Dadang Priyanto, M.Kom

MODUL
SISTEM OPERASI

SEKOLAH TINGGI MANAJEMEN INFORMATIKA DAN KOMPUTER BUMIGORA
2017

LEMBAR VERIFIKASI MODUL

Modul mata kuliah Sistem Operasi ini di lakukan verifikasi agar dapat dipergunakan untuk kegiatan proses belajar mengajar (PBM) terutama pada matakuliah Sistem Operasi Program Studi S1 ILMU KOMPUTER pada STMIK BUMIGORA MATARAM. Dengan memperhatikan isi materi pada Modul ini, kami pihak program studi menyatakan bahwa;

1. Isi materi pada modul ini telah sesuai dengan silabus matakuliah
2. Tidak terdapat tumpang tindih materi yang diberikan terhadap materi lain

Terhadap kedua hal tersebut dinyatakan bahwa modul kuliah teori ini telah dapat dipergunakan untuk kegiatan belajar mengajar di program studi Ilmu Komputer

Terverifikasi, Mataram, 20/03/2017

Kaprodi S1 Ilmu Komputer

Ni Gusti Ayu Dasriani, S.Kom,.M.Kom
NIK:10.6.133

PERTEMUAN I
PENGENALAN SISTEM OPERASI

A. TUJUAN

Mahasiswa dapat memahami dan mengoperasikan ”DOS” khususnya untuk penggantian tanggal dan waktu sistem, menampilkan direktori, membuat direktori, masuk direktori, copi file, buat file, lihat isi file, penghapusan file.

B. MATERI

1. Sistem Operasi

Sistem operasi merupakan sebuah penghubung antara pengguna dari komputer dengan perangkat keras komputer. Sebelum ada sistem operasi, orang hanya mengunakan komputerdengan menggunakan sinyal analog dan sinyal digital. Seiring dengan berkembangnya pengetahuan dan teknologi, pada saat ini terdapat berbagai sistem operasi dengan keunggulan masing-masing. Untuk lebih memahami system operasi maka sebaiknya perlu diketahui terlebih dahulu beberapa konsep dasar mengenai sistem operasi itu sendiri.

Pengertian sistem operasi secara umum ialah pengelola seluruh sumber-daya yang terdapat pada system komputer dan menyediakan sekumpulan layanan (*system calls*) ke pemakai sehingga memudahkan dan menyamankan penggunaan serta pemanfaatan sumber-daya sistem komputer.

1) Fungsi Dasar

Sistem komputer pada dasarnya terdiri dari empat komponen utama, yaitu perangkat-keras, program aplikasi, sistem-operasi, dan para pengguna. Sistem operasi berfungsi untuk mengatur dan mengawasi penggunaan perangkat keras oleh berbagai program aplikasi serta para pengguna. Sistem operasi juga sering disebut resource allocator. Satu lagi fungsi penting sistem operasi ialah sebagai program pengendali yang bertujuan untuk menghindari kekeliruan (error) dan penggunaan komputer yang tidak perlu.

2) Tujuan Mempelajari Sistem Operasi

Tujuan mempelajari sistem operasi agar dapat merancang sendiri serta dapat memodifikasi sistem yang telah ada sesuai dengan kebutuhan kita, agar dapat memilih alternatif sistem operasi, memaksimalkan penggunaan sistem operasi dan agar konsep dan teknik sistem operasi dapat diterapkan pada aplikasi-aplikasi lain.

3) Sasaran Sistem Operasi

Sistem operasi mempunyai tiga sasaran utama yaitu kenyamanan membuat penggunaan computer menjadi lebih nyaman, efisien penggunaan sumber daya sistem komputer secara efisien, serta mampu berevolusi. Sistem operasi harus dibangun sehingga memungkinkan dan memudahkan pengembangan, pengujian serta pengajuan sistem-sistem yang baru.

4) Sejarah Sistem Operasi

Menurut Tanenbaum, sistem operasi mengalami perkembangan yang sangat pesat, yang dapat dibagi kedalam empat generasi :

- **Generasi Pertama (1945-1955)**
  Generasi pertama merupakan awal perkembangan sistem komputasi elektronik sebagai pengganti sistem komputasi mekanik, hal itu disebabkan kecepatan manusia untuk menghitung terbatas dan manusia sangat mudah untuk membuat kecerobohan, kekeliruan bahkan kesalahan. Pada generasi ini belum ada sistem operasi, maka sistem komputer diberi instruksi yang harus dikerjakan secara langsung.

- **Generasi Kedua (1955-1965)**
  Generasi kedua memperkenalkan *Batch Processing System*, yaitu Job yang dikerjakan dalam satu rangkaian, lalu dieksekusi secara berurutan.Pada generasi ini sistem komputer belum dilengkapi sistem operasi, tetapi beberapa fungsi sistem operasi telah ada, contohnya fungsi sistem operasi ialah FMS dan IBSYS.

- **Generasi Ketiga (1965-1980)**
  Pada generasi ini perkembangan sistem operasi dikembangkan untuk melayani banyak pemakai sekaligus, dimana para pemakai interaktif berkomunikasi lewat terminal secara on-line ke komputer, maka system operasi menjadi *multi-user* (di gunakan banyak pengguna sekaligus) dan *multi-programming* (melayani banyak program sekali gus).

- **Generasi Keempat (Pasca 1980an)**
  Pada masa ini para pengguna juga telah dinyamankan dengan *Graphical User Interface* yaitu antar-muka komputer yang berbasis grafis yang sangat nyaman, pada masa ini juga dimulai era komputasi tersebar dimana komputasi-komputasi tidak lagi berpusat di satu titik, tetapi dipecah dibanyak komputer sehingga tercapai kinerja yang lebih baik.

5) Layanan Sistem Operasi

Sebuah sistem operasi yang baik menurut Tanenbaum harus memiliki layanan sebagai berikut:

- Pembuatan program, sistem operasi menyediakan fasilitas dan layanan untuk membantu para pemrogram untuk menulis program
- Eksekusi program, yang berarti Instruksi-instruksi dan data-data harus dimuat ke memori utama, perangkat-parangkat masukan/ keluaran dan berkas harus di-inisialisasi, serta sumber-daya yang ada harus disiapkan, semua itu harus di tangani oleh sistem operasi
- Pengaksesan *I/O Device*, artinya Sistem Operasi harus mengambil alih sejumlah instruksi yang rumit dan sinyal kendali menjengkelkan agar pemrogram dapat berfikir sederhana dan perangkat pun dapat beroperasi
- Pengaksesan terkendali terhadap berkas pengaksesan sistem, artinya disediakannya mekanisme proteksi terhadap berkas untuk mengendalikan pengaksesan terhadap berkas; Pengaksesan sistem artinya pada pengaksesan digunakan bersama (*shared system*); Fungsi pengaksesan harus menyediakan proteksi terhadap sejumlah sumber-daya dan data dari pemakai tak terdistorsi serta menyelesaikan kon ik-kon ik dalam perebutan sumber-daya;
- Deteksi dan pemberian tanggapan pada kesalahan, yaitu jika muncul permasalahan muncul pada sistem komputer maka sistem operasi harus memberikan tanggapan yang menjelaskan kesalahan yang terjadi serta dampaknya terhadap aplikasi yang sedang berjalan dan
- Akunting, artinya Sistem Operasi yang bagus mengumpulkan data statistik penggunaan beragam sumber-daya dan memonitor parameter kinerja.

PERTEMUAN II
OPERASI SISTEM KOMPUTER

A. TUJUAN

Mahasiswa dapat memahami operasi sistem komputer.

B. TEORI

1. Sistem Operasi Komputer

Sistem komputer multiguna terdiri dari CPU (Central Processing Unit); serta sejumlah device controller yang dihubungkan melalui bus yang menyediakan akses ke memori. Setiap device controller bertugas mengatur perangkat yang tertentu (contohnya disk drive, audio device dan video display). CPU dan device controller dapat dijalankan secara bersamaan, namun demikian diperlukan mekanisme sinkronisasi untuk mengatur akses ke memori. Pada saat pertama kali dijalankan atau pada saat boot, terdapat sebuah program awal yang mesti dijalankan. Program awal ini disebut program bootstrap. Program ini berisi semua aspek dari system komputer, mulai dari register CPU, device controller, sampai isi memori. Interupsi merupakan bagian penting dari sistem arsitektur komputer.

Setiap sistem komputer memiliki mekanisme yang berbeda. Interupsi bisa terjadi apabila perangkat keras (hardware) atau perangkat lunak (software) minta "dilayani" oleh prosesor. Apabila terjadi interupsi maka prosesor menghentikan proses yang sedang dikerjakannya, kemudian beralih mengerjakan service routine untuk melayani interupsi tersebut. Setelah selesai mengerjakan service routine maka prosesor kembali melanjutkan proses yang tertunda.

2. Struktur I/O

1) Interupsi I/O

Untuk memulai operasi I/O, CPU me*load* register yang bersesuaian ke *device controller*. Sebaliknya *Device controller* memeriksa isi register untuk kemudian menentukan operasi apa yang harus dilakukan. Pada saat operasi I/O dijalankan ada dua kemungkinan, yaitu *synchronous I/O* dan *asynchronous I/O*. Pada *synchronous I/O*, kendali dikembalikan ke proses pengguna setelah proses I/O selesai dikerjakan. Sedangkan pada *asynchronous I/O*, kendali dikembalikan ke proses pengguna tanpa menunggu proses I/O selesai. Sehingga proses I/O dan proses pengguna dapat dijalankan secara bersamaan.

2) Struktur DMA

Direct Memory Access (DMA) suatu metoda penanganan I/O dimana *device controller* langsung berhubungan dengan memori tanpa campur tangan CPU. Setelah men-set *buffers*, *pointers* dan *counters* untuk perangkat I/O, *device controller* mentransfer blok data langsung ke penyimpanan tanpa campur tangan CPU. DMA digunakan untuk perangkat I/O dengan kecepatan tinggi. Hanya terdapat satu interupsi setiap blok, berbeda dengan perangkat yang mempunyai kecepatan rendah dimana interupsi terjadi untuk setiap *byte* (*word*).

3. Struktur Penyimpanan

Program komputer harus berada di memori utama (biasanya RAM) untuk dapat dijalankan. Memori utama adalah satu-satunya tempat penyimpanan yang dapat diakses secara langsung oleh prosesor. Idealnya program dan data secara keseluruhan dapat disimpan dalam memori utama secara permanen. Namun demikian hal ini tidak mungkin karena :

- Ukuran memori utama relatif kecil untuk dapat menyimpan data dan program secara keseluruhan.
- Memori utama bersifat *volatile*, tidak bisa menyimpan secara permanen, apabila komputer dimatikan maka data yang tersimpan di memori utama akan hilang.

1) Memori Utama

Hanya memori utama dan register merupakan tempat penyimpanan yang dapat diakses secara langsung oleh prosesor. Oleh karena itu instruksi dan data yang akan dieksekusi harus disimpan di memori utama atau register. Untuk mempermudah akses perangkat I/O ke memori, pada arsitektur komputer menyediakan fasilitas pemetaan memori ke I/O. Dalam hal ini sejumlah alamat di memori dipetakan dengan *device*

register. Membaca dan menulis pada alamat memori ini menyebabkan data ditransfer dari dan ke *device register*. Metode ini cocok untuk perangkat dengan waktu respon yang cepat seperti *video controller*.
Register yang terdapat dalam prosesor dapat diakses dalam waktu 1 *clock cycle*. Hal ini menyebabkan register merupakan media penyimpanan dengan akses paling cepat bandingkan dengan memori utama yang membutuhkan waktu relatif lama. Untuk mengatasi perbedaan kecepatan, dibuatlah suatu penyangga (*buffer*) penyimpanan yang disebut *cache*.

2) Magnetic Disk

Magnetic Disk berperan sebagai *secondary storage* pada sistem komputer modern. *Magnetic Disk* disusun dari piringan-piringan seperti CD. Kedua permukaan piringan diselimuti oleh bahan-bahan magnetik. Permukaan dari piringan dibagi-bagi menjadi *track* yang memutar, yang kemudian dibagi lagi menjadi beberapa sektor.

4. Storage Hierarchy

Dalam *storage hierarchy structure*, data yang sama bisa tampil dalam level berbeda dari system penyimpanan. Sebagai contoh integer A berlokasi pada bekas B yang ditambahkan 1, dengan asumsi bekas B terletak pada *magnetic disk*. Operasi penambahan diproses dengan pertama kali mengeluarkan operasi I/O untuk menduplikat disk block pada A yang terletak pada memori utama Operasi ini diikuti dengan kemungkinan penduplikatan A ke dalam *cache* dan penduplikatan A ke dalam internal register. Sehingga penduplikatan A terjadi di beberapa tempat. Pertama terjadi di internal register dimana nilai A berbeda dengan yang di sistem penyimpanan. Dan nilai di A akan kembali sama ketika nilai baru ditulis ulang ke *magnetic disk*. Pada kondisi multi prosesor, situasi akan menjadi lebih rumit. Hal ini disebabkan masing-masing prosesor mempunyai *local cache*. Dalam kondisi seperti ini hasil duplikat dari A mungkin hanya ada di beberapa *cache*. Karena CPU (register-register) dapat dijalankan secara bersamaan maka kita harus memastikan perubahan nilai A pada satu *cache* akan mengubah nilai A pada semua *cache* yang ada. Hal ini disebut sebagai *Cache Coherency*.

5. Proteksi Perangkat Keras

Tanpa proteksi jika terjadi kesalahan maka hanya satu saja program yang dapat dijalankan atau seluruh output pasti diragukan. Banyak kesalahan pemprograman dideteksi oleh perangkat keras. Kesalahan ini biasanya ditangani oleh sistem operasi. Jika terjadi kesalahan program, perangkat keras akan meneruskan kepada sistem operasi dan sistem operasi akan menginterupsi dan mengakhirinya. Pesan kesalahan disampaikan dan memori dari program akan dibuang. Tapi memori yang terbuang biasanya tersimpan pada disk agar *programmer* bisa membetulkan kesalahan dan menjalankan program ulang.

1) Operasi Dual Mode

Untuk memastikan operasi berjalan baik kita harus melindungi sistem operasi, program dan data dari program-program yang salah. Proteksi ini memerlukan *share resources*. Hal ini bisa dilakukan system operasi dengan cara menyediakan pendukung perangkat keras yang mengizinkan kita membedakan mode pengeksekusian program. Mode yang kita butuhkan ada dua mode operasi yaitu :

- Mode Monitor
- Mode Pengguna.

Pada perangkat keras akan ada bit atau Bit Mode yang berguna untuk membedakan mode apa yang sedang digunakan dan apa yang sedang dikerjakan. Jika Mode Monitor maka akan benilai 0, dan jika Mode Pengguna maka akan bernilai 1. Pada saat *boot time*, perangkat keras bekerja pada mode monitor dan setelah sistem operasi di *load* maka akan mulai masuk ke mode pengguna. Ketika terjadi *trap* atau interupsi, perangkat keras akan men*switch* lagi keadaan dari mode pengguna menjadi mode monitor (terjadi perubahan *state* menjadi bit 0). Dan akan kembali menjadi mode pengguna jikalau sistem operasi mengambil alih proses dan control komputer (*state* akan berubah menjadi bit 1).

2) Proteksi I/O

Pengguna bisa mengacaukan sistem operasi dengan melakukan instruksi I/O ilegal dengan mengakses lokasi memori untuk sistem operasi atau dengan cara hendak melepaskan diri dari prosesor. Untuk mencegahnya kita menganggap semua instruksi

I/O sebagai *privilidge instruction* sehingga mereka tidak bisa mengerjakan instruksi I/O secara langsung ke memori tapi harus lewat sistem operasi terlebih dahulu. Proteksi I/O dikatakan selesai jika pengguna dapat dipastikan tidak akan menyentuh mode monitor. Jika hal ini terjadi proteksi I/O dapat dikompromikan.

3) Proteksi Memori

Salah satu proteksi perangkat keras ialah dengan proteksi memori yaitu dengan pembatasan penggunaan memori. Disini diperlukan beberapa istilah yaitu:

- Base Register yaitu alamat memori fisik awal yang dialokasikan/ boleh digunakan oleh pengguna
- Limit Register yaitu nilai batas dari alamat memori fisik awal yang dialokasikan/boleh digunakan oleh pengguna.
- Proteksi Perangkat Keras.

Sebagai contoh sebuah pengguna dibatasi mempunyai base register 300040 dan mempunyai limit register 120900 maka pengguna hanya diperbolehkan menggunakan alamat memori fisik antara 300040 hingga 420940 saja.

PERTEMUAN III

STRUKTUR SISTEM OPERASI

A. TUJUAN

Mahasiswa dapat memahami struktur sistem operasi

B. TEORI

1. Komponen-komponen Sistem

Pada kenyataannya tidak semua sistem operasi mempunyai struktur yang sama. Namun menurut Avi Silberschatz, Peter Galvin, dan Greg Gagne, umumnya sebuah sistem operasi modern mempunyai komponen sebagai berikut:

- Managemen Proses.
- Managemen Memori Utama.
- Managemen *Secondary-Storage*.
- Managemen Sistem I/O.
- Managemen Berkas.
- Sistem Proteksi.
- Jaringan.
- *Command-Interpreter system*

2. Managemen Proses

Proses adalah keadaan ketika sebuah program sedang di eksekusi. Sebuah proses membutuhkan beberapa sumber daya untuk menyelesaikan tugasnya. Sumber daya tersebut dapat berupa *CPU time*, memori, berkas-berkas, dan perangkat-perangkat I/O. Sistem operasi bertanggung jawab atas aktivitas-aktivitas yang berkaitan dengan managemen proses seperti:

- Pembuatan dan penghapusan proses pengguna dan sistem proses.
- Menunda atau melanjutkan proses.
- Menyediakan mekanisme untuk proses sinkronisasi.
- Menyediakan mekanisme untuk proses komunikasi.
- Menyediakan mekanisme untuk penanganan *deadlock*.

3. Managemen Memori Utama

Memori utama atau lebih dikenal sebagai memori adalah sebuah *array* yang besar dari *word* atau *byte*, yang ukurannya mencapai ratusan, ribuan, atau bahkan jutaan. Setiap *word* atau *byte* mempunyai alamat tersendiri. Memori Utama berfungsi sebagai tempat penyimpanan yang akses datanya digunakan oleh CPU atau perangkat I/O. Memori utama termasuk tempat penyimpanan data yang sementara (*volatile*) artinya data dapat hilang begitu sistem dimatikan. Sistem operasi bertanggung jawab atas aktivitas-aktivitas yang berkaitan dengan managemen memori seperti menjaga *track* dari memori yang sedang digunakan dan siapa yang menggunakannya, memilih program yang akan di *load* ke memori dan msengalokasikan dan meng-dealokasikan ruang memori sesuai kebutuhan.

4. Managemen Secondary-Storage

Data yang disimpan dalam memori utama bersifat sementara dan jumlahnya sangat kecil. Oleh karena itu, untuk meyimpan keseluruhan data dan program komputer dibutuhkan *secondary storage* yang bersifat permanen dan mampu menampung banyak data. Contoh dari *secondary storage* adalah *hard disk*, disket dll. Sistem operasi bertanggung-jawab atas aktivitas-aktivitas yang berkaitan dengan *disk-management* seperti: *free-space management*, alokasi penyimpanan, penjadualan disk.

5. Managemen Sistem I/O

Sering disebut *device manager*. Menyediakan "*device driver*" yang umum sehingga operasi I/O dapat seragam (membuka, membaca, menulis, menutup). Contoh: pengguna menggunakan operasi yang sama untuk membaca berkas pada *hard-disk*, CD-ROM dan *oppy disk*. Komponen Sistem Operasi untuk sistem I/O:

- *Buffer* : menampung sementara data dari/ ke perangkat I/O.
- *Spooling* : melakukan penjadualan pemakaian I/O sistem supaya lebih efisien (antrian dsb.)
- Menyediakan *driver* untuk dapat melakukan operasi "rinci" untuk perangkat keras I/O tertentu.

6. Managemen Berkas

Berkas adalah kumpulan informasi yang berhubungan sesuai dengan tujuan pembuat berkas tersebut. Berkas dapat mempunyai struktur yang bersifat hirarkis (direktori, volume, dll). Sistem operasi bertanggung-jawab :

- Pembuatan dan penghapusan berkas.
- Pembuatan dan penghapusan direktori.
- Mendukung manipulasi berkas dan direktori.
- Memetakan berkas ke *secondary storage*
- Mem*backup* berkas ke media penyimpanan yang permanen (*non-volatile*).

7. Sistem Proteksi

Proteksi mengacu pada mekanisme untuk mengontrol akses yang dilakukan oleh program, prosesor, atau pengguna ke sistem sumber daya. Mekanisme proteksi harus:

- Membedakan antara penggunaan yang sudah diberi izin dan yang belum.
- *Specify the controls to be imposed provide a means of enforcement*

8. Jaringan

Sistem terdistribusi adalah sekumpulan prosesor yang tidak berbagi memori atau *clock*. Tiap prosesor mempunyai memori sendiri. Prosesor-prosesor tersebut terhubung melalui jaringan komunikasi Sistem terdistribusi menyediakan akses pengguna ke bermacam sumber-daya sistem. Akses tersebut menyebabkan :

- *Computation speed-up* .
- *Increased data availability*
- *Enhanced reliability*

9. Command-Interpreter System

Sistem Operasi menunggu instruksi dari pengguna (*command driven*). Program yang membaca instruksi dan mengartikan *control statements* umumnya disebut : *control-card interpreter*, *command-line interpreter* dan *UNIX shell*. *Command-Interpreter System* sangat bervariasi dari satu sistem operasi ke sistem operasi yang lain dan disesuaikan dengan tujuan dan teknologi *I/O devices* yang ada. Contohnya: *CLI*, *Windows*, *Pen-based* (*touch*) dan lain-lain.

10. Layanan Sistem Operasi

Eksekusi program adalah kemampuan sistem untuk "*load*" program ke memori dan menjalankan program. Operasi I/O : pengguna tidak dapat secara langsung mengakses sumber daya perangkat keras, sistem operasi harus menyediakan mekanisme untuk melakukan operasi I/O atas nama pengguna. Sistem manipulasi berkas dalah kemampuan program untuk operasi pada berkas (membaca, menulis, membuat and menghapus berkas). Komunikasi adalah pertukaran data/ informasi antar dua atau lebih proses yang berada pada satu komputer (atau lebih). Deteksi *error* adalah menjaga kestabilan sistem dengan mendeteksi "*error* ", perangkat keras mau pun operasi. Efesisensi penggunaan system :

- *Resource allocator* adalah mengalokasikan sumber daya ke beberapa pengguna atau *job* yang jalan pada saat yang bersamaan. Proteksi menjamin akses ke sistem sumber daya dikendalikan (pengguna dikontrol aksesnya ke sistem).
- *Accounting* adalah merekam kegiatan pengguna, jatah pemakaian sumber daya (keadilan atau kebijaksanaan).

11. System Calls

System call menyediakan interface antara program (program pengguna yang berjalan) dan bagian OS. *System call* menjadi jembatan antara proses dan sistem operasi. *System call* ditulis dalam bahasa *assembly* atau bahasa tingkat tinggi yang dapat mengendalikan mesin (C).
Tiga cara memberikan parameter dari program ke sistem operasi:

- Melalui registers (sumber daya di CPU).
- Menyimpan parameter pada data struktur (table) di memori, dan alamat table tsb ditunjuk oleh *pointer* yang disimpan di register.
- *Push* (*store*) melalui "*stack*" pada memori dan OS mengambilnya melalui *pop* pada *stack* tsb.

12. Mesin Virtual

Sebuah mesin virtual (*Virtual Machine*) menggunakan misalkan terdapat sistem program => control Program yang mengatur pemakaian sumber daya perangkat keras. Control program = trap *System call* + akses ke perangkat keras. Control program memberikan fasilitas ke proses pengguna. Mendapatkan jatah CPU dan memori. Menyediakan *interface* "identik" dengan apa

yang disediakan oleh perangkat keras => *sharing devices* untuk berbagai proses. Mesin Virtual (MV) (MV) => control program yang minimal MV memberikan ilusi *multitasking*: seolah-olah terdapat prosesor dan memori ekslusif digunakan MV. MV memilah fungsi *multitasking* dan implementasi *extended machine* (tergantung proses pengguna) =>exible dan lebih mudah untuk pengaturan. Jika setiap pengguna diberikan satu MV => bebas untuk menjalankan OS (kernel) yang diinginkan pada MV tersebut. Potensi lebih dari satu OS dalam satu komputer. Contoh: IBM VM370: menyediakan MV untuk berbagai OS: CMS (interaktif), MVS, CICS, dll. Masalah: Sharing disk => OS mempunyai sistem berkas yang mungkin berbeda. IBM: virtual disk (minidisk) yang dialokasikan untuk pengguna melalui MV. Konsep MV menyediakan proteksi yang lengkap untuk sumber daya sistem, dikarenakan tiap MV terpisah dari MV yang lain. Namun, hal tersebut menyebabkan tidak adanya *sharing* sumberdaya secara langsung. MV merupakan alat yang tepat untuk penelitian dan pengembangan sistem operasi. Konsep MV susah untuk diimplementasi sehubungan dengan usaha yang diperlukan untuk menyediakan duplikasi dari mesin utama.

13. Perancangan Sistem dan Implementasi

Target untuk pengguna: sistem operasi harus nyaman digunakan, mudah dipelajari, dapat diandalkan, aman dan cepat. Target untuk sistem: sistem operasi harus gampang dirancang, diimplementasi dan dipelihara, sebagaimana eksibel, *error* dan efisien.
Mekanisme dan Kebijaksanaan:

- Mekanisme menjelaskan bagaimana melakukan sesuatu kebijaksanaan memutuskan apa yang akan dilakukan. Pemisahan kebijaksanaan dari mekanisme merupakan hal yang sangat penting; mengizinkan eksibilitas yang tinggi bila kebijaksanaan akan diubah nanti.
- Kebijaksanaan memutuskan apa yang akan dilakukan.

Pemisahan kebijaksanaan dari mekanisme merupakan hal yang sangat penting; ini mengizinkan eksibilitas yang tinggi bila kebijaksanaan akan diubah nanti. Implementasi Sistem biasanya menggunakan bahas *assembly*, system operasi sekarang dapat ditulis dengan menggunakan bahasa tingkat tinggi. Kode yang ditulis dalam bahasa tingkat tinggi: dapat dibuat dengan cepat, lebih ringkas, lebih mudah dimengerti dan didebug. Sistem operasi lebih mudah dipindahkan ke perangkat keras yang lain bila ditulis dengan bahasa tingkat tinggi.

14. System Generation (SYSGEN)

Sistem operasi dirancang untuk dapat dijalankan di berbagai jenis mesin;sistemnya harus di konfigurasi untuk tiap komputer. Program SYSGEN mendapatkan informasi mengenai konfigurasi khusus dari sistem perangkat keras. *Booting* : memulai komputer dengan me*load* kernel. *Bootstrap program* : kode yang disimpan di code ROM yang dapat menempatkan kernel, memasukkannya kedalam memori, dan memulai eksekusinya.

C. LATIHAN

1. Sebutkan tiga tujuan utama dari sistem operasi!
2. Sebutkan keuntungan dari *multiprogramming*!
3. Sebutkan perbedaan utama dari sistem operasi antara computer *mainframe* dan PC?
4. Sebutkan kendala-kendala yang harus diatasi oleh *programmer* dalam menulis sistem operasi untuk lingkungan waktu nyata?
5. Jelaskan perbedaan antara *symmetric* dan *asymmetric multiprocessing*. Sebutkan keuntungan dan kerugian dari sistem *multiprosessor*!
6. Apakah perbedaan antara *trap* dan *interrupt*? Sebutkan penggunaan dari setiap fungsi tersebut!
7. Untuk jenis operasi apakah **DMA** itu berguna? Jelaskan jawabannya!
8. Sebutkan dua kegunaan dari *memory cache*! Problem apakah yang dapat dipecahkan dan juga muncul dengan adanya *cache* tersebut?
9. Beberapa **CPU** menyediakan lebih dari dua mode operasi. Sebutkan dua kemungkinan penggunaan dari mode tersebut?
10. Sebutkan lima kegiatan utama dari sistem operasi yang berhubungan dengan managemen proses!
11. Sebutkan tiga kegiatan utama dari sistem operasi yang berhubungan dengan managemen memori!
12. Sebutkan tiga kegiatan utama dari sistem operasi yang berhubungan dengan managemen *secondary-storage*!

PERTEMUAN IV
PROSES

A. TUJUAN

Mahasiswa dapat memahami proses

B. TEORI

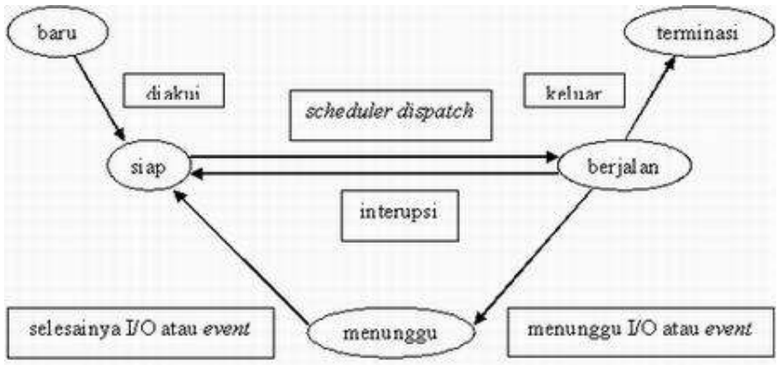
1. Konsep Dasar dan Definisi Proses

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 berisikan variabel global.

1) 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/O atau penerimaan sebuah tanda/ signal).
- *Ready* : Proses sedang menunggu untuk ditugaskan pada sebuah prosesor.
- *Terminated* : Proses telah selsesai melaksanakan tugasnya/ mengeksekusi.



Gambar 4.1. Keadaan Proses.

2) 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 4.2.

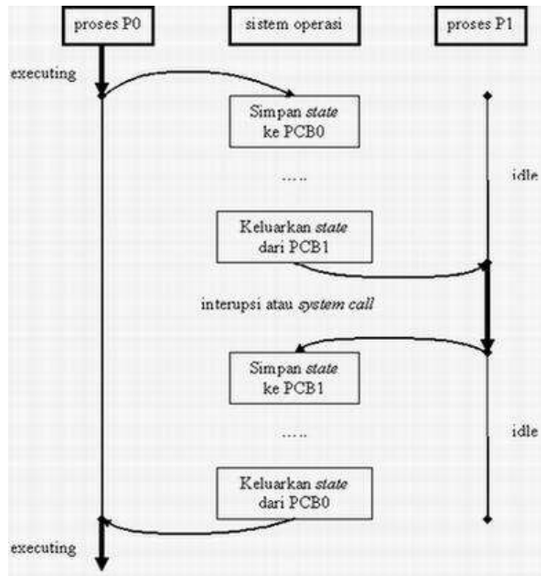
Gambar 4.2. *Process Control Block*.

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 4.3).



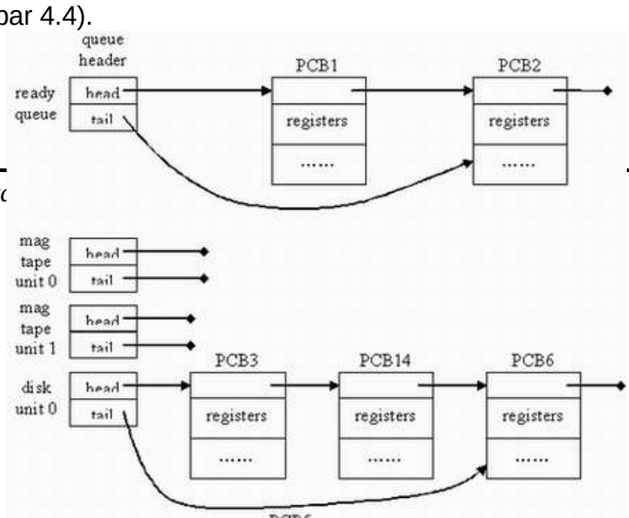
Gambar 4.3. *CPU Register*.

- 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 yang digunakan oleh sistem operasi
- 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.

2. Penjadualan Proses

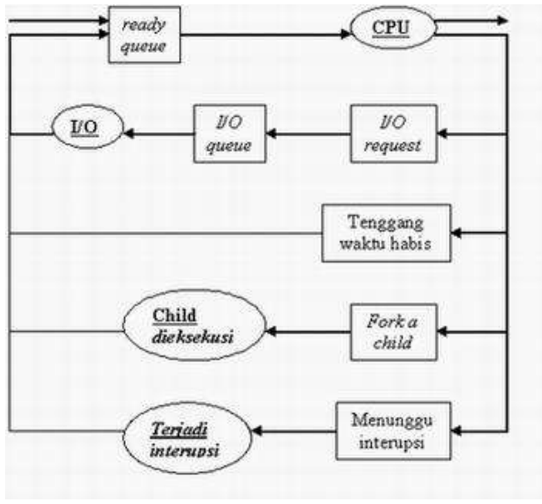
1) Penjadualan Antrian

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 4.4).



Gambar 4.4. Device Queue.

Reprensentasi umum untuk suatu diskusi mengenai penjadualan proses adalah diagram antrian, seperti pada Gambar 4.5. 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.



Gambar 4.5. Diagram Antrian.

Proses tersebut dapat digantikan secara paksa dari CPU, sebagai hasil dari suatu interupsi, dan diletakkan kembali dalam ready queue.
Dalam dua kasus pertama, proses akhirnya berganti dari waiting state menjadi ready state, lalu disaat dimana proses tersebut diganti dari seluruh queue dan memiliki PCB nya dan sumber-sumber/ resources dialokasikan kembali.

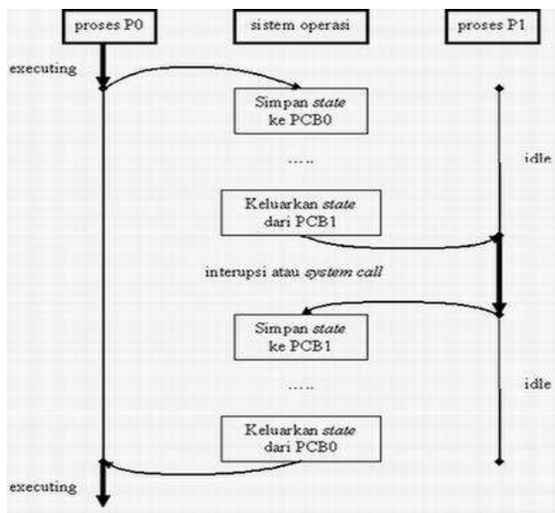
2) Penjadual Medium-term

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.

Gambar 4.6. Penjadual Medium-term.

3) Alih Konteks

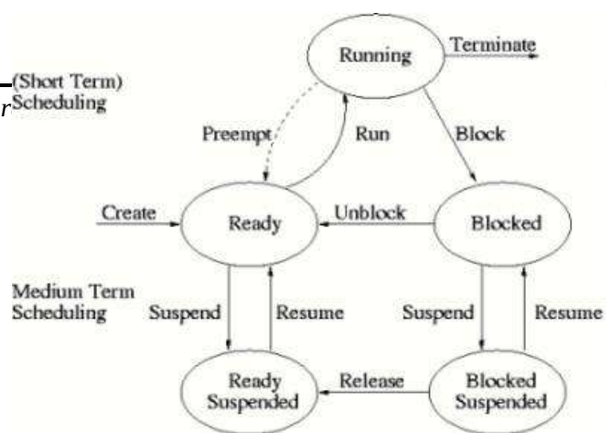
Mengganti CPU ke proses lain memerlukan penyimpanan suatu keadaan proses lama *state of old process*) dan kemudian beralih ke proses yang baru. Tugas tersebut diketahui sebagai alih konteks (*context switch*). Alih konteks sebuah proses digambarkan dalam PCB suatu proses; termasuk nilai dari CPU register, status proses (lihat Gambar 4.7) 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. Waktu alih konteks sangat begantung pada dukungan perangkat keras.



Gambar 4.7. Alih Konteks.

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 untuk pembuatan proses dan terminasi proses.



Gambar 4.8. Operasi pada Proses.

1) Pembuatan Proses

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.

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 system 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.

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.

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:

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

Program Produser Konsumer.

```
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;
}
```

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) 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 system penyampaian pesan, dan sistem-sistem pesan dapat diberikan dalam banyak cara.

5. 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.
- Pengiriman berdasarkan salinan atau referensi.
- Pesan berukuran pasti dan variabel.

1) 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) 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.

6. 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.

7. 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.

Program Produser Konsumer Alternatif. Sumber: . . .

```
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 );
}
}
}
```

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:

Keluaran Program Produser Konsumer. Sumber: . . .

```
. . .
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
```

. . .

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

Keluaran Program Produser Konsumer. Sumber: . . .

```
. . .
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
. . .
```

Situasi di atas dikenal dengan race conditions. Kita dapat menghindari situasi itu dengan mensinkronisasikan aktivitas p1 dan c1 (sehubungan dengan akses mereka ke *mailbox*).

Mailbox
Program Send/ Receive. Sumber: . . .

```
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;
}
}
}
```

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.

C. LATIHAN

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.
6. Sebutkan!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.

PERTEMUAN V
PENJADWALAN CPU

A. TUJUAN

Mahasiswa dapat memahami penjadwalan CPU

B. TEORI

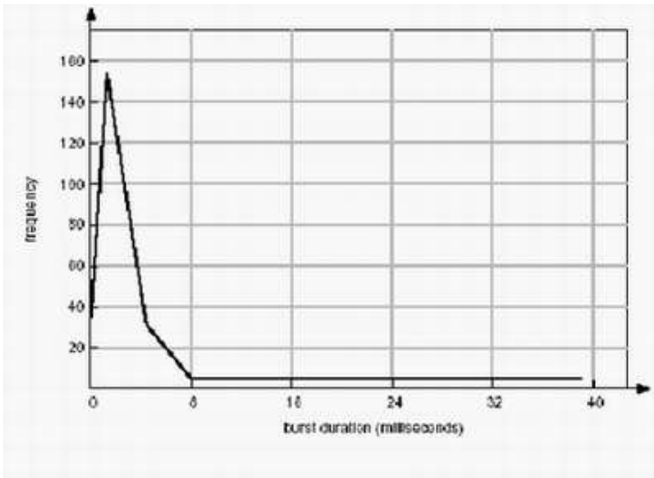
Penjadual CPU adalah basis dari multi programming sistem operasi. Dengan men-switch CPU diantara proses. Akibatnya sistem operasi bisa membuat komputer produktif. Dalam bab ini kami akan mengenalkan tentang dasar dari konsep penjadual dan beberapa algoritma penjadual. Dan kita juga memaparkan masalah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

1. Konsep Dasar

Tujuan dari multi programming adalah untuk mempunyai proses berjalan secara bersamaan, unutk 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 system 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.

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 5.1. CPU Burst

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:

- Berubah dari running ke waiting state.
- Berubah dari running ke ready state.
- Berubah dari waiting ke ready.
- 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.

3) Dispatcher

Komponen yang lain yang terlibat dalam penjadual CPU adalan *dispatcher*. *Dispatcher* adalah modul yang memberikan kontrol CPU kepada proses yang fungsinya adalah:

- Alih Konteks
- Switching to user mode.
- Lompat dari suatu bagian di progam user untuk mengulang progam

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 :

- CPU utilization : mempunyai range dari 0 ke 100 persen. Di sistem yang sebenarnya seharusnya ia mempunyai range dari 40 persen samapi 90 persen.
- 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.
- Turnaround time : dari sudut 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.
- 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.
- Response time : di sistem yang interaktif, turnaround time mungkin bukan waktu yang terbaik untuk kriteria. 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.

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.

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:

- *Non premptive* : ketika CPU memberikan kepada proses itu tidak bisa ditunda hingga selesai.
- *premptive* : bila sebuah proses datang dengan waktu proses 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).

4. Penjadual Prioritas

Penjadualan SJF (*Shortest Job First*) adalah kasus khusus untuk algoritma penjadual Prioritas. 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:

- Preemptive: proses dapat di interupsi jika terdapat prioritas lebih tinggi yang memerlukan CPU.
- Non preemptive: 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.

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 umumnya antara 10 - 100 milidetik.

- Setelah *time slice* / *quantum* maka proses akan dipreempt dan dipindahkan ke antrian ready.
- Proses ini adil dan sangat sederhana.

Jika terdapat n proses di "antrian ready" dan waktu quantum q (milidetik), maka:

- Maka setiap proses akan mendapatkan 1/n dari waktu CPU.
- Proses tidak akan menunggu lebih lama dari: (n-1)q *time units* .

Kinerja dari algoritma ini tergantung dari ukuran time quantum •

- Time Quantum dengan ukuran yang besar maka akan sama dengan FCFS
- 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.

6. 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.

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 ready 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) Penjadualan Real Tim

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 kon ik dari dispatch latency mempunyai dua komponen:

- Preemption semua proses yang berjalan di kernel.
- Lepas prioritas yang rendah untuk prioritas yang tinggi

C. LATIHAN

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

   Tabel 2-1. Tabel untuk soal 4 — 5

   Proses Burst Time Prioritas

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

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.

PERTEMUAN VI
MEMORI

A. TUJUAN

Mahasiswa dapat memahami memori management

B. TEORI

1. Latar Belakang

Memori merupakan inti dari sistem komputer modern. CPU mengambil instruksi dari memori sesuai yang ada pada program counter. Instruksi dapat berupa menempatkan/ menyimpan dari/ ke alamat dimemori, penambahan dan sebagainya. Dalam managemen memori ini, kita akan membahas bagaimana urutan alamat memori yang dibuat oleh program yang berjalan.

1) Pengikatan Alamat

Dalam banyak kasus, program akan berada dalam beberapa tahapan sebelum dieksekusi. Alamat-alamat yang dbutuhkan mungkin saja direpresentasikan dalam cara yang berbeda dalam tahapan-tahapan ini. Alamat dalam kode program masih berupa simbolik. Alamat ini

akan diikat oleh kompilator ke alamat memori yang dapat diakses (misalkan 14 byte, mulai dari sebuah modul). Kemudian *linkage editor* dan *loader*, akan mengikat alamat fisiknya (misalkan 17014). Setiap pengikatan akan memetakan suatu ruang alamat ke lainnya. Secara klasik, instruksi pengikatan dan data ke alamat memori dapat dilakukan dalam beberapa tahap: waktu *compile*: jika diketahui pada waktu compile, dimana proses ditempatkan di memori. Untuk kemudian kode absolutnya dapat di buat. Jika kemudian alamat awalnya berubah, maka harus di compile ulang waktu penempatan: Jika tidak diketahui dimana poses ditempatkan di memori, maka kompilator harus membuat kode yang dapat dialokasikan. Dalam kasus pengikatan akan ditunda sampai waktu penempatan. Jika alamat awalnya berubah, kita hanya perlu menempatkan ulang kode, untuk menyesuaikan dengan perubahan waktu eksekusi: Jika proses dapat dipindahkan dari suatu segmen memori ke lainnya selama dieksekusi. Pengikatan akan ditunda sampai *run-time*.

2) Ruang Alamat Fisik dan Logik

Alamat yang dibuat CPU akan merujuk ke sebuah alamat logik. Sedangkan alamat yang dilihat oleh memori adalah alamat yang dimasukkan ke register di memori, merujuk pada alamat fisik pada pengikatan alamat, waktu *compile* dan waktu penempatan mnghasilkan daerah dimana alamat logik dan alamat fisik sama. Sedangkan pada waktu eksekusi menghasilkan alamat fisik dan logik yang berbeda. Kita biasanya menyebut alamat logik dengan alamat virtual. Kumpulan alamat logik yang dibuat oleh program adalah ruag alamat logik. Kumpulan alamat fisik yang berkoresponddensi dengan alamat logic sibut ruang alamat fisik. Pemetaan dari virtual ke alamat fisik dialkukan oleh *Memory Management Unit* (MMU), yang merupakan sebuah perangkat keras.

Register utamanya disebut relocation-register. Nilai pada relocation register bertambah setiap alamat dibuat oleh proses pengguna, pada waktu yang sama alamat ini dikirim ke memori. Program pengguna tidak dapat langsung mengakses memori. Ketika ada program yang menunjuk ke alamat memori, kemudian mengoperasikannya, dan menaruh lagi di memori, akan di lokasikan awal oleh MMU, karena program pengguna hanya bernterkasi dengan alamat logik. Konsep untuk memisahkan ruang alamat logik dan ruang alamat fisik, adalah inti dari managemen memori yang baik.

3) Penempatan Dinamis

Telah kita ketahui seluruh proses dan data berada memori fisik ketika di eksekusi. Ukuran dari memori fisik terbatas. Untuk mendapatkan utilisasi ruang memori yang baik, kita melakukan penempatan dinamis. Dengan penempatan dinamis, sebuah rutin tidak akan ditempatkan sampai dipanggil. Semua rutin diletakan di disk, dalam format yang dapat di lokasikan ulang. Program utama di tempatkan di memori dan dieksekusi. Jika sebuah rutin memanggil rutin lainnya, maka akan di cek dulu apakah rutin yang dipanggil ada di dalam memori atau tidak, jika tidak ada maka linkage loader dipanggil untuk menempatkan rutin yang diinginkan ke memori dan memperbaharui tabel alamat program untuk menyesuaikan perubahan. Kemudian kontrol diletakan pada rutin yang baru ditempatkan. Keuntungan dari penempatan dinamis adalah rutin yang tidak digunakan tidak pernah ditempatkan.

Metode ini berguna untuk kode dalam jumlah banyak, ketika muncul kasus-kasus yang tidak lazim, seperti rutin yang salah. Dalam kode yag besar, walau pun ukuran kode besar, tapi yang ditempatkan dapat jauh lebih kecil. Penempatan Dinamis tidak didukung oleh sistem operasi. Ini adalah tanggung-jawab para pengguna untuk merancang program yang mengambil keuntungan dari metode ini. Sistem Operasi dapat membantu pembuat program dengan menyediakan libary rutin untuk mengimplementasi penempatan dinamis.

4) Perhubungan Dinamis dan Berbagi Library

Pada proses dengan banyak langkah, ditemukan juga perhubungan-perhubungan *library* yang dinamis. Beberapa sistem operasi hanya mendukung perhubungan yang dinamis, dimana sistem bahasa *library* diperlakukan seperti objek modul yang lain, dan disatukan oleh pemuat kedalam tampilan program biner. Konsep perhubungan dinamis, serupa dengan konsep penempatan dinamis. Penempatan lebih banyak ditunda selama waktu eksekusi, dari pada lama penundaan oleh perhubungan dinamis. Keistimewaan ini biasanya digunakan dalam *library* sistem, seperti *library* bahasa sub-rutin. Tanpa fasilitas ini, semua program dalam sebuah sistem, harus mempunyai kopi dari libary bahasa mereka (atau setidaknya referensi rutin oleh program) termasuk dalam tampilan yang dapat dieksekusi. Kebutuhan ini sangat boros baik untuk disk, mau pun memori utama. Dengan penempatan dinamis, sebuah potongan dimasukkan kedalam tampilan untuk setiap rujukan *library* subrutin. Potongan ini adalah sebuah bagian kecil dari kode yang menunjukan

bagaimana mealokasikan libary rutin di memori denga tepat, atau bagaimana menempatkan *library* jika rutin belum ada. Ketika potongan ini dieksekusi, dia akan memeriksa dan melihat apakah rutin yang dibutuhkan sudah ada di memory. Jika rutin yang dibutuhkan tidak ada di memori, program akan menempatkannya ke memori. Jika rutin yang dibutuhkan ada dimemori, maka potongan akan mengganti dirinya dengan alamat dari rutin, dan mengeksekusi rutin. Demikianlah, berikutnya ketika segmentasi kode dicapai, rutin *library* dieksekusi secara langsung, dengan begini tidak ada biaya untuk penghubungan dinamis. Dalam skema ini semua proses yang menggunakan sebuah *library* bahasa, mengeksekusi hanya satu dari kopi kode *library*. Fasilitas ini dapat diperluas menjadi pembaharuan *library* (seperti perbaikan bugs). Sebuah libary dapat ditempatkan lagi dengan versi yang lebih baru dan semua program yang merujuk ke *library* akan secara otomatis menggunakan versi yang baru. Tanpa penempatan dinamis, semua program akan akan membutuhkan penempatan kembali, untuk dapat mengakses *library* yang baru. Jadi semua program tidak secara sengaja mengeksekusi yang baru, perubahan versi *library*, informasi versi dapat dimasukkan kedalam memori, dan setiap program menggunakan informasi versi untuk memutuskan versi mana yang akan digunakan dari kopi *library*. Sedikit perubahan akan tetap meggunakan nomor versi yang sama, sedangkan perubhan besar akan menambah satu versi seblumnya. Karenanya program yang dikompile dengan versi yang baru akan dipengaruhi dengan perubahan yang terdapat di dalamnya. Program lain yang berhubungan sebelum *library* baru diinstal, akan terus menggunakan *library* lama. Sistem ini juga dikenal sebagai berbagi *library*.

5) Lapisan Atas

Karena proses dapat lebih besar daripada memori yang dialokasikan, kita gunakan lapisan atas. Idenya untuk menjaga agar di dalam memori berisi hanya instruksi dan data yang dibutuhkan dalam satuan waktu. Ketika instruksi lain dibutuhkan instruksi akan dimasukkan kedalam ruang yang ditempati sebelumnya oleh oleh instruksi yang tidak lagi dibutuhkan. Sebagai contoh, sebuah two-pass assembler. selama pass1 dibangun sebuah tabel simbol, kemudian selama pass2, akan membuat kode bahasa mesin. kita dapat mempartisi sebuah assembler menjadi kode pass1, kode pass2 dan simbol tabel. dan rutin biasa digunakan untuk kedua pass1 dan pass2. Untuk menempatkan semuanya sekaligus, kita akan membutuhkan 200K memori. Jika hanya 150K yang tersedia, kita tidak dapat menjalankan proses. Bagaimana pun perhatikan bahwa pass1 dan pass2 tidak harus berada di memori pada saat yang sama. Kita mendefinisikan dua lapisan atas. Lapisan atas A untuk pass1, tabel simbol dan rutin, lapisan atas 2 untuk simbol tabel, rutin, dan pass2. Kita menambahkan sebuah driver lapisan atas (10K) dan mulai dengan lapisan atas A di memori. Ketika selesai pass1, lompat ke driver, dan membaca lapisan atas B kedalam memori, meniban lapisan atas A dan mengirim kontrol ke pass2. Lapisan atas A butuh hanya 120K, dan B membutuhkan 150K memori. Kita sekarang dapat menjalankan assembler dalam 150K memori. Penempatan akan lebih cepat, karena lebih sedikit data yang ditransfer sebelum eksekusi dimulai. Jalan program akan lebih lambat, karena ekstra I/O dari kode lapisan atas B melalui kode lapisan atas A. Seperti dalam penempatan dinamis, lapisan atas tidak membutuhkan dukungan tertentu dari system operasi. Implementasi dapat dilakukan secara lengkap oleh user dengan berkas struktur yang sederhana, membasa dari berkas ke memori dan lompat dari memori tersebut, dan mengeksekusi instruksi yang baru dibaca. Sistem operasi hanya memperhatikan jika ada lebih banyak I/O dari biasanya. Di sisi lain programmer harus mendesain program dengan struktur lapisan atas yang layak. Tugas ini membutuhkan pengetahuan yang komplit tentang struktur dari program, kode dan struktur data. Pemakaian dari lapisan atas, dibatasi oleh mikrokomputer, dan sistem lain yang mempunyai batasan jumlah memori fisik, dan kurangnya dukungan perangkat keras, untuk teknik yang lebih maju. Teknik otomatis menjalankan program besar dalam dalam jumlah memori fisik yang terbatas, lebih diutamakan.

2. Penukaran (Swap)

Sebuah proses membutuhkan memori untuk dieksekusi. Sebuah proses dapat ditukar sementara keluar memori ke backing store (disk), dan kemudian dibawa masuk lagi ke memori untuk dieksekusi. Sebagai contoh, asumsi multiprogramming, dengan penjadualan algoritma CPU Round-Robin. Ketika kuantum habis, manager memori akan mulai menukar keluar proses yang selesai, dan memasukkan ke memori proses yang bebas. Sementara penjadualan CPU akan mangalokasikan waktu untuk proses lain di memori. Ketika tiap proses menghabiskan waktu kuantumnya, proses akan ditukar dengan proses lain. Idealnya memori manager, dapat menukar proses-proses cukup cepat, sehingga selalu ada proses dimemori, siap dieksekusi,

ketika penjadual CPU ingin menjadual ulang CPU. Besar kuantum juga harus cukup besar, sehingga jumlah perhitungan yang dilakukan antar pertukaran masuk akal. Variasi dari kebijakan *swapping* ini, digunakan untuk algoritma penjadualan berdasarkan prioritas. Jika proses yang lebih tinggi tiba, dan minta dilayani, memori manager dapat menukar keluar proses dengan prioritas yang lebih rendah, sehingga dapat memasukkan dan mengeksekusi proses dengan prioritas yang lebih tinggi. Ketika proses dengan prioritas lebih tinggi selesai, proses dengan prioritas yang lebih rendah, dapat ditukar masuk kembali, dan melanjutkan. Macam-macam pertukaran ini kadang disebut roll out, dan roll in. Normalnya, sebuah proses yang ditukar keluar, akan dimasukkan kembali ke tempat memori yang sama dengan yang digunakan sebelumnya. Batasan ini dibuat oleh method pengikat alamat. Jika pengikatan dilakukan saat assemble atau load time, maka proses tidak bisa dipindahkan ke lokasi yang berbeda. Jika menggunakan pengikatan waktu eksekusi, maka akan mungkin menukar proses kedalam tempat memori yang berbeda. Karena alamat fisik dihitung selama proses eksekusi. Pertukaran membutuhkan sebuah backing store. Backing store biasanya adalah sebuah disk yang cepat. Cukup besar untuk mengakomodasi semua kopi tampilan memori. Sistem memelihara *ready queue* terdiri dari semua proses yang mempunyai tampilan memori yang ada di backing store, atau di memori dan siap dijalankan. Ketika penjadual CPU memutuskan untuk mengeksekusi sebuah proses, dia akan memanggil dispatcher, yang mengecek dan melihat apakah proses berikutnya ada diantrian memori. Jika proses tidak ada, dan tidak ada ruang memori yang kosong, *dispatcher* menukar keluar sebuah proses dan memaasukan proses yang diinginkan. Kemudian memasukkan ulang register dengan normal, dan mentransfer pengendali ke proses yang diinginkan.

Konteks waktu pergantian pada sistem swapping, lumayan tinggi. Untuk efisiensi kegunaan CPU, kita ingin waktu eksekusi untuk tiap proses lebih lama dari waktu pertukaran. Karenanya digunakan CPU penjadualan roun-robin, dimana kuantumnya harus lebih besar dari waktu pertukaran. Perhatikan bahwa bagian terbesar dari waktu pertukaran, adalah waktu pengiriman. Total waktu pengiriman langsung didapat dari jumlah pertukaran memori. Proses dengan kebutuhan memori dinamis, akan membutuhkan *system call* (meminta dan melepaskan memori), untuk memberi tahu sistem operasi tentang perubahan kebutuhan memori. Ada beberapa keterbatasan *swapping*. Jika kita ingin menukar sebuah proses kita harus yakin bahwa proses sepenuhnya diam. Konsentrasi lebih jauh, jika ada penundaan I/O. Sebuah proses mungkin menunggu I/O, ketika kita ingin menukar proses itu untuk mengosongkan memori. Jika I/O secara asinkronus, mengakses memori dari I/O buffer, maka proses tidak bisa ditukar. Misalkan I/O operation berada di antrian, karena *device* sedang sibuk. Maka bila kita menukar keluar proses P1 dan memasukkan P2, mungkin saja operasi I/O akan berusaha masuk ke memori yang sekarang milik P2. Dua solusi utama masalah ini adalah

- Jangan pernah menukar proses yang sedang menunggu I/O.
- Untuk mengeksekusi operasi I/O hanya pada *buffer* sistem operasi.

Secara umum, ruang pertukaran dialokasikan sebagai potongan disk, terpisah dari sistem berkas, sehingga bisa digunakan secepat mungkin. Belakangan pertukaran standar pertukaran digunakan dibeberapa sistem. Ini membutuhkan terlalu banyak waktu untuk menukar dari pada untuk mengeksekusi untuk solusi managemen memori yang masuk akal. Modifikasi *swapping* digunakan dibanyak versi di UNIX. Pertukaran awalnya tidak bisa, tapi akan mulai bila banyak proses yang jalan dan menggunakan batas jumlah memori.

3. Alokasi Memori Yang Berdampingan

Memori biasanya dibagi menjadi dua bagian, yakni

- Sistem Operasi (*Operating System*).
- Proses Pengguna (*User Processes*).

Sistem Operasi dapat dialokasikan pada memori bagian bawah (*low memory*) mau pun memori bagian atas (*high memory*). Hal ini tergantung pada letak vektor interupsi (*interrupt vector*) pada memori tersebut. Jika vektor interupsi lebih sering berada pada memori bawah, maka sistem operasi juga biasanya diletakkan pada memori bawah. Memori memerlukan suatu perlindungan yang disebut dengan istilah *memory protection* yakni perlindungan memori terhadap:

- Sistem operasi dari proses pengguna;
- Proses pengguna yang satu dari proses pengguna lainnya.

Perlindungan memori tersebut dapat diakomadasikan menggunakan suatu register pengalokasian kembali (*relocation register*) dengan suatu register batasan (*limit register*). Register batasan berisi jarak dari alamat logik (*logical address*) sementara register pengalokasian kembali berisi nilai dari alamat fisik (*physical address*) yang terkecil. Dengan

adanya register pengalokasian kembali dan register batasan ini, mengakibatkan suatu alamat logik harus lebih kecil dari register batas an dan memori akan memetakan (*mapping*) alamat logik secara dinamik dengan menambah nilai dalam register pengalokasian kembali.

Alokasi Kembali.
register batasan register pengalokasian kembali
| |
|Prosesor|-->(alamat logik)-->|<|->(ya)-->|+|-->(alamat fisik)-->|MAR|
|(no)

perangkap: kesalahan pengalamatan
Sebagaimana telah diketahui, bahwa pengatur jadual prosesor (*CPU scheduler*) bertugas mengatur dan menyusun jadual dalam proses eksekusi proses yang ada. Dalam tugasnya, pengatur jadual prosesor akan memilih suatu proses yang telah menunggu di antrian proses (*process queue*) untuk dieksekusi. Saat memilih satu proses dari proses yang ada di antrian tersebut, *dispatcher* akan mengambil register pengalokasian kembali dan register batasan dengan nilai yang benar sebagai bagian dari skalar alih konteks. Oleh karena setiap alamat yang ditentukan oleh prosesor diperiksa berlawanan dengan register-register ini, kita dapat melindungi sistem operasi dari program pengguna lainnya dan data dari pemodifikasian oleh proses yang sedang berjalan.
Metode yang paling sederhana dalam mengalokasikan memori ke proses-proses adalah dengan cara membagi memori menjadi partisi tertentu. Secara garis besar, ada dua metode khusus yang digunakan dalam membagi-bagi lokasi memori:

a. Alokasi partisi tetap (*Fixed Partition Allocation*) yaitu metode membagi memori menjadi partisi yang telah berukuran tetap.
   Kriteria-kriteria utama dalam metode ini antara lain:
   - Alokasi memori: proses p membutuhkan k unit memori.
   - Kebijakan alokasi yaitu "sesuai yang terbaik": memilih partisi terkecil yang cukup besar (memiliki ukuran = k).
   - Fragmentasi dalam (*Internal fragmentation*) yaitu bagian dari partisi tidak digunakan.
   - Biasanya digunakan pada sistem operasi awal (*batch*).
   - Metode ini cukup baik karena dia dapat menentukan ruang proses; sementara ruang proses harus konstan. Jadi sangat sesuai dengan parti si berukuran tetap yang dihasilkan metode ini.
   - Setiap partisi dapat berisi tepat satu proses sehingga derajat dari pemrograman banyak *multiprogramming* dibatasi oleh jumlah partisi yang ada.
   - Ketika suatu partisi bebas, satu proses dipilih dari masukan antrian dan dipindahkan ke partisi tersebut.
   - Setelah proses berakhir (selesai), partisi tersebut akan tersedia (*available*) untuk proses lain.
b. Alokasi partisi variabel (*Variable Partition Allocation*) yaitu metode dimana sistem operasi menyimpan suatu tabel yang menunjukkan partisi memori yang tersedia dan yang terisi dalam bentuk s. Alokasi memori: proses p membutuhkan k unit memori.
   Kebijakan alokasi:
   - Sesuai yang terbaik: memilih lubang (*hole*) terkecil yang cukup besar untuk keperluan proses sehingga menghasilkan sisa lubang terkecil.
   - Sesuai yang terburuk: memilih lubang terbesar sehingga menghasil kan sisa lubang.
   - Sesuai yang pertama: memilih lubang pertama yang cukup besar untuk keperluan proses

Fragmentasi luar (*External Fragmentation*) yakni proses mengambil ruang, sebagian digunakan, sebagian tidak digunakan. Memori, yang tersedia untuk semua pengguna, dianggap sebagai suatu blok besar memori yang disebut dengan lubang. Pada suatu saat memori memiliki suatu daftar set ubang (*free list holes*). Saat suatu proses memerlukan memori, maka kita mencari suatu lubang yang cukup besar untuk kebutuhan proses tersebut. Jika ditemukan, kita mengalokasikan lubang tersebut ke proses tersebut sesuai dengan kebutuhan dan sisanya disimpan untuk dapat digunakan proses lain.
Suatu proses yang telah dialokasikan memori akan dimasukkan ke memori dan selanjutnya dia akan bersaing dalam mendapatkan prosesor untuk pengeksekusiannya. Jika suatu proses tersebut telah selesai, maka dia akan melepaskan kembali semua memori yang digunakan dan sistem operasi dapat mengalokasikannya lagi untuk proses lainnya yang sedang menunggu di antrian masukan. Apabila memori sudah tidak mencukupi lagi untuk kebutuhan proses, sistem

operasi akan menunggu sampai ada lubang yang cukup untuk dialokasikan ke suatu proses dalam antrian masukan. Jika suatu lubang terlalu besar, maka sistem operasi akan membagi lubang tersebut menjadi dua bagian, dimana satu bagian untuk dialokasikan ke proses tersebut dan satu lagi dikembalikan ke set lubang lainnya. Setelah proses tersebut selesai dan melepaskan memori yang digunakannya, memori tersebut akan digabungkan lagi ke set lubang.

Fragmentasi luar mempunyai kriteria antara lain:

- Ruang memori yang kosong dibagi menjadi partisi kecil.
- Ada cukup ruang memori untuk memenuhi suatu permintaan, tetapi memori itu tidak lagi berhubungan antara satu bagian dengan bagian lain (*contiguous*) karena telah dibagi-bagi.
- Kasus terburuk (*Worst case*): akan ada satu blok ruang memori yang kosong yang terbuang antara setiap dua proses.
- Aturan 50 persen: dialokasikan N blok, maka akan ada 0.5N blok yang hilang akibat fragmentasi sehingga itu berarti 1/3 memori akan tidak berguna.

Perbandingan kompleksitas waktu dan ruang tiga kebijakan alokasi memori.

Alokasi Kembali.
Waktu Ruang
==========
Sesuai yang Terbaik Buruk Baik
Sesuai yang Terburuk Buruk Buruk
Sesuai yang Pertama Baik Baik

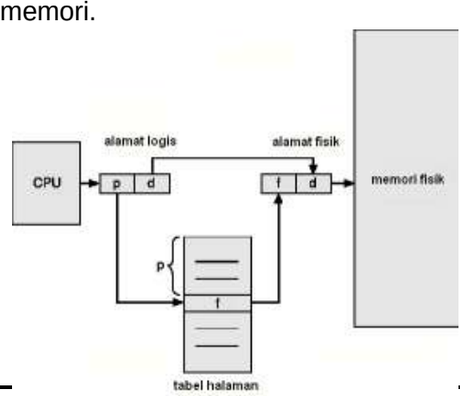
Sesuai yang pertama merupakan kebijakan alokasi memori paling baik secara praktis.

4. Pemberian Halaman

Solusi lain yang mungkin untuk permasalahan pemecahan luar adalah dengan membuat ruang alamat fisik dari sebuah proses menjadi tidak bersebelahan, jadi membolehkan sebuah proses untuk dialokasikan memori fisik bilamana nantinya tersedia. Satu cara mengimplementasikan solusi ini adalah melalui penggunaan dari skema pemberian halaman. Pemberian halaman mencegah masalah penting dari mengepaskan the ukuran bongkahan memori yang bervariasi ke dalam penyimpanan cadangan, yang mana diderita oleh kebanyakan dari skema managemen memori sebelumnya. Ketika beberapa pecahan kode dari data yang tersisa di memori utama perlu untuk di tukar keluar, harus ditemukan ruang di penyimpanan cadangan. Masalah pemecahan didiskusikan dengan kaitan bahwa meori utama juga lazim dengan penyimpanan cadangan, kecuali bahwa pengaksesanny lebih lambat, jadi kerapatan adalah tidak mungkin. Karena keuntungannya pada metode-metode sebelumnya, pemberian halaman dalam berbagai bentuk biasanya digunakan pada banyak sistem operasi.

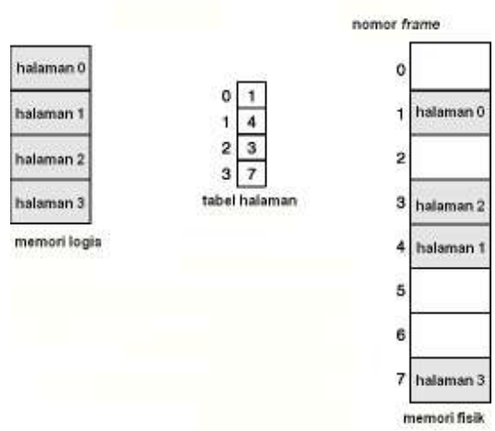
1) Metode Dasar

Memori fisik dipecah menjadi blok-blok berukuran tetap disebut sebagai *frame*. Memori logis juga dipecah menjadi blok-blok dengan ukuran yang sama disebut sebagai halaman. Ketika proses akan dieksekusi, halamannya akan diisi ke dalam *frames* memori mana saja yang tersedia dari penyimpanan cadangan. Penyimpanan cadangan dibagi-bagi menjadi blok-blok berukuran tetap yang sama besarnya dengan *frames* dimemori. Dukungan perangkat keras untuk pemberian halaman diilustrasikan pada gambar Gambar 4-3. Setiap alamat yang dihasilkan oleh CPU dibagi-bagi menjadi 2 bagian: sebuah nomor halaman (p) dan sebuah *offset* halaman (d). Nomor halaman digunakan sebagai indeks untuk tabel halaman. Tabel halaman mengandung basis alamat dari tiap tiap halaman di memori fisik. Basis ini dikombinasikan dengan *offset* halaman untuk menentukan alamat memori fisik yang dikirim ke unit memori.



Gambar 6.1. Perangkat Keras Pemberian Halaman.

Ukuran halaman (seperti halnya ukuran *frame*) didefinisikan oleh perangkat keras. Khasnya ukuran dari sebuah halaman adalah pangkat 2 yang berkisar antara 512 byte dan 8192 byte per halamannya, tergantung dari arsitektur komputernya. Penentuan pangkat 2 sebagai ukuran halaman akan memudahkan penterjemahan dari memori logis ke nomor halaman dan *offset* halaman. Jika ukuran ruang dari memori logis adalah 2 pangkat m, dan ukuran sebuah halaman adalah 2 pangkat n unit pengalamatan (byte atau word), maka pangkat tinggi m-n bit dari alamat logis manandakan *offset* dari halaman. Jadi, alamat logisnya adalah: dimana p merupakan index ke tabel halaman dan d adalah pemindahan dalam halaman. Untuk konkritnya, walau kecil sekali, contoh, lihat memori Gambar 4-4. Menggunakan ukuran halaman 4 byte dan memori fisik 32 byte (8 halaman), kami menunjukan bagaimana pandangan pengguna terhadap memori dapat dipetakan kedalam memori fisik. Alamat logis 0 adalah halaman 0, *offset* 0. Pemberian index menjadi tabel halaman, kita dapati bahwa halaman 0 berada pada frame 5. Jadi, alamat logis 0 memetakan ke alamat fisik 20 (=(5x4)+0). Alamat logis 3 (page 0, *offset* 3) memetakan ke alamat fisik 23 (=(5x4)+3). Alamat logis 4 adalah halaman 1, *offset* ; menurut tabel halaman, halaman 1 dipetakan ke *frame* 6. Jadi, alamat logis 4 memetakan ke alamat fisik 24 (=(6x4)+0). Alamat logis 13 memetakan ke alamat fisik 9.



Gambar 6.2. Model pemberian halaman dari memori fisik dan logis.

Pembentukan pemberian halaman itu sendiri adalah suatu bentuk dari penampungan dinamis. Setiap alamat logis oleh perangkat keras untuk pemberian halaman dibatasi ke beberapa alamat fisik. Pembacayang setia akan menyadari bahwa pemberian halaman sama halnya untuk menggunakan sebuah table dari basis register, satu untuk setiap *frame* di memori. Ketika kita menggunakan skema pemberian halaman, kita tidak memiliki pemecah-mecahan luar: sembarang *frame* kosong dapat dialokasikan ke proses yang membutuhkan. Bagaimana pun juga kita mungkin mempunyai beberapa pemecahan di dalam. Mengingat bahwa *frame* - *frame* dialokasikan sebagai unit. Jika kebutuhan memori dari sebuah proses tidak menurun pada batas halaman, *frame* terakhir yang dialokasikan mungkin tidak sampai penuh. Untuk contoh, jika halamannya 2048 byte, proses 72.766 byte akan membutuhkan 35 halaman tambah 1086 byte. Alokasinya menjadi 36 frame, menghasilkan fragmentasi internal dari 2048 - 1086 = 962 byte. Pada kasus terburuknya, proses akan membutuhkan n halaman tambah satu byte. Sehingga dialokasikan n+1 *frame*, menghasilkan fragmentasi internal dari hampir semua *frame*. Jika ukuran proses tidak bergantung dari ukuran halaman, kita mengharapkan fragmentasi internal hingga rata-rata setengah halaman per prosesnya. Pertimbangan ini memberi kesan bahwa ukuran halaman yang kecil sangat diperlukan sekali. Bagaimana pun juga, ada sedikit pemborosan dilibatkan dalam masukan tabel halaman, dan pemborosan ini dikurangi dengan ukuran halaman meningkat. Juga disk I/O lebih efisien ketika jumlah data yang dipindahkan lebih besar. Umumnya, ukuran halaman bertambah seiring bertambahnya waktu seperti halnya proses, himpunan data, dan memori utama telah menjadi besar. Hari ini, halaman umumnya berukuran 2 atau 4 kilobyte. Ketika proses tiba untuk dieksekusi, ukurannya yang diungkapkan di halaman itu diperiksa. Setiap pengguna membutuhkan satu *frame*. Jadi, jika proses membutuhkan n halaman, maka pasti ada n *frame* yang

tersedia di memori. Jika ada n *frame* yang tersedia, maka mereka dialokasikan di proses ini. Halamn pertama dari proses diisi ke salah satu *frame* yang sudah teralokasi, dan nomor *frame*nya diletakkan di tabel halaman untuk proses ini. Halaman berikutnya diisikan ke *frame* yang lain, dan nomor *frame*nya diletakkan ke tabel halaman, dan begitu seterusnya (gambar Gambar 6.2)

Aspek penting dari pemberian halaman adalah pemisahan yang jelas antara pandangan pengguna tentang memori dan fisik memori sesungguhnya. Program pengguna melhiat memori sebagai satu ruang berdekatan yang tunggal, hanya mengandung satu program itu. Faktanya, program pengguna terpencar-pencar didalam memori fisik, yang juga menyimpan program lain. Perbedaan antara pandangan pengguna terhadap memori dan fisik memori sesungguhnya disetarakan oleh perangkat keras penterjemah alamat. Alamat logis diterjemahkan ke alamat fisik. Pemetaan ini tertutup bagi pengguna dan dikendalikan oleh sistem operasi. Perhatikan bahwa proses pengguna dalam definisi tidak dapat mengakses memori yang bukan haknya. Tidak ada pengalamatan memori di luar tabel halamannya, dan tabelnya hanya melingkupi halaman yang proses itu miliki. Karena sistem operasi mengatur memori fisik, maka harus waspada dari rincian alokasi memori fisik: frame mana yang dialokasikan, frame mana yang tersedia, berapa banyak total frame yang ada, dan masih banyak lagi. Informasi ini umumnya disimpan di struktur data yang disebut sebagai table frame. Tabel frame punya satu masukan untuk setiap fisik halaman frame, menandakan apakah yang terakhir teralokasi ataukah tidak, jika teralokasi maka kepada halaman mana dari proses mana. Tambahan lagi sistem operasi harus waspada bahwa proses-proses pengguna beroperasi di ruang pengguna, dan semua logis alamat harus dipetakan untuk menghasilkan alamat fisik. Jika pengguna melakukan pemanggilan sistem (contohnya melakukan I/O) dan mendukung alamat sebagai parameter (contohnya penyangga), alamatnya harus dipetakan untuk menghasilkan alamat fisik yang benar. Sistem operasi mengatur salinan tabel halaman untuk tiap-tiap proses, seperti halnya ia mengatur salinan dari *counter* instruksi dan isi register. Salinan ini digunakan untuk menterjemahkan alamat fisik ke alamat logis kapan pun sistem operasi ingin memetakan alamat logis ke alamat fisik secara manual. Ia juga digunakan oleh *dispatcher* CPU untuk mendefinisikan tabel halaman perangkat keras ketika proses dialokasikan ke CPU. Oleh karena itu pemberian halaman meningkatkan waktu alih konteks.

2) Struktur Tabel Halaman

Setiap sistem operasi mempunyai metodenya sendiri untuk menyimpan tabel-tabel halaman. Sebagian besar mengalokasikan tabel halaman untuk setiap proses. Penunjuk ke tabel halaman disimpan dengan nilai register yang lain (seperti *counter* instruksi) di blok kontrol proses. Ketika pelaksana *dispatcher* mengatakan untuk memulai proses, maka harus disimpan kembali register-register pengguna dan mendefinisikan nilai tabel halaman perangkat keras yang benar dari tempat penyimpanan tabel halaman pengguna.

- **Dukungan Perangkat Keras**

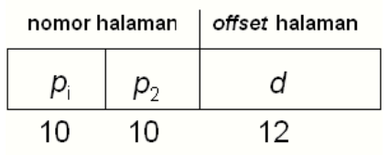
  Implementasi perangkat keras dari tabel halaman dapat dilakukan dengan berbagai cara. Kasus sederhananya, tabel halaman diimplementasikan sebagai sebuah himpunan dari register resmi. Register ini harus yang bekecepatan tinggi agar penerjemahan alamat pemberian halaman efisien Setiap pengaksesan ke memori harus melalui peta pemberian halaman, jadi ke-efisienan adalah pertimbangan utama. Pelaksana (*dispatcher*) CPU mengisi kembali register-register ini, seperti halnya ia mengisi kembali register yang lain. Instruksi untuk mengisi atau mengubah register tabel halaman adalah, tentu saja diberi hak istimewa, sehingga hanya sistem operasi yang dapat mengubah peta memori. DEC PDP-11 adalah contoh arsitektur yang demikian. Alamatnya terdiri dari 16-bit, dan ukuran halamannya 8K. Jadi tabel halaman terdiri dari 8 masukan yang disimpan di register-register cepat.

- **Pemeliharaan**

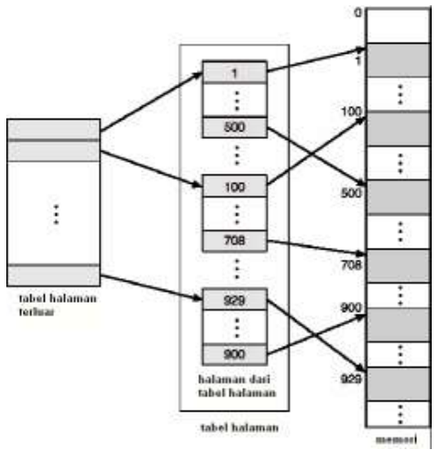
  Proteksi memori dari suatu lingkungan berhalaman diselesaikan dengan bit-bit proteksi yang diasosiasikan dengan tiap-tiap *frame*. Normalnya, bit-bit ini disimpan di tabel halaman. Satu bit dapat menentukan halaman yang akan dibaca tulis atau baca saja. Setiap referensi ke memori menggunakan tabel halaman untuk menemukan nomor *frame* yang benar. Pada saat yang sama alamat fisik diakses, bit-bit proteksi dapat dicek untuk menguji tidak ada penulisan yang sedang dilakukan terhadap halaman yang boleh dibaca saja. Suatu usaha untuk menulis ke halaman yang boleh dibaca saja akan menyebabkan perangkat keras menangkapnya ke sistem operasi.

3) Pemberian Halaman Secara Multilevel

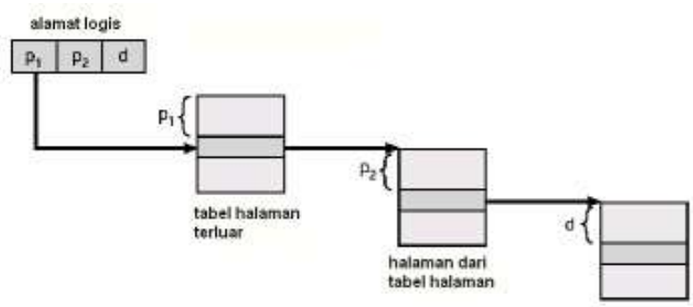
Banyak sistem komputer moderen mendukung ruang alamat logis yang sangat luas (2 pangkat 32 sampai 2 pangkat 64). Pada lingkungan seperti itu tabel halamanya sendiri menjadi sangat-sangat besat sekali. Untuk contoh, misalkan suatu sistem dengan ruang alamat logis 32-bit. Jika ukuran halaman di system seperti itu adalah 4K byte (2 pangkat 12), maka tabel halaman mungkin berisi sampai 1 juta masukan ((2^32)/(2^12)). Karena masing-masing masukan terdiri atas 4 byte, tiap-tiap proses mungkin perlu ruang alamat fisik sampai 4 megabyte hanya untuk tabel halamannya saja. Jelasnya, kita tidak akan mau mengalokasi tabel halaman secara berdekatan di dalam memori. Satu solusi sederhananya adalah dengan membagi tabel halaman menjadi potongan-potongan yang lebih kecil lagi. Ada beberapa cara yang berbeda untuk menyelesaikan ini.



Gambar 6.3. Alamat logis.

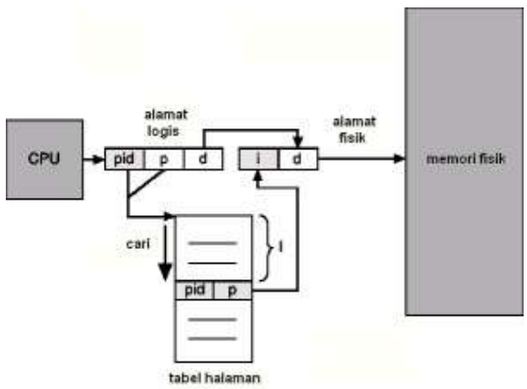


Gambar 6.4. Skema Tabel Halaman Dua Tingkat.



Gambar 6.5 . Penterjemahan alamat untuk arsitektur pemberian halaman dua tingkat 32-bit logis.

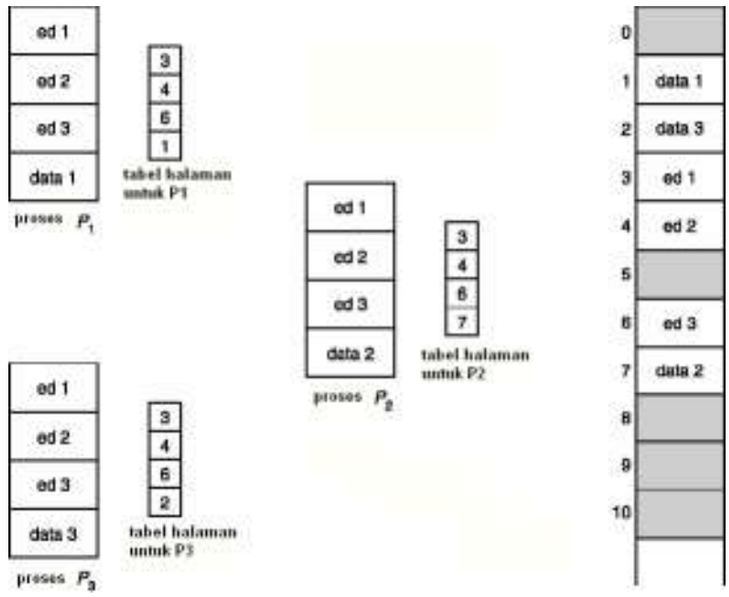
- **Tabel Halaman yang Dibalik**
  Biasanya, setiap proses mempunyai tabel halaman yang diasosiasikan dengannya. Tabel halaman hanya punya satu masukan untuk setiap halaman proses tersebut sedang gunakan (atau satu slot untuk setiap alamat maya, tanpa meperhatikan validitas terakhir). Semenjak halaman referensi proses melalui alamat maya halaman, maka representasi tabel ini adalah alami. Sistem operasi harus menterjemahkan referensi ini ke alamat memori fisik. Semenjak tabel diurutkan berdasarkan alamat maya, sistem operasi dapat menghitung dimana pada tabel yang diasosiasikan dengan masukan alamat fisik, dan untuk menggunakan nilai tersebut secara langsung. Satu kekurangan dari skema ini adalah masing masing halaman mungkin mengandung jutaan masukan. Tabel ini mungkin memakan memori fisik dalam jumlah yang besar, yang mana dibutukan untuk tetap menjaga bagaimana memori fisik lain sedang digunakan.



Gambar 6.6. Tabel halaman yang dibalik. Sumber: . . .

- **Berbagi Halaman**

Keuntungan lain dari pemberian halaman adalah kemungkinannya untuk berbagi kode yang sama. Pertimbangan ini terutama sekali penting pada lingkungan yang berbagi waktu. Pertimbangkan sebuah sistem yang mendukung 40 pengguna, yang masing-masing menjalankan aplikasi pengedit teks. Jika editor teks tadi terdiri atas 150K kode dan 50K ruang data, kita akan membutuhkan 8000K untuk mendukung 40 pengguna. Jika kodenya dimasukan ulang, bagaimana pun juga dapat dibagi bagi, seperti pada gambar Gambar 6.7. Disini kita lihat bahwa tiga halaman editor (masing-masing berukuran 50K; halaman ukuran besar digunakan untuk menyederhanakan gambar) sedang dibagi-bagi diantara tiga proses. Masing-masing proses mempunyai halaman datanya sendiri.



Gambar 6.7. Berbagi kode pada lingkungan berhalaman. Sumber: . . .

Kode pemasukan kembali (juga disebut kode murni) adalah kode yang bukan *self-modifying*. Jika kodenya dimasukan kembali, maka ia tidak akan berubah selama eksekusi. Jadi, dua atau lebih proses dapat mengeksekusi kode yang sama pada saat bersamaan. Tiap-tiap proses mempunyai register salinannya sendiri dan penyimpan data untuk menahan data bagi proses bereksekusi. Data untuk dua proses yang berbeda akan bervariasi pada tiap-tiap proses. Hanya satu salinan editor yang dibutuhkan untuk menyimpan di memori fisik. Setiap tabel halaman pengguna memetakan ke salinan fisik yang sama dari editor, tapi halaman-halaman data dipetakan ke *frame* yang berbeda. Jadi, untuk mendukung 40 pengguna, kita hanya membutuhkan satu salinannya editor (150K), ditambah 40 salinan 50K dari ruang data per pengguna. Total ruang yang dibutuhkan sekarang 2150K, daripada 8000K, penghematan yang signifikan. Program-program lain pun juga dapat dibagi-bagi: *compiler*, *system window* database system, dan masih banyak lagi. Agar dapat dibagi-bagi, kodenya harus dimasukan kembali. System yang menggunakan tabel halaman yang dibalik mempunyai kesulitan dalam mengimplementasikan berbagi memori. Berbagi memori biasanya diimplementasikan sebagai dua alamat maya yang dipetakan ke satu alamat fisik. Metode standar ini tidak dapat digunakan, bagaimana pun juga selama di situ hanya ada satu masukan halaman maya untuk setiap halaman fisik, jadi satu alamat fisik tidak dapat mempunyai dua atau lebih alamat maya yang dibagi-bagi.

5. Segmentasi

Salah satu aspek penting dari managemen memori yang tidak dapat dihindari dari pemberian halaman adalah pemisahan cara pandang pengguna dengan tentang bagaimana memori dipetakan dengan keadaan yang sebenarnya. Pada kenyataannya pemetaan tersebut memperbolehkan pemisahan antara memori logis dan memori fisik.

1) Metode Dasar

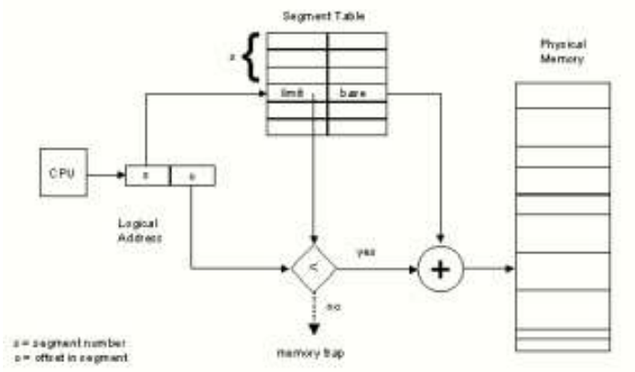
Bagaimanakah cara pandang pengguna tentang bagaimana memori dipetakan? Apakah

pengguna menganggap bahwa memori dianggap sebagai sebuah kumpulan dari byte-byte, yang mana sebagian berisi instruksi dan sebagian lagi merupakan data, atau apakah ada cara pandang lain yang lebih layak digunakan? Ternyata programmer dari sistem tidak menganggap bahwa memori adalah sekumpulan *byte-byte* yang linear. Akan tetapi, mereka lebih senang dengan menganggap bahwa memori adalah sebagai kumpulan dari segmen-segmen yang berukuran beragam tanpa adanya pengurutan penempatan dalam memori fisik. Ketika kita menulis suatu program, kita akan menganggapnya sebagai sebuah program dengan sekumpulan dari subrutin, prosedur, fungsi, atau variabel. mungkin juga terdapat berbagai macam struktur data seperti: tabel, *array* , *stack* , variabel, dsb. Tiap-tiap modul atau elemen-elemen dari data ini dapat di-referensikan dengan suatu nama, tanpa perlu mengetahui dimana alamat sebenarnya elemen-elemen tersebut disimpan di memori. dan kita juga tidak perlu mengetahui apakah terdapat urutan penempatan dari program yang kita buat. Pada kenyataannya, elemen-elemen yang terdapat pada sebuah segmen dapat ditentukan lokasinya dengan menambahkan *offset* dari awal alamat segmen tersebut. Segmentasi adalah sebuah bagian dari managemen memori yang mengatur pengalamatan dari memori yang terdiri dari segmen-segmen. *logical address space* adalah kumpulan dari segmen-segmen yang mana tiap-tiap segmen mempunyai nama dan panjang. alamat tersebut menunjukkan alamat dari segmen tersebut dan *offset*-nya didalam segmen-segmen tersebut. pengguna kemudian menentukan pengalamatan dari setiap segmen menjadi dua bentuk, nama segmen dan offset dari segmen tersebut (Hal ini berbeda dengan pemberian halaman, dimana pengguna hanya menentukan satu buah alamat, dimana pembagian alamat menjadi dua dilakukan oleh perangkat keras, semua ini tidak dapat dilihat oleh user). Untuk kemudahan pengimplementasian, segmen-segmen diberi nomor dan direferensikan dengan menggunakan penomoran tersebut, daripada dengan menggunakan nama. maka, *logical address space* terdiri dari dua *tuple* yaitu: (nomor-segmen, offset) Pada umumnya, program dari pengguna akan dikompilasi, dan kompilator tersebut akan membuat segmen-segmen tersebut secara otomatis. Jika mengambil contoh kompilator dari Pascal, maka kemungkinan kompilator tersebut akan membuat beberapa segmen yang terpisah untuk

- Variabel Global;
- Prosedur dari pemanggilan stack, untuk menyimpan parameter dan pengembalian alamat;
- Porsi dari kode untuk setiap prosedur atau fungsi; dan
- Variabel lokal dari setiap prosedur dan fungsi.

2) Perangkat Keras

Walau pun pengguna sekarang dapat mengacu ke suatu objek yang berada di dalam program dengan menggunakan pengalamatan secara dua dimensi, akan tetapi, pada kenyataannya tetap saja pada memori fisik akan dipetakan ke dalam pengalamatan satu dimensi yang terdiri dari urutan dari *byte-byte*. Maka, kita harus mendefinisikan suatu implementasi untuk memetakan pengalamatan dua dimensi yang dilakukan oleh pengguna ke dalam pengalamatan satu dimensi yang terdapat di memori fisik. Pemetaan ini dapat di lakukan dengan menggunakan tabel segmen. Setiap anggota dari tabel segmen mempunyai basis dan limit yang akan menentukan letak dari segmen tersebut di dalam memori.



Gambar 6.8. Alamat Logis

Kegunaan tabel segmen dapat dilihat pada gambar Gambar 4-10 alamat logis terdiri dari dua bagian: bagian segmen, s, dan bagian offsetnya, d. Nomor dari segmen tersebut akan digunakan sebagai index di dalam tabel segmen. Offset dari d di alamat logis sebaiknya

tidak melebihi limit dari alamat segmen, jika ini terjadi, maka sistem operasi sebaiknya dapat mengatasi hal ini, dengan melakukan *trap* .

3) Pemeliharaan dan Pembagian

Dengan dilakukannya pengelompokan antara segmen-segmen yang sama, maka pemeliharaan dari segmen tersebut dapat menjadi lebih mudah, walau pun didalam segmen tersebut sebagian berisi instruksi dan sebagian lagi berisi data. Dalam arsitektur modern, instruksi-instruksi yang digunakan tidak dapat diubah tanpa campur tangan pengguna, oleh karena itu, segmen yang berisi instruksi dapat diberi label *read only* atau hanya dapat dijalankan saja. Perangkat keras yang bertugas untuk melakukan pemetaan ke memori fisik akan melakukan pemeriksaan terhadap bit proteksi yang terdapat pada segmen, sehingga pengaksesan memori secara ilegal dapat dihindari, seperti suatu usaha untuk menulis ke area yang berstatus tidak boleh dimodifikasi. Keuntungan lain dari segmentasi adalah menyangkut masalah pembagian penggunaan kode atau data. Setiap proses mempunyai tabel segmennya sendiri, dimana ini akan digunakan oleh *dispatcher* untuk menentukan tabel segmen dari perangkat keras yang mana akan digunakan ketika proses yang bersangkutan di eksekusi oleh CPU. Segmen akan berbagi ketika anggota dari elemen table segmen yang berasal dari dua proses yang berbeda menunjuk ke lokasi fisik yang sama. Pembagian tersebut terjadi pada level segmen, maka, informasi apa pun dapat dibagi jika didefinisikan pada level segmen. Bahkan beberapa segmen pun dapat berbagi, sehingga sebuah program yang terdiri dari beberapa segmen pun dapat saling berbagi pakai.

4) Fragmentasi

Penjadwalan jangka-panjang harus mencari dan mengalokasikan memori untuk semua segmen dari program pengguna. Situasi ini mirip dengan pemberian halaman kecuali bahwa segmen-segmen ini mempunyai panjang yang variabel; sedangkan pada halaman, semua mempunyai ukuran yang sama. maka, masalah yang dihadapi adalah pengalamatan memori secara dinamis, hal ini biasanya dapat diselesaikan dengan menggunakan algoritma *best-fit* atau algoritma *first-fit*. Segmentasi dapat menyebabkan terjadi fragmentasi eksternal, ini terjadi ketika semua blok memori yang dapat dapat dialokasikan terlalu sedikit untuk mengakomodasi sebuah segmen. Dalam kasus ini, proses hanya harus menunggu sampai terdapat cukup tempat untuk menyimpan segmen tersebut di memori, atau, melakukan suatu pemampatan dapat digunakan untuk membuat ruang kosong dalam memori menjadi lebih besar. Karena segmentasi pada dasarnya adalah algoritma penempatan secara dinamis, maka kita dapat melakukan pemampatan memori kapan saja kita mau. Jika *CPU Scheduler* harus menunggu untuk satu proses, karena masalah pengalokasian memori, ini mungkin akan dilewati untuk mencari proses yang berprioritas lebih kecil untuk dieksekusi lebih dulu untuk membebaskan ruang kosong dalam memori. Seberapa seriuskah masalah fragmentasi eksternal dalam segmentasi? Jawaban dari pertanyaan ini tergantung kepada besarnya rata-rata segmen yang tersimpan didalam memori. Jika ukuran rata-rata dari segmen menggunakan sedikit tempat di memori, maka fragmentasi eksternal yang dilakukan juga akan sedikit terjadi.

6. Segmentasi Dengan Pemberian Halaman

1) Pengertian

Metode segmentasi dan *paging* yang telah dijelaskan pada sub bab sebelumnya masing-masing memiliki keuntungan dan kerugian. Selain kedua metode itu ada metode pengaturan memori lain yang berusaha menggabungkan metode segmentasi dan paging. Metode ini disebut dengan segmentation with paging. Dengan metode ini jika ukuran segmen melebihi ukuran memori utama maka segmen tersebut dibagi-bagi jadi ukuran-ukuran halaman yang sama ==> *paging* .

2) Kelebihan Segmentasi dengan Pemberian Halaman

Sesuai dengan definisinya yang merupakan gabungan dari segmentasi dan *paging*, maka metode ini memiliki keunggulan yang dimiliki baik oleh metode segmentasi mau pun yang dimiliki oleh *paging*. Tetapi selain itu segmentasi dengan pemberian halaman ini juga memiliki beberapa kelebihan yang tidak dimiliki oleh kedua metode tersebut. Kelebihan-kelebihan segmentasi dengan pemberian halaman antara lain :

- Dapat dibagi.
- Proteksi.
- Tidak ada fragmentasi luar.
- Alokasi yang cepat.

- Banyak variasinya.
- Biaya kinerja yang kecil.

3) Perbedaan Segmentasi dan Paging

Ada beberapa perbedaan antara Segmentasi dan *Paging* diantaranya adalah:

- Segmentasi melibatkan programer (programer perlu tahu teknik yang digunakan), sedangkan dengan *paging* , programer tidak perlu tahu teknik yang digunakan.
- Pada segmentasi kompilasi dilakukan secara terpisah sedangkan pada *paging*, kompilasinya tidak terpisah.
- Pada segmentasi proteksinya terpisah sedangkan pada *paging* proteksinya tidak terpisah.
- Pada segmentasi ada *shared code* sedangkan pada *paging* tidak ada *shared code*.
- Pada segmentasi terdapat banyak ruang alamat linier sedangkan pada *paging* hanya terdapat satu ruang alamat linier.
- Pada segmentasi prosedur dan data dapat dibedakan dan diproteksi terpisah sedangkan pada *paging* prosedur dan data tidak dapat dibedakan dan diproteksi terpisah.
- Pada segmentasi pengubahan ukuran tabel dapat dilakukan dengan mudah sedangkan pada *Paging* pengubahan ukuran tabel tidak dapat dilakukan dengan mudah.
- Segmentasi digunakan untuk mengizinkan program dan data dapat dipecahkan jadi ruang alamat mandiri dan juga untuk mendukung sharing dan proteksi sedangkan *paging* digunakan untuk mendapatkan ruang alamat linier yang besar tanpa perlu membeli memori fisik lebih.

4) Pengimplementasian Segmentasi dengan Pemberian Halaman Intel i386

Salah satu contoh prosesor yang menggunakan metode segmentasi dengan pemberian halaman ini diantaranya adalah Intel i386. Jumlah maksimum segmen tiap proses adalah 16 K dan besar tiap segmen adalah 4 GB. Dan ukuran halamannya adalah 4 KB.

❖ **Logical Address**

Ruang *logical address* dari suatu proses terbagi menjadi dua partisi yaitu:

- Partisi I
  Terdiri dari segmen berjumlah 8 K yang sifatnya pribadi atau rahasia terhadap proses tersebut. Informasi tentang partisi ini disimpan didalam *Local Descriptor Table*.
- Partisi II
  Terdiri dari 8 K segmen yang digunakan bersama diantara proses-proses tersebut. Informasi tentang partisi ini disimpan didalam *Global Descriptor Table* .

Tiap masukan atau entri pada *Local Descriptor Table* dan *Global Descriptor Table* terdiri dari 8 bita dengan informasi yang detil tentang segmen khusus termasuk lokasi dasar dan panjang segmen tersebut. *Logical address* merupakan sepasang:

➢ Selektor

Terdiri dari angka 16 bit:
Dimana s = jumlah segmen (13 bit)
g = mengindikasikan apakah segmen ada di
Global Descriptor Table
atau Local Descriptor Table
(1 bit)
p= proteksi(2 bit)

s g p
13 1 2

➢ *Offset*

Terdiri dari angka 32 bit yang menspesifikasikan lokasi suatu kata atau bita di dalam segmen tersebut.

Mesin memiliki 6 register segmen yang membiarkan 6 segmen dialamatkan pada suatu waktu oleh sebuah proses. Mesin meiliki register program mikro 8 bita untuk menampung *descriptor* yang bersesuaian baik dari *Global Descriptor Table* atau *Local Descriptor Table*. *Cache* ini membiarkan 386 menghindari membaca *descriptor* dari memori untuk tiap perujukan memori.

❖ **Alamat Fisik**

Alamat fisik 386 panjangnya adalah 32 bit. Mula-mula register segmen menunjuk ke masukan atau entri di *Global Descriptor Table* atau *Local Descriptor Table* . Kemudian informasi dasar dan limit tentang segmen tersebut digunakan untuk mengeneralisasikan alamat linier. Limit itu digunakan untuk mengecek keabsahan alamat. Jika alamat tidak sah maka akan terjadi memori fault yang menyebabkan terjadinya trap pada sistem operasi. Sedangkan apabila alamat itu sah maka nilai dari offset ditambahkan kenilai dasar yang menghasilkan alamat linier 32 bit. Alamat inilah yang kemudian diterjemahkan ke alamat fisik. Seperti dikemukakan sebelumnya tiap segmen dialamatkan dan tiap halaman 4 KB. Sebuah table halaman mungkin terdiri sampai satu juta masukan atau entri. Karena tiap entri terdiri dari 4 byte, tiap proses mungkin membutuhkan sampai 4 MB ruang alamat fisik untuk halaman tabel sendiri. Sudah jelas kalau kita tidak menginginkan untuk mengalokasi tabel halaman bersebelahan di memori utama. Solusi yang dipakai 386 adalah dengan menggunakan skema paging dua tingkat (*two-level paging scheme*). Alamat linier dibagi menjadi nomer halaman yang terdiri dari 20 bit dan *offset* halaman terdiri dari 12 bit. Karena kita *page* tabel halaman dibagi jadi 10 bit penunjuk halaman direktori dan 10 bit penunjuk tabel halaman sehingga *logical address* menjadi:

nomor halaman
offset halaman
p1 p2 d
10 10 12

7. Memori Virtual

Selama bertahun-tahun, pelaksanaan berbagai strategi managemen memori yang ada menuntut keseluruhan bagian proses berada di memori sebelum proses dapat mulai dieksekusi. Dengan kata lain, semua bagian proses harus memiliki alokasi sendiri pada memori fisiknya. Pada nyatanya tidak semua bagian dari program tersebut akan diproses, misalnya:

- Terdapat pernyataan-pernyataan atau pilihan yang hanya akan dieksekusi jika kondisi tertentu dipenuhi. Apabila kondisi tersebut tidak dipenuhi, maka pilihan tersebut tak akan pernah dieksekusi/ diproses. Contoh dari pilihan itu adalah: pesan-pesan error yang hanya akan muncul bila terjadi kesalahan dalam eksekusi program.
- Terdapat fungsi-fungsi yang jarang digunakan, bahkan sampai lebih dari 100x pemakaian
- Terdapat pealokasian memori lebih besar dari yang sebenarnya dibutuhkan. Contoh pada *array*, *list* dan tabel.

Hal-hal di atas telah menurunkan optimalitasi utilitas dari ruang memori fisik. Pada memori berkapasitas besar, hal ini mungkin tidak menjadi masalah. Akan tetapi, bagaimana jika memori yang disediakan terbatas? Salah satu cara untuk mengatasinya adalah dengan *overlay* dan *dynamic loading*. Namun hal ini menimbulkan masalah baru karena implementasinya yang rumit dan penulisan program yang akan memakan tempat di memori. Tujuan semula untuk menghemat memori bisa jadi malah tidak tercapai apabila program untuk *overlay* dan *dynamic loading*. malah lebih besar daripada program yang sebenarnya ingin dieksekusi. Maka sebagai solusi untuk masalah-masalah ini digunakanlah konsep memori virtual.

1) Pengertian

Memori virtual merupakan suatu teknik yang memisahkan antara memori logis dan memori fisiknya. Teknik ini mengizinkan program untuk dieksekusi tanpa seluruh bagian program perlu ikut masuk ke dalam memori. Berbeda dengan keterbatasan yang dimiliki oleh memori fisik, memori virtual dapat menampung program dalam skala besar, melebihi daya tampung dari memori utama yang tersedia. Prinsip dari memori virtual yang patut diingat adalah bahwa: "Kecepatan maksimum eksekusi proses di memori virtual dapat sama, tetapi tidak pernah melampaui kecepatan eksekusi proses yang sama di sistem tanpa menggunakan memori virtual." Konsep memori virtual pertama kali dikemukakan Fotheringham pada tahun 1961 pada sistem computer Atlas di Universitas Manchester, Inggris (Hariyanto, Bambang : 2001).

2) Keuntungan

Sebagaimana dikatakan di atas bahwa hanya sebagian dari program yang diletakkan di memori. Hal ini berakibat pada:

- Berkurangnya I/O yang dibutuhkan (lalu lintas I/O menjadi rendah). Misal, untuk

program butuh membaca dari disk dan memasukkan dalam memory setiap kali diakses.

- Berkurangnya memori yang dibutuhkan (*space* menjadi lebih leluasa). Contoh, untuk program 10 MB tidak seluruh bagian dimasukkan dalam memori. Pesan-pesan *error* hanya dimasukkan jika terjadi *error* .
- Meningkatnya respon, sebagai konsekuensi dari menurunnya beban I/O dan memori.
- Bertambahnya jumlah *user* yang dapat dilayani. Ruang memori yang masih tersedia luas memungkinkan komputer untuk menerima lebih banyak permintaan dari *user*.

3) Implementasi

Gagasan dari memori virtual adalah ukuran gabungan program, data dan *stack* melampaui jumlah memori fisik yang tersedia. Sistem operasi menyimpan bagian-bagian proses yang sedang digunakan di memori utama (*main memory*) dan sisanya ditaruh di disk. Begitu bagian di disk diperlukan, maka bagian di memori yang tidak diperlukan akan disingkirkan (*swap-out*) dan diganti (*swap-in*) oleh bagian disk yang diperlukan itu. Memori virtual diimplementasikan dalam system *multiprogramming*. Misalnya: 10 program dengan ukuran 2 Mb dapat berjalan dimemori berkapasitas 4 Mb. Tiap program dialokasikan 256 KByte dan bagian-bagian proses di *swap* masuk dan keluar memori begitu diperlukan. Dengan demikian, system *multiprogramming* menjadi lebih efisien. Memori virtual dapat dilakukan melalui dua cara:

- Permintaan pemberian halaman (*demand paging*).
- Permintaan segmentasi (*demand segmentation*). Contoh: IBM OS/2. Algoritma dari permintaan segmentasi lebih kompleks, karenanya jarang diimplementasikan.

8. Permintaan Pemberian Halaman (Demand Paging)

Merupakan implementasi yang paling umum dari memori virtual. Prinsip permintaan pemberian halaman (*demand paging*) hampir sama dengan sistem penomoran (*paging*) dengan menggunakan *swapping*. Perbedaannya adalah *page* pada permintaan pemberian halaman tidak akan pernah di*swap* ke memori sampai ia benar-benar diperlukan. Untuk itu diperlukan adanya pengecekan dengan bantuan perangkat keras mengenai lokasi dari *page* saat ia dibutuhkan.

1) Permasalahan pada Page Fault

Ada tiga kemungkinan kasus yang dapat terjadi pada saat dilakukan pengecekan pada *page* yang dibutuhkan, yaitu:

- *Page* ada dan sudah berada di memori.
- *Page* ada tetapi belum ditaruh di memori (harus menunggu sampai dimasukkan).
- *Page* tidak ada, baik di memori mau pun di disk (invalid reference --> abort).

Saat terjadi kasus kedua dan ketiga, maka proses dinyatakan mengalami *page fault*.

2) Skema Bit Valid - Tidak Valid

Dengan meminjam konsep yang sudah pernah dijelaskan dalam Bab 9, maka dapat ditentukan page mana yang ada di dalam memori dan mana yang tidak ada di dalam memori. Konsep itu adalah skema bit valid - tidak valid, di mana di sini pengertian "valid" berarti bahwa *page* legal dan berada dalam memori (kasus 1), sedangkan "tidak valid" berarti *page* tidak ada (kasus 3) atau *page* ada tapi tidak ditemui di memori (kasus 2).

Pengesetan bit:

Bit 1 -->
page berada di memori
Bit 0 -->
page tidak berada di
memori.
(Dengan inisialisasi: semua bit
di-set 0).

Apabila ternyata hasil dari translasi, bit *page* = 0, berarti *page fault* terjadi.

❖ **Penanganan Page Fault**

Prosedur penanganan *page fault* sebagaimana tertulis di buku *Operating System Concept*

5th Ed. Halaman 294 adalah sebagai berikut:

- Cek tabel internal yang dilengkapi dengan PCB untuk menentukan valid atau tidaknya bit.
- Apabila tidak valid, program akan di*terminate* (interupsi oleh *illegal address trap*).
- Memilih *frame* kosong (*free-frame*), misal dari *free-frame list*. Jika tidak ditemui ada *frame* yang kosong, maka dilakukan *swap-out* dari memori. *Frame* mana yang harus di- *swap-out* akan ditentukan oleh algoritma (lihat sub bab *Page Replacement*).
- Menjadualkan operasi disk untuk membaca *page* yang diinginkan ke *frame* yang baru dialokasikan.
- Ketika pembacaan komplit, tabel internal akan dimodifikasi dan *page* diidentifikasi ada di memori.
- Mengulang instruksi yang tadi telah sempat diinterupsi. Jika tadi *page fault* terjadi saat instruksi di- *fetch* , maka akan dilakukan *fecthing* lagi. Jika terjadi saat operan sedang di-*fetch*, maka harus dilakukan *fetch* ulang, *decode* dan *fetch* operan lagi.

❖ **Permasalahan Lain yang berhubungan dengan Demand Paging**

Sebagaimana dilihat di atas, bahwa ternyata penanganan *page fault* menimbulkan masalah masalah baru pada proses *restart instruction* yang berhubungan dengan arsitektur komputer. Masalah yang terjadi, antara lain mencakup:

- Bagaimana mengulang instruksi yang memiliki beberapa lokasi yang berbeda?
- Bagaimana pengalamatan dengan menggunakan *special-addresing mode*, termasuk *autoincrement* dan *autodecrement mode*?
- Bagaimana jika instruksi yang dieksekusi panjang (contoh: *block move*)?

Masalah pertama dapat diatasi dengan dua cara yang berbeda.

- Komputasi *microcode* dan berusaha untuk mengakses kedua ujung dari blok, agar tidak ada modifikasi *page* yang sempat terjadi.
- Memanfaatkan register sementara (*temporary register*) untuk menyimpan nilai yang sempat tertimpa/termodifikasi oleh nilai lain.

Masalah kedua diatasi dengan menciptakan suatu *special-status register* baru yang berfungsi menyimpan nomor register dan banyak perubahan yang terjadi sepanjang eksekusi instruksi. Sedangkan masalah ketiga diatasi dengan mengeset bit FPD (*first phase done*) sehingga *restart instruction* tidak akan dimulai dari awal program, melainkan dari tempat program terakhir dieksekusi.

❖ **Persyaratan Perangkat Keras**

Pemberian nomor halaman melibatkan dukungan perangkat keras, sehingga ada persyaratan perangkat keras yang harus dipenuhi. Perangkat-perangkat keras tersebut sama dengan yang digunakan untuk *paging* dan *swapping*, yaitu:

- *Page-table* , menandai bit valid-tidak valid
- *Secondary memory* , tempat menyimpan *page* yang tidak ada di memori utama

Lebih lanjut, sebagai konsekuensi dari persyaratan ini, akan diperlukan pula perangkat lunak yang dapat mendukung terciptanya pemberian nomor halaman.

9. Pemindahan Halaman

Pada dasarnya, kesalahan halaman (*page fault*) sudah tidak lagi menjadi masalah yang terlalu dianggap serius. Hal ini disebabkan karena masing-masing halaman pasti akan mengalami paling tidak satu kali kesalahan dalam pemberian halaman, yakni ketika halaman ini ditunjuk untuk pertama kalinya. Representasi seperti ini sebenarnya tidaklah terlalu akurat. Berdasarkan pertimbangan tersebut, sebenarnya proses-proses yang memiliki 10 halaman hanya akan menggunakan setengah dari jumlah seluruh halaman yang dimilikinya. Kemudian *demand paging* akan menyimpan I/O yang dibutuhkan untuk mengisi 5 halaman yang belum pernah digunakan. Kita juga dapat meningkatkan derajat *multiprogramming* dengan menjalankan banyak proses sebanyak 2 kali. Jika kita meningkatkan derajat *multiprogramming*, itu sama artinya dengan melakukan *over-allocating* terhadap memori. Jika kita menjalankan 6 proses, dengan masing-masing mendapatkan 10 halaman, walau pun sebenarnya yang digunakan hanya 5 halaman, kita akan memiliki utilisasi CPU dan *throughput* yang lebih tinggi dengan 10 *frame* yang masih kosong. Lebih jauh lagi, kita harus mempertimbangkan bahwa system memori tidak hanya digunakan untuk menangani pengalamatan suatu program. Penyangga *(buffer)* untuk I/O juga menggunakan sejumlah memori. Penggunaan ini dapat meningkatkan pemakaian algoritma dalam penempatan dimemori. Beberapa sistem

mengalokasikan secara pasti beberapa persen dari memori yang dimilikinya untuk penyangga I/O, dimana keduanya, baik proses pengguna mau pun subsistem dari I/O saling berlomba untuk memanfaatkan seluruh sistem memori.

1) Skema Dasar

Pemindahan halaman mengambil pendekatan seperti berikut. Jika tidak ada *frame* yang kosong, kita mencari *frame* yang tidak sedang digunakan dan mengosongkannya. Kita dapat mengosongkan sebuah *frame* dengan menuliskan isinya ke ruang pertukaran *(swap space)*, dan merubah tabel halaman (juga tabel-tabel lainnya) untuk mengindikasikan bahwa halaman tesebut tidak akan lama berada di memori. Sekarang kita dapat menggunakan *frame* yang kosong sebagai penyimpan halaman dari proses yang salah. Rutinitas pemindahan halaman:

- Cari lokasi dari halaman yang diinginkan pada *disk*
- Cari *frame* kosong:
  - Jika ada *frame* kosong, gunakan.
  - Jika tidak ada *frame* kosong, gunakan algoritma pemindahan halaman untuk menyeleksi *frame* yang akan digunakan.
  - Tulis halaman yang telah dipilih ke disk, ubah tabel halaman dan table *frame*.
- Baca halaman yang diinginkan kedalam *frame* kosong yang baru, ubah tabel halaman dan table *frame* .
- Ulang dari awal proses pengguna.

Jika tidak ada *frame* yang kosong, pentransferan dua halaman (satu masuk, satu keluar) akan dilakukan. Situasi ini secara efektif akan menggandakan waktu pelayanan kesalahan halaman dan meningkatkan waktu akses efektif. Kita dapat mengurangi pemborosan ini dengan menggunakan bit tambahan. Masing-masing halaman atau *frame* mungkin memiliki bit tambahan yang diasosiasikan didalam perangkat keras. Pemindahan halaman merupakan dasar dari *demand paging*. Yang menjembatani pemisahan antara memori lojik dan memori fisik. Dengan mekanisme seperti ini, memori virtual yang sangat besar dapat disediakan untuk *programmer* dalam bentuk memori fisik yang lebih kecil. Dengan *nondemand paging* , alamat dari *user* dipetakan kedalam alamat fisik, jadi 2 set alamat dapat berbeda. Seluruh halaman dari proses masih harus berada di memori fisik. Dengan *demand paging*, ukuran dari ruang alamat logika sudah tidak dibatasi oleh memori fisik. Kita harus menyelesaikan 2 masalah utama untuk mengimplementasikan *demand paging*. Kita harus mengembangkan algoritma pengalokasian *frame* dan algoritma pemindahan halaman. Jika kita memiliki banyak proses di memori, kita harus memutuskan berapa banyak *frame* yang akan dialokasikan ke masing-masing proses. Lebih jauh lagi, saat pemindahan halaman diinginkan, kita harus memilih *frame* yang akan dipindahkan. Membuat suatu algoritma yang tepat untuk menyelesaikan masalah ini adalah hal yang sangat penting.Ada beberapa algoritma pemindahan halaman yang berbeda. Kemungkinan setiap Sistem Operasi memiliki skema pemindahan yang unik. Algoritma pemindahan yang baik adalah yang memiliki tingkat kesalahan halaman terendah. Kita mengevaluasi algoritma dengan menjalankannya dalam *string* khusus di memori acuan dan menghitung jumlah kesalahan halaman. *String* dari memori acuan disebut *string* acuan *(reference string)* . Sebagai contoh, jika kita memeriksa proses khusus, kita mungkin akan mencatat urutan alamat seperti dibawah ini:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

dimana pada 100 *bytes* setiap halaman, diturunkan menjadi *string* acuan seperti berikut:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Perlu diperhatikan bahwa selama jumlah *frame* meningkat, jumlah kesalahan halaman menurun. Penambahan memori fisik akan meningkatkan jumlah *frame* .

2) Pemindahan Halaman Secara FIFO

Algoritma ini adalah algoritma paling sederhana dalam hal pemindahan halaman. Algoritma pemindahan FIFO *(First In First Out)* mengasosiasikan waktu pada saat halaman dibawa kedalam memori dengan masing-masing halaman. Pada saat halaman harus dipindahkan, halaman yang paling tua yang dipilih. Sebagai contoh:

String Acuan.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 2 2 4 4 4 0 0 0 7 7 7
0 0 0 3 3 3 2 2 2 1 1 1 0 0 0
1 1 1 0 0 0 3 3 3 2 2 2 1

frame halaman

Dari contoh diatas, terdapat 15 kesalahan halaman. Algoritma FIFO mudah untuk dipahami dan diimplementasikan. Namun *performance*nya tidak selalu bagus. Salah satu kekurangan dari algoritma FIFO adalah kemungkinan terjadinya anomali Beladi, dimana dalam beberapa kasus, tingkat kesalahan akan meningkat seiring dengan peningkatan jumlah *frame* yang dialokasikan.

3) Pemindahan Halaman Secara Optimal

Salah satu akibat dari upaya mencegah terjadinya anomali Beladi adalah algoritma pemindahan halaman secara optimal. Algoritma ini memiliki tingkat kesalahan halaman terendah dibandingkan dengan algoritma-algoritma lainnya. Algoritma ini tidak akan mengalami anomaly Belady. Konsep utama dari algoritma ini adalah mengganti halaman yang tidak akan digunakan untuk jangka waktu yang paling lama. Algoritma ini menjamin kemungkinan tingkat kesalahan terendah untuk jumlah *frame* yang tetap. Sebagai contoh:

String Acuan.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
7 7 7 2 2 2 2 2 7
0 0 0 0 4 0 0 0
1 1 3 3 3 1 1

Dari contoh diatas, terdapat 9 kesalahan halaman. Dengan hanya 9 kesalahan halaman, algoritma optimal jauh lebih baik daripada algoritma FIFO. Perlu disayangkan, algoritma optimal susah untuk diimplementasikan kedalam program, karena algoritma ini menuntut pengetahuan tentang *string* acuan yang akan muncul.

4) Pemindahan Halaman Secara LRU

Jika algoritma optimal sulit untuk dilakukan, mungkin kita dapat melakukan pendekatan terhadap algoritma tersebut. Jika kita menggunakan waktu yang baru berlalu sebagai pendekatan terhadap waktu yang akan datang, kita akan memindahkan halaman yang sudah lama tidak digunakan dalam jangka waktu yang terlama. Pendekatan ini disebut algoritma LRU *(Least Recently Used)*. Algoritma LRU mengasosiasikan dengan masing-masing halaman waktu dari halaman yang terakhir digunakan. Ketika halaman harus dipindahkan, LRU memilih halaman yang paling lama tidak digunakan pada waktu yang lalu. Inilah algoritma LRU, melihat waktu yang telah lalu, bukan waktu yang akan datang. Sebagai contoh:

String Acuan.
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
7 7 7 2 2 4 4 4 0 1 1 1
0 0 0 0 0 0 3 3 3 0 0
1 1 3 3 2 2 2 2 2 7

frame halaman

Dari contoh diatas, terdapat 12 kesalahan halaman. Meski pun algoritma ini menghasilkan 12 kesalahan halaman, algoritma ini masih lebih baik daripada algoritma FIFO, yang menghasilkan 15 kesalahan halaman. Untuk mengimplementasikan algoritma LRU, terdapat 2 implementasi yang dapat digunakan, yaitu dengan *counter* dan *stack* .

Selain algoritma optimal, algoritma LRU juga dapat terhindar dari anomali Beladi. Salah satu kelas dari algoritma pemindahan halaman adalah algoritma *stack*, yang juga tidak akan pernah mengalami anomaly Beladi. Algoritma *stack* ini menyimpan nomor-nomor halaman pada *stack* . Kapan pun suatu halaman ditunjuk, halaman ini dikeluarkan dari *stack* dan diletakkan di blok paling atas dari *stack* . Dengan cara seperti ini, blok paling atas dari *stack* selalu berisi halaman yang baru digunakan, sedangkan blok terbawah dari *stack* selalu berisi halaman yang sudah lama tidak digunakan. Karena suatu halaman dalam *stack* dapat dikeluarkan meski pun berada ditengah-tengah *stack* , maka implementasi terbaik untuk algoritma ini adalah dengan daftar mata rantai ganda (*doubly linked list*), dengan kepala dan ekor sebagai penunjuk. Pendekatan ini sangat tepat untuk

perangkat lunak atau implementasi kode mikro dari algoritma LRU. Sebagai contoh:

Gambar 4-14. String Acuan. Sumber: . . .

4 7 0 7 1 0 1 2 1 2 7 1 2
4 7 0 7 1 0 1 2 1 2 7 1 2
4 7 0 7 1 0 1 2 1 2 7 1
4 4 0 7 7 0 0 0 1 2 7
4 4 4 7 7 7 0 0 0
4 4 4 4 4 4

frame halaman.

5) Pemindahan Halaman Secara Perkiraan LRU

Hanya sedikit sistem komputer yang menyediakan perangkat lunak yang memberikan cukup dukungan terhadap algoritma pemindahan halaman secara LRU. Banyak sistem yang tidak menyediakan perangkat lunak yang memberikan dukungan terhadap algoritma LRU, sehingga terpaksa menggunakan algoritma lain, seperti FIFO. Banyak sistem menyediakan bantuan untuk menangani masalah ini, misalnya dengan bit acuan. Bit acuan untuk halaman diset oleh perangkat lunak kapan pun halaman tersebut ditunjuk. Bit acuan diasosiasikan dengan masing-masing isi dari tabel halaman. Awalnya, seluruh bit dikosongkan oleh sistem operasi. Selama proses pengguna dijalankan, bit yang diasosiasikan ke masing-masing halaman acuan diset menjadi 1 oleh perangkat keras. Setelah beberapa waktu, kita dapat menentukan halaman mana yang sudah digunakan dan halaman mana yang belum digunakan dengan menguji bit-bit acuan. Informasi tersebut memberikan informasi penting untuk banyak algoritma pemindahan halaman yang memperkirakan halaman mana yang sudah lama tidak digunakan.

❖ **Algoritma Additional-Reference-Bit**

Kita bisa mendapatkan informasi tambahan mengenai urutan dengan mencatat bit-bit acuan pada suatu interval yang tetap. Kita dapat menyimpan 8-bit byte untuk masing-masing halaman pada tabel di memori. Pada interval tertentu, pencatat waktu (*timer*) melakukan interupsi dengan mentransfer control kepada sistem operasi. Sistem operasi mengubah bit acuan untuk masing-masing halaman kedalam bit *high-order* dari 8-bit byte ini dan membuang bit *low-order*. Register pengganti 8-bit ini berisi sejarah penggunaan halaman dalam periode 8 waktu terakhir.

Sebagai contoh, seandainya register pengganti berisi 00000000, maka itu berarti halaman sudah tidak digunakan dalam periode 8 waktu terakhir, halaman yang digunakan paling tidak 1 kali akan memiliki nilai register penggati 11111111.

❖ **Algoritma Second-Chance**

Algoritma "*second-chance*" didasari oleh algoritma FIFO. Pada saat suatu halaman ditunjuk, kita akan menginspeksi bit acuannya. Jika bit acuan tersebut bernilai 0, kita memproses untuk membuang halaman ini. Jika bit acuan tersebut bernilai 1, kita berikan kesempatan kedua untuk halaman ini dan menyeleksi halaman FIFO selanjutnya. Ketika suatu halaman mendapatkan kesempatan kedua, bit acuannya dikosongkan dan waktu tibanya direset menjadi saat ini. Karena itu, halaman yang mendapatkan kesempatan kedua tidak akan dipindahkan sampai seluruh halaman dipindahkan. Tambahan lagi, jika halaman yang digunakan cukup untuk menampung 1 set bit acuan, maka halaman ini tidak akan pernah dipindahkan.

❖ **Algoritma Second-Chance (Yang Diperbaiki)**

Kita dapat memperbaiki kekurangan dari algoritma *second-chance* dengan mempertimbangkan 2 hal sekaligus, yaitu bit acuan dan bit modifikasi. Dengan 2 bit ini, kita akan mendapatkan 4 kemungkinan yang akan terjadi, yaitu:

- (0,0) tidak digunakan dan tidak dimodifikasi, bit terbaik untuk dipindahkan.
- (0,1) tidak digunakan tapi dimodifikasi, tidak terlalu baik untuk dipindahkan karena halaman ini perlu ditulis sebelum dipindahkan.
- (1,0) digunakan tapi tidak dimodifikasi, terdapat kemungkinan halaman ini akan segera digunakan lagi.
- (1,1) digunakan dan dimodifikasi, halaman ini mungkin akan segera digunakan lagi dan halaman ini perlu ditulis ke *disk* sebelum dipindahkan.

Algoritma ini digunakan dalam skema manajemen memori virtual Macintosh.

6) Dasar Perhitungan Pemindahan Halaman

Banyak algoritma-algoritma lain yang dapat digunakan untuk pemindahan halaman. Sebagai contoh, kita dapat menyimpan *counter* dari nomor acuan yang sudah dibuat untuk masing-masing halaman, dan mengembangkan 2 skema dibawah ini: ALGORITMA PEMINDAHAN HALAMAN LFU Algoritma LFU (*Least Frequently Used*) menginginkan halaman dengan nilai terkecil untuk dipindahkan. Alasannya, halaman yang digunakan secara aktif akan memiliki nilai acuan yang besar. ALGORITMA PEMINDAHAN HALAMAN MFU Algoritma MFU (*Most Frequently Used*) didasarkan pada argumen yang menyatakan bahwa halaman dengan nilai terkecil mungkin baru saja dimasukkan dan baru digunakan. Kedua algoritma diatas tidaklah terlalu umum, hal ini disebabkan karena implementasi dari kedua algoritma diatas sangatlah mahal.

7) Algoritma Page-Buffering

Prosedur lain sering digunakan untuk menambah kekhususan dari algoritma pemindahan halaman. Sebagai contoh, pada umumnya sistem menyimpan *pool* dari *frame* yang kosong. Prosedur ini memungkinkan suatu proses mengulang dari awal secepat mungkin, tanpa perlu menunggu halaman yang akan dipindahkan untuk ditulis ke *disk* karena *frame* nya telah ditambahkan kedalam *pool frame* kosong. Teknik seperti ini digunakan dalam sistem VAX/ VMS, dengan algoritma FIFO. Ketika algoritma FIFO melakukan kesalahan dengan memindahkan halaman yang masih digunakan secara aktif, halaman tersebut akan dengan cepat diambil kembali dari penyangga *frame* -kosong, untuk melakukan hal tersebut tidak ada I/O yang dibutuhkan. Metode ini diperlukan oleh VAX karena versi terbaru dari VAX tidak mengimplementasikan bit acuan secara tepat.

10. Alokasi Frame

Terdapat masalah dalam alokasi *frame* dalam penggunaan memori virtual, masalahnya yaitu bagaimana kita membagi memori yang bebas kepada berbagai proses yang sedang dikerjakan? Jika ada sejumlah frame bebas dan ada dua proses, berapakah *frame* yang didapatkan tiap proses? Kasus paling mudah dari memori virtual adalah sistem satu pemakai. Misalkan sebuah system mempunyai memori 128K dengan ukuran halaman 1K, sehingga ada 128 *frame*. Sistem operasinya menggunakan 35K sehingga ada 93 *frame* yang tersisa untuk proses tiap user. Untuk *pure demand paging*, ke-93 *frame* tersebut akan ditaruh pada daftar *frame* bebas. Ketika sebuah proses user mulai dijalankan, akan terjadi sederetan *page fault*. Sebanyak 93 *page fault* pertama akan mendapatkan *frame* dari daftar *frame* bebas. Saat *frame* bebas sudah habis, sebuah algoritma pergantian halaman akan digunakan untuk memilih salah satu dari 93 halaman di memori yang diganti dengan yang ke 94, dan seterusnya. Ketika proses selesai atau diterminasi, sembilan puluh tiga *frame* tersebut akan disimpan lagi pada daftar *frame* bebas. Terdapat macam-macam variasi untuk strategi sederhana ini, kita bisa meminta system operasi untuk mengalokasikan seluruh *buffer* dan ruang tabel-nya dari daftar *frame* bebas. Saat ruang ini tidak digunakan oleh sistem operasi, ruang ini bisa digunakan untuk mendukung paging dari user. Kita juga dapat menyimpan tiga *frame* bebas yang dari daftar *frame* bebas, sehingga ketika terjadi *page fault*, ada *frame* bebas yang dapat digunakan untuk *paging*. Saat pertukaran halaman terjadi, penggantinya dapat dipilih, kemudian ditulis ke disk, sementara proses user tetap berjalan. Variasi lain juga ada, tetapi ide dasarnya tetap yaitu proses pengguna diberikan *frame* bebas yang mana saja. Masalah lain muncul ketika *demand paging* dikombinasikan dengan *multiprogramming*. Hal ini terjadi karena *multiprogramming* menaruh dua (atau lebih) proses di memori pada waktu yang bersamaan.

1) Jumlah Frame Minimum

Tentu saja ada berbagai batasan pada strategi kita untuk alokasi *frame*. Kita tidak dapat mengalokasikan lebih dari jumlah total *frame* yang tersedia (kecuali ada *page sharing*). Ada juga jumlah minimal *frame* yang dapat di alokasikan. Jelas sekali, seiring dengan bertambahnya jumlah *frame* yang dialokasikan ke setiap proses berkurang, tingkat *page fault* bertambah dan mengurangi kecepatan eksekusi proses. Selain hal tersebut di atas, ada jumlah minimum *frame* yang harus dialokasikan. Jumlah minimum ini ditentukan oleh arsitektur set instruksi. Ingat bahwa ketika terjadi *page fault*, sebelum eksekusi instruksi selesai, instruksi tersebut harus diulang. Sehingga kita harus punya jumlah *frame* yang cukup untuk menampung semua halaman yang dirujuk oleh sebuah instruksi tunggal.

Jumlah minimum *frame* ditentukan oleh arsitektur komputer. Sebagai contoh, instruksi *move* pada PDP-11 adalah lebih dari satu kata untuk beberapa modus pengalamatan,

sehingga instruksi tersebut bisa membutuhkan dua halaman. Sebagai tambahan, tiap operannya mungkin merujuk tidak langsung, sehingga total ada enam *frame*. Kasus terburuk untuk IBM 370 adalah instruksi MVC. Karena instruksi tersebut adalah instruksi perpindahan dari penyimpanan ke penyimpanan, instruksi ini butuh 6 bit dan dapat memakai dua halaman. Satu blok karakter yang akan dipindahkan dan daerah tujuan perpindahan juga dapat memakai dua halaman, sehingga situasi ini membutuhkan enam *frame*. Kesimpulannya, jumlah minimum *frame* yang dibutuhkan per proses tergantung dari arsitektur computer tersebut, sementara jumlah maksimumnya ditentukan oleh jumlah memori fisik yang tersedia. Di antara kedua jumlah tersebut, kita punya pilihan yang besar untuk alokasi *frame* .

2) Algoritma Alokasi

Cara termudah untuk membagi m *frame* terhadap n proses adalah untuk memberikan bagian yang sama, sebanyak m/n *frame* untuk tiap proses. Sebagai contoh ada 93 *frame* tersisa dan 5 proses, maka tiap proses akan mendapatkan 18 *frame*. Frame yang tersisa, sebanyak 3 buah dapat digunakan sebagai *frame* bebas cadangan. Strategi ini disebut *equal allocation*. Sebuah alternatif yaitu pengertian bahwa berbagai proses akan membutuhkan jumlah memori yang berbeda. Jika ada sebuah proses sebesar 10K dan sebuah proses basis data 127K dan hanya kedua proses ini yang berjalan pada sistem, maka ketika ada 62 *frame* bebas, tidak masuk akal jika kita memberikan masing-masing proses 31 *frame*. Proses pertama hanya butuh 10 *frame*, 21 *frame* lain akan terbuang percuma. Untuk menyelesaikan masalah ini, kita menggunakan *proportional allocation*. Kita mengalokasikan memori yang tersedia kepada setiap proses tergantung pada ukurannya. *Let the size of the virtual memory for process pi be si, and define S = si*. Lalu, jika jumlah total dari *frame* yang tersedia adalah m, kita mengalokasikan proses ai ke proses pi, dimana ai mendekati

ai = si / S x m

Dalam kedua strategi ini, tentu saja, alokasi untuk setiap proses bisa bervariasi berdasarkan *multiprogramming level* -nya. Jika *multiprogramming level*nya meningkat, setiap proses akan kehilangan beberapa *frame* guna menyediakan memori yang dibutuhkan untuk proses yang baru. Di sisi lain, jika *multiprogramming level*nya menurun, *frame* yang sudah dialokasikan pada bagian process sekarang bisa disebar ke proses-proses yang masih tersisa. Mengingat hal itu, dengan *equal* atau pun *proportional allocation*, proses yang berprioritas tinggi diperlakukan sama dengan proses yang berprioritas rendah. Berdasarkan definisi tersebut, bagaimanapun juga, kita ingin memberi memori yang lebih pada proses yang berprioritas tinggi untuk mempercepat eksekusi-nya, *to the detriment of low-priority processes*. Satu pendekatan adalah menggunakan *proportional allocation scheme* dimana perbandingan *frame*nya tidak tergantung pada ukuran relatif dari proses, melainkan lebih pada prioritas proses, atau tergantung kombinasi dari ukuran dan prioritas.

3) Alokasi Global lawan Local

Faktor penting lain dalam cara-cara pengalokasian *frame* ke berbagai proses adalah penggantian halaman. Dengan proses-proses yang bersaing mendapatkan *frame*, kita dapat mengklasifikasikan algoritma penggantian halaman kedalam dua kategori *broad*: Penggantian Global dan Penggantian Lokal. Penggantian Global memperbolehkan sebuah proses untuk menyeleksi sebuah *frame* pengganti dari himpunan semua *frame*, meski pun *frame* tersebut sedang dialokasikan untuk beberapa proses lain; satu proses dapat mengambil sebuah *frame* dari proses yang lain. Penggantian Lokal mensyaratkan bahwa setiap proses boleh menyeleksi hanya dari himpunan *frame* yang telah teralokasi pada proses itu sendiri. Untuk contoh, pertimbangkan sebuah skema alokasi dimana kita memperbolehkan proses berprioritas tinggi untuk meyeleksi *frame* dari proses berprioritas rendah untuk penggantian. Sebuah proses dapat menyeleksi sebuah pengganti dari *frame* -nya sendiri atau dari *frame-frame* proses yang berprioritas lebih rendah. Pendekatan ini memperbolehkan sebuah proses berprioritas tinggi untuk meningkatkan alokasi *frame* -nya pada expense proses berprioritas rendah.

Dengan strategi Penggantian Lokal, jumlah *frame* yang teralokasi pada sebuah proses tidak berubah. Dengan Penggantian Global, ada kemungkinan sebuah proses hanya menyeleksi *frame-frame* yang teralokasi pada proses lain, sehingga meningkatkan jumlah *frame* yang teralokasi pada proses itu sendiri (asumsi bahwa proses lain tidak memilih *frame* proses tersebut untuk penggantian). Masalah pada algoritma Penggantian Global adalah bahwa sebuah proses tidak bisa mengontrol *page-fault*nya sendiri. Himpunan

halaman dalam memori untuk sebuah proses tergantung tidak hanya pada kelakuan paging dari proses tersebut, tetapi juga pada kelakuan *paging* dari proses lain. Karena itu, proses yang sama dapat tampil berbeda (memerlukan 0,5 detik untuk satu eksekusi dan 10,3 detik untuk eksekusi berikutnya) *due to totally external circumstances*. Dalam Penggantian Lokal, himpunan halaman dalam memori untuk sebuah proses hanya dipengaruhi kelakuan paging proses itu sendiri. Penggantian Lokal dapat menyembunyikan sebuah proses dengan membuatnya tidak tersedia bagi proses lain, menggunakan halaman yang lebih sedikit pada memori. Jadi, secara umum Penggantian Global menghasilkan sistem throughput yang lebih bagus, maka itu artinya metode yang paling sering digunakan.

PERTEMUAN VII
SISTEM FILE

A. TUJUAN

Mahasiswa dapat memahami sistem file

B. TEORI

1. Pengertian

Sistem berkas merupakan mekanisme penyimpanan *on-line* serta untuk akses, baik data mau pun program yang berada dalam Sistem Operasi. Terdapat dua bagian penting dalam sistem berkas, yaitu

- kumpulan berkas, sebagai tempat penyimpanan data serta
- struktur direktori, yang mengatur dan menyediakan informasi mengenai seluruh berkas dalam sistem.

2. Berkas

1) Konsep Dasar

Seperti yang telah kita ketahui, komputer dapat menyimpan informasi ke beberapa media penyimpanan yang berbeda, seperti *magnetic disks*, *magnetic tapes* dan *optical disks*. Agar computer dapat digunakan dengan nyaman, system operasi menyediakan sistem penyimpanan dengan sistematika yang seragam. Sistem Operasi mengabstraksi properti fisik dari media penyimpanannya dan mendefinisikan unit penyimpanan logis, yaitu berkas. Berkas dipetakan ke media fisik oleh sistem operasi. Media penyimpanan ini umumnya bersifat *non-volatile*, sehingga kandungan di dalamnya tidak akan hilang jika terjadi gagal listrik mau pun *system reboot*. Berkas adalah kumpulan informasi berkait yang diberi nama dan direkam pada penyimpanan sekunder. Dari sudut pandang pengguna, berkas merupakan bagian terkecil dari penyimpanan logis, artinya data tidak dapat ditulis ke penyimpanan sekunder kecuali jika berada di dalam berkas. Biasanya berkas merepresentasikan program (baik *source* mau pun bentuk objek) dan data. Data dari berkas dapat bersifat numerik, alfabetik, alfanumerik, atau pun biner. Format berkas juga bisa bebas, misalnya berkas teks, atau dapat juga diformat pasti. Secara umum, berkas adalah urutan bit, byte, baris, atau catatan yang didefinisikan oleh pembuat berkas dan pengguna. Informasi dalam berkas ditentukan oleh pembuatnya. Ada banyak beragam jenis informasi yang dapat disimpan dalam berkas. Hal ini disebabkan oleh struktur tertentu yang dimiliki oleh berkas, sesuai dengan jenisnya masing-masing. Contohnya:

- *Text file* ; yaitu urutan karakter yang disusun ke dalam baris-baris.
- *Source file* ; yaitu urutan *subroutine* dan fungsi, yang nantinya akan dideklarasikan.
- *Object file* ; merupakan urutan byte yang diatur ke dalam blok-blok yang dikenali oleh *linker* dari sistem.
- *Executable file* ; adalah rangkaian *code section* yang dapat dibawa loader ke dalam memori dan dieksekusi.

2) Atribut Pada Berkas

Berkas diberi nama, untuk kenyamanan bagi pengguna, dan untuk acuan bagi data yang terkandung didalamnya. Nama berkas biasanya berupa string atau karakter. Beberapa sistem membedakan penggunaan huruf besar dan kecil dalam penamaan sebuah berkas, sementara sistem yang lain menganggap kedua hal di atas sama.Ketika berkas diberi nama, maka berkas tersebut akan menjadi mandiri terhadap proses, pengguna, bahkan sistem yang membuatnya. Atribut berkas terdiri dari ;

- *Nama* ; merupakan satu-satunya informasi yang tetap dalam bentuk yang bisa dibaca oleh manusia (human-readable form)
- *Type* ; dibutuhkan untuk sistem yang mendukung beberapa type berbeda
- *Lokasi* ; merupakan pointer ke device dan ke lokasi berkas pada device tersebut
- *Ukuran (size)* ; yaitu ukuran berkas pada saat itu, baik dalam byte, huruf, atau pun blok
- *Proteksi* ; adalah informasi mengenai kontrol akses, misalnya siapa saja yang boleh membaca, menulis dan mengeksekusi berkas *Waktu, tanggal dan identifikasi pengguna* ; informasi ini biasanya disimpan untuk:
  - ✓ pembuatan berkas,
  - ✓ modifikasi terakhir yang dilakukan pada berkas, dan
  - ✓ penggunaan terakhir berkas.

Data tersebut dapat berguna untuk proteksi, keamanan, dan monitoring penggunaan dari berkas. Informasi tentang seluruh berkas disimpan dalam struktur direktori yang terdapat pada penyimpanan sekunder. Direktori, seperti berkas, harus bersifat *non-volatile*, sehingga keduanya harus disimpan pada sebuah *device* dan baru dibawa bagian per bagian ke memori pada saat dibutuhkan.

3) Operasi Pada Berkas

Sebuah berkas adalah jenis data abstrak. Untuk mendefinisikan berkas secara tepat, kita perlu melihat operasi yang dapat dilakukan pada berkas tersebut. Sistem operasi menyediakan *system calls* untuk membuat, membaca, menulis, mencari, menghapus, dan sebagainya. Berikut dapat kita lihat apa yang harus dilakukan sistem operasi pada keenam operasi dasar pada berkas.

- *Membuat sebuah berkas*: Ada dua cara dalam membuat berkas. Pertama, tempat baru di dalam sistem berkas harus di alokasikan untuk berkas yang akan dibuat. Kedua, sebuah direktori harus mempersiapkan tempat untuk berkas baru, kemudian direktori

tersebut akan mencatat nama berkas dan lokasinya pada sistem berkas.

- *Menulis pada sebuah berkas* : Untuk menulis pada berkas, kita menggunakan *system call* beserta nama berkas yang akan ditulisi dan informasi apa yang akan ditulis pada berkas. Ketika diberi nama berkas, sistem mencari ke direktori untuk mendapatkan lokasi berkas. Sistem juga harus menyimpan penunjuk tulis pada berkas dimana penulisan berikut akan ditempatkan. Penunjuk tulis harus diperbaharui setiap terjadi penulisan pada berkas.
- *Membaca sebuah berkas* : Untuk dapat membaca berkas, kita menggunakan *system call* beserta nama berkas dan di blok memori mana berkas berikutnya diletakkan. Sama seperti menulis, direktori mencari berkas yang akan dibaca dan sistem menyimpan penunjuk baca pada berkas dimana pembacaan berikutnya akan terjadi. Ketika pembacaan dimulai, penunjuk baca harus diperbaharui. Sehingga secara umum, suatu berkas ketika sedang dibaca atau ditulis, kebanyakan sistem hanya mempunyai satu penunjuk, baca dan tulis menggunakan penunjuk yang sama, hal ini menghemat tempat dan mengurangi kompleksitas sistem.
- *Menempatkan kembali sebuah berkas* : Direktori yang bertugas untuk mencari berkas yang bersesuaian, dan mengembalikan lokasi berkas pada saat itu. Menempatkan berkas tidak perlu melibatkan proses I/O. Operasi sering disebut pencarian berkas.
- *Menghapus sebuah berkas*: Untuk menghapus berkas kita perlu mencari berkas tersebut di dalam direktori. Setelah ditemukan kita membebaskan tempat yang dipakai berkas tersebut (sehingga dapat digunakkan oleh berkas lain) dan menghapus tempatnya di direktori.
- *Memendekkan berkas*: Ada suatu keadaan dimana pengguna menginginkan atribut dari berkas tetap sama tetapi ingin menghapus isi dari berkas tersebut. Fungsi ini mengizinkan semua atribut tetap sama tetapi panjang berkas menjadi nol, hal ini lebih baik dari pada memaksa pengguna untuk menghapus berkas dan membuatnya lagi.

Enam operasi dasar ini sudah mencakup operasi minimum yang dibutuhkan. Operasi umum lainnya adalah menyambung informasi baru di akhir suatu berkas, mengubah nama suatu berkas, dan lain-lain. Operasi dasar ini kemudian digabung untuk melakukan operasi lainnya. Sebagai contoh misalnya kita menginginkan salinan dari suatu berkas, atau menyalin berkas ke peralatan I/O lainnya seperti *printer*, dengan cara membuat berkas lalu membaca dari berkas lama dan menulis ke berkas yang baru. Hampir semua operasi pada berkas melibatkan pencarian berkas pada direktori. Untuk menghindari pencarian yang lama, kebanyakan sistem akan membuka berkas apabila berkas tersebut digunakan secara aktif. Sistem operasi akan menyimpan tabel kecil yang berisi informasi semua berkas yang dibuka yang disebut "tabel berkas terbuka". Ketika berkas sudah tidak digunakan lagi dan sudah ditutup oleh yang menggunakan, maka sistem operasi mengeluarkan berkas tersebut dari tabel berkas terbuka. Beberapa sistem terkadang langsung membuka berkas ketika berkas tersebut digunakan dan otomatis menutup berkas tersebut jika program atau pemakainya dimatikan. Tetapi pada sistem lainnya terkadang membutuhkan pembukaan berkas secara tersurat dengan *system call* (*open*) sebelum berkas dapat digunakan. Implementasi dari buka dan tutup berkas dalam lingkungan dengan banyak pengguna seperti UNIX, lebih rumit. Dalam sistem seperti itu pengguna yang membuka berkas mungkin lebih dari satu dan pada waktu yang hampir bersamaan. Umumnya sistem operasi menggunakan tabel internal dua level. Ada tabel yang mendata proses mana saja yang membuka berkas tersebut, kemudian tabel tersebut menunjuk ke tabel yang lebih besar yang berisi informasi yang berdiri sendiri seperti lokasi berkas pada disk, tanggal akses dan ukuran berkas. Biasanya tabel tersebut juga memiliki data berapa banyak proses yang membuka berkas tersebut. Jadi, pada dasarnya ada beberapa informasi yang terkait dengan pembukaan berkas yaitu:

- *Penunjuk Berkas* : Pada sistem yang tidak mengikutkan batas berkas sebagai bagian dari system call baca dan tulis, sistem tersebut harus mengikuti posisi dimana terakhir proses baca dan tulis sebagai penunjuk. Penunjuk ini unik untuk setiap operasi pada berkas, maka dari itu harus disimpan terpisah dari atribut berkas yang ada pada disk.
- *Penghitung berkas yang terbuka* : Setelah berkas ditutup, sistem harus mengosongkan kembali tabel berkas yang dibuka yang digunakan oleh berkas tadi atau tempat di tabel akan habis. Karena mungkin ada beberapa proses yang membuka berkas secara bersamaan dan sistem harus menunggu sampai berkas tersebut ditutup sebelum mengosongkan tempatnya di tabel. Penghitung ini mencatat banyaknya berkas yang telah dibuka dan ditutup, dan menjadi nol ketika yang terakhir membaca berkas menutup berkas tersebut barulah sistem dapat mengosongkan tempatnya di tabel.
- *Lokasi berkas pada disk* : Kebanyakan operasi pada berkas memerlukan sistem untuk

mengubah data yang ada pada berkas. Informasi mengenai lokasi berkas pada disk disimpan di memori agar menghindari banyak pembacaan pada disk untuk setiap operasi.

Beberapa sistem operasi menyediakan fasilitas untuk memetakan berkas ke dalam memori pada system memori virtual. Hal tersebut mengizinkan bagian dari berkas ditempatkan pada suatu alamat di memori virtual. Operasi baca dan tulis pada memori dengan alamat tersebut dianggap sebagai operasi baca dan tulis pada berkas yang ada di alamat tersebut. Menutup berkas mengakibatkan semua data yang ada pada alamat memori tersebut dikembalikan ke disk dan dihilangkan dari memori virtual yang digunakan oleh proses.

4) Jenis Berkas

Pertimbangan utama dalam perancangan sistem berkas dan seluruh sistem operasi, apakah sistem operasi harus mengenali dan mendukung jenis berkas. Jika suatu sistem operasi mengenali jenis dari berkas, maka ia dapat mengoperasikan berkas tersebut. Contoh apabila pengguna mencoba mencetak berkas yang merupakan kode biner dari program yang pasti akan menghasilkan sampah, hal ini dapat dicegah apabila sistem operasi sudah diberitahu bahwa berkas tersebut merupakan kode biner. Teknik yang umum digunakan dalam implementasi jenis berkas adalah menambahkan jenis berkas dalam nama berkas. Nama dibagi dua, nama dan akhiran (ekstensi), biasanya dipisahkan dengan karakter titik. Sistem menggunakan akhiran tersebut untuk mengindikasikan jenis berkas dan jenis operasi yang dapat dilakukan pada berkas tersebut. Sebagai contoh hanya berkas yang berakhiran *bat*, *.exe* atau *.com* yang bisa dijalankan (eksekusi). Program aplikasi juga menggunakan akhiran tersebut untuk mengenal berkas yang dapat dioperasikannya. Akhiran ini dapat ditimpa atau diganti jika diperbolehkan oleh sistem operasi. Beberapa sistem operasi menyertakan dukungan terhadap akhiran, tetapi beberapa menyerahkan kepada aplikasi untuk mengatur akhiran berkas yang digunakan, sehingga jenis dari berkas dapat menjadi petunjuk aplikasi apa yang dapat mengoperasikannya. Sistem UNIX tidak dapat menyediakan dukungan untuk akhiran berkas karena menggunakan angka ajaib yang disimpan di depan berkas untuk mengenali jenis berkas. Tidak semua berkas memiliki angka ini, jadi sistem tidak bisa bergantung pada informasi ini. Tetapi UNIX memperbolehkan akhiran berkas tetapi hal ini tidak dipaksakan atau tergantung sistem operasi, kebanyakan hanya untuk membantu pengguna mengenali jenis isi dari suatu berkas.

Tabel Jenis Berkas

Jenis berkas	**Akhiran**	**Fungsi**
executable	exe, com, bat,	bin program yang siap dijalankan
objek	obj, o	bahasa mesin, kode terkompilasi
kode asal (source code)	c, cc, pas, java, asm, a	kode asal dari berbagai bahasa
batch	bat, sh	perintah pada shell
text	txt, doc	data text, document
pengolah kata	wpd, tex, doc	format jenis pengolah data
library	lib, a, DLL	library untuk rutin program
print, gambar	ps, dvi, gif	format aSCII atau biner untuk dicetak
archive	arc, zip, tar	beberapa berkas yang dikumpulkan menjadi satu berkas. Terkadang dimampaatkan untukpenyimpanan

5) Struktur Berkas

Kita juga dapat menggunakan jenis berkas untuk mengidentifikasi struktur dalam dari berkas. Berkas berupa source dan objek memiliki struktur yang cocok dengan harapan program yang membaca berkas tersebut. Suatu berkas harus memiliki struktur yang dikenali oleh sistem operasi. Sebagai contoh, system operasi menginginkan suatu berkas yang dapat dieksekusi memiliki struktur tertentu agar dapat diketahui dimana berkas tersebut akan ditempatkan di memori dan di mana letak instruksi pertama berkas tersebut. Beberapa sistem operasi mengembangkan ide ini sehingga mendukung beberapa struktur berkas, dengan beberapa operasi khusus untuk memanipulasi berkas dengan struktur tersebut. Kelemahan memiliki dukungan terhadap beberapa struktur berkas adalah: Ukuran dari sistem operasi dapat menjadi besar, jika sistem operasi mendefinisikan lima struktur berkas yang berbeda maka ia perlu menampung kode untuk yang diperlukan untuk mendukung semuanya. Setiap berkas harus dapat menerapkan salah satu struktur berkas tersebut. Masalah akan timbul ketika terdapat aplikasi yang membutuhkan struktur informasi yang tidak didukung oleh sistem operasi tersebut. Beberapa sistem operasi

menerapkan dan mendukung struktur berkas sedikit struktur berkas. Pendekatan ini digunakan pada MS-DOS dan UNIX. UNIX menganggap setiap berkas sebagai urutan 8-bit byte, tidak ada interpretasi sistem operasi terhadap dari bit-bit ini. Skema tersebut menawarkan eksibilitas tinggi tetapi dukungan yang terbatas. Setiap aplikasi harus menambahkan sendiri kode untuk menerjemahkan berkas masukan ke dalam struktur yang sesuai. Walau bagaimana pun juga sebuah sistem operasi harus memiliki minimal satu struktur berkas yaitu untuk berkas yang dapat dieksekusi sehingga sistem dapat memuat berkas dalam memori dan menjalankannya. Sangat berguna bagi sistem operasi untuk mendukung struktur berkas yang sering digunakan karena akan menghemat pekerjaan pemrogram. Terlalu sedikit struktur berkas yang didukung akan mempersulit pembuatan program, terlalu banyak akan membuat sistem operasi terlalu besar dan pemrogram akan bingung.

6) Struktur Berkas Pada Disk

Menempatkan batas dalam berkas dapat menjadi rumit bagi sistem operasi. Sistem disk biasanya memiliki ukuran blok yang sudah ditetapkan dari ukuran sektor. Semua I/O dari disk dilakukan dalam satuan blok dan semua blok (’*physical record* ’) memiliki ukuran yang sama. Tetapi ukuran dari ’ *physical record* ’ tidak akan sama dengan ukuran ’ *logical record* ’. Ukuran dari ’ *logical record*’ akan bervariasi. Memuatkan beberapa ’*logical record* ’ ke dalam ’*physical record* ’ merupakan solusi umum dari masalah ini. Sebagai contoh pada sistem operasi UNIX, semua berkas didefinisikan sebagai kumpulan byte. Setiap byte dialamatkan menurut batasnya dari awal berkas sampai akhir. Pada kasus ini ukuran ’ *logical record* ’ adalah 1 byte. Sistem berkas secara otomatis memuatkan byte-byte tersebut kedalam blok pada disk. Ukuran ’ *logical record* ’, ukuran blok pada disk, dan teknik untuk memuatkannya menjelaskan berapa banyak ’ *logical record* ’ dalam tiap-tiap ’physical record’. Teknik memuatkan dapat dilakukan oleh aplikasi pengguna atau oleh sistem operasi. Berkas juga dapat dianggap sebagai urutan dari beberapa blok pada disk. Konversi dari ’ *logical record* ’ ke ’physical record’ merupakan masalah perangkat lunak.

Tempat pada disk selalu berada pada blok, sehingga beberapa bagian dari blok terakhir yang ditempati berkas dapat terbuang. Jika setiap blok berukuran 512 byte, sebuah berkas berukuran 1.949 byte akan menempati empat blok (2.048 byte) dan akan tersisa 99 byte pada blok terakhir. Byte yang terbuang tersebut dipertahankan agar ukuran dari unit tetap blok bukan byte disebut fragmentasi dalam disk. Semua sistem berkas pasti mempunyai fragmentasi dalam disk, semakin besar ukuran blok akan semakin besar fragmentasi dalam disknya.

7) Penggunaan Berkas Secara Bersama-sama

Konsistensi semantik adalah parameter yang penting untuk evaluasi sistem berkas yang mendukung penggunaan berkas secara bersama. Hal ini juga merupakan karakterisasi dari sistem yang menspesifikasi semantik dari banyak pengguna yang mengakses berkas secara bersama sama. Lebih khusus, semantik ini seharusnya dapat menspesifikasi kapan suatu modifikasi suatu data oleh satu pengguna dapat diketahui oleh pengguna lain. Terdapat beberapa macam konsistensi semantik. Di bawah ini akan dijelaskan kriteria yang digunakan dalam UNIX. Berkas sistem UNIX mengikuti konsistensi semantik:

- Penulisan ke berkas yang dibuka oleh pengguna dapat dilihat langsung oleh pengguna lain yang sedang mengakses ke berkas yang sama.
- Terdapat bentuk pembagian dimana pengguna membagi pointer lokasi ke berkas tersebut. Sehingga perubahan pointer satu pengguna akan mempengaruhi semua pengguna *sharing* nya.

3. Metode Akses

1) Akses Secara Berurutan

Ketika digunakan, informasi penyimpanan berkas harus dapat diakses dan dibaca ke dalam memori komputer. Beberapa sistem hanya menyediakan satu metode akses untuk berkas. Pada system yang lain, contohnya IBM, terdapat banyak dukungan metode akses yang berbeda. Masalah pada sistem tersebut adalah memilih yang mana yang tepat untuk digunakan pada satu aplikasi tertentu. *Sequential Access* merupakan metode yang paling sederhana. Informasi yang disimpan dalam berkas diproses berdasarkan urutan. Operasi dasar pada suatu berkas adalah tulis dan baca. Operasi baca membaca berkas dan meningkatkan pointer berkas selama di jalur lokasi I/O. Operasi tulis menambahkan ke akhir berkas dan meningkatkan ke akhir berkas yang baru. Metode ini didasarkan pada tape model sebuah berkas, dan dapat bekerja pada kedua jenis *device* akses (urut mau

pun acak).

2) Akses Langsung

Direct Access merupakan metode yang membiarkan program membaca dan menulis dengan cepat pada berkas yang dibuat dengan *fixed-length logical order* tanpa adanya urutan. Metode ini sangat berguna untuk mengakses informasi dalam jumlah besar. Biasanya database memerlukan hal seperti ini. Operasi berkas pada metode ini harus dimodifikasi untuk menambahkan nomor blok sebagai parameter. Pengguna menyediakan nomor blok ke sistem operasi biasanya sebagai nomor blok relatif, yaitu indeks relatif terhadap awal berkas. Penggunaan nomor blok relatif bagi sistem operasi adalah untuk memutuskan lokasi berkas diletakkan dan membantu mencegah pengguna dari pengaksesan suatu bagian sistem berkas yang bukan bagian pengguna tersebut.

3) Akses Dengan Menggunakan Indeks

Metode ini merupakan hasil dari pengembangan metode *direct access*. Metode ini memasukkan indeks untuk mengakses berkas. Jadi untuk mendapatkan suatu informasi suatu berkas, kita mencari dahulu di indeks, lalu menggunakan pointer untuk mengakses berkas dan mendapatkan informasi tersebut. Namun metode ini memiliki kekurangan, yaitu apabila berkas-berkas besar, maka indeks berkas tersebut akan semakin besar. Jadi solusinya adalah dengan membuat 2 indeks, indeks primer dan indeks sekunder. Indeks primer memuat pointer ke indeks sekunder, lalu indeks sekunder menunjuk ke data yang dimaksud.

4. Struktur Direktori

1) Operasi Pada Direktori

Operasi-operasi yang dapat dilakukan pada direktori adalah:

- Mencari berkas, kita dapat menemukan sebuah berkas didalam sebuah struktur direktori. Karena berkas-berkas memiliki nama simbolik dan nama yang sama dapat mengindikasikan keterkaitan antara setiap berkas-berkas tersebut, mungkin kita berkeinginan untuk dapat menemukan seluruh berkas yang nama-nama berkas membentuk pola khusus.
- Membuat berkas, kita dapat membuat berkas baru dan menambahkan berkas tersebut kedalam direktori.
- Menghapus berkas, apabila berkas sudah tidak diperlukan lagi, kita dapat menghapus berkas tersebut dari direktori.
- Menampilkan isi direktori, kita dapat menampilkan seluruh berkas dalam direktori, dan kandungan isi direktori untuk setiap berkas dalam daftar tersebut.
- Mengganti nama berkas, karena nama berkas merepresentasikan isi dari berkas kepada user, maka user dapat merubah nama berkas ketika isi atau penggunaan berkas berubah. Perubahan nama dapat merubah posisi berkas dalam direktori.
- Melintasi sistem berkas, ini sangat berguna untuk mengakses direktori dan berkas didalam struktur direktori.

2) Direktori Satu Tingkat

Ini adalah struktur direktori yang paling sederhana. Semua berkas disimpan di dalam direktori yang sama. Struktur ini tentunya memiliki kelemahan jika jumlah berkasnya bertambah banyak, karena tiap berkas mesti memiliki nama yang unik.

3) Direktori Dua Tingkat

Kelemahan yang ada pada direktori tingkat satu dapat diatas pada sistem direktori dua tingkat. Caranya ialah dengan membuat direktori secara terpisah. Pada direktori tingkat dua, setiap pengguna memiliki direktori berkas sendiri (UFD). Setiap UFD memiliki struktur yang serupa, tapi hanya berisi berkas-berkas dari seorang pengguna. Ketika seorang pengguna *login*, *master* direktori berkas (MFD) dicari. Isi dari MFD adalah indeks dari nama pengguna atau nomor rekening, dan tiap entri menunjuk pada UFD untuk pengguna tersebut. Ketika seorang pengguna ingin mengakses suatu berkas, hanya UFD-nya sendiri yang diakses. Jadi pada setiap UFD yang berbeda, boleh terdapat nama berkas yang sama.

4) Direktori Dengan Struktur Tree

Struktur direktori dua tingkat bisa dikatakan sebagai pohon dua tingkat. Sebuah direktori dengan struktur pohon memiliki sejumlah berkas atau subdirektori lagi. Pada penggunaan

yang normal setiap pengguna memiliki direktorinya sendiri-sendiri. Selain itu pengguna tersebut dapat memiliki subdirektori sendiri lagi. Dalam struktur ini dikenal dua istilah, yaitu *path* relatif dan *path* mutlak. *Path* relatif adalah *path* yang dimulai dari direktori yang aktif. Sedangkan *path* mutlak adalah *path* yang dimulai dari direktori akar.

5) Direktori Dengan Struktur Acyclic-Graph

Jika ada sebuah berkas yang ingin diakses oleh dua pengguna atau lebih, maka struktur ini menyediakan fasilitas " *sharing* ", yaitu penggunaan sebuah berkas secara bersama-sama. Hal ini tentunya berbeda dengan struktur pohon, dimana pada struktur tersebut penggunaan berkas atau direktori secara bersama-sama dilarang. Pada struktur "*Acyclic-Graph* ", penggunaan berkas atau direktori secara bersama-sama diperbolehkan. Tapi pada umumnya struktur ini mirip dengan struktur pohon.

6) Direktori Dengan Struktur Graph

Masalah yang sangat utama pada struktur direktori "*Acyclic-Graph* " adalah kemampuan untuk memastikan tidak-adanya siklus. Jika pada struktur 2 tingkat direktori, seorang pengguna dapat membuat subdirektori, maka akan kita dapatkan direktori dengan struktur pohon. Sangatlah mudah untuk tetap mempertahankan sifat pohon setiap kali ada penambahan berkas atau subdirektori pada direktori dengan struktur pohon. Tapi jika kita menambahkan sambungan pada direktori dengan struktur pohon, maka akan kita dapatkan direktori dengan struktur *graph* sederhana. Proses pencarian pada direktori dengan struktur "*Acyclic-Graph*", apabila tidak ditangani dengan baik (algoritma tidak bagus) dapat menyebabkan proses pencarian yang berulang dan menghabiskan banyak waktu. Oleh karena itu, diperlukan skema pengumpulan sampah ("*garbage-collection scheme* "). Skema ini menyangkut memeriksa seluruh sistem berkas dengan menandai tiap berkas yang dapat diakses. Kemudian mengumpulkan apa pun yang tidak ditandai sebagai tempat kosong. Hal ini tentunya dapat menghabiskan banyak waktu.

5. Proteksi Berkas

Ketika kita menyimpan informasi dalam sebuah sistem komputer, ada dua hal yang harus menjadi perhatian utama kita. Hal tersebut adalah :

a. Reabilitas dari sebuah sistem

Maksud dari reabilitas system adalah kemampuan sebuah sistem untuk melindungi informasi yang telah disimpan agar terhindar dari kerusakan, dalam hal ini adalah perlindungan secara fisik pada sebuah berkas. Reabilitas sistem dapat dijaga dengan membuat cadangan dari setiap berkas secara manual atau pun otomatis, sesuai dengan layanan yang dari sebuah sistem operasi.

b. Proteksi (Perlindungan) terhadap sebuah berkas

Perlindungan terhadap berkas dapat dilakukan dengan berbagai macam cara. Pada bagian ini, kita akan membahas secara detil mekanisme yang diterapkan dalam melindungi sebuah berkas.

1) Tipe Akses Pada Berkas

Salah satu cara untuk melindungi berkas dalam komputer kita adalah dengan melakukan pembatasan akses pada berkas tersebut. Pembatasan akses yang dimaksudkan adalah kita, sebagai pemilik dari sebuah berkas, dapat menentukan operasi apa saja yang dapat dilakukan oleh pengguna lain terhadap berkas tersebut. Pembatasan ini berupa sebuah permission atau pun not permitted operation, tergantung pada kebutuhan pengguna lain terhadap berkas tersebut. Di bawah ini adalah beberapa operasi berkas yang dapat diatur aksesnya:

- Read: Membaca dari berkas
- Write: Menulis berkas
- Execute: Meload berkas kedalam memori untuk dieksekusi.
- Append: Menambahkan informasi kedalam berkas di akhir berkas.
- Delete: Menghapus berkas.
- List: Mendaftar properti dari sebuah berkas.
- Rename: Mengganti nama sebuah berkas.
- Copy: Menduplikasikan sebuah berkas.
- Edit: Mengedit sebuah berkas.

Selain operasi-operasi berkas diatas, perlindungan terhadap berkas dapat dilakukan dengan mekanisme yang lain. Namun setiap mekanisme memiliki kelebihan dan kekurangan. Pemilihan mekanisme sangatlah tergantung pada kebutuhan dan spesifikasi sistem.

2) Akses List dan Group

Hal yang paling umum dari sistem proteksi adalah membuat akses tergantung pada identitas pengguna yang bersangkutan. Implementasi dari akses ini adalah dengan membuat daftar akses yang berisi keterangan setiap pengguna dan keterangan akses berkas dari pengguna yang bersangkutan. Daftar akses ini akan diperiksa setiap kali seorang pengguna meminta akses ke sebuah berkas. Jika pengguna tersebut memiliki akses yang diminta pada berkas tersebut, maka diperbolehkan untuk mengakses berkas tersebut. Proses ini juga berlaku untuk hal yang sebaliknya. Akses pengguna terhadap berkas akan ditolak, dan sistem operasi akan mengeluarkan peringatan *Protection Violation*. Masalah baru yang timbul adalah panjang dari daftar akses yang harus dibuat. Seperti telah disebutkan, kita harus mendaftarkan semua pengguna dalam daftar akses tersebut hanya untuk akses pada satu berkas saja. Oleh karena itu, teknik ini mengakibatkan 2 konsekuensi yang tidak dapat dihindarkan:

- Pembuatan daftar yang sangat panjang ini dapat menjadi pekerjaan yang sangat melelahkan sekaligus membosankan, terutama jika jumlah pengguna dalam sistem tidak dapat diketahui secara pasti.
- Manajemen ruang *harddisk* yang lebih rumit, karena ukuran sebuah direktori dapat berubah-ubah, tidak memiliki ukuran yang tetap.

Kedua konsekuensi diatas melahirkan sebuah teknik daftar akses yang lebih singkat. Teknik ini mengelompokkan pengguna berdasarkan tiga kategori:

- Owner: User yang membuat berkas.
- Group: Sekelompok pengguna yang memiliki akses yang sama terhadap sebuah berkas, atau men-share sebuah berkas.
- Universe: Seluruh pengguna yang terdapat dalam sistem komputer.

Dengan adanya pengelompokkan pengguna seperti ini, maka kita hanya membutuhkan tiga *field* untuk melindungi sebuah berkas. Field ini diasosiasikan dengan 3 buah bit untuk setiap kategori. Dalam system UNIX dikenal bit rwx dengan bit r untuk mengontrol akses baca, bit w sebagai kontrol menulis dan bit x sebagai bit kontrol untuk pengeksekusian. Setiap *field* dipisahkan dengan *field separator*. Dibawah ini adalah contoh dari sistem proteksi dengan daftar akses pada sistem UNIX.

Contoh sistem daftar akses pada UNIX

drwx rwx rwx 1 pbg staff 512 Apr 16 22.25 bekas.txt

owner group universe group owner ukuran waktu nama berkas

3) Pendekatan Sistem Proteksi yang Lain

Sistem proteksi yang lazim digunakan pada sistem komputer selain diatas adalah dengan menggunakan password (kata sandi) pada setiap berkas. Beberapa sistem operasi mengimplementasikan hal ini bukan hanya pada berkas, melainkan pada direktori. Dengan sistem ini, sebuah berkas tidak akan dapat diakses selain oleh pengguna yang telah mengetahui password untuk berkas tersebut. Akan tetapi, masalah yang muncul dari sistem ini adalah jumlah password yang harus diingat oleh seorang pengguna untuk mengakses berkas dalam sebuah sistem operasi. Masalah yang lain adalah keamanan password itu sendiri. Jika hanya satu password yang digunakan, maka kebocoran password tersebut merupakan malapetaka bagi pengguna yang bersangkutan. Sekali lagi, maka kita harus menggunakan password yang berbeda untuk setiap tingkatan yang berbeda.

6. Struktur Sistem Berkas

Disk menyediakan sebagian besar tempat penyimpanan dimana sistem berkas dikelola dikelola. Untuk meningkatkan efisiensi I/O, pengiriman data antara memori dan disk dilakukan dalam setiap blok. Setiap blok merupakan satu atau lebih sektor. Setiap disk memiliki ukuran yang berbeda-beda, biasanya berukuran 512 bytes. Disk memiliki dua karakteristik penting yang menjadikan disk sebagai media yang tepat untuk menyimpan berbagai macam berkas, yaitu:

- Disk tersebut dapat ditulis ulang di disk tersebut, hal ini memungkinkan untuk membaca,

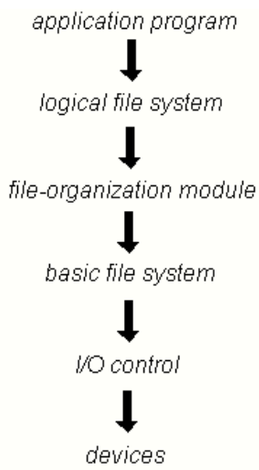
memodifikasi, dan menulis di disk tersebut.

- Dapat diakses langsung ke setiap blok di disk. Hal ini memudahkan untuk mengakses setiap berkas baik secara berurut mau pun tidak berurut, dan berpindah dari satu berkas ke berkas lain dengan hanya mengangkat head disk dan menunggu disk berputar.

1) Organisasi Sistem Berkas

Sistem operasi menyediakan sistem berkas agar data mudah disimpan, diletakkan dan diambil kembali dengan mudah. Terdapat dua masalah desain dalam membangun suatu sistem berkas. Masalah pertama adalah definisi dari sistem berkas. Hal ini mencakup definisi berkas dan atributnya, operasi ke berkas, dan struktur direktori dalam mengorganisasikan berkas-berkas. Masalah kedua adalah membuat algoritma dan struktur data yang memetakan struktur logikal sistem berkas ke tempat penyimpanan sekunder. Pada dasarnya sistem berkas tersusun atas beberapa tingkatan, yaitu (dari yang terendah):

- *I/O control*, terdiri atas *driver device* dan *interrupt handler*. *Driver device* adalah perantara komunikasi antara sistem operasi dengan perangkat keras.
- *Basic file system* , diperlukan untuk mengeluarkan perintah generik ke *device driver* untuk baca dan tulis pada suatu blok dalam disk.
- *File-organization module*, informasi tentang alamat logika dan alamat fisik dari berkas tersebut. Modul ini juga mengatur sisa disk dengan melacak alamat yang belum dialokasikan dan menyediakan alamat tersebut saat user ingin menulis berkas ke dalam disk.
- *Logical file system* , tingkat ini berisi informasi tentang simbol nama berkas, struktur dari direktori, dan proteksi dan sekuriti dari berkas tersebut.



Gambar 7.1. Lapisan pada sistem berkas.

2) Mounting Sistem Berkas

Seperti halnya sebuah berkas yang harus dibuka terlebih dahulu sebelum digunakan, sistem berkas harus di *mount* terlebih dahulu sebelum sistem berkas tersebut siap untuk memproses dalam sistem. Sistem operasi diberikan sebuah alamat mounting (*mount point*) yang berisi nama *device* yang bersangkutan dan lokasi dari *device* tersebut.

7. Metode Alokasi Berkas

Kemudahan dalam mengakses langsung suatu disk memberikan eksibilitas dalam mengimplementasikan sebuah berkas. Masalah utama dalam implementasi adalah bagaimana mengalokasikan berkas-berkas ke dalam disk, sehingga disk dapat terutilisasi dengan efektif dan berkas dapat diakses dengan cepat. Ada tiga metode utama, menurut buku "*Applied Operating System Concepts: First Edition* " oleh Avi Silberschatz, Peter Galvin dan Greg Gagne untuk mengalokasi ruang disk yang digunakan secara luas yaitu *contiguous* , *linked* , dan *indexed* .

1) Alokasi Secara Berdampingan (Contiguous Allocation)

Metode ini menempatkan setiap berkas pada satu himpunan blok yang berurut di dalam disk. Alamat disk menyatakan sebuah urutan linier. Dengan urutan linier ini maka *head disk* hanya bergerak jika mengakses dari sektor terakhir suatu silinder ke sektor pertama silinder berikutnya. Waktu pencarian (*seek time*) dan banyak *disk seek* yang dibutuhkan untuk mengakses berkas yang dialokasi secara berdampingan ini sangat minimal. Contoh dari sistem operasi yang menggunakan *contiguous allocation* adalah IBM VM/CMS karena

pendekatan ini menghasilkan performa yang baik. *Contiguous allocation* dari suatu berkas diketahui melalui alamat dan panjang disk (dalam unit blok) dari blok pertama. Jadi, misalkan ada berkas dengan panjang n blok dan mulai dari lokasi b maka berkas tersebut menempati blok b, b+1, b+2, ..., b+n-1. Direktori untuk setiap berkas mengindikasikan alamat blok awal dan panjang area yang dialokasikan untuk berkas tersebut. Terdapat dua macam cara untuk mengakses berkas yang dialokasi dengan metode ini, yaitu:

- *Sequential access* , sistem berkas mengetahui alamat blok terakhir dari disk dan membaca blok berikutnya jika diperlukan.
- *Direct access* , untuk akses langsung ke blok i dari suatu berkas yang dimulai pada blok b, dapat langsung mengakses blok b+i.

Kesulitan dari metode alokasi secara berdampingan ini adalah menemukan ruang untuk berkas baru. Masalah pengalokasian ruang disk dengan metode ini merupakan aplikasi masalah *dynamic storage-allocation* (alokasi tempat penyimpanan secara dinamik), yaitu bagaimana memenuhi permintaan ukuran n dari daftar ruang kosong. Strategi-strategi yang umum adalah *first fit* dan *best fit* . Kedua strategi tersebut mengalami masalah fragmentasi eksternal, dimana jika berkas dialokasi dan dihapus maka ruang kosong disk terpecah menjadi kepingan-kepingan kecil. Hal ini akan menjadi masalah ketika banyak kepingan kecil tidak dapat memenuhi permintaan karena kepingan-kepingan kecil tidak cukup besar untuk menyimpan berkas, sehingga terdapat banyak ruang yang terbuang. Masalah yang lain adalah menentukan berapa banyak ruang yang diperlukan untuk suatu berkas. Ketika berkas dibuat, jumlah dari ruang berkas harus ditentukan dan dialokasikan. Jika ruang yang dialokasikan terlalu kecil maka berkas tidak dapat diperbesar dari yang telah dialokasikan. Untuk mengatasi hal ini ada dua kemungkinan. Pertama, program pengguna dapat diakhiri dengan pesan error yang sesuai. Lalu, pengguna harus mengalokasikan tambahan ruang dan menjalankan programnya lagi, tetapi hal ini *cost* yang dihasilkan lebih mahal. Untuk mengatasinya, pengguna dapat melakukan estimasi yang lebih terhadap ruang yang harus dialokasikan pada suatu berkas tetapi hal ini akan membuang ruang disk. Kemungkinan yang kedua adalah mencari ruang kosong yang lebih besar, lalu menyalin isi dari berkas ke ruang yang baru dan mengkosongkan ruang yang sebelumnya. Hal ini menghabiskan waktu yang cukup banyak. Walau pun jumlah ruang yang diperlukan untuk suatu berkas dapat diketahui, pengalokasian awal akan tidak efisien. Ukuran berkas yang bertambah dalam periode yang lama harus dapat dialokasi ke ruang yang cukup untuk ukuran akhirnya, walau pun ruang tersebut tidak akan digunakan dalam waktu yang lama. Hal ini akan menyebabkan berkas dengan jumlah fragmentasi internal yang besar. Untuk menghindari hal-hal tersebut, beberapa sistem operasi memodifikasi skema metode alokasi secara berdampingan, dimana kepingan kecil yang berurut dalam ruang disk diinisialisasi terlebih dahulu, kemudian ketika jumlah ruang disk kurang besar, kepingan kecil yang berurut lainnya, ditambahkan pada alokasi awal. Kejadian seperti ini disebut perpanjangan. Fragmentasi internal masih dapat terjadi jika perpanjangan-perpanjangan ini terlalu besar dan fragmentasi eksternal masih menjadi masalah begitu perpanjangan-perpanjangan dengan ukuran yang bervariasi dialokasikan dan didealokasi.

2) Alokasi Secara Berangkai (Linked Allocation)

Metode ini menyelesaikan semua masalah yang terdapat pada contiguous allocation. Dengan metode ini, setiap berkas merupakan *linked list* dari blok-blok disk, dimana blok-blok disk dapat tersebar di dalam disk. Setiap direktori berisi sebuah penunjuk (*pointer*) ke awal dan akhir blok sebuah berkas. Setiap blok mempunyai penunjuk ke blok berikutnya. Untuk membuat berkas baru, kita dengan mudah membuat masukan baru dalam direktori. Dengan metode ini, setiap direktori masukan mempunyai penunjuk ke awal blok disk dari berkas. Penunjuk ini diinisialisasi menjadi *nil* (nilai penunjuk untuk akhir dari list) untuk menandakan berkas kosong. Ukurannya juga diset menjadi 0. Penulisan suatu berkas menyebabkan ditemukannya blok yang kosong melalui sistem manajemen ruang kosong (*free-space management system*), dan blok baru ini ditulis dan disambungkan ke akhir berkas. Untuk membaca suatu berkas, cukup dengan membaca blok-blok dengan mengikuti pergerakan penunjuk. Metode ini tidak mengalami fragmentasi eksternal dan kita dapat menggunakan blok kosong yang terdapat dalam daftar ruang kosong untuk memenuhi permintaan pengguna. Ukuran dari berkas tidak perlu ditentukan ketika berkas pertama kali dibuat, sehingga ukuran berkas dapat bertambah selama masih ada blok-blok kosong.

Metode ini tentunya mempunyai kerugian, yaitu metode ini hanya dapat digunakan secara efektif untuk pengaksesan berkas secara sequential (*sequential-access file*). Untuk mencari blok ke- *i* dari suatu berkas, harus dimulai dari awal berkas dan mengikuti penunjuk sampai

berada di blok ke- *i*. Setiap akses ke penunjuk akan membaca disk dan kadang melakukan pencarian disk (*disk seek*). Hal ini sangat tidak efisien untuk mendukung kemampuan akses langsung (*direct-access*) terhadap berkas yang menggunakan metode alokasi link. Kerugian yang lain dari metode ini adalah ruang yang harus disediakan untuk penunjuk. Solusi yang umum untuk masalah ini adalah mengumpulkan blok-blok persekutuan terkecil dinamakan *clusters* dan mengalokasikan *cluster-cluster* daripada blok. Dengan solusi ini maka, penunjuk menggunakan ruang disk berkas dengan persentase yang sangat kecil. Metode ini membuat *mapping* logikal ke fisikal blok tetap sederhana, tetapi meningkatkan *disk throughput* dan memperkecil ruang yang diperlukan untuk alokasi blok dan management daftar kosong (*free-list management*). Akibat dari pendekatan ini adalah meningkatnya fragmentasi internal, karena lebih banyak ruang yang terbuang jika sebuah *cluster* sebagian penuh daripada ketika sebuah blok sebagian penuh. Alasan *cluster* digunakan oleh kebanyakan sistem operasi adalah kemampuannya yang dapat meningkatkan waktu akses disk untuk berbagai macam algoritma. Masalah yang lain adalah masalah daya tahan metode ini. Karena semua berkas saling berhubungan dengan penunjuk yang tersebar di semua bagian disk, apa yang terjadi jika sebuah penunjuk rusak atau hilang. Hal ini menyebabkan berkas menyambung ke daftar ruang kosong atau ke berkas yang lain. Salah satu solusinya adalah menggunakan *linked list* ganda atau menyimpan nama berkas dan nomor relative blok dalam setiap blok, tetapi solusi ini membutuhkan perhatian lebih untuk setiap berkas. Variasi penting dari metode ini adalah penggunaan *file allocation table (FAT)* , yang digunakan oleh sistem operasi *MS-DOS* dan *OS/2* . Bagian awal disk pada setiap partisi disingkirkan untuk menempatkan tabelnya. Tabel ini mempunyai satu masukkan untuk setiap blok disk, dan diberi indeks oleh nomor blok. Masukkan direktori mengandung nomor blok dari blok awal berkas. Masukkan tabel diberi indeks oleh nomor blok itu lalu mengandung nomor blok untuk blok berikutnya dari berkas. Rantai ini berlanjut sampai blok terakhir, yang mempunyai nilai akhir berkas yang khusus sebagai masukkan tabel. Blok yang tidak digunakan diberi nilai 0. Untuk mengalokasi blok baru untuk suatu berkas hanya dengan mencari nilai 0 pertama dalam tabel, dan mengganti nilai akhir berkas sebelumnya dengan alamat blok yang baru. Metode pengalokasian FAT ini dapat menghasilkan jumlah pencarian *head disk* yang signifikan, jika berkas tidak di cache. *Head disk* harus bergerak dari awal partisi untuk membaca FAT dan menemukan lokasi blok yang ditanyakan, lalu menemukan lokasi blok itu sendiri. Kasus buruknya, kedua pergerakan terjadi untuk setiap blok. Keuntungannya waktu random akses meningkat, akibat dari *head disk* dapat mencari lokasi blok apa saja dengan membaca informasi dalam FAT.

3) Alokasi Dengan Indeks (Indexed Allocation)

Metode alokasi dengan berangkai dapat menyelesaikan masalah fragmentasi eksternal dan pendeklarasian ukuran dari metode alokasi berdampingan. Bagaimana pun tanpa FAT, metode alokasi berangkai tidak mendukung keefisiensian akses langsung, karena penunjuk ke bloknya berserakan dengan bloknya didalam disk dan perlu didapatkan secara berurutan. Metode alokasi dengan indeks menyelesaikan masalah ini dengan mengumpulkan semua penunjuk menjadi dalam satu lokasi yang dinamakan blok indeks (*index block*). Setiap berkas mempunyai blok indeks, yang merupakan sebuah larik *array* dari alamat-alamat disk-blok. Direktori mempunyai alamat dari blok indeks. Ketika berkas dibuat, semua penunjuk dalam blok indeks di set menjadi *nil*. Ketika blok ke- *I* pertama kali ditulis, sebuah blok didapat dari pengatur ruang kosong *free-space manager* dan alamatnya diletakkan ke dalam blok indeks ke-*i*. Metode ini mendukung akses secara langsung, tanpa mengalami fragmentasi eksternal karena blok kosong mana pun dalam disk dapat memenuhi permintaan ruang tambahan. Tetapi metode ini dapat menyebabkan ada ruang yang terbuang. Penunjuk yang berlebihan dari blok indeks secara umum lebih besar dari yang terjadi pada metode alokasi berangkai. Mekanisme untuk menghadapi masalah berapa besar blok indeks yang diperlukan sebagai berikut:

- *Linked scheme* : untuk berkas-berkas yang besar, dilakukan dengan menyambung beberapa blok indeks menjadi satu.
- *Multilevel index* : sebuah varian dari representasi yang berantai adalah dengan menggunakan blok indeks level pertama menunjuk ke himpunan blok indeks level kedua, yang akhirnya menunjuk ke blok-blok berkas.
- *Combined scheme* : digunakan oleh sistem *BSD UNIX* yaitu dengan menetapkan 15 penunjuk dari blok indeks dalam blok indeksnya berkas. 12 penunjuk pertama menunjuk ke *direct blocks* yang menyimpan alamat-alamat blok yang berisi data dari berkas. 3 penunjuk berikutnya menunjuk ke *indirect blocks*. Penunjuk *indirect blok* yang pertama adalah alamat dari *single indirect block*, yang merupakan blok indeks

yang berisi alamat-alamat blok yang berisi data. Lalu ada penunjuk *double indirect block* yang berisi alamat dari sebuah blok yang berisi alamat-alamat blok yang berisi penunjuk ke blok data yang sebenarnya.

4) Kinerja Sistem Berkas

Salah satu kesulitan dalam membandingkan performa sistem adalah menentukan bagaimana system tersebut akan digunakan. Sistem yang lebih banyak menggunakan akses sekuensial (berurutan) akan memakai metode yang berbeda dengan sistem yang lebih sering menggunakan akses random (acak). Untuk jenis akses apa pun, alokasi yang berdampingan hanya memerlukan satu akses untuk mendapatkan sebuah blok disk. Karena kita dapat menyimpan *initial address* dari berkas di dalam memori, maka alamat disk pada blok ke *i* dapat segera dikalkulasi dan dibaca secara langsung.

Untuk alokasi berangkai *(linked list)*, kita juga dapat menyimpan alamat dari blok selanjutnya ke dalam memori, lalu membacanya secara langsung. Metode ini sangat baik untuk akses sekuensial, namun untuk akses langsung, akses menuju blok ke- *i* kemungkinan membutuhkan pembacaan disk sebanyak *I* kali. Masalah ini mengindikasikan bahwa alokasi berangkai sebaiknya tidak digunakan untuk aplikasi yang membutuhkan akses langsung. Oleh sebab itu, beberapa sistem mendukung akses langsung dengan menggunakan alokasi berdampingan (*contiguous allocation*), serta akses berurutan dengan alokasi berangkai. Untuk sistem-sistem tersebut, jenis akses harus dideklarasikan pada saat berkas dibuat. Berkas yang dibuat untuk akses sekuensial (berurutan) akan dirangkaikan dan tidak dapat digunakan untuk akses langsung. Berkas yang dibuat untuk akses langsung akan berdampingan dan dapat mendukung baik akses langsung mau pun akses berurutan, dengan mendeklarasikan jarak maksimum. Perhatikan bahwa sistem operasi harus mendukung struktur data dan algoritma yang sesuai untuk mendukung kedua metode alokasi di atas. Alokasi dengan menggunakan indeks lebih rumit lagi. Jika blok indeks telah terdapat dalam memori, akses dapat dilakukan secara langsung. Namun, menyimpan blok indeks dalam memori memerlukan ruang (*space*) yang besar. Jika ruang memori tidak tersedia, maka kita mungkin harus membaca blok indeks terlebih dahulu, baru kemudian blok data yang diinginkan. Untuk indeks dua tingkat, pembacaan dua blok indeks mungkin diperlukan. Untuk berkas yang berukuran sangat besar, mengakses blok di dekat akhir suatu berkas akan membutuhkan pembacaan seluruh blok indeks agar dapat mengikuti rantai penunjuk sebelum blok data dapat dibaca. Dengan demikian, performa alokasi dengan menggunakan indeks ditentukan oleh: struktur indeks, ukuran berkas, dan posisi dari blok yang diinginkan. Beberapa system mengkombinasikan alokasi berdampingan dengan alokasi indeks. Caranya adalah dengan menggunakan alokasi berdampingan untuk berkas berukuran kecil (3-4 blok), dan beralih secara otomatis ke alokasi indeks jika berkas semakin membesar.

8. Managemen Ruang Kosong (Free Space)

Semenjak hanya tersedia tempat yang terbatas pada disk maka sangat berguna untuk menggunakan kembali tempat dari berkas yang dihapus untuk berkas baru, jika dimungkinkan,karena pada media yang sekali tulis (media optik) hanya dimungkinkan sekali menulis dan menggunakannyanya kembali secara fisik tidak mungkin. Untuk mencatat tempat kosong pada disk, system mempunyai daftar tempat kosong (*free space list*). Daftar ini menyimpan semua blok disk yang kosong yang tidak dialokasikan pada sebuah berkas atau direktori. Untuk membuat berkas baru, sistem mencari ke daftar tersebut untuk mencarikan tempat kosong yang di butuhkan, lalu tempat tersebut dihilangkan dari daftar. Ketika berkas dihapus, alamat berkas tadi ditambahkan pada daftar.

1) Menggunakan Bit Vektor

Seringnya daftar raung kosong diimplementasikan sebagai bit map atau bit vektor. Tiap blok direpresentasikan sebagai 1 bit. Jika blok tersebut kosong maka isi bitnya 1 dan jika bloknya sedang dialokasikan maka isi bitnya 0. Sebagai contoh sebuah disk dimana blok 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 dan 27 adalah kosong, dan sisanya dialokasikan. Bit mapnya akan seperti berikut:

00111100111111000110000011100000...

Keuntungan utama dari pendekatan ini adalah relatif sederhana dan efisien untuk mencari blok pertama yang kosong atau berturut-turut n blok yang kosong pada disk. Banyak komputer yang menyediakan instruksi manipulasi bit yang dapat digunakan secara efektif untuk tujuan ini. Sebagai contohnya, dari keluarga prosesor Intel dimulai dari 80386 dan keluarga Motorola dimulai dari 68020 (prosesor yang ada di PC dan Macintosh) mempunyai instruksi yang mengembalikan jarak di *word* dari bit pertama dengan nilai 1.

Sistem operasi Apple Macintosh menggunakan metode bit vektor untuk mengalokasikan tempat pada disk. Dalam hal ini perangkat keras mendukung perangkat lunak tetapi bit vektor tidak efisien kecuali seluruh vektor disimpan dalam memori utama (dan ditulis di disk untuk kebutuhan pemulihan). Menyimpan dalam memori utama dimungkinkan untuk disk yang kecil pada mikro komputer, tetapi tidak untuk disk yang besar. Sebuah *disk* 1,3 GB dengan 512-byte blok akan membutuhkan *bit map* sebesar 332K untuk mencatat blok yang kosong.

2) Linked List

Pendekatan lain adalah untuk menghubungkan semua blok yang kosong, menyimpan *pointer* ke blok pertama yang kosong di tempat yang khusus pada disk dan menyimpannya di memori. Blok pertama ini menyimpan *pointer* ke blok kosong berikutnya dan seterusnya. Pada contoh sebelumnya kita akan menyimpan *pointer* ke blok ke 2 sebagai blok kosong pertama, blok 2 akan menyimpan *pointer* ke blok 3, yang akan menunjuk ke blok 4 dan seterusnya. Bagaimana pun metode ini tidak efisien karena untuk *traverse* daftar tesebut kita perlu membaca tiap blok yang membutuhkan waktu I/O. Untungnya traverse ini tidak sering digunakan. Umumnya, sistem operasi membutuhkan blok kosong untuk mengalokasikan blok tersebut ke berkas, maka blok pertama pada daftar ruang kosong digunakan.

3) Grouping

Modifikasi lainnya adalah dengan menyimpan alamat dari n blok kosong pada blok kosong pertama. Pada n-1 pertama dari blok-blok ini adalah kosong. Blok terakhir menyimpan alamat n blok kosong lainnya dan seterusnya. Keuntungannya dari implementasi seperti ini adalah alamat dari blok kosong yang besar sekali dapat ditemukan dengan cepat, tidak seperti pendekatan standar *linked-list* .

4) Counting

Pendekatan lain adalah dengan mengambil keuntungan dari fakta bahwa beberapa blok yang berkesinambungan akan dialokasikan atau dibebaskan secara simultan. Maka dari itu dari pada menyimpan daftar dari banyak alamat disk, kita dapat menyimpan alamat dari blok kosong pertama dan jumlah dari blok kosong yang berkesinambungan yang mengikuti blok kosong pertama. Tiap isi dari daftar menyimpan alamat disk dan penghitung (*counter*). Meski pun setiap isi membutuhkan tempat lebih tetapi secara keseluruhan daftar akan lebih pendek, selama *count* lebih dari satu.

9. Implementasi Direktori

Pemilihan dalam algoritma alokasi direktori dan manajemen direktori mempunyai efek yang besar dalam efisiensi, performa, dan kehandalan dari sistem berkas.

1) Linear List

Metode paling sederhana dalam mengimplementasikan sebuah direktori adalah dengan menggunakan *linear list* dari nama berkas dengan penunjuk ke blok data. *Linear list* dari direktori memerlukan pencarian searah untuk mencari suatu direktori didalamnya. Metode sederhana untuk diprogram tetapi memakan waktu lama ketika dieksekusi. Untuk membuat berkas baru kita harus mencari di dalam direktori untuk meyakinkan bahwa tidak ada berkas yang bernama sama. Lalu kita tambahkan sebuah berkas baru pada akhir direktori. Untuk menghapus sebuah berkas, kita mencari berkas tersebut dalam direktori, lalu melepaskan tempat yang dialokasikan untuknya. Untuk menggunakan kembali suatu berkas dalam direktori kita dapat melakukan beberapa hal. Kita dapat menandai berkas tersebut sebagai tidak terpakai (dengan menamainya secara khusus, seperti nama yang kosong, atau bit terpakai atau tidak yang ditambahkan pada berkas), atau kita dapat menambahkannya pada daftar direktori bebas. Alternatif lainnya kita dapat menyalin ke tempat yang dikosongkan pada direktori. Kita juga bisa menggunakan linked list untuk mengurangi waktu untuk menghapus berkas. Kelemahan dari linear list ini adalah percarian searah untuk mencari sebuah berkas. Direktori yang berisi informasi sering digunakan, implementasi yang lambat pada cara aksesnya akan menjadi perhatian pengguna. Faktanya, banyak sistem operasi mengimplementasikan', *software cache'* untuk menyimpan informasi yang paling sering digunakan. Penggunaan '*cache*' menghindari pembacaan informasi berulang-ulang pada disk. Daftar yang telah diurutkan memperbolehkan pencarian biner dan mengurangi waktu rata-rata pencarian. Bagaimana pun juga penjagaan agar daftar tetap terurut dapat merumitkan operasi pembuatan dan

penghapusan berkas, karena kita perlu memindahkan sejumlah direktori untuk mengurutkannya. *Tree* yang lebih lengkap dapat membantu seperti B-tree. Keuntungan dari daftar yang terurut adalah kita dapatkan daftar direktori yang terurut tanpa pengurutan yang terpisah.

2) Hash Table

Struktur data lainnya yang juga digunakan untuk direktori berkas adalah *hash table*. Dalam metode ini linear list menyimpan direktori, tetapi struktur data hash juga digunakan. *Hash table* mengambil nilai yang dihitung dari nama berkas dan mengembalikan sebuah penunjuk ke nama berkas yang ada di *linear list*. Maka dari itu dapat memotong banyak biaya pencarian direktori. Memasukkan dan menghapus berkas juga lebih mudah dan cepat. Meski demikian beberapa aturan harus dibuat untuk mncegah tabrakan, situasi dimana dua nama berkas pada *hash* mempunyai tempat yang sama. Kesulitan utama dalam *hash table* adalah ukuran tetap dari *hash table* dan ketergantungan dari fungsi *hash* dengan ukuran *hash table*. Sebagai contoh, misalkan kita membuat suatu *linear-probing hash table* yang dapat menampung 64 data. Fungsi *hash* mengubah nama berkas menjadi nilai dari 0 sampai 63. Jika kita membuat berkas ke 65 maka ukuran table *hash* harus diperbesar sampai misalnya 128 dan kita membutuhkan suatu fungsi *hash* yang baru yang dapat memetakan nama berkas dari jangkauan 0 sampai 127, dan kita harus mengatur data direktori yang sudah ada agar memenuhi fungsi *hash* yang baru. Sebagai alternatif dapat digunakan *chained-over ow hash table*, setiap *hash table* mempunyai daftar yang terkait (*linked list*) dari pada nilai individual dan kita dapat mengatasi tabrakan dengan menambah tempat pada daftar terkait tersebut. Pencarian dapat menjadi lambat, karena pencarian nama memerlukan tahap pencarian pada daftar terkait. Tetapi operasi ini lebih cepat dari pada pencarian linear terhadap seluruh direktori.

10. Efisiensi dan Unjuk Kerja

Setelah kita membahas alokasi blok dan pilihan manajemen direktori maka dapat dibayangkan bagaimana efek mereka dalam keefisiensian dan unjuk kerja penggunaan disk. Hal ini dikarenakan disk selalu menjadi "bottle-neck" dalam unjuk kerja sistem.

1) Efisiensi

Disk dapat digunakan secara efisien tergantung dari teknik alokasi disk serta algoritma pembentukan direktori yang digunakan. Contoh, pada UNIX, direktori berkas dialokasikan terlebih dahulu pada partisi. Walau pun disk yang kosong pun terdapat beberapa persen dari ruangnya digunakan untuk direktori tersebut. Unjuk kerja sistem berkas meningkat akibata dari pengalokasian awal dan penyebaran direktori ini pada partisi. Sistem berkas UNIX melakukan ini agar blok-blok data berkas selalu dekat dengan blok direktori berkas sehingga waktu pencariannya berkurang. Ada pula keefesiensian pada ukuran penunjuk yang digunakan untuk mengakses data. Masalahnya dalam memilih ukuran penunjuk adalah merencanakan efek dari perubahan teknologi. Masalah ini diantisipasi dengan menginisialisasi terlebih dahulu sistem berkasnya dengan alasan keefisiensian.

Pada awal, banyak struktur data dengan panjang yang sudah ditentu kan dan dialokasi pada ketika system dijalankan. Ketika tabel proses penuh maka tidak ada proses lain yang dapat dibuat. Begitu juga dengan tabel berkas ketika penuh, tidak ada berkas yang dapat dibuka. Hal ini menyebabkan sistem gagal melayani permintaan pengguna. Ukuran tabel-tabel ini dapat ditingkatkan hanya dengan mengkompilasi ulang kernel dan boot ulang sistemnya. Tetapi sejak dikeluarkannya Solaris 2, hampir setiap struktur kernel dialokasikan secara dinamis sehingga menghapus batasan buatan pada unjuk kerja sistem.

2) Kinerja

Ketika metode dasar disk telah dipilih, maka masih ada beberapa cara untuk meningkatkan unjuk kerja. Salah satunya adalah dengan mengguna kan cache, yang merupakan memori lokal pada pengendali disk, dimana cache cukup besar untuk menampung seluruh track pada satu waktu. Beberapa sistem mengatur seksi terpisah dari memori utama untuk disk-cache, yang diasumsikan bahwa blok-blok disimpan karena mereka akan digunakan dalam waktu dekat. Ada juga sistem yang menggunakan memori fisik yang tidak digunakan sebagai penyangga yang dibagi atas sistem halaman (paging) dan sistem disk-blok cache. Suatu sistem melakukan banyak operasi I/O akan menggunakan sebagian banyak memorinya sebagai blok cache, dimana suatu sistem mengeksekusi banyak program akan menggunakan sebagian besar memori-nya untuk ruang halaman. Beberapa sistem mengoptimalkan disk-cache nya dengan menggunakan berbagai macam algoritma penempatan ulang (replacement algorithms), tergantung dari macam tipe akses dari

berkas. Pada akses yang sekuen sial dapat dioptimasi dengan teknik yang dikenal dengan nama free-behind dan read-ahead. Free-behind memindahkan sebuah blok dari penyangga secepatnya ketika blok berikutnya diminta. Hal ini dilakukan karena blok sebelumnya tidak lagi digunakan sehingga akan membuang ruang yang ada di penyangga. Sedangkan dengan read ahead, blok yang diminta dan beberapa blok berikutnya dibaca dan ditempatkan pada cache. Hal ini dilakukan karena kemungkinan blok-blok berikutnya akan diminta setelah blok yang sedang diproses. Hal ini juga mem beri dampak pada waktu yang digunakan akan lebih cepat.

Metode yang lain adalah dengan membagi suatu seksi dari memori untuk disk virtual atau RAM disk. Pada RAM disk terdapat operasi-operasi standar yang terdapat pada disk, tetapi semua operasi tersebut terjadi di dalam suatu seksi memori, bukan pada disk. Tetapi, RAM disk hanya berguna untuk penyimpanan sementara, karena jika komputer di boot ulang atau listrik mati maka isi dalam RAM disk akan terhapus. Perbedaan antara RAM disk dan disk cache adalah dalam masalah siapa yang mengendalikan disk tersebut. RAM disk dikendalikan oleh peng guna sepenuhnya, sedangkan disk cache dikendalikan oleh sistem operasi.

11. Recovery

Karena semua direktori dan berkas disimpan di dalam memori utama dan disk, maka kita perlu memastikan bahwa kegagalan pada sistem tidak menyebabkan hilangnya data atau data menjadi tidak konsiten.

1) Pemeriksaan Rutin

Informasi direktori pada memori utama pada umumnya lebih up to date daripada informasi yang terdapat di disk dikarenakan penulisan dari informasi direktori cached ke disk tidak langsung terjadi pada saat setelah peng-update-an terjadi. Consistency checker membandingkan data yang terdapat di struktur direktori dengan blok-blok data pada disk, dan mencoba memperbaiki semua ketidak konsistensian yang terjadi akibat crash-nya komputer. Algoritma pengalokasian dan management ruang kosong menentukan tipe dari masalah yang ditemukan oleh checker dan seberapa sukses dalam memperbaiki masalah-masalah tersebut.

2) Back Up and Restore

Karena kadang-kadang magnetik disk gagal, kita harus memastikan bahwa datanya tidak hilang selamanya. Karena itu, kita menggunakan program sistem untuk mem-back up data dari disk ke alat penyimpanan yang lain seperti oopy disk, magnetic tape, atau optical disk. Pengembalian berkas-berkas yang hilang hanya masalah menempatkan lagi data dari back up data yang telah dilakukan. Untuk meminimalisir penyalinan, kita dapat menggunakan informasi dari setiap masukan direktori berkas. Umpamanya, jika program back up mengetahui bahwa back up terakhir dari berkas sudah selesai dan penulisan terakhir pada berkas dalam direktori menandakan berkas tidak terjadi perubahan maka berkas tidak harus disalin lagi. Penjadualan back up yang umum sebagai berikut:

- Hari 1: Salin ke tempat penyimpanan back up semua berkas dari disk, disebut sebuah full backup.
- Hari 2: Salin ke tempat penyimpanan lain semua berkas yang berubah sejak hari 1, disebut incremental backup.
- Hari 3: Salin ke tempat peyimpanan lain semua berkas yang berubah sejak hari 2.
- Hari N: salin ke tempat penyimpanan lain semua berkas yang berubah sejak hari N-1, lalu kembali ke hari 1.

Keuntungan dari siklus backup ini adalah kita dapat menempatkan kembali berkas mana pun yang tidak sengaja terhapus pada waktu siklus dengan mendapatkannya dari back up hari sebelumnya. Panjang dari siklus disetujui antara banyaknya tempat penyimpanan backup yang diperlukan dan jumlah hari kebelakang dari penempatan kembali dapat dilakukan.

Ada juga kebiasaan untuk mem-backup keseluruhan dari waktu ke waktu untuk disimpan selamanya daripada media backupnya digunakan kembali. Ada bagusnya menyimpan backup-backup permanent ini di lokasi yang jauh dari backup yang biasa, untuk menghindari kecelakaan seperti kebakaran dan lain-lain. Dan jangan menggunakan kembali media backup terlalu lama karena media tersebut akan rusak jika terlalu sering digunakan kembali.