat mempengaruhi kinerja dari aplikasi maka kebanyakan dari implementasi model ini membatasi jumlah thread yang didukung oleh sistem. Model one-to-one diimplementasikan oleh Windows NT dan OS/2.



Gambar 2-19. Model One to One.

2.6.3. Model *Many to Many*

Beberapa tingkatan *thread* pengguna dapat menggunakan jumlah kernel thread yang lebih kecil atau sama dengan jumlah *thread* pengguna. Jumlah dari kernel *thread* dapat dispesifikasikan untuk beberapa aplikasi dan beberapa mesin (suatu aplikasi dapat dialokasikan lebih dari beberapa kernel thread dalam multiprosesor daripada dalam uniprosesor) dimana model many-to-one mengizinkan pengembang untuk membuat thread pengguna sebanyak mungkin, konkurensi tidak dapat tercapai karena hanya satu thread yang dapat dijadualkan oleh kernel dalam satu waktu. Model one-to-one mempunyai konkurensi yang lebih tinggi, tetapi pengembang harus hati-hati untuk tidak membuat terlalu banyak thread tanpa aplikasi dan dalam kasus tertentu mungkin jumlah thread yang dapat dibuat dibatasi.



Gambar 2-20. Model Many to Many.

2.6.4. *Thread* Dalam Solaris 2

Solaris 2 merupakan salah satu versi dari UNIX yang sampai dengan tahun 1992 hanya masih mendukung proses berat (*heavyweight*) dengan kontrol oleh satu buah *thread*. Tetapi sekarang Solaris 2 sudah berubah menjadi sistem operasi yang modern yang mendukung threads di dalam level kernel dan pengguna, multiprosesor simetrik (SMP), dan penjadualan *real-time*.

Threads di dalam Solaris 2 sudah dilengkapi dengan library mengenai API-API untuk pembuatan dan managemen thread. Di dalam Solaris 2 terdapat juga level tengah thread. Di antara level pengguna dan level kernel thread terdapat proses ringan/ *lightweight* (LWP). Setiap proses yang ada setidaknya mengandung minimal satu buah LWP. Library thread memasangkan beberapa thread level pengguna ke ruang LWP-LWP untuk diproses, dan hanya satu user-level thread yang sedang terpasang ke suatu LWP yang bisa berjalan. Sisanya bisa diblok mau pun menunggu untuk LWP yang bisa dijalankan.

Operasi-operasi di kernel seluruhnya dieksekusi oleh kernel-level threads yang standar. Terdapat satu kernel-level thread untuk tiap LWP, tetapi ada juga beberapa kernel-level threads yang berjalan di bagian kernel tanpa diasosiasikan dengan suatu LWP (misalnya thread untuk pengalokasian disk). Thread kernel-level merupakan satu-satunya objek yang dijadualkan ke dalam sistem (lihat Bagian 2.7 mengenai scheduling). Solaris menggunakan model many-to-many.

Thread level pengguna dalam Solaris bisa berjenis bound mau pun unbound. Suatu bound thread level pengguna secara permanen terpasang ke suatu LWP. Jadi hanya thread tersebut yang bekerja di LWP, dan dengan suatu permintaan, LWP tersebut bisa diteruskan ke suatu prosesor. Dalam beberapa situasi yang membutuhkan waktu respon yang cepat (seperti aplikasi real-time), mengikat suatu thread sangatlah berguna. Suatu thread yang unbound tidak secara permanen terpasang ke suatu LWP. Semua threads unbound dipasangkan (secara multiplex) ke dalam suatu ruang yang berisi LWP-LWP yang tersedia untuk aplikasi. Secara default thread-thread yang ada adalah unbound. Misalnya sistem sedang beroperasi, setiap proses bisa mempunyai threads level pengguna yang banyak. User-user level thread ini bisa dijadual dan diganti di antara LWP-LWP-nya oleh thread library tanpa intervensi dari kernel. User-level threads sangatlah efisien karena tidak dibutuhkan bantuan kerja kernel oleh thread library untuk menukar dari satu user-level thread ke yang lain.

Setiap LWP terpasang dengan tepat satu kernel-level thread, dimana setiap user-level thread tidak tergantung dari kernel. Suatu proses mungkin mempunyai banyak

LWP, tetapi mereka hanya dibutuhkan ketika thread harus berkomunikasi dengan kernel. Misalnya, suatu LWP akan dibutuhkan untuk setiap thread yang bloknya konkuren di sistem pemanggilan. Anggap ada lima buah pembacaan berkas yang muncul. Jadi dibutuhkan lima LWP, karena semuanya mungkin mengunggu untuk penyelesaian proses I/O di kernel. Jika suatu proses hanya mempunyai empat LWP, maka permintaan yang kelima harus menunggu unuk salah satu LWP kembali dari kernel. Menambah LWP yang keenam akan sia-sia jika hanya terdapat tempat untuk lima proses.

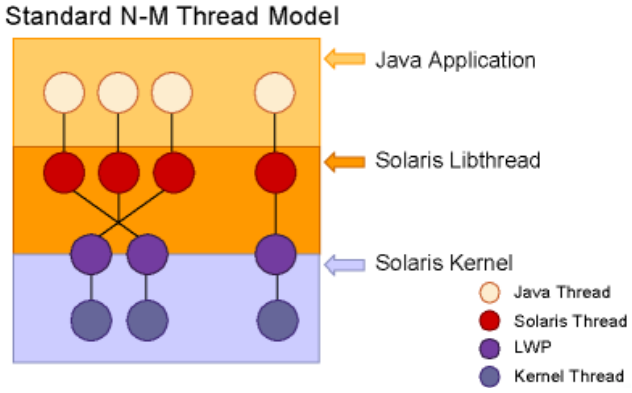
Kernel-kernel threads dijadual oleh penjadual kernel dan dieksekusi di CPU atau CPU-CPU dalam sistemnya. Jika suatu kernel thread memblok (misalnya karena menunggu penyelesaian suatu proses I/O), prosesor akan bebas untuk menjalankan kernel thread yang lain. Jika thread yang sedang terblok sedang menjalankan suatu bagian dari LWP, maka LWP tersebut akan ikut terblok. Di tingkat yang lebih atas lagi, user-level thread yang sedang terpasang ke LWP tersebut akan terblok juga. Jika suatu proses mempunyai lebih dari satu LWP, maka LWP lain bisa dijadual oleh kernel.

Para pengembang menggunakan struktur-struktur data sebagai berikut untuk mengimplementasikan thread-thread dalam Solaris 2:

- Suatu user-level thread mempunyai thread ID, himpunan register (mencakup suatu PC dan stack pointer), stack dan prioritas (digunakan oleh library untuk penjadualan). Semua struktur data tersebut berasal dari ruang user.

- Suatu LWP mempunyai suatu himpunan register untuk user-level thread yang ia jalankan, juga memori dan informasi pencatatan. LWP merupakan suatu struktur data dari kernel, dan bertempat pada ruang kernel.
- Suatu kernel thread hanya mempunyai struktur data yang kecil dan sebuah stack. Struktur datanya melingkupi copy dari kernel-kernel registers, suatu pointer yang menunjuk ke LWP yang terpasang dengannya, dan informasi tentang prioritas dan penjadualan.

Setiap proses dalam Solaris 2 mempunyai banyak informasi yang terdapat di process control block (PCB). Secara umum, suatu proses di Solaris mempunyai suatu proses id (PID), peta memori, daftar dari berkas yang terbuka, prioritas, dan pointer yang menunjuk ke daftar LWP yang terasosiasi kedalam proses.



Gambar 2-21. Thread Solaris dan Java.

2.6.5. *Thread* Java

Seperti yang telah kita lihat, thread didukung selain oleh sistem operasi juga oleh paket library thread. Sebagai contoh, Win32 library mempunyai API untuk multithreading aplikasi Windows, dan Pthreads mempunyai fungsi manajmen thread untuk sistem POSIX-compliant. Java adalah unik dalam mendukung tingkatan bahasa untuk membuat dan managemen thread.

Semua program java mempunyai paling sedikit satu kontrol thread. Bahkan program java yang sederhana mempunyai hanya satu main() method yang berjalan dalam thread tunggal dalam JVM. Java menyediakan perintah-perintah yang mendukung pengembang untuk membuat dan memanipulasi kontrol thread pada program.

Satu cara untuk membuat thread secara eksplisit adalah dengan membuat kelas baru yang diturunkan dari kelas thread, dan menimpa run() method dari kelas Thread tersebut.

Object yang diturunkan dari kelas tersebut akan menjalankan sebagian thread control dalam JVM. Bagaimana pun, membuat suatu objek yang diturunkan dari kelas Thread tidak secara spesifik membuat thread baru, tetapi start() method lah yang sebenarnya membuat thread baru.

Memanggil start() method untuk objek baru mengalokasikan memori dan menginisialisasikan thread baru dalam JVM dan memanggil run() method membuat thread pantas untuk dijalankan oleh JVM.
(Catatan: jangan pernah memanggil run() method secara langsung. Panggil start() method dan ini secara langsung akan memanggil run() method).

Ketika program ini dijalankan, dua thread akan dibuat oleh JVM. Yang pertama dibuat adalah thread yang berasosiasi dengan aplikasi-thread tersebut mulai dieksekusi pada main() method. Thread kedua adalah runner thread secara ekspilisit dibuat dengan start() method. Runner thread memulai eksekusinya dengan run() method.

Pilihan lain untuk membuat sebuah thread yang terpisah adalah dengan mendefinisikan suatu kelas yang mengimplementasikan runnable interface. Runnable interface tersebut didefinisikan sebagai berikut:

```
Public interface Runnable
{
    Public abstract void run();
}
```

Gambar 2-22. Runnable.

Sehingga, ketika sebuah kelas diimplementasikan dengan runnable, kelas tersebut harus mendefinisikan run() method. Kelas thread yang berfungsi untuk mendefinisikan static dan instance method, juga mengimplementasikan runnable interface. Itu menerangkan bahwa mengapa sebuah kelas diturunkan dari thread harus mendefinisikan run() method.

Implementasi dari runnable interface sama dengan mengekstend kelas thread, satu-satunya kemungkinan untuk mengganti "extends thread" dengan "implements runnable".

```
Class worker2 implements Runnable
{
      Public void run() {
          System. Out. Println ("I am a worker thread. ");
      }
}
```

Gambar 2-23. ClassWorker2.

Membuat sebuah thread dari kelas yang diimplementasikan oleh runnable berbeda dengan membuat thread dari kelas yang mengekstend thread. Selama kelas baru tersebut tidak mengekstend thread, dia tidak mempunyai akses ke objek static atau instance method —seperti start() method— dari kelas thread. Bagaimana pun, sebuah objek dari kelas thread adalah tetap dibutuhkan, karena yang membuat sebuah thread baru dari kontrol adalah start() method.

Di kelas kedua, sebuah objek thread baru dibuat melalui runnable objek dalam konstruktornya. Ketika thread dibuat oleh start() method, thread baru mulai dieksekusi pada run() method dari runnable objek. Kedua method dari pembuatan thread tersebut adalah cara yang paling sering digunakan.

2.6.6. Managemen *Thread*

Java menyediakan beberapa fasilitas API untuk mengatur thread — thread, diantaranya adalah:

- Suspend(): berfungsi untuk menunda eksekusi dari thread yang sedang berjalan.
- Sleep(): berfungsi untuk menempatkan thread yang sedang berjalan untuk tidur dalam beberapa waktu.
- Resume(): hasil eksekusi dari thread yang sedang ditunda.
- Stop(): menghentikan eksekusi dari sebuah thread; sekali thread telah dihentikan dia tidak akan memulainya lagi.

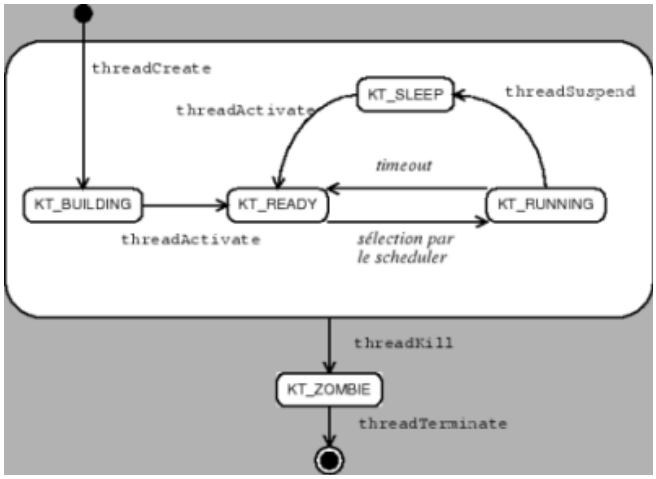
Setiap method yang berbeda untuk mengontrol keadaan dari thread mungkin akan berguna dalam situasi tertentu. Sebagai contoh: Applets adalah contoh alami untuk *multithreading* karena mereka biasanya memiliki grafik, animasi, dan audio—semuanya sangat baik untuk mengatur berbagai thread yang terpisah. Bagaimana pun, itu tidak akan mungkin bagi sebuah applet untuk berjalan ketika dia sedang tidak ditampilkan, jika applet sedang menjalankan CPU secara intensif. Sebuah cara untuk menangani situasi ini adalah dengan menjalankan applet sebagai thread terpisah dari kontrol, menunda thread ketika applet sedang tidak ditampilkan dan melaporkannya ketika applet ditampilkan kembali.

Anda dapat melakukannya dengan mencatat bahwa start() method dari sebuah applet dipanggil ketika applet tersebut pertama kali ditampilkan. Apabila user meninggalkan halaman web atau applet keluar dari tampilan, maka method stop() pada applet dipanggil (ini merupakan suatu keuntungan karena start() dan stop() keduanya terasosiasi dengan thread dan applet). Jika user kembali ke halaman web applet, kemudian start() method dipanggil kembali. Destroy() method dari sebuah applet dipanggil ketika applet tersebut dipindahkan dari cache-nya browser. Ini memungkinkan untuk mencegah sebuah applet berjalan ketika applet tersebut sedang tidak ditampilkan pada sebuah web browser dengan menggunakan stop() method dari applet yang ditunda dan melaporkan eksekusi tersebut pada thread di applet start() method.

2.6.7. Keadaan *Thread*

Sebuah thread java dapat menjadi satu dari 4 kemungkinan keadaan:

1. new: sebuah thread pada keadaan ini ada ketika objek dari thread tersebut dibuat.
2. runnable: memanggil start() method untuk mengalokasikan memori bagi thread baru dalam JVM dan memanggil run() method untuk membuat objek.
3. block: sebuah thread akan diblok jika menampilkan sebuah kalimat pengeblokan. Contohnya: sleep() atau suspend().
4. dead: sebuah thread dipindahkan ke keadaan dead ketika run() method berhenti atau ketika stop() method dipanggil.



Gambar 2-24. Keadaan Thread.

2.6.8. *Thread* dan JVM

Pada penambahannya ke java program mengandung beberapa thread yang berbeda dari kontrol, disini ada beberapa thead yang sedang berjalan secara tidak sinkron untuk kepentingan dari penanganan sistem tingkatan JVM seperti managemen memori dan grafik kontrol. *Garbage Collector* mengevaluasi objek ketika JVM untuk dilihat ketika mereka sedang digunakan. Jika tidak, maka itu akan kembali ke memori
dalam sistem.

2.6.9. JVM dan Sistem Operasi

Secara tipikal implementasi dari JVM adalah pada bagian atas terdapat host sistem operasi, pengaturan ini mengizinkan JVM untuk menyembunyikan detail implementasi dari sistem operasi dan menyediakan sebuah kekonsistenan, lingkungan yang abstrak tersebut mengizinkan program-program java untuk beroprasi pada berbagai sistem operasi yang mendukung sebuah JVM. Spesifikasi bagi JVM tidak mengidentifikasi bagaimana java thread dipetakan ke dalam sistem operasi.

2.6.10. Contoh Solusi *Multithreaded*

Pada bagian ini, kita memperkenalkan sebuah solusi multithreaded secara lengkap kepada masalah produser konsumer yang menggunakan penyampaian pesan. Kelas server pertama kali membuat sebuah mailbox untuk mengumpulkan pesan, dengan

menggunakan kelas message queue kemudian dibuat produser dan konsumer threads secara terpisah dan setiap thread mereferensi ke dalam mailbox bersama. Thread produser secara bergantian antara tidur untuk sementara, memproduksi item, dan memasukkan item ke dalam mailbox. Konsumer bergantian antara tidur dan mengambil suatu item dari mailbox dan mengkonsumsinya. Karena receive() method dari kelas message queue adalah tanpa pengeblokan, konsumer harus mencek apakah pesan yang diambilnya tersebut adalah nol.

2.7. Penjadual CPU

Penjadual CPU adalah basis dari multi programming sistem operasi. Dengan men-switch CPU diantara proses. Akibatnya sistem operasi bisa membuat komputer produktif. Dalam bab ini kami akan mengenalkan tentang dasar dari konsep penjadual dan beberapa algoritma penjadual. Dan kita juga memaparkan masalah dalam memilih algoritma dalam suatu sistem.

2.7.1. Konsep Dasar

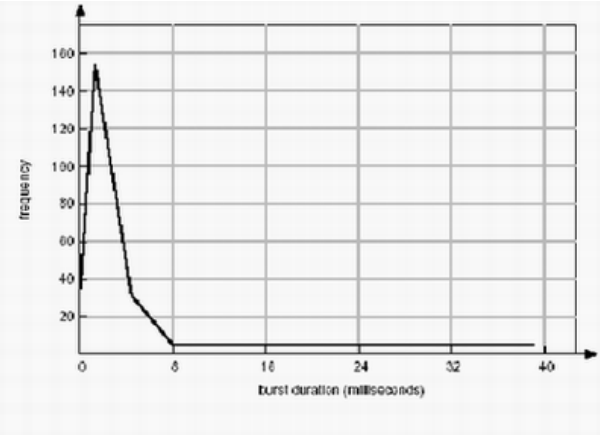
Tujuan dari multi programming adalah untuk mempunyai proses berjalan secara bersamaan, untuk memaksimalkan kinerja dari CPU. Untuk sistem uniprosesor, tidak pernah ada proses yang berjalan lebih dari satu. Bila ada proses yang lebih dari satu maka yang lain harus mengantri sampai CPU bebas. Ide dari multi porgamming sangat sederhana. Ketika sebuah proses dieksekusi yang lain harus menunggu sampai selesai. Di sistem komputer yang sederhana CPU akan banyak dalam posisi idle. Semua waktu ini sangat terbuang. Dengan multiprogamming kita mencoba menggunakan waktu secara produktif. Beberapa proses di simpan dalam memori dalam satu waktu. Ketika proses harus menuggu. Sistem operasi mengmbil CPU untuk memproses proses tersebut dan meninggalkan proses yang sedang dieksekusi.

Penjadual adalah fungsi dasar dari suatu sistem operasi. Hampir semua sumber komputer dijadual sebelum digunakan. CPU salah satu sumber dari komputer yang penting yang menjadi sentral dari sentral penjadual di sistem operasi.

2.7.1.1. Siklus Burst CPU-I/O

Keberhasilan dari penjadual CPU tergantung dari beberapa properti prosesor. Proses eksekusi mengandung siklus CPU ekskusi dan I/o Wait. Proses hanya akan bolak-balik dari dua state ini. Poses eksekusi dimulai dengan CPU Burst, setelah itu diikuti oleh I/O burst, dan dilakukan secara bergiliran.

Durasi dari CPU bust ini ditelah diukur secara ekstensif, walau pun mereka sangat berbeda dari proses ke proses. Mereka mempunyai frekeunsi kurva yang sama seperti yang diperlihatkan gambar dibawah ini.



Gambar 2-25. CPU Burst.

2.7.1.2. Penjadual CPU

Kapan pun CPU menjadi idle, sistem opersai harus memilih salah satu proses untuk masuk kedalam antrian ready (siap) untuk dieksekusi. Pemilihan tersebut dilakukan oleh penjadual short term. Penjadual memilih dari sekian proses yang ada di memori yang sudah siap dieksekusi, den mengalokasikan CPU untuk mengeksekusinya

Penjadual CPU mungkin akan dijalankan ketika proses:
1. Berubah dari running ke waiting state.
2. Berubah dari running ke ready state.
3. Berubah dari waiting ke ready.
4. Terminates.

Penjadual dari no 1 sampai 4 non premptive sedangkan yang lain premptive. Dalam penjadual*nonpreemptive* sekali CPU telah dialokasikan untuk sebuah proses, maka tidak bisa di ganggu, penjadual model seperti ini digunakan oleh Windows 3.x; Windows 95 telah menggunakan penjadual preemptive.

2.7.1.3. *Dispatcher*

Komponen yang lain yang terlibat dalam penjadual CPU adalan *dispatcher*. *Dispatcher* adalah modul yang memberikan kontrol CPU kepada proses yang fungsinya adalah:
1. Alih Konteks
2. Switching to user mode.
3. Lompat dari suatu bagian di progam user untuk mengulang progam.

Dispatcher seharusnya secepat mungkin.

2.7.1.4. Kriteria Penjadual

Algoritma penjadual CPU yang berbeda mempunyai property yang berbeda. Dalam memilih algoritma yang digunakan untuk situasi tertentu, kita harus memikirkan properti yang berbeda untuk algoritma yang berbeda. Banyak kriteria yang dianjurkan utnuk membandingkan penjadual CPU algoritma. Kritria yang biasanya digunakan dalam memilih adalah:

1. CPU utilization: kita ingin menjaga CPU sesibuk mungkin. CPU utilization akan mempunyai range dari 0 ke 100 persen. Di sistem yang sebenarnya seharusnya ia mempunyai range dari 40 persen samapi 90 persen.

2. Throughput: jika CPU sibuk mengeksekusi proses, jika begitu kerja telah dilaksanakan. Salah satu ukuran kerja adalah banyak proses yang diselesaikan per unit waktu, disebut througput. Untuk proses yang lama mungkin 1 proses per jam; untuk proses yang sebentar mungkin 10 proses perdetik.

3. Turnaround time: dari sudur pandang proses tertentu, kriteria yang penting adalah berapa lama untuk mengeksekusi proses tersebut. Interval dari waktu yang diizinkan dengan waktu yang dibutuhkan untuk menyelesaikan sebuah prose disebut turn-around time. Trun around time adalah jumlah periode untuk menunggu untuk bisa ke memori, menunggu di ready queue, eksekusi di CPU, dan melakukan I/O.

4. Waiting time: algoritma penjadual CPU tidak mempengaruhi waktu untuk melaksanakan proses tersebut atau I/O; itu hanya mempengaruhi jumlah waktu yang dibutuhkan proses di antrian ready. Waiting time adalah jumlah periode menghabiskan di antrian ready.

5. Response time: di sistem yang interaktif, turnaround time mungkin bukan waktu yang terbaik untukkriteria. Sering sebuah proses bisa memproduksi output diawal, dan bisa meneruskan hasil yang baru sementara hasil yang sebelumnya telah diberikan ke user. Ukuran yang lain adalah waktu dari pengiriamn permintaan sampai respon yang pertama di berikan. Ini disebut response time, yaitu waktu untuk memulai memberikan respon, tetapi bukan waktu yang dipakai output untu respon tersebut.

Biasanya yang dilakukan adalah memaksimalkan CPU utilization dan throughput, dan minimalkan turnaround time, waiting time, dan response time dalam kasus tertentu kita

2.7.2. Algoritma Penjadual *First Come, First Served*

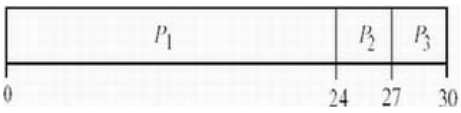
Penjadual CPU berurusan dengan permasalahan memutuskan proses mana yang akan dillaksanakan, oleh karena itu banyak bermacam algoritma penjadual, di seksi ini kita akan mendiskripsikan beberapa algoritma. Ini merupakan algoritma yang paling sederhana, dengan skema proses yang meminta CPU mendapat prioritas. Implementasi dari FCFS mudah diatasi dengan FIFO queue.

Contoh:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Gambar 2-26. Kedatangan Proses.

misal urutan kedatangan adalah P1, P2, P3 Gantt Chart untuk ini adalah:

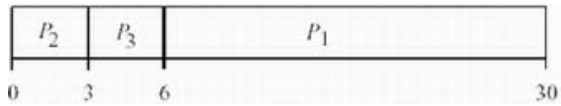


Gambar 2-27. Gannt Chart Kedatangan Proses I.

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

Gambar 2-28. Gannt Chart Kedatangan Proses II.

misal proses dibalik sehingga urutan kedatangan adalah P3, P2, P1.
Gantt chartnya adalah:



Gambar 2-29. Gannt Chart Kedatangan Proses III.

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$

Gambar 2-30. Gannt Chart Kedatangan Proses IV.

Dari dua contoh diatas bahwa kasus kedua lebih baik dari kasus pertama, karena pengaruh kedatangan disamping itu FCFS mempunyai kelemahan yaitu convoy effect dimana seandainya ada sebuah proses yang kecil tetapi dia mengantri dengan proses yang membutuhkan waktu yang lama mengakibatkan proses tersebut akan lama dieksekusi.

Penjadual FCFS algoritma adalah *nonpremptive*. Ketika CPU telah dialokasikan untuk sebuah proses, proses tetap menahan CPU sampai selssai. FCFS algortima jelas merupakan masalah bagi sistem time-sharing, dimana sangat penting untuk user mendapatkan pembagian CPU pada regular interval. Itu akan menjadi bencana untuk megizinkan satu proses pada CPU untuk waktu yang tidak terbatas

2.7.3. Penjadual *Shortest Job First*

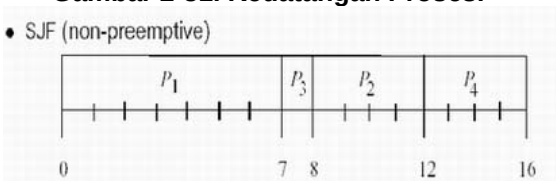
Salah satu algoritma yang lain adalah Shortest Job First. Algoritma ini berkaitan dengan waktu setiap proses. Ketika CPU bebas proses yang mempunyai waktu terpendek untuk menyelesaikannya mendapat prioritas. Seandainya dua proses atau lebih mempunyai waktu yang sama maka FCFS algoritma digunakan untuk menyelsaikan masalah tersebut.

Ada dua skema dalam SJFS ini yaitu:
1. *nonpremptive*— ketika CPU memberikan kepada proses itu tidak bisa ditunda hingga selesai.
2. *premptive*— bila sebuah proses datang dengan waktu prose lebih rendah dibandingkan dengan waktu proses yang sedang dieksekusi oleh CPU maka proses yang waktunya lebih rendah mendapatkan prioritas. Skema ini disebut juga Short - Remaining Time First (SRTF).

Contoh:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Gambar 2-31. Kedatangan Proses.



Gambar 2-32. Gannt Chart SJF Non-Preemtive.

Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Gambar 2-33. Rata-rata Menunggu

SJF algoritma mungkin adalah yang paling optimal, karena ia memberikan rata-rata minimum waiting untuk kumpulan dari proses yang mengantri. Dengan mengeksekusi waktu yang paling pendek baru yang paling lama. Akibatnya rata-rata waktu mnenuggu menurun.

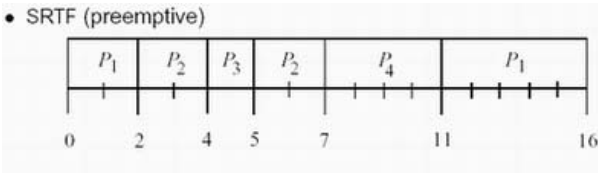
Hal yang sulit dengan SJF algoritma adalah mengethaui waku dari proses berikutnya. Untuk penjadual long term (lama) di sistem batch, kita bisa menggunakan panjang batas waktu proses yang user sebutkan ketika dia mengirim pekerjaan. Oleh karena itu sjf sering digunakan di penjadual long term. Walau pun SJF optimal tetapi ia tidak bisa digunakan untuk penjadual CPU short term. Tidak ada jalan untuk mengetahui panjang dari CPU burst berikutnya. Salah satu cara untuk mengimplementasikannya adalah dengan memprediksikan CPU burst berikutnya.

Contoh SJF premptive:

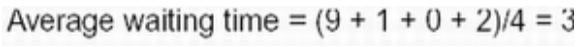
SJF algoritma mungkin adalah yang paling optimal, karena ia memberikan rata-rata minimum waiting untuk kumpulan dari proses yang mengantri.

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Gambar 2-34. Kedatangan Proses.



Gambar 2-35. Gannt Chart SJF Preemtive.



Gambar 2-36. Rata-rata Menunggu.

Kita lihat bahwa dengan premptive lebih baik hasilnya daripada non preemptive.

2.7.4. Penjadual Prioritas

Penjadualan SJF (*Shortest Job First*) adalah kasus khusus untuk algoritma penjadual Prioritas. Prioritas dapat diasosiasikan masing-masing proses dan CPU dialokasikan untuk proses dengan prioritas tertinggi. Untuk proritas yang sama dilakukan dengan FCFS.

Ada pun algoritma penjadual prioritas adalah sebagai berikut:

- Setiap proses akan mempunyai prioritas (bilangan integer). Beberapa sistem menggunakan integer dengan urutan kecil untuk proses dengan prioritas rendah, dan sistem lain juga bisa menggunakan integer urutan kecil untuk proses dengan prioritas tinggi. Tetapi dalam teks ini diasumsikan bahwa integer kecil merupakan prioritas tertinggi.

- CPU diberikan ke proses dengan prioritas tertinggi (integer kecil adalah prioritas tertinggi).

- Dalam algoritma ini ada dua skema yaitu:
  1. Preemptive: proses dapat di interupsi jika terdapat prioritas lebih tinggi yang memerlukan CPU.
  2. Nonpreemptive: proses dengan prioritas tinggi akan mengganti pada saat pemakain time-slice habis.

- SJF adalah contoh penjadual prioritas dimana prioritas ditentukan oleh waktu pemakaian CPU berikutnya. Permasalahan yang muncul dalam penjadualan prioritas adalah indefinite blocking atau starvation.

- Kadang-kadang untuk kasus dengan prioritas rendah mungkin tidak pernah dieksekusi. Solusi untuk algoritma penjadual prioritas adalah aging

- Prioritas akan naik jika proses makin lama menunggu waktu jatah CPU.

2.7.5. Penjadual *Round Robin*

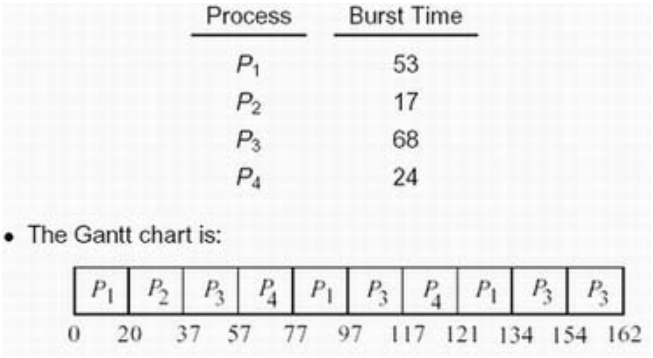
Algoritma *Round Robin* (RR) dirancang untuk sistem *time sharing*. Algoritma ini mirip dengan penjadual FCFS, namun preemption ditambahkan untuk switch antara proses. Antrian ready diperlakukan atau dianggap sebagai antrian sirkular. CPU menglilingi antrian ready dan mengalokasikan masing-masing proses untuk interval waktu tertentu sampai satu *time slice*/ *quantum*.

Berikut algritma untuk penjadual *Round Robin*:

- Setiap proses mendapat jatah waktu CPU (*time slice*/ *quantum*) tertentu Time slice/quantum umumnyaantara 10 - 100 milidetik.
  1. Setelah *time slice*/ *quantum* maka proses akan di-preempt dan dipindahkan ke antrian ready.
  2. Proses ini adil dan sangat sederhana.

- Jika terdapat n proses di "antrian ready" dan waktu quantum q (milidetik), maka:
  1. Maka setiap proses akan mendapatkan 1/n dari waktu CPU.
  2. Proses tidak akan menunggu lebih lama dari: (n-1)q *time units*.

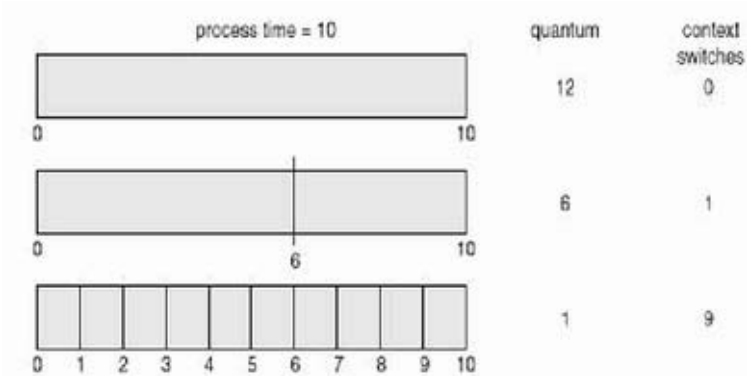
- Kinerja dari algoritma ini tergantung dari ukuran time quantum
  1. Time Quantum dengan ukuran yang besar maka akan sama dengan FCFS
  2. Time Quantum dengan ukuran yang kecil maka time quantum harus diubah ukurannya lebih besar dengan respek pada alih konteks sebaliknya akan memerlukan ongkos yang besar.



Gambar 2-37. Round Robin.

Tipikal: lebih lama waktu rata-rata turnaround dibandingkan SJF, tapi mempunyai response terhadap user lebih cepat.

Time Quantum Vs Alih Konteks



Gambar 2-38. Time Quantum dan Alih Konteks.

2.8. Penjadualan *Multiprocessor*

Multiprocessor membutuhkan penjadualan yang lebih rumit karena mempunyai banyak kemungkinan yang dicoba tidak seperti pada processor tunngal. Tapi saat ini kita hanya fokus pada processor yang homogen (sama) sesuai dengan fungsi masing-masing dari processor tersebut. Dan juga kita dapat menggunakan processor yang tersedia untuk menjalankan proses didalam antrian.

2.8.1. Penjadualan *Multiple Processor*

Diskusi kita sampai saat ini di permasalahan menjadualkan CPU di single prosesor. Jika multiple prosesor ada. Penjadualan menjadi lebih kompleks banyak kemungkinan telah dicoba dan telah kita lihat dengan penjadualan satu prosesor, tidak ada solusi yang terbaik. Pada kali ini kita hanya membahas secara sekilas tentang panjadualan di multiprosesor dengan syarat prosesornya identik.

Jika ada beberapa prosesor yang identik tersedia maka load sharing akan terjadi. Kita bisa menyediakan queue yang terpisah untuk setiap prosesor. Dalam kasus ini, bagaimana pun, satu prosesor bisa menjadi *idle* dengan antrian yang kosong sedangkan yang lain sangat sibuk. Untuk mengantisipasi hal ini kita menggunakan *ready queue* yang biasa. Semua proses pergi ke satu queue dan dijadualkan untuk prosesor yang bisa dipakai.

Dalam skema tersebut, salah satu penjadualan akan digunakan. Salah satu cara menggunakan symmetric multiprocessing (SMP). Dimana setiap prosesor menjadualkan diri sendiri. Setiap prosesor memeriksa raedy queue dan memilih proses yang akan dieksekusi.

Beberapa sistem membawa struktur satu langkah kedepan, dengan membawa semua keputusan penjadualan, I/O prosesing, dan aktivitas sistem yang lain ditangani oleh satu prosesor yang bertugas sebagai master prosesor. Prosesor yang lain mengeksekusi hanya user code yang disebut asymmetric multiprosessing jauh lebih mudah.

2.8.2. Penjadualan *Real Time*

Dalam bab ini, kita akan mendeskripsikan fasilitas penjadualan yang dibutuhkan untuk mendukung real time computing dengan bantuan sistem komputer.

Terdapat dua jenis real time computing: sistem hard real time dibutuhkan untuk menyelesaikan critical task dengan jaminan waktu tertentu. Secara umum, sebuah proses di kirim dengan sebuah pernyataan jumlah waktu dimana dibutuhkan untuk menyelesaikan atau menjalankan I/O. Kemudian penjadual bisa menjamin proses untuk selesai atau menolak permintaan karena tidak mungkin dilakukan. Karena itu setiap operasi harus dijamin dengan waktu maksimum.

Soft real time computing lebih tidak ketat. Itu membutuhkan bahwa proses yang kritis menerima prioritas dari yang lain. Walau pun menambah fungsi soft real time ke sistem time sharing mungkin akan mengakibatkan pembagian sumber yang tidak adildan mengakibatkan delay yang lebih lama, atau mungkin pembatalan bagi proses tertentu, Hasilnya adalah tujuan secara umum sistem yang bisa mendukung multimedia, graphic berkecepatan tinggi, dan variasi tugas yang tidak bisa diterima di lingkungan yang tidak mendukunng soft real time computing

Mengimplementasikan fungsi soft real time membutuhkan design yang hati-hati dan aspek yang berkaitan dengan sistem operasi. Pertama, sistem harus punya prioritas penjadualan, dan proses real time harus tidak melampaui waktu, walau pun prioritas non real time bisa terjadi. Kedua, dispatch latency harus lebih kecil. Semakin kecil latency, semakin cepat real time proses mengeksekusi.

Untuk menjaga dispatch tetap rendah. Kita butuh agar system call untuk preemptible. Ada beberapa cara untuk mencapai tujuan ini. Satu untuk memasukkan preemption

points di durasi yang lama system call, yang mana memeriksa apakah prioritas yang utama butuh untuk dieksekusi. Jika satu sudah, maka alih konteks mengambil alih; ketika high priority proses selesai, proses yang diinterupsi meneruskan dengan system call. Points premption bisa diganti hanya di lokasi yang aman di kernel — hanya kernel struktur tidak bisa dimodifikasi walau pun dengan preemption points, dispatch latency bisa besar, karena pada prakteknya untuk menambah beberapa preemption points untuk kernel.

Metoda yang lain untuk berurusan dengan preemption untuk membuat semua kernel preemptible. Karena operasi yang benar bisa dijamin, semua data kernel struktur dengan di proteksi. Dengan metode ini, kernel bisa selalu di preemptible, karena semua kernel bisa diupdate di proteksi.

Apa yang bisa diproteksi jika prioritas yang utama butuh untuk dibaca atau dimodifisikasi yang bisa dibutuhkan oleh yang lain, prioritas yang rendah? Prioritas yang tinggi harus menunggu menunggu untuk menyelesaikan prioritas yang rendah.

Fase konflik dari dispatch latency mempunyai dua komponen:
1. Preemption semua proses yang berjalan di kernel.
2. Lepas prioritas yang rendah untuk prioritas yang tinggi.

2.8.3. Penjadualan *Thread*

Di Bagian 2.5, kita mengenalkan threads untuk model proses, hal itu mengizinkan sebuah proses untuk mempunyai kontrol terhadap multiple threads. Lebih lanjut kita membedakan antara user-level dan kernel level threads. User level threads diatur oleh thread library. Untuk menjalankan di CPU, user level threads di mapping dengan asosiasi kernel level *thread*, walau pun mapping ini mungkin bisa *indirect* dan menggunakan *lightweight*.

2.9. Java *Thread* dan Algoritmanya

Penjadualan thread yang Runnable oleh Java Virtual Machine dilakukan dengan konsep preemptive dan mempunyai prioritas tertinggi. Dalam algoritma evaluasi ditentukan terlebih dahulu kriteria-kriterianya seperti utilisasinya dilihat dari segi waktu tunggu yang digunakan dan throughput yang disesuaikan dengan waktu turnaroundnya.

2.9.1. Penjadualan Java Thread

Java Virtual Machine menjadualkan thread menggunakan preemptive, berdasarkan prioritas algoritma penjadualan. Semua Java Thread diberikan sebuah prioritas dan Java Virtual Machine menjadualkan thread yang Runnable dengan menggunakan prioritas tertinggi saat eksekusi. Jika ada dua atau lebih thread yang Runnable yang mempunyai prioritas tertinggi, Java Virtual Machine akan menjadualkan thread tersebut menggunakan sebuah antrian secara FIFO.

2.9.1.1. Keunggulan Penjadualan Java *Thread*

1. Java *Virtual Machine* menggunakan prioritas preemtive berdasarkan algoritma penjadualan.

2. Semua thread Java mempunyai prioritas dan thread dengan proritas tertinggi dijadualkan untuk dieksekusi oleh Java Virtual Machine.

3. Jika terjadi dua thread dengan prioritas sama maka digunakan algoritma First In First Out.

Thread lain dijalankan bila terjadi hal-hal berikut ini:

- Thread yang sedang dieksekusi keluar dari status runnable misalnya diblok atau berakhir

- Thread dengan prioritas yang lebih tinggi dari thread yang sedang dieksekusi memasuki statusrunnable. Maka thread dengan prioritas yang lebih rendah ditunda eksekusinya dan digantikan oleh thread dengan prioritas lebih tinggi.

Time slicing tergantung pada implementasinya. Sebuah thread dapat memberi kontrol pada yield() method. Saat thread memberi sinyal pada CPU untuk mengontrol thread yang lain dengan prioritas yang sama maka thread tersebut dijadualkan untuk dieksekusi. Thread yang memberi kontrol pada CPU disebut Cooperative Multitasking.

2.9.1.2. Prioritas *Thread*

Java Virtual Machine memilih thread yang runnable dengan prioritas tertinggi. Semua thread java mempunyai prioritas dari 1 sampai 10. Prioritas tertinggi 10 dan berakhir dengan 1 sebagai prioritas terendah. Sedangkan prioritas normal adalah 5.

- Thread.MIN_PRIORITY = thread dengan prioritas terendah.
- Thread.MAX_PRIORITY = thread dengan prioritas tertinggi.
- Thread.NORM_PRIORITY = thread dengan prioritas normal.

Saat thread baru dibuat ia mempunyai prioritas yang sama dengan thread yang menciptakannya. Prioritas thread dapat diubah dengan menggunakan setpriority() method.

2.9.1.3. Penjadualan *Round-Robin* dengan Java

```
public class Scheduler extends Thread {
    public Scheduler() {
        timeSlice = DEFAULT_TIME_SLICE;
        queue = new Circularlist();
     }

     public Scheduler(int quantum) {
        timeSlice = quantum;
        queue = new Circularlist();
     }

     public addThread(Thread t) {
        t.setPriority(2);
        queue.additem(t);
    }

    private void schedulerSleep() {
        try{
            Thread.sleep(timeSlice );
        } catch (InterruptedException e){};
      }

      public void run(){
        Thread current;
        This.setpriority(6);
        while (true) {
            // get the next thread
```

```
                current = (Thread)qeue.getnext();
                if ( current != null) && (current.isAlive())) {
                        current.setPriority(4);
                        schedulerSleep();
                        current.setPriority(2)
                 }
              }
      }

      private CircularList queue;
      private int timeSlice;
      private static final int DEFAULT_TIME_SLICE = 1000;
}

public class TesScheduler{
    public static void main()String args[]) {
            Thread.currentThread().setpriority(Thread.Max_Priority);
            Schedular CPUSchedular = new Scheduler ();
            CPUSchedular.start()
            TestThread t1 = new TestThread("Thread 1");
            t1.start()
            CpuSchedular.addThread(t1);
            TestThread t2 = new TestThread("Thread 2");
            t2.start()
            CpuSchedular.addThread(t2);
            TestThread t3 = new TestThread("Thread 1");
            t3.start()
            CpuSchedular.addThread(t3);
    }
}
```

Gambar 2-39. Round Robin.

2.9.2. Evaluasi Algoritma

Bagaimana kita memilih sebuah algoritma penjadualan CPU untuk sistem-sistem tertentu. Yang menjadipokok masalah adalah kriteria seperti apa yang digunakan untuk memilih sebuah algoritma. Untuk memilih suatu algoritma, pertama yang harus kita lakukan adalah menentukan ukuran dari suatu kriteria berdasarkan:

- Memaksimalkan penggunaan CPU dibawah maksimum waktu responnya yaitu 1 detik.
- Memaksimalkan throughput karena waktu turnaroundnya bergerak secara linier pada saat eksekusi proses.

2.9.2.1. Sinkronisasi dalam Java

Setiap objek dalam java mempunyai kunci yang unik yang tidak digunakan biasanya. Saat method dinyatakan sinkron, maka method dipanggil untuk mendapatkan kunci untuk objek tersebut. Saat kunci tersebut dipunyai thread yang lain maka thread tersebut diblok dan dimasukkan kedalam kumpulan kunci objek, misalnya:

```
public synchronized void enter(Object item) {
      while (count == BUFFER_SIZE)
          ;
      Thread.yeild();
      ++count;
      buffer[in] = item;
      in = (in+1) % BUFFER_SIZE;
```

```
}

public synchronized void remove (){
        Object item;
        while (count == 0)
          ;
        Thread.yeild();
        --count;
        item = buffer[out]
        out = (out+1) % BUFFER_SIZE;
        return item
}
```

Gambar 2-40. Sinkronisasi.

2.9.2.2. Metoda Wait() dan Notify()

Thread akan memanggil method wait() saat:

1. Thread melepaskan kunci untuk objek.
2. Status thread diblok.
3. Thread yang berada dalam status wait menunggu objek.

Thread akan memanggil method notify() saat: Thread yang dipilih diambil dari thread yang ada pada himpunan wait. Dengan cara:

1. Pindahkan thread yang dipilih dari wait set ke entry set.
2. Atur status dari thread yang dipilih dari blocked menjadi runnable.

2.9.2.3. Contoh Metoda Wait() dan Notify()

```
public synchronized void enter(Object item){
   while (count == BUFFER_SIZE) {
        try{
            wait();
         } catch (InterruptedException e) {}
     }
    // add an item to the buffer
    ++count;
    buffer[in] = item;
   in = (in+1) % BUFFER_SIZE;
   notify();
}
public synchronized void remove(Object item){
     while (count == 0) {
        try {
           wait();
        }
        catch (InterruptedException e) {}
      }
     // remove an item to the buffer
     --count;
     item = buffer[out];
     out = (out+1) % BUFFER_SIZE;
     notify();
     return item;
}
```

Gambar 2-41. Contoh Wait() dan Notify().

2.10. Ringkasan

2.10.1. Proses

Sebuah proses adalah sebuah peristiwa adanya sebuah proses yang dapat dieksekusi. Sebagai sebuah eksekusi proses, maka hal tersebut membutuhkan perubahan keadaan. Keadaan dari sebuah proses dapat didefinisikan oleh aktivitas proses tertentu tersebut. Setiap proses mungkin menjadi satu dari beberapa state berikut, antara lain: *new*, *ready*, *running*, *waiting*, atau *terminated*. Setiap proses direpresentasikan ada sistem operasi berdasarkan proses-control-block (PCB)-nya.

Sebuah proses, ketika sedang tidak dieksekusi, ditempatkan pada antrian yang sama. Disini ada 2 kelas besar dari antrian dalam sebuah sistem operasi: permintaan antrian I/O dan ready queue. Ready queue memuat semua proses yang siap untuk dieksekusi dan yang sedang menunggu untuk dijalankan pada CPU. Setiap proses direpresentasikan oleh sebuah PCB, dan PCB tersebut dapat digabungkan secara bersamaan untuk mencatat sebuah ready queue. Penjadualan Long-term adalah pilihan dari proses-proses untuk diberi izin menjalankan CPU. Normalnya, penjadualan long-term memiliki pengaruh yang sangat besar bagi penempatan sumber, terutama managemen memori. Penjadualan Short-term adalah pilihan dari satu proses dari ready queue.

Proses-proses pada sistem dapat dieksekusi secara berkelanjutan. Disini ada beberapa alasan mengapa proses tersebut dapat dieksekusi secara berkelanjutan: pembagian informasi, penambahan kecepatan komputasi, modularitas, dan kenyamanan atau kemudahan. Eksekusi secara berkelanjutan menyediakan sebuah mekanisme bagi proses pembuatan dan penghapusan.

Pengeksekusian proses-proses pada operating system mungkin dapat digolongkan menjadi proses independent dan kooperasi. Proses kooperasi harus memiliki beberapa alat untuk mendukung komunikasi antara satu dengan yang lainnya. Prinsipnya adalah ada dua rencana komplementer komunikasi: pembagian memori dan sistem pesan. Metode pembagian memori menyediakan proses komunikasi untuk berbagi beberapa variabel. Proses-proses tersebut diharapkan dapat saling melakukan tukar-menukar informasi seputar pengguna variabel yang terbagi ini. Pada sistem pembagian memori, tanggung jawab bagi penyedia komunikasi terjadi dengan programmer aplikasi; sistem operasi harus menyediakan hanya pembagian memori saja. Metode sistem pesan mengizinkan proses-proses untuk tukar-menukar pesan. Tanggung jawab bagi penyedia komunikasi ini terjadi dengan sistem operasi tersebut.

2.10.2. *Thread*

Thread adalah sebuah alur kontrol dari sebuah proses. Suatu proses yang multithreaded mengandung beberapa perbedaan alur kontrol dengan ruang alamat yang sama. Keuntungan dari multithreaded meliputi peningkatan respon dari user, pembagian sumber daya proses, ekonomis, dan kemampuan untuk mengambil keuntungan dari arsitektur multiprosesor. User level thread adalah thread yang tampak oleh programmer dan tidak diketahui oleh kernel. User level thread secara tipikal dikelola oleh sebuah library thread di ruang user. Kernel level thread didukung dan dikelola oleh kernel sistem operasi. Secara umum, user level thread lebih cepat dalam pembuatan dan pengelolaan dari pada kernel thread. Ada tiga perbedaan tipe dari model yang berhubungan dengan user dan kernel thread.

- Model many to one: memetakan beberapa user level thread hanya ke satu buah kernel thread.
- Model one to one: memetakan setiap user thread ke dalam satu kernel thread. berakhir.
- Model many to many: mengizinkan pengembang untuk membuat user thread sebanyak mungkin, konkurensi tidak dapat tercapai karena hanya satu thread yang dapat dijadualkan oleh kernel dalam satu waktu.

Java adalah unik karena telah mendukung thread didalam tingkatan bahasanya. Semua program Java sedikitnya terdiri dari kontrol sebuah thread tunggal dan mempermudah membuat kontrol untuk multiple thread dengan program yang sama. JAVA juga menyediakan library berupa API untuk membuat thread, termasuk method untuk suspend dan resume suatu thread, agar thread tidur untuk jangka waktu tertentu dan menghentikan thread yang berjalan. Sebuah java thread juga mempunyai empat kemungkinan keadaan, diantaranya: New, Runnable, Blocked dan Dead. Perbedaan API untuk mengelola thread seringkali mengganti keadaan thread itu sendiri.

2.10.3. Penjadualan CPU

Penjadualan CPU adalah pemilihan proses dari antrian ready untuk dapat dieksekusi. Algoritma yang digunakan dalam penjadulan CPU ada bermacam-macam. Diantaranya adalah First Come First Serve (FCFS), merupakan algoritma sederhana dimana proses yang datang duluan maka dia yang dieksekusi pertama kalinya. Algoritma lainnya adalah Sorthest Job First (SJF), yaitu penjadualan CPU dimana proses yang paling pendek dieksekusi terlebih dahulu.

Kelemahan algoritma SJF adalah tidak dapat menghindari starvation. Untuk itu diciptakan algoritma Round Robin (RR). Penjadulan CPU dengan Round Robin adalah membagi proses berdasarkan waktu tertentu yaitu waktu quantum q. Setelah proses menjalankan eksekusi selama q satuan waktu maka akan digantikan oleh proses yang lain. Permasalahannya adalah bila waktu quantumnya besar sedang proses hanya membutuhkan waktu sedikit maka akan membuang waktu. Sedang bila waktu quantum kecil maka akan memakan waktu saat alih konteks.

Penjadualan FCFS adalah non-preemptive yaitu tidak dapat diinterupsi sebelum proses dieksekusi seluruhnya. Penjadualan RR adalah preemtive yaitu dapat dieksekusi saat prosesnya masih dieksekusi. Sedangkan penjadualan SJF dapat berupa nonpreemptive dan preemptive.

2.11. Soal-soal Latihan

2.11.1. Proses

1. Sebutkan lima aktivitas sistem operasi yang merupakan contoh dari suatu managemen proses. !
2. Definisikan perbedaan antara penjadualan short term, medium term dan long term.
3. Jelaskan tindakan yang diambil oleh sebuah kernel ketika alih konteks antar proses.
4. Informasi apa saja yang disimpan pada tabel proses saat alih konteks dari satu proses ke proses lain.

5. Di sistem UNIX terdapat banyak status proses yang dapat timbul (transisi) akibat event (eksternal) OS dan proses tersebut itu sendiri. Transisi state apa sajakah yang dapat ditimbulkan oleh proses itu sendiri. Sebutkan!
6. Apa keuntungan dan kekurangan dari:
   - • Komunikasi Simetrik dan asimetrik
   - • Automatic dan explicit buffering
   - • Send by copy dan send by reference
   - • Fixed-size dan variable sized messages
7. Jelaskan perbedaan short-term, medium-term dan long-term?
8. Jelaskan apa yang akan dilakukan oleh kernel kepada alih konteks ketika proses sedang berlangsung?
9. Beberapa single-user mikrokomputer sistem operasi seperti MS-DOS menyediakan sedikit atau tidak sama sekali arti dari pemrosesan yang konkuren. Diskusikan dampak yang paling mungkin ketika pemrosesan yang konkuren dimasukkan ke dalam suatu sistem operasi?
10. Perlihatkan semua kemungkinan keadaan dimana suatu proses dapat sedang berjalan, dan gambarkan diagram transisi keadaan yang menjelaskan bagaimana proses bergerak diantara state.
11. Apakah suatu proses memberikan 'issue' ke suatu disk I/O ketika, proses tersebut dalam 'ready' state, jelaskan?
12. Kernel menjaga suatu rekaman untuk setiap proses, disebut Proses Control Blocks (PCB). Ketika suatu proses sedang tidak berjalan, PCB berisi informasi tentang perlunya melakukan restart suatu proses dalam CPU. Jelaskan dua informasi yang harus dipunyai PCB.

2.11.2. *Thread*

1. Tunjukkan dua contoh pemrograman dari multithreading yang dapat meningkatkan sebuah solusi thread tunggal.
2. Tunjukkan dua contoh pemrograman dari multithreading yang tidak dapat meningkatkan sebuah solusi thread tunggal.
3. Sebutkan dua perbedaan antara user level thread dan kernel thread. Saat kondisi bagaimana salah satu dari thread tersebut lebih baik
4. Jelaskan tindakan yang diambil oleh sebuah kernel saat alih konteks antara kernel level thread.
5. Sumber daya apa sajakah yang digunakan ketika sebuah thread dibuat? Apa yang membedakannya dengan pembentukan sebuah proses.
6. Tunjukkan tindakan yang diambil oleh sebuah thread library saat alih konteks antara user level thread.

2.11.3. Penjadualan CPU

1. Definisikan perbedaan antara penjadualan secara preemptive dan nonpreemptive!
2. Jelaskan mengapa penjadualan strict nonpreemptive tidak seperti yang digunakan di sebuah komputer pusat.
3. Apakah keuntungan menggunakan time quantum size di level yang berbeda dari sebuah antrian sistem multilevel?

   Pertanyaan nomor 4 sampai dengan 5 dibawah menggunakan soal berikut:

   Misal diberikan beberapa proses dibawah ini dengan panjang CPU burst (dalam milidetik) Semua proses diasumsikan datang pada saat t=0

Proses	Burst Time	Prioritas

P1	**10**	**3**
P2	**1**	**1**
P3	**2**	**3**
P4	**1**	**4**
P5	**5**	**2**

Tabel 2-1. Tabel untuk soal 4—5

4. Gambarkan 4 diagram Chart yang mengilustrasikan eksekusi dari proses-proses tersebut menggunakan FCFS, SJF, prioritas nonpreemptive dan round robin.
5. Hitung waktu tunggu dari setiap proses untuk setiap algoritma penjadualan.
6. Jelaskan perbedaan algoritma penjadualan berikut:
   - • FCFS
   - • Round Robin
   - • Antrian Multilevel feedback
7. Penjadualan CPU mendefinisikan suatu urutan eksekusi dari proses terjadual. Diberikan n buah proses yang akan dijadualkan dalam satu prosesor, berapa banyak kemungkinan penjadualan yang berbeda? berikan formula dari n.
8. Tentukan perbedaan antara penjadualan preemptive dan nonpreemptive (cooperative). Nyatakan kenapa nonpreemptive scheduling tidak dapat digunakan pada suatu komputer center. Di sistem komputer nonpreemptive, penjadualan yang lebih baik digunakan.

2.12. Rujukan

1. Avi Silberschatz, Peter Galvin, dan Greg Gagne, *Applied Operationg System Concepts*, 1stEd., John Wiley & Sons, Inc. , 2002
2. William Stallings, *Operating Systems -- Fourth Edition*, Prentice Hall. , 2001
3. *http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/Proses.PDF* (http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/Proses.PDF)
4. *http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/CPU-Scheduler.PDF* (http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/CPU-Scheduler.PDF)
5. *http://www.cs.nyu.edu/courses/spring02/v22.0202-002/lecture-03.html* (http://www.cs.nyu.edu/courses/spring02/v22.0202-002/lecture-03.html)
6. *http://www.risc.uni-linz.ac.at/people/schreine/papers/idimt97/multithread.gif* (http://www.risc.uni-linz.ac.at/people/schreine/papers/idimt97/multithread.gif)
7. *http://www.unet.univie.ac.at/aix/aixprggd/genprogc/figures/genpr68.jpg* (http://www.unet.univie.ac.at/aix/aixprggd/genprogc/figures/genpr68.jpg)
8. *http://www.unet.univie.ac.at/aix/aixprggd/genprogc/understanding_threads.htm*
9. *http://www.etnus.com/Support/docs/rel5/html/cli_guide/images/procs_n_threads8a.gif* (http://www.etnus.com/Support/docs/rel5/html/cli_guide/images/procs_n_threads8a.gif)
10. *http://www.etnus.com/Support/docs/rel5/html/cli_guide/procs_n_threads5.html* (http://www.etnus.com/Support/docs/rel5/html/cli_guide/procs_n_threads5.html)
11. *http://www.crackinguniversity2000.it/boooks/1575211025/ch6.htm* (http://www.crackinguniversity2000.it/boooks/1575211025/ch6.htm)
12. *http://lass.cs.umass.edu/~shenoy/courses/fall01/labs/talab2.html* (http://lass.cs.umass.edu/~shenoy/courses/fall01/labs/talab2.html)
13. *http://www.isbiel.ch/~myf/opsys1/Exercises/Chap4/Problems1.html* (http://www.isbiel.ch/~myf/opsys1/Exercises/Chap4/Problems1.html)
14. *http://www.cee.hw.ac.uk/courses/5nm1/Exercises/2.htm* (http://www.cee.hw.ac.uk/courses/5nm1/Exercises/2.htm)

15. *http://www.cs.wisc.edu/~cao/cs537/midterm-answers1.txt*
(http://www.cs.wisc.edu/~cao/cs537/midterm-answers1.txt)