B a b 6

I/O dan Disk

Tujuan Pelajaran

Setelah mempelajari bab ini, Anda diharapkan :

- Memahami konsep perangkat keras I/O
- Memahami konsep DMA
- Memahami interface yang ada pada aplikasi I/O
- Memahami kinerja I/O
- Memahami struktur disk
- Memahami penjadualan disk
- Memahami manajemen disk
- Memahami masalah yang berkaitan dengan sisten operasi

6.1. Perangkat Keras *I/O*

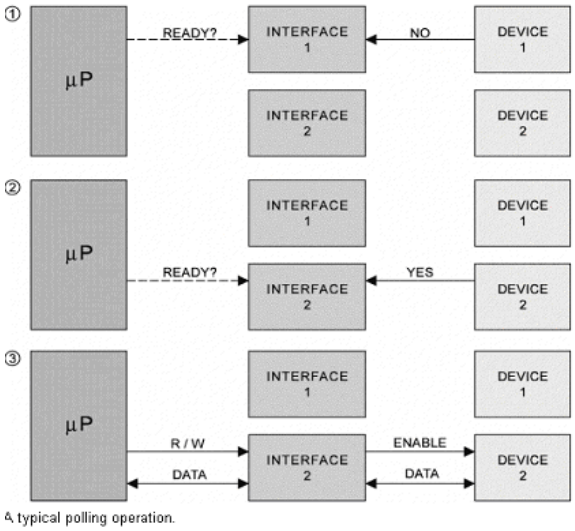
Secara umum, terdapat beberapa jenis seperti *device* penyimpanan (*disk*, *tape*), *transmission device* (*network card, modem*), dan *human-interface device* (*screen, keyboard, mouse*). *Device* tersebut dikendalikan oleh instruksi *I/O*. Alamat-alamat yang dimiliki oleh *device* akan digunakan oleh *direct I/O instruction* dan *memory-mapped I/O*.

Beberapa konsep yang umum digunakan ialah *port, bus (daisy chain/ shared direct access)*, dan *controller* (*host adapter*). *Port* adalah koneksi yang digunakan oleh *device* untuk berkomunikasi dengan mesin. *Bus* adalah koneksi yang menghubungkan beberapa *device* menggunakan kabel-kabel. *Controller* adalah alat-alat elektronik yang berfungsi untuk mengoperasikan *port, bus, dan device*.

Langkah yang ditentukan untuk *device* adalah *command-ready, busy*, dan *error*. *Host* mengeset *command-ready* ketika perintah telah siap untuk dieksekusi oleh *controller*. *Controller* mengeset *busy* ketika sedang mengerjakan sesuatu, dan men *clear busy* ketika telah siap untuk menerima perintah selanjutnya. *Error* diset ketika terjadi kesalahan.

6.1.1. *Polling*

Busy-waiting/ polling adalah ketika *host* mengalami *looping* yaitu membaca status register secara terus-menerus sampai status *busy* di-clear. Pada dasarnya *polling* dapat dikatakan efisien. Akan tetapi polling menjadi tidak efisien ketika setelah berulang-ulang melakukan *looping*, hanya menemukan sedikit *device* yang siap untuk men-*service*, karena CPU *processing* yang tersisa belum selesai.



Gambar 6-1. Polling Operation.

6.1.2. Interupsi

6.1.2.1. Mekanisme Dasar Interupsi

Ketika CPU mendeteksi bahwa sebuah *controller* telah mengirimkan sebuah sinyal ke *interrupt request line* (membangkitkan sebuah interupsi), CPU kemudian menjawab interupsi tersebut (juga disebut menangkap interupsi) dengan menyimpan beberapa

informasi mengenai *state* terkini CPU--contohnyanilai instruksi *pointer*, dan memanggil *interrupt handler* agar *handler* tersebut dapat melayani *controller* atau alat yang mengirim interupsi tersebut.

6.1.2.2. Fitur Tambahan pada Komputer Modern

Pada arsitektur komputer modern, tiga fitur disediakan oleh CPU dan *interrupt controller* (pada perangkat keras) untuk dapat menangani interrupsi dengan lebih bagus. Fitur-fitur ini antara lain adalah kemampuan menghambat sebuah proses *interrupt handling* selama prosesi berada dalam *critical state*, efisiensi penanganan interupsi sehingga tidak perlu dilakukan polling untuk mencari *device* yang mengirimkan interupsi, dan fitur yang ketiga adalah adanya sebuah konsep *multilevel* interupsi sedemikian rupa sehingga terdapat prioritas dalam penanganan interupsi (diimplementasikan dengan *interrupt priority level system*).

6.1.2.3. *Interrupt Request Line*

Pada peranti keras CPU terdapat kabel yang disebut *interrupt request line*, kebanyakan CPU memiliki dua macam *interrupt request line*, yaitu *nonmaskable interrupt* dan *maskable interrupt*. *Maskable interrupt* dapat dimatikan/ dihentikan oleh CPU sebelum pengeksekusian deretan *critical instruction (critical instruction sequence)* yang tidak boleh diinterupsi. Biasanya, interrupt jenis ini digunakan oleh *device controller* untuk meminta pelayanan CPU.

6.1.2.4. *Interrupt Vector* dan *Interrupt Chaining*

Sebuah mekanisme interupsi akan menerima alamat *interrupt handling routine* yang spesifik dari sebuah set, pada kebanyakan arsitektur komputer yang ada sekarang ini, alamat ini biasanya berupa sekumpulan bilangan yang menyatakan *offset* pada sebuah tabel (biasa disebut *interrupt vector*). Tabel ini menyimpan alamat-alamat *interrupt handler* spesifik di dalam memori. Keuntungan dari pemakaian vektor adalah untuk mengurangi kebutuhan akan sebuah *interrupt handler* yang harus mencari semua kemungkinan sumber interupsi untuk menemukan pengirim interupsi.

Akan tetapi, *interrupt vector* memiliki hambatan karena pada kenyataannya, komputer yang ada memiliki *device* (dan *interrupt handler*) yang lebih banyak dibandingkan dengan jumlah alamat pada *interrupt vector*. Karena itulah, digunakanlah teknik *interrupt chaining* dimana setiap elemen dari *interrupt vector* menunjuk/ merujuk pada elemen pertama dari sebuah daftar *interrupt handler*. Dengan teknik ini, *overhead* yang dihasilkan oleh besarnya ukuran tabel dan inefisiensi dari penggunaan sebuah *interrupt handler* (fitur pada CPU yang telah disebutkan sebelumnya) dapat dikurangi, sehingga keduanya menjadi kurang lebih seimbang.

6.1.2.5. Penyebab Interupsi

Interupsi dapat disebabkan berbagai hal, antara lain *exception, page fault*, interupsi yang dikirimkan oleh *device controllers*, dan *system call Exception* adalah suatu kondisi dimana terjadi sesuatu/ dari sebuah operasi didapat hasil tertentu yang dianggap khusus sehingga harus mendapat perhatian lebih, contoh nya pembagian dengan 0 (nol), pengaksesan alamat memori yang *restricted* atau bahkan tidak valid, dan lain-lain. *System call* adalah sebuah fungsi pada aplikasi (perangkat lunak) yang dapat mengeksekusikan instruksi khusus berupa *software interrupt* atau *trap*.

6.1.3. DMA

6.1.3.1. Definisi

DMA adalah sebuah prosesor khusus (*special purpose processor*) yang berguna untuk menghindari pembebanan CPU utama oleh program *I/O* (PIO).

Gambar 6-2. DMA Interface.

6.1.3.2. Transfer DMA

Untuk memulai sebuah transfer DMA, *host* akan menuliskan sebuah DMA *command block* yang berisi *pointer* yang menunjuk ke sumber transfer, *pointer* yang menunjuk ke tujuan/ destinasi transfer, dan jumlah *byte* yang ditransfer, ke memori. CPU kemudian menuliskan alamat *command block* ini ke DMA *controller*, sehingga DMA *controller* dapat kemudian mengoperasikan *bus* memori secara langsung dengan menempatkan alamat-alamat pada *bus* tersebut untuk melakukan transfer tanpa bantuan CPU.

Tiga langkah dalam transfer DMA:

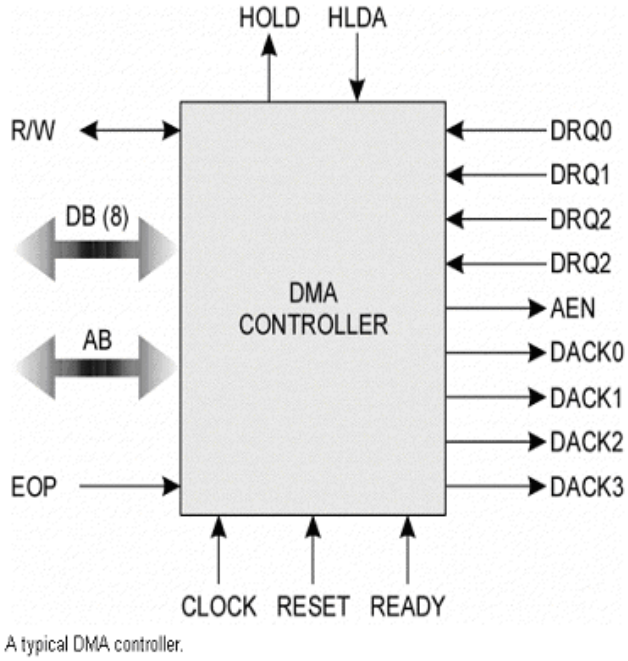
1. Prosesor menyiapkan DMA transfer dengan menyedia kan data-data dari *device*, operasi yang akan ditampilkan, alamat memori yang menjadi sumber dan tujuan data, dan banyaknya byte yang di transfer.

2. DMA *controller* memulai operasi (menyiapkan bus, menyediakan alamat, menulis dan membaca data), sampai seluruh blok sudah di transfer.

3. DMA *controller* meng-interupsi prosesor, dimana selanjutnya akan ditentukan tindakan berikutnya.

Pada dasarnya, DMA mempunyai dua metode yang berbeda dalam mentransfer data. Metode yang pertama adalah metode yang sangat baku dan simple disebut HALT, atau *Burst Mode* DMA, karena DMA *controller* memegang kontrol dari sistem bus dan mentransfer semua blok data ke atau dari memori pada *single burst*. Selagi transfer masih dalam progres, sistem mikroprosessor di-set *idle*, tidak melakukan instruksi operasi untuk menjaga internal register. Tipe operasi DMA seperti ini ada pada kebanyakan komputer.

Metode yang kedua, mengikut-sertakan DMA *controller* untuk memegang kontrol dari sistem bus untuk jangka waktu yang lebih pendek pada periode dimana mikroprosessor sibuk dengan operasi internal dan tidak membutuhkan akses ke sistem bus. Metode DMA ini disebut *cycle stealing mode*. *Cycle stealing* DMA lebih kompleks untuk diimplementasikan dibandingkan HALT DMA, karena DMA controller harus mempunyai kepintaran untuk merasakan waktu pada saat sistem bus terbuka.



Gambar 6-3. DMA Controller.

6.1.3.3. *Handshaking*

Proses *handshaking* antara DMA *controller* dan *device controller* dilakukan melalui sepasang kabel yang disebut DMA-*request* dan DMA-*acknowledge*. Device controller mengirimkan sinyal melalui DMA-*request* ketika akan mentransfer data sebanyak satu *word*. Hal ini kemudian akan mengakibatkan DMA *controller* memasukkan alamat-alamat yang dinginkan ke kabel alamat memori, dan mengirimkan sinyal melalui kabel DMA-*acknowledge*. Setelah sinyal melalui kabel DMA-*acknowledge* diterima, *device controller* mengirimkan data yang dimaksud dan mematikan sinyal pada DMA-*request*.

Hal ini berlangsung berulang-ulang sehingga disebut *handshaking*. Pada saat DMA *controller* mengambil alih memori, CPU sementara tidak dapat mengakses memori (dihalangi), walau pun masih dapat mengaksees data pada cache primer dan sekunder. Hal ini disebut *cycle stealing*, yang walau pun memperlambat komputasi CPU, tidak menurunkan kinerja karena memindahkan pekerjaan data transfer ke DMA *controller* meningkatkan performa sistem secara keseluruhan.

6.1.3.4. Cara-cara Implementasi DMA

Dalam pelaksanaannya, beberapa komputer menggunakan memori fisik untuk proses DMA , sedangkan jenis komputer lain menggunakan alamat virtual dengan melalui tahap "penerjemahan" dari alamat memori virtual menjadi alamat memori fisik, hal ini disebut *direct virtual-memory address* atau DVMA.

Keuntungan dari DVMA adalah dapat mendukung transfer antara dua *memory mapped device* tanpa intervensi CPU.

6.2. *Interface* Aplikasi I/O

Ketika suatu aplikasi ingin membuka data yang ada dalam suatu disk, sebenarnya aplikasi tersebut harus dapat membedakan jenis disk apa yang akan diaksesnya. Untuk mempermudah pengaksesan, sistem operasi melakukan standarisasi cara pengaksesan pada peralatan I/O. Pendekatan inilah yang dinamakan *interface* aplikasi I/O.

Interface aplikasi I/O melibatkan abstraksi, enkapsulasi, dan *software layering*. Abstraksi dilakukan dengan membagi-bagi detail peralatan-peralatan I/O ke dalam kelas-kelas yang lebih umum. Dengan adanya kelas-kelas yang umum ini, maka akan lebih mudah untuk membuat fungsi-fungsi standar (*interface*) untuk mengaksesnya. Lalu kemudian adanya *device driver* pada masing-masing peralatan I/O, berfungsi untuk enkapsulasi perbedaan-perbedaan yang ada dari masing-masing anggota kelas-kelas yang umum tadi. *Device driver* mengenkapsulasi tiap -tiap peralatan I/O ke dalam masing-masing 1 kelas yang umum tadi (*interface* standar). Tujuan dari adanya lapisan *device driver* ini adalah untuk menyembunyikan perbedaan-perbedaan yang ada pada *device controller* dari subsistem I/O pada kernel. Karena hal ini, subsistem I/O dapat bersifat independen dari *hardware*.

Karena subsistem I/O independen dari *hardware* maka hal ini akan sangat menguntungkan dari segi pengembangan *hardware*. Tidak perlu menunggu vendor sistem operasi untuk mengeluarkan support code untuk *hardware-hardware* baru yang akan dikeluarkan oleh vendor *hardware*.

6.2.1. Peralatan *Block* dan Karakter

Peralatan block diharapkan dapat memenuhi kebutuhan akses pada berbagai macam disk drive dan juga peralatan *block* lainnya. *Block device* diharapkan dapat memenuhi/mengerti perintah baca, tulis dan juga perintah pencarian data pada peralatan yang memiliki sifat *random-access*.

Keyboard adalah salah satu contoh alat yang dapat mengakses *stream*-karakter. *System call* dasar dari *interface* ini dapat membuat sebuah aplikasi mengerti tentang bagaimana cara untuk mengambil dan menuliskan sebuah karakter. Kemudian pada pengembangan lanjutannya, kita dapat membuat *library* yang dapat mengakses data/pesan per-baris.

6.2.2. Peralatan Jaringan

Karena adanya perbedaan dalam kinerja dan pengalamatan dari jaringan I/O, maka biasanya sistem operasi memiliki *interface* I/O yang berbeda dari baca, tulis dan pencarian pada disk. Salah satu yang banyak digunakan pada sistem operasi adalah *interface socket*.

Socket berfungsi untuk menghubungkan komputer ke jaringan. *System call* pada *socket* interface dapat memudahkan suatu aplikasi untuk membuat *local socket*, dan menghubungkannya ke *remote socket*. Dengan menghubungkan komputer ke socket, maka komunikasi antar komputer dapat dilakukan.

6.2.3. Jam dan *Timer*

Adanya jam dan timer pada *hardware* komputer, setidaknya memiliki tiga fungsi, memberi informasi waktu saat ini, memberi informasi lamanya waktu sebuah proses, sebagai trigger untuk suatu operasi pada suatu waktu. Fungsi fungsi ini sering digunakan

oleh sistem operasi. Sayangnya, *system call* untuk pemanggilan fungsi ini tidak di-standarisasi antar sistem operasi *Hardware* yang mengukur waktu dan melakukan operasi *trigger* dinamakan *programmable interval timer*. Dia dapat di set untuk menunggu waktu tertentu dan kemudian melakukan interupsi. Contoh penerapannya ada pada *scheduler*, dimana dia akan melakukan interupsi yang akan memberhentikan suatu proses pada akhir dari bagian waktunya.

Sistem operasi dapat mendukung lebih dari banyak *timer request* daripada banyaknya jumlah *hardware timer*. Dengan kondisi seperti ini, maka kernel atau *device driver* mengatur list dari interupsi dengan urutan yang duluan datang yang duluan dilayani.

6.2.4. *Blocking* dan *Nonblocking* I/O

Ketika suatu aplikasi menggunakan sebuah blocking *system call*, eksekusi aplikasi itu akan diberhentikan untuk sementara. aplikasi tersebut akan dipindahkan ke *wait queue*. Dan setelah *system call* tersebut selesai, aplikasi tersebut dikembalikan ke *run queue*, sehingga pengeksekusian aplikasi tersebut akan dilanjutkan. *Physical action* dari peralatan I/O biasanya bersifat *asynchronous*. Akan tetapi, banyak sistem operasi yang bersifat *blocking*, hal ini terjadi karena *blocking application* lebih mudah dimengerti dari pada *nonblocking application*.

6.3. *Kernel I/O Subsystem*

Kernel menyediakan banyak service yang berhubungan dengan *I/O*. Pada bagian ini, kita akan mendeskripsikan beberapa service yang disediakan oleh kernel *I/O subsystem*, dan kita akan membahas bagaimana caranya membuat infrastruktur *hardware* dan *device-driver*. Service yang akan kita bahasadalah *I/O scheduling, buffering, caching, spooling, reservasi device, error handling*.

6.3.1. *I/O Scheduling*

Untuk menjadualkan sebuah set permintaan *I/O*, kita harus menetukan urutan yang bagus untuk mengeksekusi permintaan tersebut. *Scheduling* dapat meningkatkan kemampuan sistem secara keseluruhan, dapat membagi device secara rata di antara proses-proses, dan dapat mengurangi waktu tunggu rata-rata untuk menyelesaikan *I/O*. Ini adalah contoh sederhana untuk menggambarkan definisi di atas. Jika sebuah *arm* disk terletak di dekat permulaan disk, dan ada tiga aplikasi yang memblokir panggilan untuk membaca untuk disk tersebut. Aplikasi 1 meminta sebuah blok dekat akhir disk, aplikasi 2 meminta blok yang dekat dengan awal, dan aplikasi 3 meminta bagian tengah dari disk. Sistem operasi dapat mengurangi jarak yang harus ditempuh oleh *arm* disk dengan melayani aplikasi tersebut dengan urutan 2, 3, 1. Pengaturan urutan pekerjaan kembali dengan cara ini merupakan inti dari *I/O* scheduling. Sistem operasi mengembangkan implementasi scheduling dengan menetapkan antrian permintaan untuk tiap device. Ketika sebuah aplikasi meminta sebuah *blocking* sistem *I/O*, permintaan tersebut dimasukkan ke dalam antrian untuk device tersebut. *Scheduler I/O* mengatur urutan antrian untuk meningkatkan efisiensi dari sistem dan waktu respon rata-rata yang harus dialami oleh aplikasi. Sistem operasi juga mencoba untuk bertindak secara adil, seperti tidak ada aplikasi yang menerima service yang buruk, atau dapat seperti memberi prioritas service untuk permintaan penting yang ditunda. Contohnya, pemintaan dari subsistem mungkin akan mendapatkan prioritas lebih tinggi daripada permintaan dari aplikasi. Beberapa algoritma scheduling untuk disk *I/O* akan dijelaskan ada bagian *Disk Scheduling*.

Satu cara untuk meningkatkan efisiensi *I/O* subsistem dari sebuah komputer adalah dengan mengatur operasi *I/O*. Cara lain adalah dengan menggunakan tempat penyimpanan pada memori utama atau pada disk, melalui teknik yang disebut *buffering, caching, dan spooling*.

6.3.2. *Buffering*

Buffer adalah area memori yang menyimpan data ketika mereka sedang dipindahkan antara dua *device* atau antara *device* dan aplikasi. *Buffering* dilakukan untuk tiga buah alasan. Alasan pertama adalah untuk men-*cope* dengan kesalahan yang terjadi karena perbedaan kecepatan antara produsen dengan konsumen dari sebuah *stream data*. Sebagai contoh, sebuah file sedang diterima melalui modem dan ditujukan ke media penyimpanan di *hard disk*. Kecepatan modem tersebut kira-kira hanyalah 1/1000 daripada *hard disk*. Jadi *buffer* dibuat di dalam memori utama untuk mengumpulkan jumlah *byte* yang diterima dari modem. Ketika keseluruhan data di *buffer* sudah sampai, buffer tersebut dapat ditulis ke disk dengan operasi tunggal. Karena penulisan disk tidak terjadi dengan instan dan modem masih memerlukan tempat untuk menyimpan data yang berdatangan, maka dipakai 2 buah *buffer*. Setelah modem memenuhi *buffer* pertama, akan terjadi request untuk menulis di disk. Modem kemudian mulai memenuhi *buffer* kedua sementara *buffer* pertama dipakai untuk penulisan ke disk. Pada saat modem sudah memenuhi *buffer* kedua, penulisan ke disk dari *buffer* pertama seharusnya sudah selesai, jadi modem akan berganti kembali memenuhi *buffer* pertama dan *buffer* kedua dipakai untuk menulis. Metode *double buffering* ini membuat pasangan ganda antara produsen dan konsumen sekaligus mengurangi kebutuhan waktu di antara mereka.

Alasan kedua dari *buffering* adalah untuk menyesuaikan device-device yang mempunyai perbedaan dalam ukuran transfer data. Hal ini sangat umum terjadi pada jaringan komputer, dimana *buffer* dipakai secara luas untuk fragmentasi dan pengaturan kembali pesan-pesan yang diterima. Pada bagian pengirim, sebuah pesan yang besar akan dipecah ke paket-paket kecil. Paket-paket tersebut dikirim melalui jaringan, dan penerima akan meletakkan mereka di dalam *buffer* untuk disusun kembali.

Alasan ketiga untuk *buffering* adalah untuk mendukung *copy semantics* untuk aplikasi *I/O*. Sebuah contoh akan menjelaskan apa arti dari *copy semantics*. Jika ada sebuah aplikasi yang mempunyai *buffer* data yang ingin dituliskan ke disk. Aplikasi tersebut akan memanggil sistem penulisan, menyediakan pointer ke *buffer*, dan sebuah *integer* untuk menunjukkan ukuran bytes yang ingin ditulis. Setelah pemanggilan tersebut, apakah yang akan terjadi jika aplikasi tersebut merubah isi dari *buffer*, dengan *copy semantics*, keutuhan data yang ingin ditulis sama dengan data waktu aplikasi ini memanggil sistem untuk menulis, tidak tergantung dengan perubahan yang terjadi pada *buffer*. Sebuah cara sederhana untuk sistem operasi untuk menjamin *copy semantics* adalah membiarkan sistem penulisan untuk mengkopi data aplikasi ke dalam *buffer* kernel sebelum mengembalikan kontrol kepada aplikasi. Jadi penulisan ke disk dilakukan pada *buffer* kernel, sehingga perubahan yang terjadi pada *buffer* aplikasi tidak akan membawa dampak apa-apa. Mengcopy data antara *buffer* kernel data aplikasi merupakan sesuatu yang umum pada sistem operasi, kecuali *overhead* yang terjadi karena operasi ini karena *clean semantics*. Kita dapat memperoleh efek yang sama yang lebih efisien dengan memanfaatkan virtual-memori mapping dan proteksi *copy-on-wire* dengan pintar.

6.3.3. *Caching*

Sebuah *cache* adalah daerah memori yang cepat yang berisikan data kopian. Akses ke sebuah kopian yang di-*cached* lebih efisien daripada akses ke data asli. Sebagai

contoh, instruksi-instruksi dari proses yang sedang dijalankan disimpan ke dalam disk, dan ter-*cached* di dalam memori *physical*, dan kemudian dicopy lagi ke dalam *cache secondary and primary* dari CPU. Perbedaan antara sebuah *buffer* dan ache adalah *buffer* dapat menyimpan satu-satunya informasi datanya sedangkan sebuah cache secara definisi hanya menyimpan sebuah data dari sebuah tempat untuk dapat diakses lebih cepat.

Caching dan *buffering* adalah dua fungsi yang berbeda, tetapi terkadang sebuah daerah memori dapat digunakan untuk keduanya. sebagai contoh, untuk menghemat *copy semantics* dan membuat scheduling *I/O* menjadi efisien, sistem operasi menggunakan *buffer* pada memori utama untuk menyimpan data.

Buffer ini juga digunakan sebagai cache, untuk meningkatkan efisiensi *I/O* untuk file yang digunakan secara bersama-sama oleh beberapa aplikasi, atau yang sedang dibaca dan ditulis secara berulang-ulang.

Ketika kernel menerima sebuah permintaan file *I/O*, kernel tersebut mengakses *buffer cache*untuk melihat apakah daerah memori tersebut sudah tersedia dalam memori utama. Jika iya, sebuah *physical disk I/O* dapat dihindari atau tidak dipakai. penulisan disk juga terakumulasi ke dalam *buffer cache* selama beberapa detik, jadi transfer yang besar akan dikumpulkan untuk mengefisiensikan *schedule* penulisan. Cara ini akan menunda penulisan untuk meningkatkan efisiensi *I/O* akan dibahas pada bagian *Remote File Access*.

6.3.4. *Spooling* dan *Reservasi Device*

Sebuah spool adalah sebuah *buffer* yang menyimpan *output* untuk sebuah *device*, seperti printer, yang tidak dapat menerima *interleaved data streams*. Walau pun printer hanya dapat melayani satu pekerjaan pada waktu yang sama, beberapa aplikasi dapat meminta printer untuk mencetak, tanpa harus mendapatkan hasil *output* mereka tercetak secara bercampur. Sistem operasi akan menyelesaikan masalah ini dengan meng-*intercept* semua output kepada printer. Tiap *output* aplikasi sudah di-*spooled* ke disk file yang berbeda. Ketika sebuah aplikasi selesai mengeprint, sistem spooling akan melanjutkan ke antrian berikutnya. Di dalam beberapa sistem operasi, *spooling* ditangani oleh sebuah sistem proses daemon. Pada sistem operasi yang lain, sistem ini ditangani oleh *in-kernel thread*. Pada kedua kasus, sistem operasi menyediakan *interface*kontrol yang membuat *users and system administrator* dapat menampilkan antrian tersebut, untuk mengenyahkan antrian-antrian yang tidak diinginkan sebelum mulai di-print.

Untuk beberapa *device*, seperti *drive tape*dan printer tidak dapat me-*multiplex* permintaan *I/O* dari beberapa aplikasi. *Spooling* merupakan salah satu cara untuk mengatasi masalah ini. Cara lain adalah dengan membagi koordinasi untuk *multiple concurrent* ini. Beberapa sistem operasi menyediakan dukungan untuk akses *device* secara eksklusif, dengan mengalokasikan proses ke *device idle* dan membuang *device* yang sudah tidak diperlukan lagi. Sistem operasi lainnya memaksakan limit suatu file untuk menangani *device* ini. Banyak sistem operasi menyediakan fungsi yang membuat proses untuk menangani koordinat *exclusive* akses diantara mereka sendiri.

6.3.5. *Error Handling*

Sebuah sistem operasi yang menggunakan *protected memory* dapat menjaga banyak kemungkinan *error* akibat *hardware* mau pun aplikasi. *Devices* dan transfer *I/O* dapat gagal dalam banyak cara, bisa karena alasan *transient*, seperti overloaded pada *network*, mau pun alasan *permanen* yang seperti kerusakan yang terjadi pada *disk controller*. Sistem operasi seringkali dapat mengkompensasikan untuk kesalahan

transient. Seperti, sebuah kesalahan baca pada disk akan mengakibatkan pembacaan ulang kembali dan sebuah kesalahan pengiriman pada *network* akan mengakibatkan pengiriman ulang apabila protokolnya diketahui. Akan tetapi untuk kesalahan *permanent*, sistem operasi pada umumnya tidak akan bisa mengembalikan situasi seperti semula.

Sebuah ketentuan umum, yaitu sebuah sistem *I/O* akan mengembalikan satu *bit* informasi tentang status panggilan tersebut, yang akan menandakan apakah proses tersebut berhasil atau gagal. Sistem operasi pada UNIX menggunakan *integer* tambahan yang dinamakan *errno* untuk mengembalikan kode kesalahan sekitar 1 dari 100 nilai yang mengindikasikan sebab dari kesalahan tersebut. Akan tetapi, beberapa perangkat keras dapat menyediakan informasi kesalahan yang detail, walau pun banyak sistem operasi yang tidak mendukung fasilitas ini.

6.3.6. *Kernel Data Structure*

Kernel membutuhkan informasi *state* tentang penggunakan komponen *I/O*. Kernel menggunakan banyak struktur yang mirip untuk melacak koneksi jaringan, komunikasi karakter-*device*, dan aktivitas *I/O* lainnya.

UNIX menyediakan akses sistem file untuk beberapa entiti, seperti file *user*, *raw devices*, dan alamat tempat proses. Walau pun tiap entiti ini didukung sebuah operasi baca, *semantics*-nya berbeda untuk tiap entiti. Seperti untuk membaca file user, kernel perlu memeriksa *buffer cache* sebelum memutuskan apakah akan melaksanakan *I/O* disk. Untuk membaca sebuah *raw disk*, kernel perlu untuk memastikan bahwa ukuran permintaan adalah kelipatan dari ukuran sektor disk, dan masih terdapat di dalam batas sektor. Untuk memproses citra, cukup perlu untuk mengkopi data ke dalam memori. UNIX mengkapsulasikan perbedaan-perbedaan ini di dalam struktur yang uniform dengan menggunakan teknik *object oriented*.

Beberapa sistem operasi bahkan menggunakan metode *object oriented* secara lebih extensif. Sebagai contoh, Windows NT menggunakan implementasi *message-passing* untuk *I/O*. Sebuah permintaan *I/O* akan dikonversikan ke sebuah pesan yang dikirim melalui kernel kepada *I/O manager* dan kemudian ke *device driver*, yang masing-masing bisa mengubah isi pesan. Untuk output, isi *message* adalah data yang akan ditulis. Untuk input, *message* berisikan *buffer* untuk menerima data. Pendekatan *message-passing* ini dapat menambah *overhead*, dengan perbandingan dengan teknik prosedural yang men-*share* struktur data, tetapi akan mensederhanakan struktur dan *design* dari sistem *I/O* tersebut dan menambah fleksibilitas.

Kesimpulannya, subsistem *I/O* mengkoordinasi kumpulan-kumpulan service yang banyak sekali, yang tersedia dari aplikasi mau pun bagian lain dari kernel. Subsistem I/O mengawasi:

1. Manajemen nama untuk file dan *device*.
2. Kontrol akses untuk file dan *device*.
3. Kontrol operasi, contoh: model yang tidak dapat dikenali.
4. Alokasi tempat sistem file.
5. Alokasi *device*.
6. *Buffering, caching, spooling*.
7. *I/O scheduling*
8. Mengawasi status *device*, *error handling*, dan kesalahan dalam *recovery*.
9. Konfigurasi dan utilisasi *driver device*.

6.4. Penanganan Permintaan I/O

Di bagian sebelumnya, kita mendeskripsikan *handshaking* antara *device driver* dan *device controller*, tapi kita tidak menjelaskan bagaimana Sistem Operasi menyambungkan permintaan aplikasi untuk menyiapkan jaringan menuju sektor disk yang spesifik.

Sistem Operasi yang modern mendapatkan fleksibilitas yang signifikan dari tahapan-tahapan tabel *lookup* di jalur diantara permintaan dan *physical device controller*. Kita dapat mengenalkan *device* dan *driver* baru ke komputer tanpa harus meng-*compile* ulang kernelnya. Sebagai fakta, ada beberapa sistem operasi yang mampu untuk me-*load device drivers* yang diinginkan. Pada waktu *boot*, sistem mula-mula meminta bus piranti keras untuk menentukan *device* apa yang ada, kemudian sistem me-*load* ke dalam *driver* yang sesuai; baik sesegera mungkin, mau pun ketika diperlukan oleh sebuah permintaan *I/O*.

UNIX Sistem V mempunyai mekanisme yang menarik, yang disebut *streams*, yang membolehkan aplikasi untuk men-*assemble pipeline* dari kode *driver* secara dinamis. Sebuah *stream* adalah sebuah koneksi *full duplex* antara sebuah *device driver* dan sebuah proses *user-level*. *Stream* terdiri atas sebuah *stream head* yang merupakan antarmuka dengan *user process*, sebuah *driver end* yang mengontrol *device*, dan nol atau lebih *stream modules* diantara mereka. *Modules* dapat didorong ke stream untuk menambah fungsionalitas di sebuah *layered fashion*. Sebagai gambaran sederhana, sebuah proses dapat membuka sebuah alat port serial melalui sebuah *stream*, dan dapat mendorong ke sebuah modul untuk memegang edit input. *Stream* dapat digunakan untuk interproses dan komunikasi jaringan. Faktanya, di Sistem V, mekanisme soket diimplementasikan dengan stream.

Berikut dideskripsikan sebuah *lifecycle* yang tipikal dari sebuah permintaan pembacaan blok.

1. Sebuah proses mengeluarkan sebuah *blocking read system call* ke sebuah file deskriptor dari berkas yang telah dibuka sebelumnya.
2. Kode *system-call* di kernel mengecek parameter untuk kebenaran. Dalam kasus input, jika data telah siap di buffer cache, data akan dikembalikan ke proses dan permintaan *I/O* diselesaikan.
3. Jika data tidak berada dalam *buffer cache*, sebuah *physical I/O* akan bekerja, sehingga proses akan dikeluarkan dari antrian jalan (*run queue*) dan diletakkan di antrian tunggu (*wait queue*) untuk alat, dan permintaan *I/O* pun dijadwalkan. Pada akhirnya, subsistem *I/O* mengirimkan permintaan ke *device driver*. Bergantung pada sistem operasi, permintaan dikirimkan melalui *call* subrutin atau melalui pesan *in-kernel*.
4. *Device driver* mengalokasikan ruang *buffer* pada kernel untuk menerima data, dan menjadwalkan *I/O*. Pada akhirnya, *driver* mengirim perintah ke *device controller* dengan menulis ke register *device control*.
5. *Device controller* mengoperasikan piranti keras *device* untuk melakukan transfer data.
6. *Driver* dapat menerima status dan data, atau dapat menyiapkan transfer DMA ke memori kernel. Kita mengasumsikan bahwa transfer diatur oleh sebuah *DMA controller*, yang meggunakan interupsi ketika transfer selesai.
7. *Interrupt handler* yang sesuai menerima interupsi melalui tabel vektor-interupsi, menyimpan sejumlah data yang dibutuhkan, menandai *device driver*, dan kembali dari interupsi.

8. *Device driver* menerima tanda, menganalisa permintaan *I/O* mana yang telah diselesaikan, menganalisa status permintaan, dan menandai subsistem *I/O* kernel yang permintaannya telah terselesaikan.
9. Kernel mentransfer data atau mengembalikan kode ke ruang alamat dari proses permintaan, dan memindahkan proses dari antrian tunggu kembali ke antrian siap.
10. Proses tidak diblok ketika dipindahkan ke antrian siap. Ketika penjadwal (*scheduler*) mengembalikan proses ke CPU, proses meneruskan eksekusi pada penyelesaian dari *system call*.

6.5. Kinerja *I/O*

6.5.1. Pengaruh *I/O* pada Kinerja

I/O sangat berpengaruh pada kinerja sebuah sistem komputer. Hal ini dikarenakan *I/O* sangat menyita CPU dalam pengeksekusian *device driver* dan penjadwalan proses, demikian sehingga alih konteks yang dihasilkan membebani CPU dan cache perangkat keras. Selain itu, *I/O* juga memenuhi bus memori saat mengkopi data antara *controller* dan *physical memory*, serta antara buffer pada kernel dan *application space data*. Karena besarnya pengaruh *I/O* pada kinerja komputer inilah bidang pengembangan arsitektur komputer sangat memperhatikan masalah-masalah yang telah disebutkan diatas.

6.5.2. Cara Meningkatkan Efisiensi *I/O*

1. Menurunkan jumlah alih konteks.
2. Mengurangi jumlah pengkopian data ke memori ketika sedang dikirimkan antara *device* dan aplikasi.
3. Mengurangi frekuensi interupsi, dengan menggunakan ukuran transfer yang besar, *smart controller*, dan *polling*.
4. Meningkatkan *concurrency* dengan *controller* atau *channel* yang mendukung DMA.
5. Memindahkan kegiatan processing ke perangkat keras, sehingga operasi kepada *device controller* dapat berlangsung bersamaan dengan CPU.
6. Menyeimbangkan antara kinerja CPU, *memory subsystem, bus*, dan *I/O*.

6.5.3. Implementasi Fungsi I/O

Pada dasarnya kita mengimplementasikan algoritma *I/O* pada level aplikasi. Hal ini dikarenakan kode aplikasi sangat fleksible, dan *bugs* aplikasi tidak mudah menyebabkan sebuah sistem *crash*. Lebih lanjut, dengan mengembangkan kode pada level aplikasi, kita akan menghindari kebutuhan untuk *reboot* atau *reload device driver* setiap kali kita mengubah kode. Implementasi pada level aplikasi juga bisa sangat tidak efisien. Tetapi, karena *overhead* dari alih konteks dan karena aplikasi tidak bisa mengambil keuntungan dari struktur data kernel internal dan fungsionalitas dari kernel (misalnya, efisiensi dari kernel *messaging, threading dan locking*.

Pada saat algoritma pada level aplikasi telah membuktikan keuntungannya, kita mungkin akan mengimplementasikannya di kernel. Langkah ini bisa meningkatkan kinerja tetapi perkembangannya dari kerja jadi lebih menantang, karena besarnya kernel dari sistem operasi, dan kompleksnya sistem sebuah perangkat lunak. Lebih lanjut , kita harus men-*debug* keseluruhan dari implementasi *in-kernel* untuk menghindari korupsi sebuah data dan sistem *crash*.

Kita mungkin akan mendapatkan kinerja yang optimal dengan menggunakan implementasi yang special pada perangkat keras, selain dari *device* atau *controller*. Kerugian dari implementasi perangkat keras termasuk kesukaran dan biaya yang ditanggung dalam membuat kemajuan yang lebih baik dalam mengurangi *bugs*, perkembangan waktu yang maju dan fleksibilitas yang meningkat. Contohnya, RAID *controller* pada perangkat keras mungkin tidak akan menyediakan sebuah efek pada kernel untuk mempengaruhi urutan atau lokasi dari individual *block reads* dan *write*, meski pun kernel tersebut mempunyai informasi yang spesial mengenai *workload* yang dapat mengaktifkan kernel untuk meningkatkan kinerja dari *I/O*.

6.6. Struktur Disk

Disk menyediakan penyimpanan sekunder bagi sistem komputer modern. *Magnetic tape* sebelumnya digunakan sebagai media penyimpanan sekunder, tetapi waktu aksesnya lebih lambat dari disk. Oleh karena itu, sekarang *tape* digunakan terutama untuk *backup*, untuk penyimpanan informasi yang tidak sering, sebagai media untuk mentransfer infromasi dari satu sistem ke sistem yang lain, dan untuk menyimpan sejumlah data yang terlalu besar untuk sistem disk.

Disk drive modern dialamatkan sebagai suatu array satu dimensi yang besar dari blok lojik, dimana blok lojik merupakan unit terkecil dari transfer. Ukuran dari blok lojik biasanya adalah 512 *bytes*, walau pun sejumlah disk dapat diformat di level rendah (*low level formatted*) untuk memilih sebuah ukuran blok lojik yang berbeda, misalnya 1024 *bytes*.

Array satu dimensi dari blok lojik dipetakan ke bagian dari disk secara sekuensial. Sektor 0 adalah sektor pertama dari trek pertama di silinder paling luar (*outermost cylinder*). Pemetaan kemudian memproses secara berurutan trek tersebut, kemudian melalui trek selanjutnya di silinder tersebut, dan kemudian sisa silinder dari yang paling luar sampai yang paling dalam.

Dengan menggunakan pemetaan, kita dapat minimal dalam teori mengubah sebuah nomor blok logikal ke sebuah alamat disk yang bergaya lama (*old-style disk address*) yang terdiri atas sebuah nomor silinder, sebuah nomor trek di silinder tersebut, dan sebuah nomor sektor di trek tersebut. Dalam prakteknya, adalah sulit untuk melakukan translasi ini, dengan 2 alasan. Pertama, kebanyakan disk memiliki sejumlah sektor yang rusak, tetapi pemetaan menyembunyikan hal ini dengan mensubstitusikan dengan sektor yang dibutuhkan dari mana-mana di dalam disk. Kedua, jumlah dari sektor per trek tidaklah konstan. Semakin jauh sebuah trek dari tengah disk, semakin besar panjangnya, dan juga semakin banyak sektor yang dipunyainya. Oleh karena itu, disk modern diatur menjadi zona-zona silinder. Nomor sektor per trek adalah konstan dalam sebuah zona. Tetapi seiring kita berpindah dari zona dalam ke zona luar, nomor sektor per trek bertambah. Trek di zona paling luar tipikalnya mempunyai 40 persen sektor lebih banyak daripada trek di zona paling dalam.

Nomor sektor per trek telah meningkat seiring dengan peningkatan teknologi disk, dan adalah lazim untuk mempunyai lebih dari 100 sektor per trek di zona yang lebih luar dari disk. Dengan analogi yang sama, nomor silinder per disk telah meningkat, dan sejumlah ribuan silinder adalah tak biasa.

6.7. Penjadualan Disk

Salah satu tanggung jawab sistem operasi adalah menggunakan *hardware* dengan efisien. Khusus untuk *disk drives*, efisiensi yang dimaksudkan di sini adalah dalam hal waktu akses yang cepat dan aspek *bandwidth disk*. Waktu akses memiliki dua komponen utama yaitu waktu pencarian dan waktu rotasi disk. Waktu pencarian adalah waktu yang dibutuhkan *disk arm* untuk menggerakkan *head* ke bagian silinder *disk* yang mengandung sektor yang diinginkan. Waktu rotasi *disk* adalah waktu tambahan yang dibutuhkan untuk menunggu rotasi atau perputaran *disk*, sehingga sektor yang diinginkan dapat dibaca oleh *head*. Pengertian *Bandwidth* adalah total jumlah *bytes* yang ditransfer dibagi dengan total waktu antara permintaan pertama sampai seluruh bytes selesai ditransfer. Untuk meningkatkan kecepatan akses dan bandwidth, kita dapat melakukan penjadualan pelayanan atas permintaan *I/O* dengan urutan yang tepat.

Sebagaimana kita ketahui, jika suatu proses membutuhkan pelayanan *I/O* dari atau menuju *disk*, maka proses tersebut akan melakukan *system call* ke sistem operasi.

Permintaan tersebut membawa informasi-informasi antara lain:
1. Apakah operasi *input* atau *output*.
2. Alamat *disk* untuk proses tersebut.
3. Alamat memori untuk proses tersebut
4. Jumlah *bytes* yang akan ditransfer

Jika *disk drive* beserta *controller* tersedia untuk proses tersebut, maka proses akan dapat dilayani dengan segera. Jika ternyata *disk drive* dan *controller* tidak tersedia atau sedang sibuk melayani proses lain,maka semua permintaan yang memerlukan pelayanan *disk* tersebut akan diletakkan pada suatu antrian penundaan permintaan untuk *disk* tersebut. Dengan demikian, jika suatu permintaan telah dilayani, maka sistem operasi memilih permintaan tertunda dari antrian yang selanjutnya akan dilayani.

6.7.1. Penjadualan FCFS

Bentuk paling sederhana dalam penjadualan *disk* adalah dengan sistem antrian (queue) atau *First Come First Served* (FCFS). Algoritma ini secara intrinsik bersifat adil, tetapi secara umum algoritma ini pada kenyataannya tidak memberikan pelayanan yang paling cepat. Sebagai contoh, antrian permintaan pelayanan *disk* untuk proses *I/O* pada blok dalam silinder adalah sebagai berikut: 98, 183, 37, 122, 14, 124, 65, 67. Jika *head* pada awalnya berada pada 53, maka *head* akan bergerak dulu dari 53 ke 98, kemudian 183, 37, 122, 14, 124, 65, dan terakhir 67, dengan total pergerakan *head* sebesar 640 silinder.

Permasalahan dengan menggunakan penjadualan jenis ini dapat diilustrasikan dengan pergerakan dari 122 ke 14 dan kembali lagi ke 124. Jika permintaan terhadap silinder 37 dan 14 dapat dikerjakan/ dilayani secara bersamaan, baik sebelum mau pun setelah permintaan 122 dan 124, maka pergerakan total *head* dapat dikurangi secara signifikan, sehingga dengan demikian pendayagunaan akan meningkat.

6.7.2. Penjadualan SSTF

Sangat beralasan jika kita menutup semua pelayanan pada posisi *head* saat ini, sebelum menggerakkan *head* ke tempat lain yang jauh untuk melayani suatu permintaan. Asumsi ini mendasari algoritma penjadualan kita yang kedua yaitu *shortest-seek-time-first* (SSTF). Algoritma ini memilih permintaan dengan berdasarkan waktu pencarian atau *seek time* paling minimum dari posisi *head* saat itu. Karena waktu pencarian meningkat

seiring dengan jumlah silinder yang dilewati oleh *head*, maka SSTF memilih permintaan yang paling dekat posisinya di *disk* terhadap posisi *head* saat itu.

Perhatikan contoh antrian permintaan yang kita sajikan pada penjadualan FCFS, permintaan paling dekat dengan posisi *head* saat itu (53) adalah silinder 65. Jika kita enuhi permintaan 65, maka yang terdekat berikutnya adalah silinder 67. Dari 67, silinder 37 letaknya lebih dekat ke 67 dibandingkan silinder 98, jadi 37 dilayani duluan. Selanjutnya, dilanjutkan ke silinder 14, 98, 122, 124, dan terakhir adalah 183. Metode penjadualan ini hanya menghasilkan total pergerakan *head* sebesar 236 silinder -- kira-kira sepertiga dari yang dihasilkan penjadualan FCFS. Algoritma SSTF ini memberikan peningkatan yang cukup signifikan dalam hal pendayagunaan atau *performance* sistem.

Penjadualan SSTF merupakan salah satu bentuk dari penjadualan *shortest-job-first* (SJF), dan karena itu maka penjadualan SSTF juga dapat mengakibatkan *starvation* pada suatu saat tertentu. Kita ketahui bahwa permintaan dapat datang kapan saja. Anggap kita memiliki dua permintaan dalam antrian, yaitu untuk silinder 14 dan 186. Selama melayani permintaan 14, kita anggap ada permintaan baru yang letaknya dekat dengan 14. Karena letaknya lebih dekat ke 14, maka permintaan ini akan dilayani dulu sementara permintaan 186 menunggu gilirannya. Jika kemudian berdatangan lagi permintaan-permintaan yang letaknya lebih dekat dengan permintaan terakhir yang dilayani jika dibandingkan dengan 186, maka permintaan 186 bisa saja menunggu sangat lama. Kemudian jika ada lagi permintaan yang lebih jauh dari 186, maka juga akan menunggu sangat lama untuk dapat dilayani.

Walau pun algoritma SSTF secara substansial meningkat jika dibandingkan dengan FCFS, tetapi algoritma SSTF ini tidak optimal. Seperti contoh diatas, kita dapat menggerakkan *head* dari 53 ke 37, walau pun bukan yang paling dekat, kemudian ke 14, sebelum menuju 65, 67, 98, 122, dan 183. Strategi ini dapat mengurangi total gerakan *head* menjadi 208 silinder.

6.7.3. Penjadualan SCAN

Pada algoritma SCAN, pergerakan *disk arm* dimulai dari salah satu ujung *disk*, kemudian bergerak menuju ujung yang lain sambil melayani permintaan setiap kali mengunjungi masing-masing silinder. Jika telah sampai di ujung *disk*, maka *disk arm* bergerak berlawanan arah, kemudian mulai lagi melayani permintaan-permintaan yang muncul. Dalam hal ini *disk arm* bergerak bolak-balik melalui *disk*.

Kita akan menggunakan contoh yang sudah dibarikan diatas. Sebelum melakukan SCAN untuk melayani permintaan-permintaan 98, 183, 37, 122, 14, 124, 65, dan 67, kita harus mengetahui terlebih dahulu pergerakan *head* sebagai langkah awal dari 53. Jika *disk arm* bergerak menuju 0, maka *head* akan melayani 37 dan kemudian 14. Pada silinder 0, *disk arm* akan bergerak berlawanan arah dan bergerak menuju ujung lain dari *disk* untuk melayani permintaan 65, 67, 98, 122, 124, dan 183. Jika permintaan terletak tepat pada *head* saat itu, maka akan dilayani terlebih dahulu, sedangkan permintaan yang datang tepat dibelakang *head* harus menunggu dulu *head* mencapai ujung *disk*, berbalik arah, baru kemudian dilayani.

Algoritma SCAN ini disebut juga algoritma lift/ elevator, karena kelakuan *disk arm* sama seperti elevator dalam suatu gedung, melayani dulu orang-orang yang akan naik ke atas, baru kemudian berbalik arah untuk melayani orang-orang yang ingin turun ke bawah.

Kelemahan algoritma ini adalah jika banyak permintaan terletak pada salah satu ujung *disk*, sedangkan permintaan yang akan dilayani sesuai arah *arm disk* jumlahnya sedikit

atau tidak ada, maka mengapa permintaan yang banyak dan terdapat pada ujung yang berlawanan arah dengan gerakan *disk arm* saat itu tidak dilayani duluan? Ide ini akan mendasari algoritma penjadualan berikut yang akan kita bahas.

6.7.4. Penjadualan C-SCAN

Circular-SCAN adalah varian dari algoritma SCAN yang sengaja didesain untuk menyediakan waktu tunggu yang sama. Seperti halnya SCAN, C-SCAN akan menggerakkan *head* dari satu ujung *disk* ke ujung lainnya sambil melayani permintaan yang terdapat selama pergerakan tersebut. Tetapi pada saat *head* tiba pada salah satu ujung, maka *head* tidak berbalik arah dan melayani permintaan-permintaan, melainkan akan kembali ke ujung *disk* asal pergerakannya. Jika *head* mulai dari ujung 0, maka setelah tiba di ujung disk yang lainnya, maka *head* tidak akan berbalik arah menuju ujung 0, tetapi langsung bergerak ulang dari 0 ke ujung satunya lagi.

6.7.5. Penjadualan LOOK

Perhatikan bahwa SCAN dan C-SCAN menggerakkan *disk arm* melewati lebar seluruh *disk*. Pada kenyataanya algoritma ini tidak diimplementasikan demikian (pergerakan melewati lebar seluruh *disk*). Pada umumnya, *arm disk* bergerak paling jauh hanya pada permintaan terakhir pada masing-masin arah pergerakannya. Kemudian langsung berbalik arah tanpa harus menuju ujung *disk*. Versi SCAN dan C-SCAN yang berprilaku seperti ini disebut LOOK SCAN dan LOOK C-SCAN, karena algoritma ini melihat dulu permintaan-permintaan sebelum melanjutkan arah pergerakannya.

6.7.6. Pemilihan Algoritma Penjadualan *Disk*

Dari algoritma-algoritma diatas, bagaimanakah kita memilih algoritma terbaik yang akan digunakan? SSTF lebih umum dan memiliki prilaku yang lazim kita temui. SCAN dan C-SCAN memperlihatkan kemampuan yang lebih baik bagi sistem yang menempatkan beban pekerjaan yang berat kepada *disk*, karena algoritma tersebut memiliki masalah *starvation* yang paling sedikit. Untuk antrian permintaan tertentu, mungkin saja kita dapat mendefinisikan urutan akses dan pengambilan data dari *disk* yang optimal, tapi proses komputasi membutuhkan penjadualan optimal yang tidak kita dapatkan pada SSTF atau SCAN.

Dengan algoritma penjadualan yang mana pun, kinerja sistem sangat tergantung pada jumlah dan tipe permintaan. Sebagai contoh, misalnya kita hanya memiliki satu permintaan, maka semua algoritma penjadualan akan dipaksa bertindak sama, karena algoritma-algoritma tersebut hanya punya satu pilihan dari mana menggerakkan *disk head*: semuanya berprilaku seperti algoritma penjadualan FCFS.

Perlu diperhatikan pula bahwa pelayanan permintaan *disk* dapat dipengaruhi pula oleh metode alokasi file. Sebuah program yang membaca alokasi file secara terus menerus mungkin akan membuat beberapa permintaan yang berdekatan pada *disk*, menyebabkan pergerakan *head* menjadi terbatas. File yang memiliki link atau indeks, dilain pihak, mungkin juga memasukkan blok-blok yang tersebar luas pada *disk*, menyebabkan pergerakan *head* yang sangat besar.

Lokasi blok-blok indeks dan *directory* juga tidak kalah penting. Karena file harus dibuka sebelum digunakan, proses pembukaan file membutuhkan pencarian pada struktur *directory*, dengan demikian *directory* akan sering diakses. Kita anggap catatan *directory* berada pada awal silinder, sedangkan data file berada pada silinder terakhir. Pada kasus ini, *disk head* harus bergerak melewati sepanjang lebar *disk*. Membuat tempat penyimpanan sementara dari blok-blok indeks dan *directory* ke dalam memori

dapat membantu mengurangi pergerakan *disk arm*, khususnya untuk permintaan membaca *disk*.

Karena kerumitan inilah, maka algoritma penjadualan *disk* harus ditulis dalam modul terpisah dari sistem operasi, jadi dapat saling mengganti dengan algoritma lain jika diperlukan. Baik SSTF mau pun LOOK keduanya merupakan pilihan yang paling masuk akal sebagai algoritma yang paling dasar.

6.8. Managemen Disk

6.8.1. Memformat Disk

Sebuah disk magnetik yang baru sebenarnya hanyalah sebuah *slate* kosong yang berupa piringan magnetik untuk menyimpan sesuatu. Sebelum disk tersebut dapat menyimpan data, harus dilakukan proses *low-level formatting/ physical formatting*, yaitu membagi disk menjadi beberapa sektor dan mengisinya dengan struktur data tertentu (biasanya *header, area data*, dan *trailer*) agar dapat dibaca dan ditulis oleh *disk controller*.

Salah satu informasi yang dibutuhkan oleh *disk controller* adalah *error-correcting code* (ECC). Disebut seperti itu karena jika terdapat satu atau dua bit data yang *corrupt, controller* dapat mengidentifikasi bit mana yang berubah dan mengoreksi nya. Proses ini otomatis dilakukan oleh *controller* setiap membaca atau menulis pada disk.

Low-level formatting berfungsi agar pihak manufaktur dapat mengetes disk dan menginisialisasi mapping dari lojikal nomor blok ke pendeteksi sektor kosong. Semakin besar ukuran sektor yang diformat, semakin sedikit sektor yang dapat diisi pada masing-masing *track* dan semakin sedikit *header* dan *trailer* yang ditulis pada setiap *track*. Hal ini berarti ruang yang dapat digunakan untuk data semakin besar.

Agar disk dapat menggunakan suatu berkas, sistem operasi membutuhkan untuk menyimpan struktur datanya pada disk. Langkah pertama adalah membagi disk menjadi satu/ lebih silinder (*partition*), sehingga sistem operasi dapat memperlakukannya sebagai disk yang terpisah. Langkah kedua adalah logical formatting, atau membuat sistem berkas. Pada langkah ini, sistem operasi menyimpan struktur data yang telah diinisialisasi ke disk.

Raw I/O adalah *array* pada blok lojikal yang memiliki kemampuan untuk menggunakan suatu partisi disk tanpa struktur data dari sistem berkas. Dengan partisi *raw* ini, untuk beberapa aplikasi tertentu akan lebih efisien dari segi penyimpanan. Tetapi kebanyakan aplikasi akan berjalan lebih baik dengan servis sistem berkas biasa.

6.8.2. *Boot Block*

Ketika pertama kali menjalankan komputer, dibutuhkan program yang sudan diinisialisasi, yaitu *bootstrap*. Yang diinisialisasi adalah segala aspek sistem, dari *CPU register* sampai *device controller* dan isi dari *main memory*, kemudian menjalankan sistem operasi. Untuk itu *bootstrap* mencari *kernel* sistem operasi pada disk, me-*load*-nya ke memori, dan menggunakan alamat yang telah diinisialisasi untuk mulai menjalankan sistem operasi.

Hampir semua komputer menyimpan *bootstrap* pada *Read-Only Memory* (ROM). Alasannya karena ROM tidak membutuhkan inisialisasi dan berada pada lokasi yang

tetap dimana prosesor tetap dapat mengeksekusinya ketika komputer baru dinyalakan/ di-reset. Kelebihan lainnya karena ROM *read-only*, ia tidak dapat terkena virus. Tetapi masalah yang timbul adalah jika kita mengubah kode *bootstrap* berarti mengubah *chip* ROM juga. Untuk mengatasinya, sistem menyimpan *bootstrap loader* di ROM, yang hanya berfungsi untuk memasukkan seluruh program *bootstrap* dari disk. *Boot blocks* adalah suatu partisi untuk menyimpan seluruh program *bootstrap*. *Boot disk* atau *system disk* adalah disk yang
memiliki partisi *boot*.

6.8.3. *Bad Blocks*

Bad blocks adalah satu/lebih sektor yang rusak pada suatu disk. Pada disk sederhana, *bad blocks* diatasi secara manual. Untuk disk yang lebih kompleks seperti disk SCSI, *bad blocks* diatasi dengan *sector sparing* atau *forwarding*, yaitu *controller* dapat mengganti sektor yang rusak dengan sebuah sektor yang terpisah. Alternatif lainnya adalah mengganti sektor tersebut dengan cara *sector slipping*.

Mengganti blok yang rusak bukan sepenuhnya merupakan proses yang otomatis, karena data-data yang tersimpan sebelum nya akan terhapus.

6.9. Penanganan *Swap-Space*

Penanganan (*management*) *swap-space* (tempat pertukaran; tetapi karena istilah *swap-space* sudah umum dipakai, maka untuk seterusnya kita tetap memakai istilah *swap-space*) adalah salah satu dari *low-level* task pada sebuah sistem operasi. Memori Virtual menggunakan *disk space* sebagai perpanjangan (atau *space* tambahan) dari memori utama. Karena kecepatan akses disk lebih lambat daripada kecepatan akses memori, menggunakan *swap-space* akan mengurangi performa sistem secara signifikan. Tujuan utama dari perancangan dan implementasi *swap-space* adalah untuk menghasilkan kinerja memori virtual yang optimal. Dalam sub-bab ini, kita akan membicarakan bagaimana *swap-space* digunakan, dimana letak *swap-space* pada disk, dan bagaimana penanganan *swap-space*.

6.9.1. Penggunaan *Swap-Space*

Penggunaan *swap-space* pada berbagai macam sistem operasi berbeda-beda, tergantung pada algoritma *memory management* yang diimplementasikan. Sebagai contoh, sistem yang mengimplementasikan *swapping* mungkin akan menggunakan *swap-space* untuk menyimpan (dan mengerjakan) sebuah proses, termasuk segmen kode dan datanya. Sistem yang menggunakan *paging* hanya akan menyimpan *page* (atau "halaman " proses) yang sudah dikeluarkan dari memori utama. Besarnya *swap-space* yang dibutuhkan sebuah sistem bermacam-macam, tergantung dari banyaknya *physical memory* (RAM, seperti EDO DRAM, SDRAM, RD RAM), memori virtual yang disimpan di *swap-space*, dan caranya memori virtual digunakan. Besarnya bervariasi, antara beberapa megabytes sampai ratusan megabytes atau lebih.

Beberapa sistem operasi, seperti UNIX, menggunakan *swap-space* sebanyak yang diperlu kan. *Swap-space* ini biasanya disimpan dalam beberapa *disk* yang terpisah, jadi beban yang diterima oleh sistem I/O dari *paging* dan *swapping* bisa didistribusikan ke berbagai I/O *device* pada sistem.

Harap dicatat bahwa menyediakan *swap-space* yang berlebih lebih aman daripada kekurangan *swap-space*, karena bila kekurangan maka ada kemungkinan sistem

terpaksa menghentikan sebuah atau lebih proses atau bahkan membuat sistem menjadi *crash*. *Swap-space* yang berlebih memang membuang *disk space* yang sebenarnya bisa digunakan untuk menyimpan berkas (*file*), tapi setidaknya tidakmenimbulkan resiko yang lain.

6.9.2. Lokasi *Swap-Space*

Ada dua tempat dimana *swap-space* bisa berada: *swap-space* bisa diletakkan pada partisi yang sama dengan sistem operasi, atau pada partisi yang berbeda. Apabila *swap-space* yang dipakai hanya berupa sebuah berkas yang besar di dalam sistem berkas, maka sistem berkas yang dipakai bisa digunakan untuk membuat, menamakan, dan mengalokasikan tempat *swap-space*. Maka dari itu, pendekatan seperti ini mudah untuk diimplementasikan. Sayangnya, juga tidak efisien. Menelusuri struktur direktori dan struktur data alokasi disk memakan waktu, dan berpotensi untuk mengakses disk lebih banyak dari yang diperlukan. Fragmentasi eksternal bisa membuat *swapping* lebih lama dengan memaksakan pencarian sekaligus banyak (*multiple seeks*) ketika sedang membaca atau menulis sebuah proses. Kita bisa meningkatkan performa dengan meng-*cache* informasi lokasi blok pada *physical memory*, dan dengan menggunakan aplikasi khusus untuk mengalokasikan blok-blok yang *contiguous* (tidak terputus) untuk berkas *swap*-nya, dengan waktu tambahan untuk menelusuri struktur data *file-system* masih tetap ada.

Metode yang lebih umum adalah untuk membuat *swap-space* di partisi yang terpisah. Tidak ada sistem *file* atau struktur direktori di dalam partisi ini. Justru sebuah *swap-space storage manager* yang terpisah digunakan untuk mengalokasikan dan melepaskan blok-blok yang digunakan. *Manager* ini menggunakan algoritma yang dioptimalkan untuk kecepatan, daripada efisiensi tempat. Fragmentasi internal mungkin akan meningkat, tetapi ini bisa diterima karena data dalam *swap-space* biasanya umurnya lebih singkat daripada data-data di sistem *file*, dan *swap area*-nya diakses lebih sering.

Pendekatan ini membuat besar *swap-space* yang tetap selagi mempartisi *disk*. Menambah jumlah *swap-space* bisa dilakukan hanya me lalui mempartisi ulang *disk* (dimana juga termasuk memindahkan atau menghancurkan dan mengembalikan partisi *file-system* lainnya dari *backup*), atau dengan menambahkan *swap-space* di tempat lain.

Beberapa sistem operasi cukup fleksibel dan bisa *swapping* baik di partisi mentah (*raw*, belum di-format) dan di *file-system*. Contohnya Solaris 2. *Policy* dan implementasinya terpisah, sehingga administrator mesinnya (komputernya) bisa memutus kan mana yang akan digunakan. Pertimbangannya adalah antara kemudahan alokasi dan pengelolaan *file-system*, dan performa dari *swapping* pada partisi yang *raw*.

6.9.3. Pengelolaan *Swap-Space*

Untuk mengilustrasikan metode-metode yang digunakan untuk mengelola *swap-space*, kita sekarang akan mengikuti evolusi dari *swapping* dan *paging* pada GNU/ Linux. GNU/ Linux memulai dengan implementasi *swapping* yang menyalin seluruh proses antara daerah disk yang *contiguous* (tidak terputus) dan memori. UNIX berevolusi menjadi kombinasi dari *swapping* dan *paging* dengan tersedianya *hardware* untuk *paging*.

Dalam 4.3BSD, *swap-space* dialokasikan untuk proses ketika sebuah proses dimulai. Tempat yang cukup disediakan untuk menampung program, yang juga dikenal sebagai halaman-halaman teks (*text pages*) atau segmen teks, dan segmen data dari proses itu. Alokasi dini tempat yang dibutuhkan dengan cara seperti ini umumnya mencegah sebuah

proses untuk kehabisan *swap-space* selagi proses itu dikerjakan. Ketika proses mulai, teks di dalamnya di-*page* dari file system. Halaman-halaman (*pages*) ini akan ditaruh di *swap* bila perlu, dan dibaca kembali dari sana, jadi sistem *file* akan diakses sekali untuk setiap *text page*. Halaman-halaman dari segmen data dibaca dari sistem *file*, atau dibuat (bila belum sebelumnya), dan ditaruh di *swap space* dan di-*page* kembali bila perlu. Satu contoh optimisasi (sebagai contoh, ketika dua pengguna menggunakan editor yang sama) adalah proses-proses dengan text page yang identik membagi halaman-halaman (*pages*) ini, baik di memori mau pun di *swap-space*.

Dua peta *swap* untuk setiap proses digunakan oleh kernel untuk melacak penggunaan *swap-space*. Segmen teks besarnya tetap, maka *swap space* yang dialokasikan sebesar 512K setiap potong (*chunks*), kecuali untuk potongan terakhir, yang menyimpan sisa halaman-halaman (*pages*) tadi, dengan kenaikan (*increments*) sebesar 1K.

Peta swap dari Segmen data lebih rumit, karena segmen data bisa mem besar setiap saat. Petanya sendiri besarnya tetap, tapi menyimpan a lamat-alamat swap untuk blok-blok yang besarnya bervariasi. Misalkan ada index I, dengan besar maksimun 2 megabytes. Data struktur ini ditunjukkan oleh gambar 13.8. (Besar minimum dan maksi mum blok bervariasi, dan bisa diubah ketika me-reboot sistem.) Ketika sebuah proses mencoba untuk memperbesar segmen datanya melebihi blok yang dialokasikan di tempat swap, sistem operasi mengalokasikan blok lain lagi, dua kali besarnya yang pertama. Skema ini menyebab kan proses-proses yang kecil menggunakan blok-blok kecil. Ini juga meminimalisir fragmentasi. Blok-blok dari proses yang besar bisa di temukan dengan cepat, dan peta swap tetap kecil.

Pada Solaris 1 (SunOS 4), para pembuatnya membuat perubahan pada me tode standar UNIX untuk meningkatkan efisiensi dan untuk mencermin kan perubahan teknologi. Ketika sebuah proses berjalan, halaman-hala man (pages) dari segmen teks dibawa kembali dari sistem berkas, diak ses di memori utama, dan dibuang bila diputuskan untuk di-pageout. A kan lebih efisien untuk membaca ulang sebuah halaman (page) dari sis tem berkas daripada menaruhnya di swap-space dan membacanya ulang dari sana. Lebih banyak lagi perubahan pada Solaris 2. Perubahan terbesar ada lah Solaris 2 mengalokasikan swap-space hanya ketika sebuah halaman (page) dipaksa keluar dari memori, daripada ketika halaman (page) da ri memori virtual pertama kali dibuat. Perubahan ini memberikan per forma yang lebih baik pada komputer-komputer modern, yang sudah mem punyai memori lebih banyak daripada komputer-komputer dengan sistem yang sudah lama, dan lebih jarang melakukan paging.

6.10. Kehandalan Disk

Disk memiliki resiko untuk mengalami kerusakan. Kerusakan ini dapat berakibat turunnya performa atau pun hilangnya data. Meski pun terdapat *backup* data, tetap saja ada kemungkinan data yang hilang karena adanya perubahan setelah terakhir kali data di-*backup*. Karenanya reliabilitas dari suatu disk harus dapat terus ditingkatkan.

Berikut adalah beberapa macam penyebab terjadinya hilangnya data:

1. Ketidaksengajaan dalam menghapus.
   Bisa saja pengguna secara tidak sengaja menghapus suatu berkas, hal ini dapat dicegah seminimal mungkin dengan cara melakukan *backup* data secara reguler.

2. Hilangnya tenaga listrik
   Hilangnya tenaga listrik dapat mengakibatkan adanya *corrupt* data.

3. Blok rusak pada disk.
   Rusaknya blok pada disk dapat saja disebabkan dari umur disk tersebut. Seiring dengan waktu, banyaknya blok pada disk yang rusak dapat terus terakumulasi. Blok yang rusak pada disk, tidak akan dapat dibaca.

4. Rusaknya Disk.
   Bisa saja karena suatu kejadian disk rusak total. Sebagai contoh, dapat saja disk jatuh atau pun ditendang ketika sedang dibawa.

5. *System Corrupt*.
   Ketika komputer sedang dijalankan, bisa saja terjadi OS *error*, program *error*, dan lain sebagainya. Hal ini tentu saja dapat menyebabkan hilangnya data.

Berbagai macam cara dilakukan untuk meningkatkan kinerja dan juga reliabilitas dari disk. Biasanya untuk meningkatkan kinerja, dilibatkan banyak disk sebagai satu unit penyimpanan. Tiap-tiap blok data dipecah ke dalam beberapa subblok, dan dibagi-bagi ke dalam disk-disk tersebut. Ketika mengirim data disk-disk tersebut bekerja secara pararel. Ditambah dengan sinkronisasi pada rotasi masing-masing disk, maka kinerja dari disk dapat ditingkatkan. Cara ini dikenal sebagai RAID (*Redundant Array of Independent Disks*). Selain masalah kinerja RAID juga dapat meningkatkan reabilitas dari disk dengan jalan melakukan redundansi data.

Salah satu cara yang digunakan pada RAID adalah dengan *mirroring* atau *shadowing*, yaitu dengan membuat duplikasi dari tiap-tiap disk. Pada cara ini, berarti diperlukan media penyimpanan yang dua kali lebih besar daripada ukuran data sebenarnya. Akan tetapi, dengan cara ini pengaksesan disk yang dilakukan untuk membaca dapat ditingkatkan dua kali lipat. Hal ini dikarenakan setengah dari permintaan membaca dapat dikirim ke masing-masing disk. Cara lain yang digunakan pada RAID adalah *block interleaved parity*. Pada cara ini, digunakan sebagian kecil dari disk untuk penyimpanan *parity block*. Sebagai contoh, dimisalkan terdapat 10 disk pada array. Karenanya setiap 9 data *block* yang disimpan pada array, 1 parity *block* juga akan disimpan. Bila terjadi kerusakan pada salah satu *block* pada disk maka dengan adanya informasi pada *parity block* ini, ditambah dengan data *block* lainnya, diharapkan kerusakan pada disk tersebut dapat ditanggulangi, sehingga tidak ada data yang hilang. Penggunaan *parity block* ini juga akan menurunkan kinerja sama seperti halnya pada *mirroring*. Pada *parity block* ini, tiap kali *subblock* data ditulis, akan terjadi perhitungan dan penulisan ulang pada *parity block*.

6.11. Implementasi *Stable-Storage*

Pada bagian sebelumnya, kita sudah membicarakan mengenai write-ahead log, yang membutuhkan ketersediaan sebuah storage yang stabil. Berdasarkan definisi, informasi yang berada di dalam stable storage tidak akan pernah hilang. Untuk mengimplementasikan storage seperti itu, kita perlu mereplikasi informasi yang dibutuhkan ke banyak peralatan storage (biasanya disk-disk) dengan failure modes yang independen. Kita perlu mengkoordinasikan penulisan update-update dalam sebuah cara yang menjamin bila terjadi kegagalan selagi meng-update tidak akan membuat semua kopi yang ada menjadi rusak, dan bila sedang recover dari sebuah kegagalan, kita bisa memaksa semua kopi yang ada ke dalam keadaan yang bernilai benar dan konsisten, bahkan bila ada kegagalan lain yang terjadi ketika sedang recovery.

Untuk selanjutnya, kita akan membahas bagaimana kita bisa mencapai kebutuhan kita.

Sebuah disk write menyebabkan satu dari tiga kemungkinan:

1. Successful completion.
   Data disimpan dengan benar di dalam disk.

2. Partial failure.
   Kegagalan terjadi di tengah-tengah transfer, menyebabkan hanya bebe rapa sektor yang diisi dengan data yang baru, dan sektor yang diisi ketika terjadi kegagalan menjadi rusak.

3. Total failure.
   Kegagalan terjadi sebelum disk write dimulai, jadi data yang sebe lumnya ada pada disk masih tetap ada.

Kita memerlukan, kapan pun sebuah kegagalan terjadi ketika sedang me nulis ke sebuah blok, sistem akan mendeteksinya dan memanggil sebuah prosedur recovery untuk me-restore blok tersebut ke sebuah keadaan yang konsisten. Untuk melakukan itu, sistem harus menangani dua blok physical untuk setiap blok logical. Sebuah operasi output dieksekusi seperti berikut:

1. Tulis informasinya ke blok physical yang pertama.
2. Ketika penulisan pertama berhasil, tulis informasi yang sama ke blok physical yang kedua.
3. Operasi dikatakan berhasil hanya jika penulisan kedua berhasil.

Pada saat recovery dari sebuah kegagalan, setiap pasang blok physi cal diperiksa. Jika keduanya sama dan tidak terdeteksi adanya kesa lahan, tetapi berbeda dalam isi, maka kita mengganti isi dari blok yang pertama dengan isi dari blok yang kedua. Prosedur recovery se perti ini memastikan bahwa sebuah penulisan ke stable storage akan sukses atau tidak ada perubahan sama sekali.

Kita bisa menambah fungsi prosedur ini dengan mudah untuk memboleh kan penggunaan dari kopi yang banyak dari setiap blok pada stable storage. Meski pun sejumlah besar kopi semakin mengurangi kemungkin an untuk terjadinya sebuah kegagalan, maka biasanya wajar untuk men simulasi stable storage hanya dengan dua kopi. Data di dalam stable storage dijamin aman kecuali sebuah kegagalan menghancurkan semua kopi yang ada.

6.12. *Tertiary-Storage Structure*

Ciri-ciri Tertiary-Storage Structure:
- Biaya produksi lebih murah.
- Menggunakan *removable media*.
- Data yang disimpan bersifat permanen.

6.12.1. Macam-macam *Tertiary-Strorage Structure*

6.12.1.1. *Floppy Disk*

Gambar 6-4. Floppy Disk.

Floopy disk adalah fleksible disk yang tipis, dilapisi material yang bersifat magnet, dan ditutupi oleh plastik.

Ciri-ciri:
• Umumnya mempunyai kapasitas antara 1-2 MB.
• Kemampuan akses hampir seperti *hardisk*.

6.12.1.2. *Magneto-optic disk*

Gambar 6-5. Magneto Optic.

Magneto-optic Disk adalah Piringan *optic* yang keras dilapisi oleh material yang bersifat magnet, kemudian dilapisi pelindung dari plastik atau kaca yang berfungsi untuk menahan *head* yang hancur.

Drive ini mempunyai medan magnet. Pada suhu kamar, medan magnet terlalu kuat dan terlalu lemah untuk memagnetkan satu *bit* ke disk. Untuk menulis satu *bit*, *disk head* akan mengeluarkan sinar laser ke permukaan disk. Sinar laser ini ditujukan pada *spot* yang kecil. *Spot* ini adalah tempat yang ingin kita tulis satu *bit*. *Spot* yang ditembak sinar laser menjadi rentan terhadap medan magnet sehingga menulis satu *bit* dapat dilakukan baik pada saat medan magnet kuat mau pun lemah.

Magneto-optic disk head berjarak lebih jauh dari permukaan disk daripada *magnetic disk head*. Walau pun demikian, *drive* tetap dapat membaca *bit*, yaitu dengan bantuan sinar laser (disebut *Kerr effect*).

6.12.1.3. *Optical Disk*

Gambar 6-6. Optical Disk.

Disk ini tidak menggunakan sifat magnet, tetapi menggunakan bahan khusus yang dimodifikasi menggunakan sinar laser. Setelah dimodifikasi dengan dengan sinar laser pada *disk* akan terdapat *spot* yang gelap atau terang. *Spot* ini menyimpan satu *bit*.

Optical-disk teknologi terbagi atas:

1. *Phase-change disk*, dilapisi oleh material yang dapat membeku menjadi *crystalline* atau *amorphous state*. Kedua state ini memantulkan sinar laser dengan kekuatan yang berbeda. *Drive* menggunakan sinar laser pada kekuatan yang berbeda untuk mencairkan dan membekukan *spot* di disk sehingga *spot* berubah antara *crystalline* atau *amorphous state*.

2. *Dye-polimer disk*, merekam data dengan membuat *bump*. Disk dilapisi plastik yang mengandung *dye* yang dapat menyerap sinar laser. Sinar laser membakar *spot* yang kecil sehingga *spot* membengkak dan membentuk *bump*. Sinar laser juga dapat menghangatkan *bump* sehingga *spot* menjadi lunak dan *bump* menjadi datar.

6.12.1.4. WORM Disk (*Write Once, Read Many Times*)

Gambar 6-7.Worm Disk.

WORM adalah Aluminium film yang tipis dilapisi oleh piringan plastik atau kaca pada bagian atas dan bawahnya. Untuk menulis *bit*, *drive* tersebut menggunakan sinar laser untuk membakar *hole* yang kecil pada aluminium. *Hole* ini tidak dapat diubah seperti sebelumnya. Oleh karena itu, *disk* hanya dapat ditulis sekali.
Ciri-ciri:
• Data hanya dapat ditulis sekali.
• Data lebih tahan lama dan dapat dipercaya.
• *Read Only disk*, seperti *CD-ROM* dan *DVD* yang berasal dari pabrik sudah berisi data.

6.12.1.5. *Tapes*

Gambar 6-8. Tape.

Walau pun harga *tape drive* lebih mahal daripada *disk drive*, harga *tape cartridge* lebih murah daripada *disk cartridge* apabila dilihat dari kapasitas yang sama. Jadi, untuk penggunaan yang lebih ekonomis lebih baik digunakan *tape*. *Tape drive* dan *disk drive* mempunyai *transfer rate* yang sama. Tetapi, *random access tape* lebih lambat daripada disk karena *tape* menggunakan operasi *forward* dan *rewind*.

Seperti disebutkan diatas, *tape* adalah media yang ekonomis apabila media yang ingin digunakan tidak membutuhkan kemampuan *random access*, contoh: *backup* data dari data disk, menampung data yang besar. *Tape* digunakan oleh *supercomputer center* yang besar untuk menyimpan data yang besar. Data ini gunakan oleh badan penelitian ilmiah dan perusahaan komersial yang besar.

Pemasangan *tape* yang besar menggunakan *robotic tape changers*. R*obotic tape changers* memindahkan beberapa *tape* antara beberapa *tape drive* dan beberapa *slot* penyimpanan yang berada di dalam *tape library*. *Library* yang menyimpan beberapa tape disebut *tape stacker*. *Library* yang menyimpan ribuan tape disebut *tape silo*.

Robotic tape library mengurangi biaya penyimpanan data. *File* yang ada di disk dapat dipindahkan ke *tape* dengan tujuan mengurangi biaya penyimpanan. Apabila *file* itu ingin digunakan, maka komputer akan memindahkan *file* tadi ke disk.

6.12.2. Masalah-Masalah yang Berkaitan Dengan Sistem Operasi

1. Tugas terpenting dari sistem operasi adalah mengatur *physical devices* dan menampilkan abstarksi mesin virtual dari aplikasi (*Interface aplikasi*).

2. Untuk *hardisk*, OS menyediakan dua abstaksi, yaitu:
   - *Raw device* = *array* dari beberapa data blok.
   - *File sistem* = sistem operasi mengantrikan dan menjadwalkan beberapa permintaan *interleaved* yang berasal dari beberapa aplikasi.

6.12.3. *Interface Aplikasi*

Kebanyakan sistem operasi menangani *removable media* hampir sama dengan *fixed disk*, yaitu *cartridge* di *format* dan dibuat *file sistem* yang kosong pada disk.
Tapes ditampilkan sebagai media *raw storage* dan aplikasi tidak membuka file pada *tape*, tetapi *tapes* dibuka kesemuanya sebagai *raw device*. Biasanya *tape drive* disediakan untuk penggunaan khusus dari suatu aplikasi sampai aplikasi berakhir atau menutup *tape drive*. Penggunaan khusus ini dikarenakan *random access tape* membutuhkan waktu yang lama. Jadi, *interleaving random access* oleh *tape* oleh beberapa aplikasi akan menyebabkan *thrashing*. Sistem operasi tidak menyediakan *file system* sehingga aplikasi harus memutuskan bagaimana cara menggunakan *array* dari blok-blok. Sebagai contoh, program yang mem-*backup hardisk* ke *tape* akan mendaftar nama file dan kapasitas file pada permulaan *tape*. Kemudian, program meng-*copy* data file ke *tape*. Setiap aplikasi mempunyai caranya masing-masing untuk mengatur *tape* sehingga *tape* yang telah penuh terisi data hanya dapat digunakan oleh program yang membuatnya. Operasi dasar *tape drive* berbeda dengan operasi dasar disk drive. Contoh operasi dasar *tape drive*:

- Operasi *locate* berfungsi untuk menetapkan posisi *tape head* ke sebuah *logical blok*. Operasi ini mirip operasi yang ada di disk, yaitu: operasi *seek*. Operasi *seek* berfungsi untuk menetapkan posisi semua *track*.

- Operasi *read position* berfungsi memberitahu posisi *tape head* dengan menunjukkan nomor *logical blok*.

- Operasi *space* berfungsi memindahkan posisi *tape head*. Misalnya operasi *space* -2 akan memindahkan posisi *tape head* sejauh dua blok ke belakang.

Kapasitas blok ditentukan pada saat blok ditulis. Apabila terdapat area yang rusak pada saat blok ditulis, maka area yang rusak itu tidak dipakai dan penulisan blok dilanjutkan setelah daerah yang rusak tersebut.

Tape drive "append-only" devices, maksudnya adalah apabila kita meng-*update* blok yang ada di tengah berarti kita akan menghapus semua data sebelumnya pada blok tersebut. Oleh karena itu, *meng-update* blok tidak diperbolehkan.

Untuk mencegah hal tadi digunakan tanda *EOT* (*end-of-tape*). Tanda EOT ditaruh setelah sebuah blok ditulis. *Drive* menolak ke lokasi sebelum tanda EOT, tetapi *drive* tidak menolak ke lokasi tanda EOT kemudian *drive* mulai menulis data. Setelah selesai menulis data, tanda EOT ditaruh setelah blok yang baru ditulis tadi.

6.12.4. Penamaan Berkas

Menamakan berkas pada *removable media* cukup sulit terutama pada saat kita menulis data pada *removable cartridge* pada suatu komputer, kemudian menggunakan *cartridge* ini pada komputer yang lain. Jika jenis komputer yang digunakan sama dan jenis *cartridge* yang digunakan sama, maka permasalahannya adalah mengetahui isi dan *data layout* dari *cartridge*. Tetapi, bila jenis komputer yang digunakan dan jenis *drive* yang digunakan berbeda, maka berbagai masalah akan muncul. Apabila hanya jenis *drive* yang digunakan sama, komputer yang berbeda menyimpan *bytes* dengan berbagai cara dan juga menggunakan *encoding* yang berbeda untuk *binary number* atau huruf.

Pada umumnya sistem operasi sekarang tidak memperdulikan masalah penamaan *space* pada *removable media*. Hal ini tergantung kepada aplikasi dan user bagaimana cara mengakses dan menterjemahkan data. Tetapi, beberapa jenis *removable media* (contoh: CDs) distandarkan cara menggunakannya untuksemua jenis komputer.

6.12.5. Managemen Penyimpanan Hirarkis

Managemen Penyimpanan Hirarkis (*Hierachical Storage management*) menjelaskan *storage hierarchy* antara *primary memory* dan *secondary storage* untuk membentuk *tertiary storage*. *Tertiary storage* biasanya diimplementasikan sebagai *jukebox* dari *tapes* atau *removable media*.

Walau pun *tertiary storage* dapat memepergunakan sistem *virtual-memory*, cara ini tidak baik. Karena pengambilan data dari *jukebox* membutuhkan waktu yang agak lama. Selain itu diperlukan waktu yang agak lama untuk *demand paging* dan untuk bentuk lain dari penggunaan *virtual-memory*.

File yang kapasitasnya kecil dan sering digunakan disimpan di disk. Sedangkan file yang kapasitasnya besar, sudah lama, dan tidak aktif akan diarsipkan di *jukebox*. Pada beberapa sistem *file-archiving*, *directory entry* untuk file selalu ada, tetapi isi file tidak berada di *secondary storage*. Jika aplikasi mencoba membuka file, pemanggilan *open system* akan ditunda sampai isi file dikirim dari *tertiary storage*. Ketika isi file sudah ada di *secondary storage*, operasi *open* dikembalikan ke aplikasi.

Hierachical Storage management biasanya ditemukan pada pusat *supercomputing* dan *installasi* besar lainnya yang mempunyai data yang besar.

6.13. Rangkuman

6.13.1. *I/O*

Dasar dari elemen perangkat keras yang terkandung pada *I/O* adalah *bus*, *device controller*, dan *I/O* itu sendiri. Kinerja kerja pada data yang bergerak antara device dan memori utama di jalankan oleh CPU, di program oleh *I/O* atau mungkin DMA *controller*. Modul kernel yang mengatur *device* adalah *device driver*. *System-call interface* yang disediakan aplikasi dirancang untuk menghandle beberapa dasar kategori dari perangkat keras, termasuk *block devices, character devices, memory mapped files, network sockets* dan *programmed interval timers*.

Subsistem *I/O* kernel menyediakan beberapa servis. Diantaranya adalah *I/O schedulling, buffering, spooling, error handling* dan *device reservation*. Salah satu servis dinamakan *translation*, untuk membuat koneksi antara perangkat keras dan nama file yang digunakan oleh aplikasi.

I/O system calls banyak dipakai oleh CPU, dikarenakan oleh banyaknya lapisan dari perangkat lunak antara *physical device* dan aplikasi. Lapisan ini mengimplikasikan *overhead* dari alih konteks untuk melewati *kernel's protection boundary*, dari sinyal dan *interrupt handling* untuk melayani *I/O devices*.

6.13.2. Disk

Disk drives adalah *major secondary-storage I/O device* pada kebanyakan komputer. Permintaan untuk disk *I/O* digenerate oleh sistem file dan sistem virtual memori. Setiap permintaan menspesifikasikan alamat pada disk untuk dapat direferensikan pada *form* di *logical block number*.

Algoritma *disk schedulling* dapat meningkatkan efektifitas *bandwith*, *average response time*, dan *variance response time*. Algoritma seperti SSTF, SCAN, C-SCAN, LOOK dan C-LOOK didesain untuk membuat perkembangan dengan menyusun ulang antrian disk untuk meningkatkan total waktu pencarian.

Kinerja dapat rusak karena *external fragmentation*. Satu cara untuk menyusun ulang disk untuk mengurangi fragmentasi adalah untuk *back up* dan *restore* seluruh disk atau partisi. Blok-blok dibaca dari lokasi yang tersebar, me-*restore* tulisan mereka secara berbeda. Beberapa sistem mempunyai kemampuan untuk men-*scan* sistem file untuk mengidentifikasi file terfragmentasi, lalu menggerakan blok-blok mengelilingi untuk meningkatkan fragmentasi. Men-defragmentasi file yang sudah di fragmentasi (tetapi hasilnya kurang optimal) dapat secara signifikan meningkatkan kinerja, tetapi sistem ini secara umum kurang berguna selama proses defragmentasi sedang berjalan. Sistem operasi me-*manage* blok-blok pada disk. Pertama, disk baru di format secara *low level* untuk menciptakan sektor pada perangkat keras yang masih belum digunakan. Lalu, disk dapat di partisi dan sistem file diciptakan, dan blok-blok boot dapat dialokasikan. Terakhir jika ada blok yang terkorupsi, sistem harus mempunyai cara untuk me-*lock out* blok tersebut, atau menggantikannya dengan cadangan.

Tertiary storage di bangun dari disk dan *tape drives* yang menggunakan media yang dapat dipindahkan. Contoh dari *tertiary storage* adalah *magnetic tape*, *removable magnetic*, dan *magneto-optic disk*.

Untuk *removable disk*, sistem operasi secara general menyediakan servis penuh dari sistem file *interface*, termasuk *space management* dan *request-queue schedulling*. Untuk

tape, sistem operasi secara general hanya menyediakan *interface* yang baru. Banyak sistem operasi yang tidak memiliki *built-in support* untuk *jukeboxes*. *Jukebox support* dapat disediakan oleh *device driver*.

6.14. Soal Latihan

Perangkat Keras I/O

1. Gambarkan diagram dari Interrupt Driven I/O Cycle.
2. Sebutkan langkah-langkah dari transfer DMA!
3. Apakah perbedaan dari polling dan interupsi?
4. Apa hubungan arsitektur kernel yang di-thread dengan implemen tasi interupsi?

Interface Aplikasi I/O

1. Kenapa dibutuhkan interface pada aplikasi I/O?
2. Apa tujuan adanya device driver? Berikan contoh keuntungan yang kita dapatkan dengan adanya hal ini!

Kernel I/O Subsystem

1. Apakah yang dimaksud dengan proses *pooling*? (jelaskan dengan jelas)
2. Mengapa diperlukan proses *pooling*?
3. Apakah yang dimaksud dengan *buffer*?
4. Jelaskan dengan singkat mengenai *I/O Scheduling*!

Penanganan Permintaan I/O

1. Apakah kegunaan dari Streams pada Sistem V UNIX?
2. Jelaskan lifecycle dari permintaan pembacaan blok!

Performa I/O

1. Gambarkan bagan mengenai komunikasi antar komputer
2. Bagaimana cara meningkatkan efisiensi performa I/O
3. Jelaskan mengenai implementasi dari fungsi I/O

Struktur Disk

1. Sebutkan bagian-bagian dari disk
2. Apa keuntungan penggunaan pemetaan pada disk?

Penjadualan Disk

1. Buatlah dengan pemikiran Anda sendiri, strategi penjadualan disk yang tepat dan efisien menurut Anda
2. Menurut Anda, diantara algoritma-algoritma penjadualan disk diatas manakah yang paling cepat, manakah yang paling efisien (hemat/tidak mahal), dan manakah yang paling lambat dan tidak efisien? Jelaskan!

Managemen Disk

1. Bagaimana cara disk SCSI me-recovery blok yang rusak? Jelaskan selengkap mungkin!

Penanganan *Swap-Space*

1. Bagaimana penanganan *swap space* pada disk?
2. Bagaimana pengelolaan *swap space* pada disk?

Reabilitas Disk

1. Terangkan bagaimana RAID dapat meningkatkan reabilitas dari disk?
2. Adakah batas waktu hidup suatu disk? Jika ada, berapa lama? Jika tidak, kenapa?

Implementasi *Stable-Storage*

1. Sebutkan kemungkinan-kemungkinan dari disk write!
2. Bagaimanakah suatu operasi output dieksekusi?

Tertiary-Storage Structure

1. Sebutkan kelebihan *tertiary storage structure*?
2. Apakah kegunaan EOT pada *tapes*? Jelaskan cara kerjanya?
3. Jelaskan tugas sistem operasi terhadap *tertiary-storage structure*?

6.15. Rujukan

1. Applied Operating System Concept, Silberschatz, Galvin, Gagne, 1999
2. *DMA Interface* (http://www.eso.org/projects/iridt/irace/aboutirace.html)
3. *I/O Transfer Method* (http://www.ebiz.com.pk/pakistan/dma.doc)

6.16. Daftar Istilah

I/O = I/O (Input/Output)
hardware -> perangkat keras
device = device
storage device -> device penyimpanan
disk = disk
transmission = transmission
processor -> prosesor
human-interface device = human-interface device
instruction -> instruksi
direct I/O instruction = direct I/O instruction
memory-mapped I/O = memory-mapped I/O
port = port (perangkat keras)
bus = bus (perangkat keras)
daisy chain = daisy chain
shared direct access = shared direct access
controller = controller
host adapter = host adapter
command-ready =command-ready
busy = busy
error = error
host = host
polling = polling

looping = looping
status register -> register status
service = service
CPU processing = CPU processing
Interrupt -> Interupsi
request line = request line
pointer = pointer
interrupt handler/ing = interrupt handler/ing
interrupt controller = interrupt controller
critical state = critical state, efisiensi
interrupt priority level system = interrupt priority level system
interrupt request line = interrupt request line
nonmaskable interrupt = nonmaskable interrupt
maskable interrupt = maskable interrupt
critical instruction sequence = critical instruction sequence
interrupt vector = interrupt vector
interrupt chaining = interrupt chaining
offset = offset
overhead = overhead
exception = exception
page fault = page fault
system call = system call
software interrupt = software interrupt
trap = trap
DMA = Direct Memory Access
command block = command block
transfer destination -> destinasi transfer
address -> alamat (istilah komputer dalam penunjukkan lokasi)
block -> blok
burst mode = burst mode
single burst = single burst
microprocessor -> mikroprosesor
idle = idle
cycle stealing mode = cycle stealing mode
handshaking = handshaking
DMA request = DMA request
DMA acknowledge = DMA acknowledge
memory-address -> alamat memori
cycle stealing = cycle stealing
virtual address -> alamat virtual
physical memory -> memori fisik
performance -> performa
device driver = device driver
memory bus -> bus memori
controller = controller
physical memory = physical memory
application space data = application space data
context switch = alih konteks
device = device
interrupt -> interupsi
smart controller = smart controller
polling = polling
concurrency = concurrency
channel = channel
memory subsystem = memory subsystem
bus = bus

application code = kode aplikasi
bugs = bugs
reboot = reboot
reload = reload
overhead = overhead
internal kernel -> kernel internal
messaging = messaging
threading = threading
locking = locking
debug = debug
crash = crash
block reads = block reads
write = write
workload = workload
secondary storage -> penyimpanan sekunder
magnetic tape = magnetic tape
tape = tape
backup = backup
disk drive = disk drive
logic block -> blok lojik
bytes = bytes
low level formatted = low level formatted
logical block number -> nomor blok lojikal
disk address -> alamat disk
sector -> sektor
hardware = hardware
disk drives = disk drives
bandwith disk = bandwith disk
seek time -> waktu pencarian
disk arm = disk arm
head = head
disk = disk
bandwith = bandwith
bytes = bytes
input = input
output = output
controller = controller
memory address = alamat memori
First-come First-serve = First-com First-serve
shortest-seek-time-first = shortest-seek-time-first
shortest-job-first = shortest-job-first
starvation = starvation
schedulling -> penjadwalan
disk arm = disk arm
Circular-SCAN = Circular-SCAN
variance -> varian
index -> indeks
directory = directory
disk head = disk head
magnetic disk = disk magnetik
slate = slate
low-level formatting = low-level formatting
physical formatting = physical formatting
trailer = trailer
disk controller = disk controller
partition = partition

I/O = I/O
logical block -> blok lojikal
raw I/O = raw I/O
main memory = memori utama
bootstrap = boostrap
boot disk = boot disk
bad blocks = bad blocks
sector slipping = sector slipping
interface = interface
I/O Application -> aplikasi I/O
software layering = software layering
device driver = device driver
layer -> lapisan
disk drive = disk drive
block device = block device
random-access = random-access
stream character -> karakter stream
library = library
network device -> peralatan jaringan
interface socket = interface socket
local socket = local socket
remote socket = remote socket
clock -> jam
timer = timer
trigger = trigger
programmable interval timer = programmable interval timer
scheduler = scheduler
timer request = timer request
hardware timer = hardware timer
blocking (application) = blocking (application)
nonblocking (application) = nonblocking (application)
wait queue = wait queue
run queue = run queue
physical action = physical action
asynchronous = asynchronous