PROCESS MANAGEMENT

Tugas Kelompok 3

Disusun Guna Memenuhi Salah Satu Tugas Mata Kuliah Sistem Operasi

CCS210 – Sistem Operasi Sesi KJ010

Dosen Pengajar : NUGROHO BUDHISANTOSA

DISUSUN OLEH:

TOMMY SOEMITRO	**(20170801246)**
REIO OCTAVIANUS	**(20170801220)**
DENDY REFLY YOANES	**(20170801289)**
DODOT NANDA T	**(20170801224)**
JUNISAI DAUD	**(20170801**

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS ESA UNGGUL
TAHUN 2017-2018

VERSI BAHASA INDONESIA

OPERATING SYSTEM CONCEPTS

Abraham Silberschatz
Peter Baer Galvin
Greg Gagne

Ninth Edition

DAFTAR ISI

Halaman

BAGIAN II
PROCESS MANAGEMENT

Suatu proses dapat dianggap sebagai program dalam eksekusi. Proses akan membutuhkan sumber daya tertentu — seperti waktu CPU, memori, file, dan perangkat I / O — untuk menyelesaikan tugasnya. Sumber daya ini dialokasikan untuk proses baik ketika dibuat atau ketika sedang mengeksekusi.

Suatu proses adalah unit kerja di sebagian besar sistem. Sistem terdiri dari kumpulan proses: proses sistem operasi mengeksekusi kode sistem, dan proses pengguna mengeksekusi kode pengguna. Semua proses ini dapat dijalankan secara bersamaan.

Meskipun secara tradisional sebuah proses hanya berisi satu rangkaian kontrol ketika dijalankan, kebanyakan sistem operasi modern sekarang mendukung proses yang memiliki banyak rangkaian.

Sistem operasi bertanggung jawab untuk beberapa aspek penting dari proses dan manajemen rangkaian : pembuatan dan penghapusan proses pengguna dan sistem; penjadwalan proses; dan penyediaan mekanisme untuk sinkronisasi, komunikasi, dan penanganan kebuntuan untuk proses.

.

BAB III
PROCESSES

Disaat awal, Komputer hanya mengizinkan satu program untuk dieksekusi pada suatu waktu. Program ini memiliki kontrol penuh terhadap sistem dan memiliki akses ke semua sumber daya sistem. Sebaliknya, sistem komputer kontemporer memungkinkan banyak program dimuat ke dalam memori dan dijalankan secara bersamaan. Evolusi ini membutuhkan kontrol yang lebih kuat dan lebih banyak kompartementalisasi dari berbagai program; dan kebutuhan ini menghasilkan gagasan proses, yang merupakan program dalam eksekusi. Suatu proses adalah unit kerja dalam sistem pembagian waktu modern.

Semakin kompleks sistem operasi, semakin diharapkan untuk dilakukan atas nama penggunanya. Meskipun perhatian utamanya adalah eksekusi program pengguna, itu juga perlu untuk mengurus berbagai tugas sistem yang lebih baik dibiarkan di luar kernel itu sendiri. Oleh karena itu sistem terdiri dari kumpulan proses: operasi proses sistem mengeksekusi kode sistem dan proses pengguna mengeksekusi kode pengguna. Secara potensial, semua proses ini dapat dijalankan secara bersamaan, dengan CPU (atau CPU) digandakan di antara mereka. Dengan mengalihkan CPU antar proses, sistem operasi dapat membuat komputer lebih produktif. Dalam bab ini, Anda akan membaca tentang apa itu proses dan bagaimana cara kerjanya.

TUJUAN BAB

- Untuk memperkenalkan gagasan tentang suatu proses — program dalam eksekusi, yang membentuk dasar dari semua perhitungan.
- Untuk menggambarkan berbagai fitur proses, termasuk penjadwalan, pembuatan, dan penghentian.
- Untuk mengeksplorasi komunikasi interproses menggunakan memori bersama dan pengiriman pesan.
- Untuk menggambarkan komunikasi dalam sistem client-server.

3.1 PROCESS CONCEPT

Sebuah pertanyaan yang muncul dalam membahas sistem operasi melibatkan apa yang disebut semua kegiatan CPU. Sistem batch menjalankan pekerjaan, sedangkan sistem time-shared memiliki program pengguna, atau tugas. Bahkan pada sistem pengguna tunggal, pengguna mungkin dapat menjalankan beberapa program sekaligus: pengolah kata, browser Web, dan paket e-mail. Dan bahkan jika pengguna dapat menjalankan hanya satu program dalam satu waktu, seperti pada perangkat yang disematkan yang tidak mendukung multitasking, sistem operasi mungkin perlu mendukung aktivitas internal yang diprogram sendiri, seperti manajemen memori. Dalam banyak hal, semua kegiatan ini serupa, jadi kami menyebut semuanya proses.

Istilah Job dan Process digunakan hampir secara bergantian dalam teks ini. Meskipun kami secara pribadi lebih menyukai proses istilah, banyak teori dan terminologi sistem operasi yang dikembangkan selama waktu ketika aktivitas utama sistem operasi adalah pemrosesan pekerjaan. Akan menyesatkan untuk menghindari penggunaan istilah yang diterima umum yang mencakup pekerjaan kata (seperti penjadwalan pekerjaan) hanya karena proses telah menggantikan pekerjaan.

3.1.1 THE PROCESS

Secara informal, seperti yang disebutkan sebelumnya, proses adalah program dalam eksekusi. Suatu proses lebih dari kode program, yang kadang-kadang dikenal sebagai text section. Ini juga termasuk aktivitas saat ini, sebagaimana diwakili oleh nilai program counter dan isi register prosesor. Suatu proses umumnya juga mencakup proses tumpukan(stack), yang berisi data sementara (seperti parameter fungsi, alamat pengirim, dan variabel lokal), dan bagian data, yang berisi variabel global. Suatu proses dapat juga termasuk suatu heap, yang merupakan memori yang secara tersirat dialokasikan selama waktu kerja. Struktur memori dalam memori ditunjukkan pada Gambar 3.1.

Kami menekankan bahwa program itu sendiri bukanlah sebuah proses. Program adalah entitas pasif, seperti file yang berisi daftar instruksi yang disimpan di disk (sering disebut file yang dapat dieksekusi/ executable file). Sebaliknya, proses adalah entitas aktif, dengan penghitung program yang menetapkan instruksi berikutnya untuk mengeksekusi dan satu set sumber daya terkait. Suatu program menjadi suatu proses ketika file yang dapat dieksekusi dimuat ke dalam memori. Dua teknik umum untuk memuat file yang dapat dieksekusi

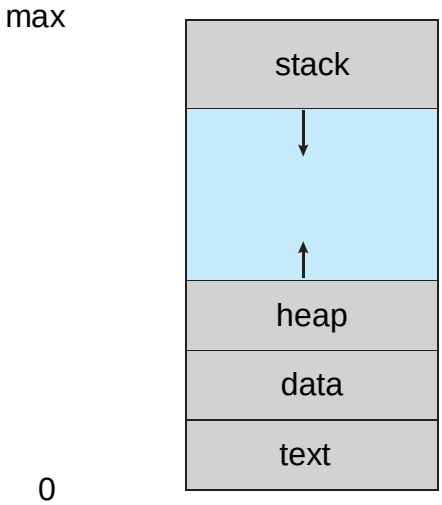


Figure 3.1 Process in memory.

mengklik dua kali ikon yang mewakili file yang dapat dieksekusi dan memasukkan nama file yang dapat dieksekusi pada baris perintah (seperti dalam prog.exe atau a.out).
Meskipun dua proses mungkin terkait dengan program yang sama, mereka tetap dianggap sebagai dua urutan eksekusi terpisah. Misalnya, beberapa pengguna mungkin menjalankan salinan program email yang berbeda, atau pengguna yang sama dapat meminta banyak salinan program browser web. Masing-masing ini adalah proses yang terpisah; dan meskipun bagian teks setara, bagian data, tumpukan, dan tumpukan bervariasi. Ini juga umum untuk memiliki proses yang memunculkan banyak proses saat berjalan. Kami membahas hal-hal tersebut di Bagian 3.4.

Perhatikan bahwa proses itu sendiri dapat menjadi lingkungan eksekusi untuk kode lain. Lingkungan pemrograman Java memberikan contoh yang baik. Dalam sebagian besar keadaan, program Java yang dapat dieksekusi dijalankan dalam mesin virtual Java (JVM). JVM mengeksekusi sebagai proses yang menafsirkan kode Java yang dimuat dan mengambil tindakan (melalui instruksi mesin asli) atas nama kode itu. Misalnya, untuk menjalankan Program Java yang dikompilasi Program .Class, kita akan masuk

Java Program

Perintah java menjalankan JVM sebagai proses biasa, yang pada gilirannya menjalankan Program Program Java di mesin virtual. Konsepnya sama dengan simulasi, kecuali bahwa kode, alih-alih ditulis untuk set instruksi yang berbeda, ditulis dalam bahasa Java.

3.1.2 PROCESS STATE

Saat proses dijalankan, ia mengubah status. Keadaan suatu proses didefinisikan sebagian oleh aktivitas saat ini dari proses itu. Suatu proses dapat berada di salah satu dari status berikut:

- **New.** Proses sedang dibuat.
- **Running**. Instruksi sedang dieksekusi.
- **Waiting.** Prosesnya menunggu beberapa peristiwa terjadi (seperti penyelesaian I / O atau penerimaan sinyal).
- **Ready**. Prosesnya menunggu untuk ditugaskan ke prosesor.
- **Terminated**. Prosesnya telah selesai eksekusi.

Nama-nama ini beragam, dan mereka bervariasi di seluruh sistem operasi. Negara-negara yang mereka wakili ditemukan pada semua sistem, namun. Sistem operasi tertentu juga lebih menggambarkan status proses. Penting untuk disadari bahwa hanya satu proses yang dapat berjalan pada prosesor apa pun secara instan. Banyak proses yang mungkin siap dan menunggu. Diagram keadaan yang sesuai dengan keadaan ini disajikan pada Gambar 3.2.

3.1.3 PROCESS CONTROL BLOCK

Setiap proses direpresentasikan dalam sistem operasi oleh Process Control Block (PCB) —juga disebut Task Control Block. PCB ditunjukkan pada Gambar3.3. Berisi banyak informasi yang terkait dengan proses tertentu, termasuk ini:

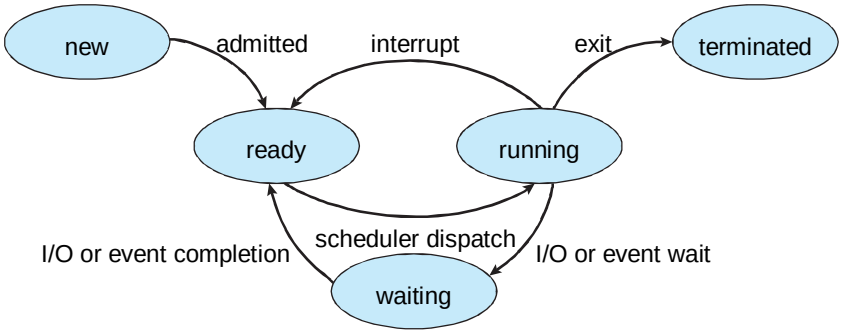
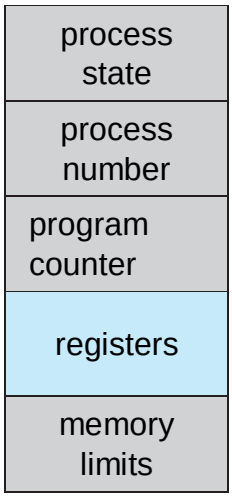


Figure3.2 Diagram of process state.

- **Process State.** Negara mungkin baru, siap, berjalan, menunggu, dihentikan, dan seterusnya.
- **Program Counter**. Penghitung menunjukkan alamat instruksi berikutnya yang akan dieksekusi untuk proses ini.
- **CPU Registers.** Register bervariasi dalam jumlah dan jenis, tergantung pada arsitektur komputer. Mereka termasuk akumulator, register indeks, penunjuk tumpukan, dan register tujuan umum, ditambah informasi kode kondisi apa pun. Bersama dengan penghitung program, informasi status ini harus disimpan ketika interrupt terjadi, untuk memungkinkan proses dilanjutkan dengan benar sesudahnya (Gambar 3.4).
- **CPU-scheduling information.** Informasi ini mencakup prioritas proses, pointer ke antrian penjadwalan, dan parameter penjadwalan lainnya. (Bab 6 menjelaskan penjadwalan proses.)
- **Memory-management information**. Informasi ini dapat mencakup item-item seperti nilai register dasar dan batas serta tabel halaman, atau tabel segmen, tergantung pada sistem memori yang digunakan oleh sistem operasi (Bab 8).



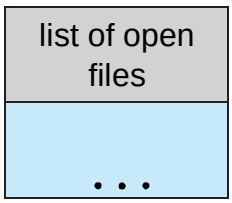


Figure 3.3 Process control block (PCB).

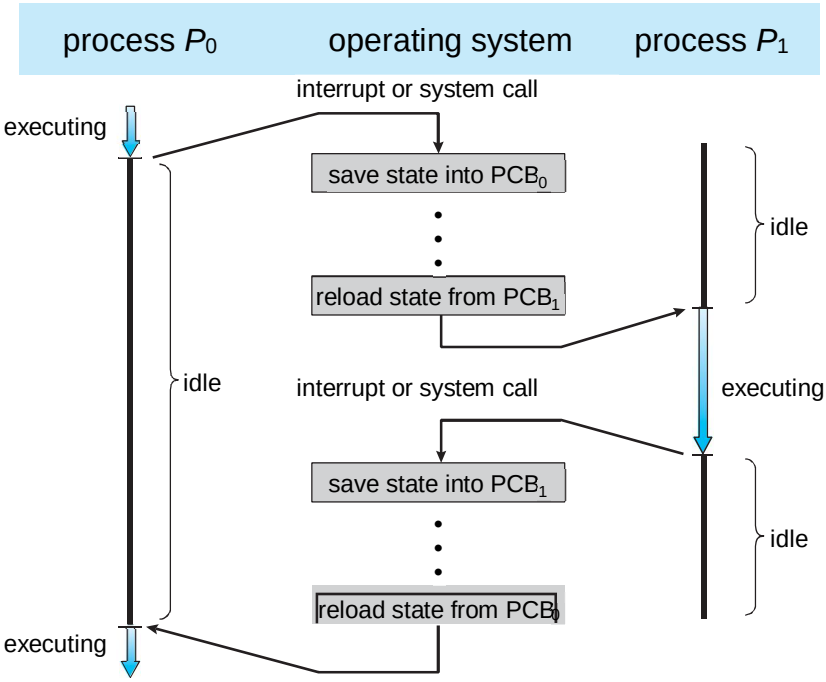


Figure 3.4 Diagram showing CPU switch from process to process.

- **Accounting Information**. Informasi ini termasuk jumlah CPU dan waktu nyata yang digunakan, batas waktu, nomor rekening, nomor pekerjaan atau proses, dan sebagainya.
- **I/O Status Information.** Informasi ini termasuk daftar perangkat I / O yang dialokasikan untuk proses, daftar file yang terbuka, dan sebagainya.

Singkatnya, PCB hanya berfungsi sebagai repositori untuk setiap informasi yang dapat bervariasi dari proses ke proses.

3.1.4 THREADS (RANGKAIAN)

Model proses yang dibahas sejauh ini menyiratkan bahwa suatu proses adalah program yang melakukan satu rangkaian eksekusi. Misalnya, ketika sebuah proses sedang menjalankan program pengolah kata, satu utas instruksi sedang dijalankan. Satu utas kontrol ini memungkinkan proses untuk melakukan hanya satu tugas pada satu waktu. Pengguna tidak dapat secara bersamaan mengetik karakter dan menjalankan pemeriksa ejaan dalam proses yang sama, misalnya. Sebagian besar sistem operasi modern telah memperluas konsep proses untuk memungkinkan suatu

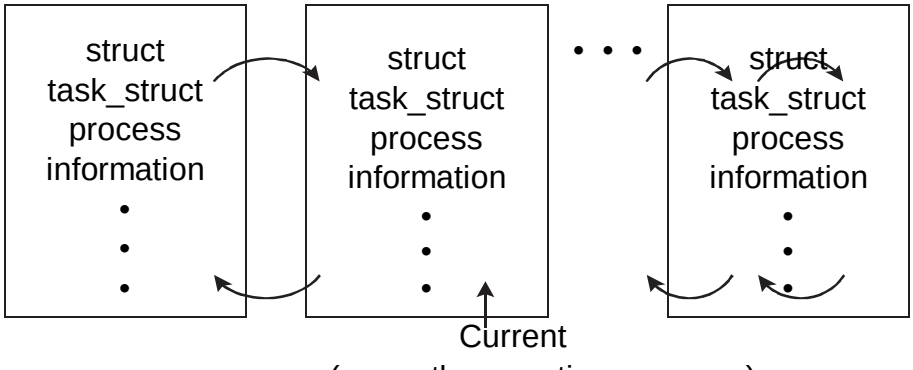
proses untuk memiliki beberapa utas eksekusi dan dengan demikian melakukan lebih dari satu tugas pada satu waktu. Fitur ini sangat bermanfaat pada sistem multicore, di mana beberapa utas dapat berjalan secara paralel. Pada sistem yang mendukung untaian, PCB diperluas untuk memasukkan informasi untuk setiap utas. Perubahan lain di seluruh sistem juga diperlukan untuk mendukung untaian. Bab 4 mengeksplorasi untaian secara detail.

REPRESENTASI PROSES DI LINUX

Blok kontrol proses dalam sistem operasi Linux diwakili oleh struktur C task_struct , yang ditemukan dalam<linux / sched.h> termasuk file di dalam direktori kode sumber kernel. Struktur ini berisi semua informasi yang diperlukan untuk mewakili suatu proses, termasuk keadaan proses, penjadwalan dan informasi manajemen memori, daftar file yang terbuka, dan pointer ke process's parent dan daftar childre dan sibling. (process's parent adalah proses yang membuatnya; children adalah proses apa pun yang dibuatnya. Sibling adalah chilren dengan proses induk (parent)yang sama.) Beberapa bidang ini meliputi:

long state; /* state of the process */ struct sched entity se; /* scheduling information */ struct task struct *parent; /* this process's parent */ struct list head children; /* this process's children */ struct files struct *files; /* list of open files */ struct mm struct *mm; /* address space of this process */

Sebagai contoh, keadaan suatu proses diwakili oleh kondisi medan panjang dalam struktur ini. Di dalam kernel Linux, semua proses yang aktif diwakili menggunakan daftar tugas ganda yang terkait. Kernel mempertahankan pointer- arus - ke proses yang saat ini mengeksekusi pada sistem, seperti yang ditunjukkan di bawah ini:



Sebagai gambaran tentang bagaimana kernel dapat memanipulasi salah satu bidang dalam task_struct untuk proses tertentu, mari kita asumsikan sistem ingin mengubah keadaan proses yang saat ini berjalan ke nilai new_state. Jika saat ini adalah penunjuk ke proses yang sedang dieksekusi, statusnya diubah dengan yang berikut:

current->state = new state;

3.2 PROCESS SCHEDULING

Tujuan dari multiprogramming adalah untuk memiliki beberapa proses yang berjalan setiap saat, untuk memaksimalkan pemanfaatan CPU. Tujuan pembagian waktu adalah untuk mengganti CPU di antara proses sehingga sering kali pengguna dapat berinteraksi dengan setiap program

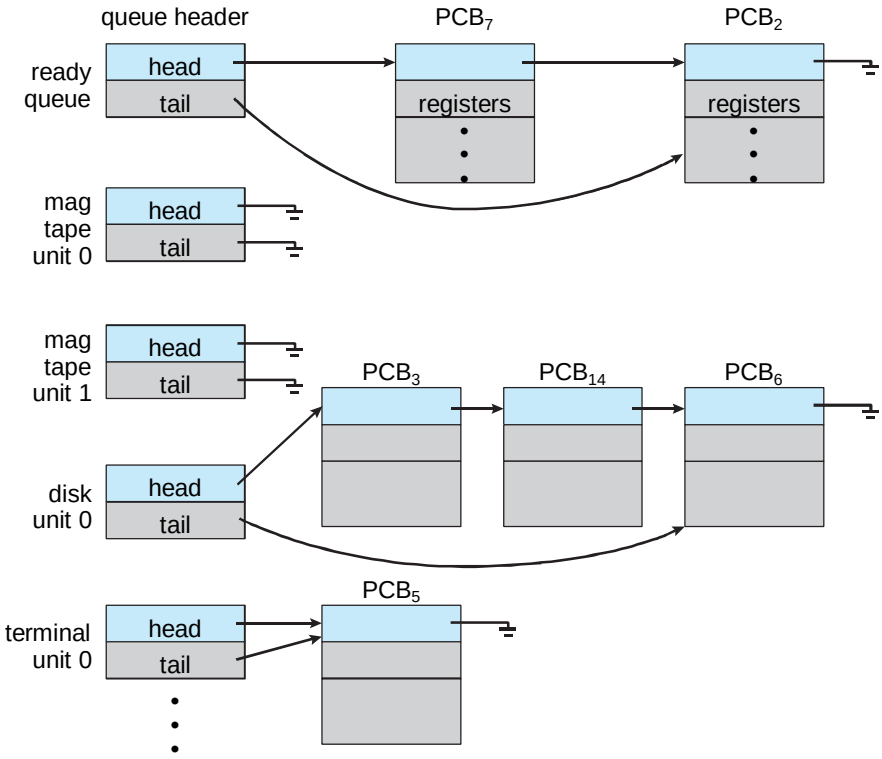


Figure 3.5 The ready queue and various I/O device queues.

saat sedang berjalan. Untuk memenuhi tujuan ini, penjadwal proses memilih proses yang tersedia (mungkin dari serangkaian proses yang tersedia) untuk eksekusi program pada CPU. Untuk sistem prosesor tunggal, tidak akan pernah ada lebih dari satu proses yang berjalan. Jika ada lebih banyak proses, selebihnya harus menunggu sampai CPU gratis dan dapat dijadwal ulang.

3.2.1 SCHEDULING QUEUES

Ketika proses memasuki sistem, mereka dimasukkan ke dalam antrian pekerjaan, yang terdiri dari semua proses dalam sistem. Proses yang berada di memori utama dan siap dan menunggu untuk dijalankan disimpan dalam daftar yang disebut antrean siap. Antrean ini umumnya disimpan sebagai daftar tertaut. Header siap-antrian berisi pointer ke PCB pertama dan terakhir dalam daftar. Setiap PCB termasuk bidang pointer yang menunjuk ke PCB berikutnya di antrian siap.

Sistem ini juga termasuk antrean lainnya. Ketika suatu proses dialokasikan CPU, ia mengeksekusi untuk sementara waktu dan akhirnya berhenti, terganggu, atau menunggu terjadinya peristiwa tertentu, seperti penyelesaian permintaan I / O.

Misalkan proses membuat permintaan I / O ke perangkat bersama, seperti disk. Karena ada banyak proses dalam sistem, disk mungkin sibuk dengan permintaan I / O dari beberapa proses lainnya. Oleh karena itu proses mungkin harus menunggu disk. Daftar proses yang menunggu perangkat I / O tertentu disebut antrean perangkat. Setiap perangkat memiliki antrian perangkat sendiri (Gambar 3.5).

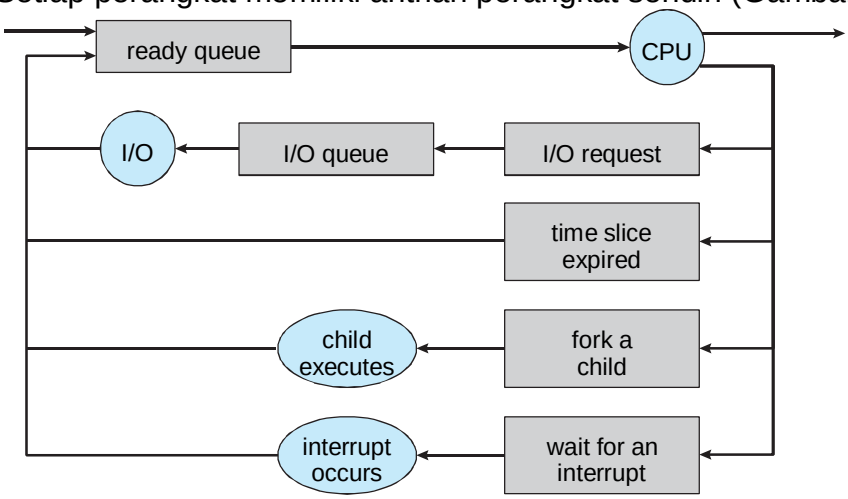


Figure 3.6 Queueing-diagram representation of process scheduling.

Representasi umum penjadwalan proses adalah diagram antrian, seperti pada Gambar 3.6. Setiap kotak persegi panjang mewakili suatu antrian. Dua jenis antrian hadir: antrian siap dan seperangkat antrian perangkat. Lingkaran mewakili sumber daya yang melayani antrian, dan panah menunjukkan aliran proses dalam sistem.

Proses baru pada awalnya dimasukkan ke antrean siap. Ia menunggu di sana sampai ia dipilih untuk dieksekusi, atau dikirim. Setelah proses tersebut dialokasikan CPU dan mengeksekusi, salah satu dari beberapa peristiwa dapat terjadi:

- Proses ini dapat mengeluarkan permintaan I / O dan kemudian ditempatkan dalam antrean I / O.
- Prosesnya dapat membuat proses anak baru dan menunggu penghentian anak.
- Proses ini dapat dikeluarkan secara paksa dari CPU, sebagai akibat interupsi, dan dimasukkan kembali ke antrean siap.

Dalam dua kasus pertama, proses akhirnya beralih dari status menunggu ke keadaan siap dan kemudian dimasukkan kembali ke antrean siap. Sebuah proses melanjutkan siklus ini sampai berakhir, pada saat itu dihapus dari semua antrian dan memiliki PCB dan sumber daya yang dialokasikan.

3.2.2 SCHEDULERS

Sebuah proses bermigrasi di antara berbagai antrian penjadwalan sepanjang masa hidupnya. Sistem operasi harus memilih, untuk tujuan penjadwalan, proses dari antrean ini dengan cara tertentu. Proses seleksi dilakukan oleh penjadwal yang sesuai.

Seringkali, pada batch system, proses lebih dapat disubmit dan dieksekusi segera.Proses ini di-spool ke perangkat penyimpanan massal (biasanya disk), di mana mereka disimpan untuk eksekusi nanti. Penjadwal jangka panjang, atau penjadwal pekerjaan, memilih proses dari kumpulan ini dan memuatnya ke dalam memori untuk eksekusi.
Penjadwal jangka pendek, atau penjadwal CPU, memilih dari antara proses yang siap dijalankan dan mengalokasikan CPU ke salah satunya.

Perbedaan utama antara kedua penjadwal ini terletak pada frekuensi eksekusi. Penjadwal jangka pendek harus memilih proses baru untuk CPU secara berkala. Sebuah proses dapat dijalankan hanya untuk beberapa milidetik sebelum menunggu permintaan I / O. Seringkali, penjadwal jangka pendek mengeksekusi setidaknya sekali setiap 100 milidetik. Karena waktu singkat antara eksekusi, penjadwal jangka pendek harus cepat. Jika diperlukan 10 milidetik untuk memutuskan untuk melaksanakan proses selama 100 milidetik, maka 10 / (100 + 10) = 9 persen dari CPU sedang digunakan (terbuang) hanya untuk menjadwalkan pekerjaan.

Penjadwal jangka panjang mengeksekusi lebih jarang; menit dapat memisahkan pembuatan satu proses baru dan berikutnya. Penjadwal jangka panjang mengontrol tingkat multiprogramming (jumlah proses dalam memori). Jika tingkat multiprogramming stabil, maka tingkat rata-rata pembuatan proses harus sama dengan tingkat keberangkatan rata-rata proses yang meninggalkan sistem. Dengan demikian, penjadwal jangka panjang mungkin perlu dipanggil hanya ketika suatu proses meninggalkan sistem. Karena interval yang lebih lama antara eksekusi, penjadwal jangka panjang dapat mengambil lebih banyak waktu untuk memutuskan proses mana yang harus dipilih untuk eksekusi.

Penting bahwa penjadwal jangka panjang membuat seleksi yang cermat. Secara umum, sebagian besar proses dapat digambarkan sebagai I / O bound atau CPU bound. Proses I / O-bound adalah salah satu yang menghabiskan lebih banyak waktu melakukan I / O daripada menghabiskan melakukan perhitungan. CPU-bound process, sebaliknya, menghasilkan permintaan I / O yang jarang, menggunakan lebih banyak waktunya melakukan perhitungan. Penting bahwa penjadwal jangka panjang memilih proses campuran yang baik dari proses I / O-bound dan CPU-bound. Jika semua proses terikat I / O, antrean siap hampir selalu kosong, dan penjadwal jangka pendek tidak banyak yang harus dilakukan. Jika semua proses terikat dengan CPU, antrian tunggu I / O hampir selalu kosong, perangkat akan tidak digunakan, dan lagi sistem akan tidak seimbang. Sistem dengan kinerja terbaik akan memiliki kombinasi CPU-bound dan I/O-bound process.

Pada beberapa sistem, penjadwal jangka panjang mungkin tidak ada atau minimal. Sebagai contoh, sistem pembagian waktu seperti sistem UNIX dan Microsoft Windows sering tidak memiliki penjadwal jangka panjang tetapi hanya menempatkan setiap proses baru dalam memori untuk penjadwal jangka pendek. Stabilitas sistem ini tergantung pada keterbatasan fisik (seperti jumlah terminal yang tersedia) atau pada sifat pengguna manusia yang menyesuaikan diri. Jika kinerja menurun ke tingkat yang tidak dapat diterima pada sistem multi-pengguna, beberapa pengguna hanya akan berhenti.

Beberapa sistem operasi, seperti sistem pembagian waktu, dapat memperkenalkan penjadwalan tambahan tingkat menengah. Penjadwal jangka menengah ini digambarkan pada Gambar 3.7. Ide utama di balik scheduler jangka menengah adalah bahwa kadang-kadang dapat menguntungkan untuk menghapus proses dari memori (dan dari pernyataan aktif untuk CPU) dan dengan demikian mengurangi tingkat multiprogramming. Kemudian, proses itu dapat diperkenalkan kembali ke dalam memori, dan eksekusinya dapat dilanjutkan di tempat yang ditinggalkannya. Skema ini disebut swapping. Proses ini ditukar, dan kemudian diganti, oleh penjadwal jangka menengah. Menukar mungkin diperlukan untuk meningkatkan bauran proses atau karena perubahan dalam persyaratan memori memiliki memori yang tersedia secara berlebihan, yang membutuhkan memori untuk dibebaskan. Pertukaran dibahas dalam Bab 8.

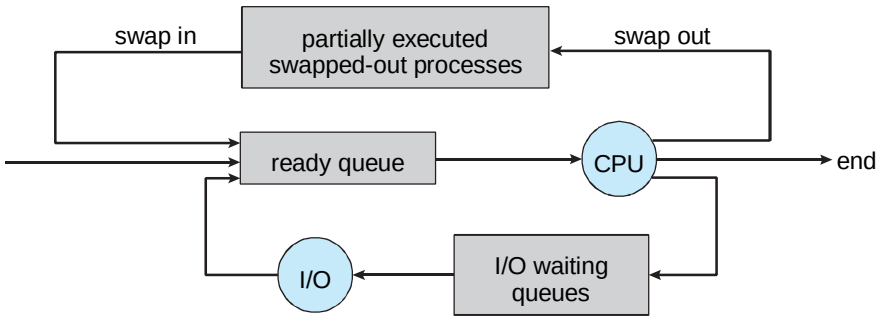


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

3.2.3 CONTEXT SWITCH

Seperti yang disebutkan di Bagian 1.2.1, interupsi menyebabkan sistem operasi mengubah CPU dari tugasnya saat ini dan menjalankan rutin kernel. Operasi seperti itu sering terjadi pada sistem tujuan umum. Ketika interupsi terjadi, sistem perlu menyimpan konteks saat ini dari proses yang berjalan pada CPU sehingga dapat mengembalikan konteks itu ketika prosesnya selesai, pada dasarnya menangguhkan proses dan kemudian melanjutkannya. Konteks diwakili dalam PCB proses. Ini termasuk nilai register CPU, status proses (lihat Gambar 3.2), dan informasi manajemen memori. Secara umum, kami melakukan state save untuk keadaan CPU saat ini, baik itu dalam kernel atau mode pengguna, dan kemudian mengembalikan keadaan(state restore) untuk melanjutkan operasi.

Mengalihkan CPU ke proses lain membutuhkan melakukan penyelamatan keadaan dari proses saat ini dan pemulihan keadaan dari proses yang berbeda. Tugas ini dikenal sebagai pengalih konteks/ context switch. Ketika konteks beralih terjadi, kernel menyimpan konteks proses lama di PCB dan memuat konteks tersimpan dari proses baru yang dijadwalkan untuk berjalan. Waktu pengalih konteks adalah overhead murni, karena sistem tidak berguna saat beralih. Kecepatan beralih bervariasi dari mesin ke mesin, tergantung pada kecepatan memori, jumlah register yang harus disalin, dan keberadaan instruksi khusus (seperti instruksi tunggal untuk memuat atau menyimpan semua register). Kecepatan tipikal adalah beberapa milidetik.

Waktu beralih konteks sangat tergantung pada dukungan perangkat keras. Sebagai contoh, beberapa prosesor (seperti Sun UltraSPARC) menyediakan beberapa set register. Sebuah saklar konteks di sini hanya membutuhkan mengubah pointer ke set register saat ini. Tentu saja, jika ada proses yang lebih aktif daripada set register, sistem resor menyalin data register ke dan dari memori, seperti sebelumnya. Juga, semakin kompleks sistem operasi, semakin besar jumlah pekerjaan yang harus dilakukan selama sakelar konteks. Seperti yang akan kita lihat di Bab 8, teknik manajemen memori lanjutan mungkin mengharuskan data tambahan dialihkan dengan setiap konteks. Misalnya, ruang alamat dari proses saat ini harus dipertahankan sebagai ruang tugas berikutnya disiapkan untuk digunakan. Bagaimana ruang alamat dipertahankan, dan berapa jumlah pekerjaan yang diperlukan untuk melestarikannya, tergantung pada metode manajemen memori dari sistem operasi.

MULTITASKING IN MOBILE SYSTEMS

Karena kendala yang dikenakan pada perangkat seluler, versi awal iOS tidak menyediakan aplikasi pengguna multitasking; hanya satu aplikasi yang berjalan di latar depan dan semua aplikasi pengguna lainnya ditangguhkan. Tugas sistem operasi adalah multitasked karena ditulis oleh Apple dan berperilaku baik. Namun, dimulai dengan iOS 4, Apple kini menyediakan bentuk terbatas multitasking untuk aplikasi pengguna, sehingga memungkinkan aplikasi latar depan tunggal untuk berjalan bersamaan dengan beberapa aplikasi latar belakang. (Pada perangkat seluler, aplikasi latar depan adalah aplikasi yang saat ini terbuka dan muncul di layar. Aplikasi latar belakang tetap berada di memori, tetapi tidak menempati layar tampilan.) API pemrograman iOS 4 menyediakan dukungan untuk multitasking, sehingga memungkinkan proses untuk berjalan di latar belakang tanpa ditangguhkan. Namun, terbatas dan hanya tersedia untuk sejumlah jenis aplikasi terbatas, termasuk aplikasi

- Menjalankan satu tugas terbatas, panjang (seperti menyelesaikan unduhan konten dari jaringan)
- Menerima pemberitahuan dari suatu peristiwa yang terjadi (seperti pesan email baru);
- Dengan tugas-tugas latar belakang yang berjalan lama (seperti pemutar audio.)

Apple mungkin membatasi multitasking karena masalah penggunaan baterai dan memori. CPU tentu memiliki fitur untuk mendukung multitasking, tetapi Apple memilih untuk tidak memanfaatkan beberapa dari mereka untuk lebih baik mengelola penggunaan sumber daya.

Android tidak menempatkan batasan semacam itu pada jenis aplikasi yang dapat berjalan di latar belakang. Jika aplikasi membutuhkan pemrosesan saat berada di latar belakang, aplikasi harus menggunakan layanan, komponen aplikasi terpisah yang berjalan atas nama proses latar belakang. Pertimbangkan aplikasi audio streaming: jika aplikasi berpindah ke latar belakang, layanan akan terus mengirim file audio ke driver perangkat audio atas nama aplikasi latar belakang. Bahkan, layanan akan terus berjalan meski aplikasi latar belakangnya ditangguhkan. Layanan tidak memiliki antarmuka pengguna dan memiliki jejak memori kecil, sehingga memberikan teknik yang efisien untuk multitasking di lingkungan seluler.

3.3 OPERATION ON PROCESSES

Proses di sebagian besar sistem dapat dijalankan secara bersamaan, dan mereka dapat dibuat dan dihapus secara dinamis. Dengan demikian, sistem ini harus menyediakan mekanisme untuk pembuatan dan pemutusan proses. Pada bagian ini, kami mengeksplorasi mekanisme yang terlibat dalam pembuatan proses dan menggambarkan penciptaan proses pada sistem UNIX dan Windows.

3.3.1 PROCESS CREATION

Selama proses eksekusi, suatu proses dapat menciptakan beberapa proses baru. Seperti disebutkan sebelumnya, proses pembuatan disebut proses induk, dan proses baru disebut anak-anak dari proses itu. Masing-masing proses baru ini pada gilirannya dapat menciptakan proses lain, membentuk pohon proses.

Sebagian besar sistem operasi (termasuk UNIX, Linux, dan Windows) mengidentifikasi proses sesuai dengan pengenal proses yang unik (atau pid), yang biasanya merupakan bilangan integer. Pid memberikan nilai unik untuk setiap proses dalam sistem, dan dapat digunakan sebagai indeks untuk mengakses berbagai atribut proses dalam kernel.

Gambar 3.8 mengilustrasikan pohon proses khas untuk sistem operasi Linux, menunjukkan nama setiap proses dan pid-nya. (Kami menggunakan proses istilah agak longgar, karena Linux lebih menyukai istilah tugas sebagai gantinya.) Init process (yang selalu memiliki pid dari 1) berfungsi sebagai proses induk root untuk semua proses pengguna. Setelah sistem di-boot, maka proses ini dapat menyebabkan berbagai proses pengguna, seperti web atau server cetak, ssh server, dan sejenisnya. Pada Gambar 3.8, kita melihat dua child dari init — kthreadd dan sshd. Proses kthreadd bertanggung jawab untuk membuat proses tambahan yang melakukan tugas atas nama kernel (dalam situasi ini, khelper dan pdflush). Proses sshd bertanggung jawab untuk mengelola klien yang terhubung ke sistem dengan menggunakan ssh (yang merupakan kependekan dari secureshell). Membantu proses penerimaan yang bertanggung jawab dari sistem yang secara langsung masuk ke sistem. Dalam contoh ini, klien telah masuk dan menggunakan bash shell, yang telah ditetapkan pid 8416. Menggunakan antarmuka baris perintah bash, pengguna ini telah membuat proses ps serta editor emacs.

Pada sistem UNIX dan Linux, kita dapat memperoleh daftar proses dengan menggunakan perintah ps. Misalnya, perintah

ps –el

akan mencantumkan informasi lengkap untuk semua proses yang saat ini aktif dalam sistem. Sangat mudah untuk membangun sebuah pohon proses yang mirip dengan yang ditunjukkan pada Gambar 3.8 dengan melacak secara rekursif proses induk semua jalan ke proses init.

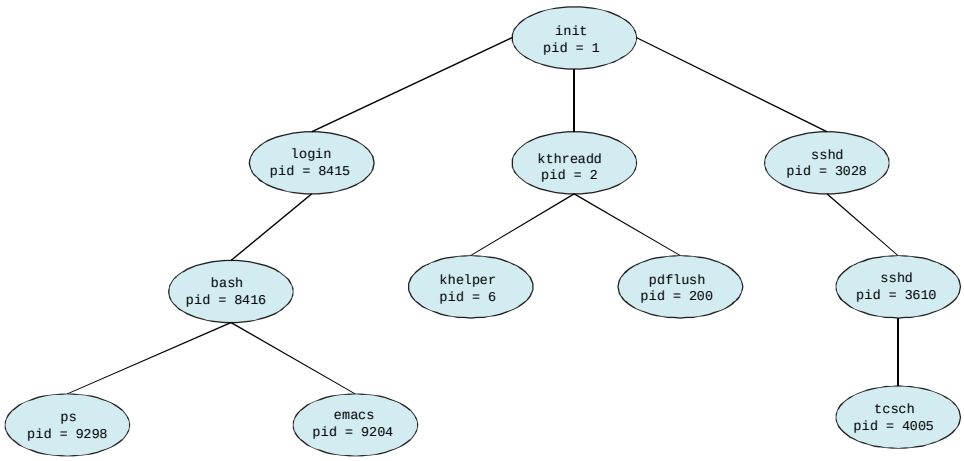


Figure 3.8 A tree of processes on a typical Linux system.

Secara umum, ketika suatu proses menciptakan proses anak, proses anak itu akan memerlukan sumber daya tertentu (waktu CPU, memori, file, perangkat I / O) untuk menyelesaikan tugasnya. Proses anak mungkin dapat memperoleh sumber dayanya langsung dari sistem operasi, atau mungkin dibatasi ke subkumpulan sumber daya dari proses induk. Orang tua mungkin harus membagi sumber dayanya di antara anak-anaknya, atau mungkin dapat berbagi beberapa sumber daya (seperti memori atau file) di antara beberapa anaknya. Membatasi proses anak ke subkumpulan sumber daya orang tua mencegah proses apa pun dari membebani sistem dengan membuat terlalu banyak proses anak.

Selain menyediakan berbagai sumber daya fisik dan logis, proses induk dapat meneruskan data inisialisasi (input) ke proses child. Sebagai contoh, pertimbangkan proses yang fungsinya untuk menampilkan isi dari suatu file —sebaiknya, image.jpg — pada layar terminal. Ketika proses dibuat, itu akan mendapatkan, sebagai masukan dari proses induknya, nama file image.jpg. Menggunakan nama file itu, ia akan membuka file dan menulis isinya. Mungkin juga mendapatkan nama perangkat output. Atau, beberapa sistem operasi memberikan sumber daya ke proses anak. Pada sistem seperti itu, proses baru mungkin mendapatkan dua file terbuka, image.jpg dan perangkat terminal, dan hanya dapat mentransfer datum antara keduanya.

Ketika suatu proses menciptakan proses baru, dua kemungkinan eksekusi ada:

- parent terus mengeksekusi secara bersamaan dengan childnya.
- parent menunggu sampai beberapa atau semua childnya berhenti.

  Ada juga dua kemungkinan ruang alamat untuk proses baru:
- Proses Child adalah duplikat dari proses induk (ia memiliki program dan data yang sama dengan parent).
- Proses Child memiliki program baru yang dimasukkan ke dalamnya.

Untuk mengilustrasikan perbedaan ini, pertama-tama perhatikan sistem operasi UNIX. Di UNIX, seperti yang telah kita lihat, setiap proses diidentifikasi oleh pengenal prosesnya, yang merupakan bilangan bulat unik. Proses baru dibuat oleh panggilan system *fork ( )*. Proses baru terdiri dari salinan ruang alamat dari proses asli.

Mekanisme ini memungkinkan proses parent untuk berkomunikasi dengan mudah dengan proses childnya. Kedua proses (parent dan child) melanjutkan eksekusi pada instruksi setelah fork (), dengan satu perbedaan: kode kembali untuk fork () adalah nol untuk proses (child) baru, sedangkan identifier proses (bukan nol) dari anak dikembalikan ke parent.

Setelah panggilan sistem fork (), salah satu dari dua proses biasanya menggunakan panggilan systemexec ()untuk menggantikan ruang memori proses dengan program baru. Panggilan sistem exec () memuat file biner ke dalam memori (menghancurkan gambar memori dari program yang berisi panggilan system exec ()) dan memulai eksekusinya. Dengan cara ini, kedua proses dapat berkomunikasi dan kemudian berpisah. Orang tua kemudian dapat menciptakan lebih banyak anak; atau, jika tidak ada hal lain yang dapat dilakukan saat anak berjalan, ia dapat mengeluarkan panggilan system *wait()*untuk memindahkan dirinya sendiri dari antrean siap sampai penghentian anak. Karena

```
#include <sys/types.h>
#include <stdio.h> #include <unistd.h> int
main()
{
pid t pid;
    /* fork a child process */ pid = fork();
    if (pid < 0) { /* error occurred */ fprintf(stderr, "Fork Failed"); return
        1;
    }
    else if (pid == 0) { /* child process */ execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */ wait(NULL);
        printf("Child Complete");
    } return 0; }
```

Figure 3.9 Creating a separate process using the UNIX fork() system call.

panggil ke exec () melumpuhkan ruang alamat proses dengan program baru, panggilan ke exec () tidak mengembalikan kontrol kecuali terjadi kesalahan.

Program C yang ditunjukkan pada Gambar 3.9 mengilustrasikan panggilan sistem UNIX yang dijelaskan sebelumnya. Kami sekarang memiliki dua proses berbeda menjalankan salinan dari program yang sama. Satu-satunya perbedaan adalah bahwa nilai pid (pengidentifikasi proses) untuk proses anak adalah nol, sedangkan untuk orang tua adalah nilai integer lebih besar dari nol (sebenarnya, itu adalah pid yang sebenarnya dari proses anak). Proses anak mewarisi hak istimewa dan atribut penjadwalan dari orang tua, serta sumber daya tertentu, seperti file terbuka. Proses

anak kemudian melapisi ruang alamatnya dengan perintah UNIX / bin / ls (digunakan untuk mendapatkan daftar direktori) menggunakan panggilan system execlp () (execlp () adalah versi dari memanggil system exec ()). Parent menunggu proses child selesai dengan memanggil sistem wait (). Ketika proses chiild selesai (baik secara implisit atau secara eksplisit memanggil exit ()), proses parent melanjutkan dari panggilan wait (), di mana ia selesai menggunakan exit () system call. Ini juga diilustrasikan pada Gambar 3.10.

Tentu saja, tidak ada yang mencegah achild untuk tidak memohon exec () dan malah terus mengeksekusi sebagai salinan proses induk/parent. Dalam skenario ini, parent dan child adalah proses konkuren yang menjalankan kode yang sama

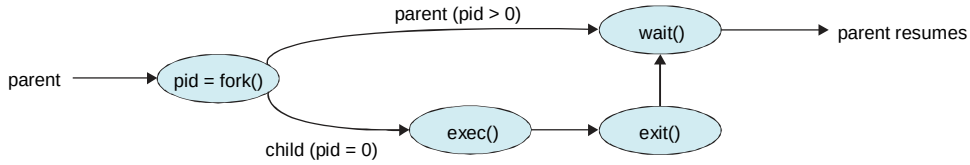


Figure 3.10 Process creation using the fork() system call.

Instruksi-instruksi . Karena child adalah salinan parent, setiap proses memiliki salinan data apa pun.

Sebagai contoh alternatif, kami selanjutnya mempertimbangkan pembuatan proses di Windows. Proses dibuat di Windows API menggunakan fungsi CreateProcess (), yang mirip dengan fork () di mana parent membuat proses child baru. Namun, sedangkan fork () memiliki proses child yang mewarisi ruang alamat induk\parentnya, CreateProcess () membutuhkan pemuatan program tertentu ke dalam ruang alamat proses child pada proses penciptaan. Selanjutnya, sedangkan fork () dilewatkan tanpa parameter, CreateProcess () mengharapkan tidak kurang dari sepuluh parameter.

Program C yang ditunjukkan pada Gambar 3.11 mengilustrasikan fungsi CreateProcess (), yang membuat proses child yang memuat aplikasi mspaint.exe. Kami memilih banyak nilai default dari sepuluh parameter yang dikirimkan ke CreateProcess (). Pembaca yang tertarik untuk mengejar detail pembuatan dan pengelolaan proses di Windows API dianjurkan untuk membaca catatan bibliografi di akhir bab ini.

Kedua parameter yang dilewatkan ke fungsi CreateProcess () adalah contoh dari STARTUPINFO dan struktur PROCESS_INFORMATION . STARTUPINFO menentukan banyak properti dari proses baru, seperti ukuran dan penampilan jendela dan menangani ke file input dan output standar. Struktur PROCESS_INFORMATION berisi pegangan dan pengidentifikasi untuk proses yang baru dibuat dan rangkaiannya. Kami memanggil fungsi ZeroMemory () untuk mengalokasikan memori untuk masing-masing struktur ini sebelum melanjutkan dengan CreateProcess ().

Dua parameter pertama yang dilewatkan ke CreateProcess () adalah nama aplikasi dan parameter baris perintah. Jika nama aplikasi adalah NULL (seperti dalam kasus ini), parameter baris perintah menentukan aplikasi untuk memuat. Dalam contoh ini, kami memuat aplikasi mspaint.exe Microsoft Windows. Di luar dua parameter awal ini, kami menggunakan parameter default untuk mewarisi proses dan pegangan benang serta menentukan bahwa tidak akan ada bendera kreasi. Kami juga menggunakan blok lingkungan orang tua yang ada dan direktori awal. Terakhir, kami menyediakan dua petunjuk ke STARTUPINFO dan struktur PROCESS_INFORMATION yang dibuat di awal program. Pada Gambar 3.9, proses induk/parent menunggu child untuk menyelesaikan dengan memohon panggilan sistem wait (). Setara dengan ini di Windows adalah WaitForSingleObject (), yang melewati suatu pegangan proses child — pi.hProcess — dan menunggu proses ini selesai. Setelah proses child keluar, kontrol kembali dari fungsi WaitForSingleObject () dalam proses parent.

```
#include <stdio.h> #include <windows.h> int
main(VOID)
{
STARTUPINFO si;
PROCESS INFORMATION pi;

    /* allocate memory */ ZeroMemory(&si, sizeof(si)); si.cb =
    sizeof(si); ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
      "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
      NULL, /* don't inherit process handle */
      NULL, /* don't inherit thread handle */ FALSE, /* disable handle
      inheritance */
      0, /* no creation flags */
      NULL, /* use parent's environment block */
      NULL, /* use parent's existing directory */
      &si, &pi))
    { fprintf(stderr, "Create Process Failed"); return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE); printf("Child Complete");



    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread); }
```

Figure 3.11 Creating a separate process using the Windows API.

3.3.2 PROCESS TERMINATION

Sebuah proses berakhir ketika selesai mengeksekusi pernyataan akhir dan meminta sistem operasi untuk menghapusnya dengan menggunakan panggilan sistem exit (). Pada titik itu, proses dapat mengembalikan nilai status (biasanya integer) ke proses induknya (melalui panggilan system wait ()). Semua sumber daya proses — termasuk memori fisik dan virtual, file terbuka, dan buffer I / O — dinonaktifkan oleh sistem operasi.

Penghentian dapat terjadi dalam keadaan lain juga. Suatu proses dapat menyebabkan penghentian proses lain melalui panggilan sistem yang sesuai (misalnya, TerminateProcess () di Windows). Biasanya, panggilan sistem semacam itu dapat dipanggil hanya oleh perbedaan proses yang ditimbulkan kembali. Jika tidak, pengguna dapat secara sewenang-wenang membunuh pekerjaan masing-masing. Perhatikan bahwa parent perlu mengetahui identitas childnya jika ingin mengakhirinya. Jadi, ketika satu proses menciptakan proses baru, identitas proses yang baru dibuat dilewatkan ke induk/parentnya.

Parent dapat menghentikan eksekusi salah satu childnya karena berbagai alasan, seperti ini:

- child telah melebihi penggunaan beberapa sumber daya yang telah dialokasikan. (Untuk menentukan apakah ini telah terjadi, parent harus memiliki mekanisme untuk memeriksa keadaan anak-anaknya.)
- Tugas yang diberikan kepada child tidak lagi diperlukan.
- Parent keluar, dan sistem operasi tidak mengizinkan child melanjutkan jika parentnya berhenti.

Beberapa sistem tidak mengizinkan child ada jika parentnya telah berhenti. Dalam sistem seperti itu, jika suatu proses berakhir (baik normal atau tidak normal), maka semua childnya juga harus dihentikan. Fenomena ini, yang disebut sebagai penghentian kaskade (cascading termination), biasanya dimulai oleh sistem operasi.Untuk mengilustrasikan proses eksekusi dan penghentian, pertimbangkan bahwa, dalam sistem Linux dan UNIX, kita dapat menghentikan proses dengan menggunakan panggilan system exit () untuk menyediakan status keluar sebagai parameter:

/* exit with status 1 */ exit(1);

Bahkan, di bawah pengakhiran normal, exit () dapat disebut baik secara langsung (seperti yang ditunjukkan di atas) atau tidak langsung (oleh pernyataan return di main ()).

Proses parent dapat menunggu penghentian proses child dengan menggunakan panggilan sistem wait (). Panggilan system wait () dilewatkan parameter yang memungkinkan parent untuk mendapatkan status keluar dari child. Panggilan sistem ini juga mengembalikan pengidentifikasi proses dari child yang diakhiri sehingga parent dapat mengetahui mana dari childnyatelah dihentikan:
Pid_ t pid; int status;

pid = wait(&status);

Ketika suatu proses berakhir, sumber dayanya dialihkan oleh sistem operasi. Namun, entri dalam tabel proses harus tetap ada hingga parent memanggil wait(), karena tabel proses berisi status keluar proses. Suatu proses yang telah dihentikan, tetapi yang parentnya belum disebut wait (), dikenal sebagai proses zombie. Semua proses transisi ke keadaan ini ketika mereka mengakhiri, tetapi umumnya mereka hanya eksis sebagai zombie. Setelah parent memanggil wait(), proses identifier proses zombie dan entri dalam tabel proses dilepaskan.

Sekarang pertimbangkan apa yang akan terjadi jika parent tidak meminta wait () dan sebagai gantinya dihentikan, dengan demikian meninggalkan proses childnya sebagai orphans/tidak punya parent. Linux dan UNIX mengatasi skenario ini dengan menetapkan proses init sebagai parent baru untuk proses orphans. (Ingat kembali dari Gambar 3.8 bahwa proses init adalah akar dari hirarki proses dalam sistem UNIX dan Linux.) Proses init secara berkala memanggil wait (), sehingga memungkinkan status keluar dari setiap proses orphans yang akan dikumpulkan dan melepaskan identifier proses orphans dan entri proses-tabel.

3.4 INTERPROCESS COMMUNICATION

Proses yang dijalankan bersamaan dalam sistem operasi dapat berupa proses independen atau proses kerja sama. Suatu proses bersifat independen jika tidak dapat mempengaruhi atau dipengaruhi oleh proses lain yang dieksekusi dalam sistem. Setiap proses yang tidak berbagi data dengan proses lain adalah independen. Suatu proses bekerja sama jika dapat mempengaruhi atau dipengaruhi oleh proses lain yang dieksekusi dalam sistem. Jelas, proses apa pun yang berbagi data dengan proses lain adalah proses yang bekerja sama.

Ada beberapa alasan untuk menyediakan lingkungan yang memungkinkan kerja sama proses:

- Information sharing. Karena beberapa pengguna mungkin tertarik pada bagian informasi yang sama (misalnya, file bersama), kita harus menyediakan lingkungan untuk memungkinkan akses bersamaan ke informasi tersebut.
- Computation speedup. Jika kita ingin tugas tertentu berjalan lebih cepat, kita harus memecahnya menjadi subtugas, yang masing-masing akan dieksekusi secara paralel dengan yang lain. Perhatikan bahwa percepatan seperti itu dapat dicapai hanya jika komputer memiliki beberapa inti pemrosesan.
- Modularity. Kita mungkin ingin membangun sistem secara modular, membagi fungsi sistem menjadi proses atau utas yang terpisah, seperti yang kita diskusikan pada Bab 2.
- Convenience Bahkan seorang pengguna individu dapat mengerjakan banyak tugas pada saat yang bersamaan. Misalnya, pengguna mungkin mengedit, mendengarkan musik, dan menyusun secara paralel.

Proses kerjasama membutuhkan mekanisme ***Interprocess Communication*** (IPC) yang memungkinkan mereka untuk bertukar data dan informasi. Ada dua model dasar komunikasi interproses: shared memory dan message passing. Dalam model shared memory yaitu suatu wilayah memori yang dibagi dengan proses bekerja sama

didirikan. Proses kemudian dapat bertukar informasi dengan membaca dan menulis data ke wilayah bersama. Dalam model message passing, komunikasi terjadi melalui pesan yang dipertukarkan antara proses yang bekerja sama. Kedua model komunikasi dikontraskan pada Gambar 3.12.

Kedua model yang disebutkan di atas sudah umum dalam sistem operasi, dan banyak sistem yang mengimplementasikan keduanya. Message passing berguna untuk bertukar data dalam jumlah yang lebih kecil, karena tidak ada konflik yang perlu dihindari. Melewati pesan juga lebih mudah untuk diterapkan dalam sistem terdistribusi daripada memori bersama. (Meskipun sistem lain yang menyediakan distribusi bersama memori, kita tidak mempertimbangkannya dalam teks ini.) shared memory dapat lebih cepat daripada message passing, karena sistem pengiriman pesan biasanya diimplementasikan menggunakan panggilan sistem.

MULTIPROCESS ARCHITECTURE – CHROME BROWSER

Banyak situs web berisi konten aktif seperti JavaScript, Flash, dan HTML5 untuk memberikan pengalaman menjelajah web yang kaya dan dinamis. Sayangnya, aplikasi web ini juga mengandung bug perangkat lunak, yang dapat mengakibatkan waktu respons yang lamban dan bahkan dapat menyebabkan browser web macet. Ini bukan masalah besar dalam peramban web yang hanya menampilkan konten dari satu situs web. Tetapi sebagian besar peramban web kontemporer menyediakan penjelajahan tab, yang memungkinkan satu contoh dari aplikasi peramban web untuk membuka beberapa situs web pada saat yang sama, dengan masing-masing situs di tab terpisah. Untuk beralih antar situs yang berbeda, pengguna hanya perlu mengklik pada tab yang sesuai. Pengaturan ini diilustrasikan di bawah ini:

Masalah dengan pendekatan ini adalah untuk menerapkannya di semua tab crashes, seluruh proses — termasuk semua tab lain yang menampilkan situs web tambahan —crashes juga.

Browser web Google Chrome dirancang untuk mengatasi masalah ini dengan menggunakan arsitektur multiproses. Chrome mengidentifikasi tiga jenis proses yang berbeda: browser, renderers, dan plug-ins

- Proses browser bertanggung jawab untuk mengelola antarmuka pengguna serta disk dan jaringan I / O. Proses browser baru dibuat saat Chrome dimulai. Hanya satu proses browser yang dibuat.
- Proses Renderers berisi logika untuk menampilkan halaman web. Dengan demikian, mereka mengandung logika untuk menangani HTML, Javascript, gambar, dan sebagainya. Sebagai aturan umum, proses perender baru dibuat untuk setiap situs web yang dibuka di tab baru, sehingga beberapa proses perender dapat aktif pada saat yang bersamaan.
- Proses plug-in dibuat untuk setiap jenis plug-in (seperti Flash atau QuickTime) yang digunakan. Proses plug-in berisi kode untuk plug-in serta kode tambahan yang memungkinkan plug-in untuk berkomunikasi dengan proses penyaji terkait dan proses browser.

Keuntungan dari pendekatan multiprocess adalah situs web dijalankan secara terpisah dari satu sama lain. Jika satu situs web macet, hanya proses perender yang terpengaruh; semua proses lainnya tetap tidak terganggu. Selanjutnya, proses penyaji berjalan di sand box, yang berarti bahwa akses ke disk dan jaringan I / O dibatasi, meminimalkan efek dari setiap eksploitasi keamanan.

dan dengan demikian membutuhkan lebih banyak tugas yang memakan waktu dari intervensi kernel. Dalam sistem memori bersama, panggilan sistem diperlukan hanya untuk membuat wilayah shared memory. Setelah shared memory dibuat, semua akses diperlakukan sebagai akses memori rutin, dan tidak perlu bantuan dari kernel.

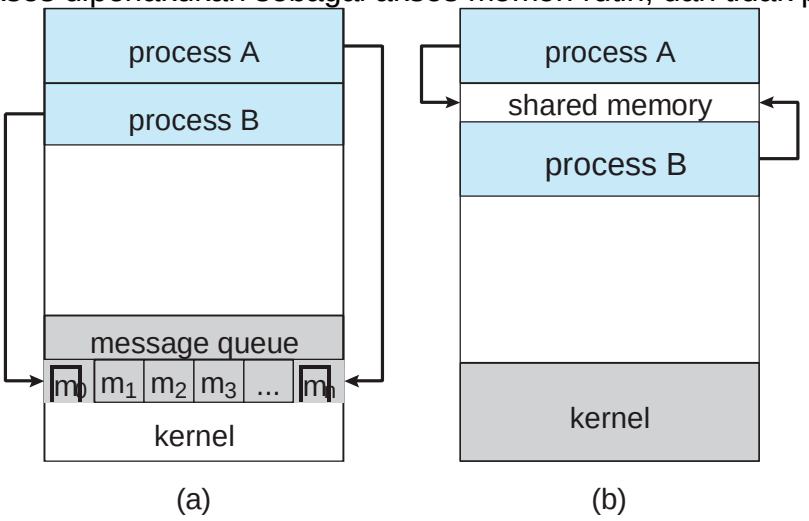


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Penelitian terbaru pada sistem dengan beberapa inti pemrosesan menunjukkan bahwa message passing memberikan kinerja yang lebih baik daripada shared memory pada sistem tersebut. Shared memoru mengalami masalah koherensi cache, yang muncul karena shared data bermigrasi di antara beberapa cache. Karena jumlah inti pemrosesan pada sistem meningkat, ada kemungkinan bahwa kita akan melihat message passing sebagai mekanisme yang disukai untuk IPC.

Di sisa bagian ini, kami mengeksplorasi sistem shared-memory dan messagepassing secara lebih rinci.

3.4.1 Shared-Memory Systems

Interprocess Commumnication menggunakan shared memory membutuhkan proses berkomunikasi untuk membangun wilayah memori bersama. Biasanya, wilayah shared memory berada di ruang alamat proses yang menciptakan segmen shared memory. Proses lain yang ingin berkomunikasi menggunakan segmen shared memory ini harus melampirkannya ke ruang alamat mereka. Ingat bahwa, biasanya, sistem operasi mencoba mencegah satu proses mengakses memori proses lain. Shared memory mengharuskan dua proses atau lebih setuju untuk menghapus batasan ini. Mereka kemudian dapat bertukar informasi dengan membaca dan

menulis data di area bersama. Bentuk data dan lokasi ditentukan oleh proses ini dan tidak berada di bawah kendali sistem operasi. Proses juga bertanggung jawab untuk memastikan bahwa mereka tidak menulis ke lokasi yang sama secara bersamaan.

Untuk mengilustrasikan konsep cooperating process, mari kita pertimbangkan masalah produsen-konsumen, yang merupakan paradigma umum untuk cooperating process. Proses produsen menghasilkan informasi yang dikonsumsi oleh proses konsumen. Sebagai contoh, kompiler dapat menghasilkan kode assembly yang dikonsumsi oleh assembler. Perakit, pada gilirannya, dapat menghasilkan modul objek yang dikonsumsi oleh loader. Masalah produsen-konsumen

```
item next produced;

while (true) {

  /* produce an item in next produced */

  while (((in + 1) % BUFFER SIZE) == out) ; /* do nothing */

  buffer[in] = next produced;
  in = (in + 1) % BUFFER SIZE;
}
```

Figure 3.13 The producer process using shared memory.

juga menyediakan metafora yang berguna untuk paradigma client-server. Kami umumnya menganggap server sebagai produsen dan klien sebagai konsumen. Sebagai contoh, server web menghasilkan (yaitu, menyediakan) file HTML dan gambar, yang dikonsumsi (yaitu, dibaca) oleh browser web klien yang meminta sumber daya.

Satu solusi untuk masalah produsen-konsumen menggunakan memori bersama. Untuk memungkinkan produsen dan proses konsumen untuk berjalan secara bersamaan, kita harus memiliki penyangga(buffer) barang yang dapat diisi oleh produsen dan dikosongkan oleh konsumen. Penyangga(buffer) ini akan berada di wilayah memori yang dibagikan oleh produsen dan proses konsumen. Seorang produsen dapat menghasilkan satu barang sementara konsumen mengkonsumsi barang lain. Produsen dan konsumen harus disinkronkan, sehingga konsumen tidak mencoba untuk mengkonsumsi barang yang belum diproduksi.

Dua jenis buffer dapat digunakan. Tempat buffer tak terbatas(unbounded buffer) tidak ada batas praktis pada ukuran buffer. Konsumen mungkin harus menunggu barang baru, tetapi produser selalu dapat menghasilkan barang baru. Buffer yang dibatasi (bounded buffer) mengasumsikan ukuran buffer tetap. Dalam hal ini, konsumen harus menunggu jika buffer kosong, dan produser harus menunggu jika buffer sudah penuh.
Mari kita lihat lebih dekat bagaimana buffer yang dibatasi menggambarkan

komunikasi interprocess menggunakan memori bersama. Variabel berikut berada di wilayah memori yang dibagikan oleh produsen dan proses konsumen:

```
#define BUFFERSIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFERSIZE]; int in = 0; int out = 0;
```

Shared Buffer diimplementasikan sebagai array melingkar dengan dua pointer logis: masuk dan keluar. Variabel dalam poin ke posisi bebas berikutnya di buffer; out poin ke posisi first full di buffer. Buffer kosong ketika in == out; buffer penuh ketika ((in+1)% BUFFER _SIZE) == out.

Kode untuk proses produsen ditunjukkan pada Gambar 3.13, dan kode untuk proses konsumen ditunjukkan pada Gambar 3.14. Proses produsen memiliki variabel lokal berikutnya yang diproduksi di mana barang baru yang akan diproduksi disimpan. Proses konsumen memiliki variabel lokal yang selanjutnya dikonsumsi di mana barang yang akan dikonsumsi disimpan.

```
item next consumed; while (true) { while (in ==
out)
        ; /* do nothing */
    next consumed = buffer[out]; out = (out + 1) % BUFFER_SIZE; /*
    consume the item in next consumed */
}
```

Figure 3.14 The consumer process using shared memory.

Skema ini memungkinkan paling banyak BUFFER SIZE - 1 item dalam buffer pada saat yang bersamaan. Kami membiarkannya sebagai latihan bagi Anda untuk memberikan solusi di mana item BUFFER SIZE dapat berada di buffer pada saat yang bersamaan. Dalam Bagian 3.5.1, kami mengilustrasikan API POSIX untuk memori bersama.

Satu isu ilustrasi ini tidak membahas masalah situasi di mana proses produsen dan proses konsumen mencoba untuk mengakses buffer bersama secara bersamaan. Dalam Bab 5, kita membahas bagaimana sinkronisasi antara proses kerja sama dapat diimplementasikan secara efektif dalam lingkungan kenangan bersama.

3.4.2 MESSAGE-PASSING SYSTEMS

Dalam Bagian 3.4.1, kami menunjukkan bagaimana cooperating process dapat berkomunikasi dalam lingkungan memori bersama. Skema ini mensyaratkan bahwa proses ini berbagi wilayah memori dan bahwa kode untuk mengakses dan memanipulasi memori bersama ditulis secara eksplisit oleh programmer aplikasi. Cara lain untuk mencapai efek yang sama adalah sistem operasi untuk menyediakan sarana dimana cooperating process untuk berkomunikasi satu sama lain melalui fasilitas pengiriman pesan.
Message passing menyediakan mekanisme untuk memungkinkan proses untuk berkomunikasi dan untuk menyinkronkan tindakan mereka tanpa berbagi ruang alamat yang sama. Ini sangat berguna dalam lingkungan terdistribusi, di mana proses berkomunikasi dapat berada pada komputer yang berbeda yang terhubung oleh jaringan. Misalnya, program obrolan Internet dapat dirancang agar peserta obrolan berkomunikasi satu sama lain dengan bertukar pesan.

Fasilitas pengiriman pesan menyediakan setidaknya dua operasi:

send(message) receive(message)

Pesan yang dikirim oleh suatu proses dapat berupa ukuran tetap atau variabel. Jika hanya pesan berukuran tetap yang dapat dikirim, implementasi tingkat sistem sangat mudah. Pembatasan ini, bagaimanapun, membuat tugas pemrograman menjadi lebih sulit. Sebaliknya, pesan berukuran variabel memerlukan implementasi tingkat sistem yang lebih kompleks, tetapi tugas pemrograman menjadi lebih sederhana. Ini adalah jenis tradeoff yang umum terlihat di seluruh desain sistem operasi.

Jika proses P dan Q ingin berkomunikasi, mereka harus mengirim pesan ke dan menerima pesan dari satu sama lain: sebuah communication link harus ada di antara mereka. Tautan ini dapat diimplementasikan dalam berbagai cara. Kami khawatir di sini bukan dengan penerapan fisik tautan (seperti memori bersama, bus perangkat keras, atau jaringan, yang dibahas di Bab 17) tetapi dengan implementasi logisnya. Berikut adalah beberapa metode untuk secara logis menerapkan tautan dan operasi send () / receive ():

- Komunikasi langsung atau tidak langsung
- Komunikasi sinkron atau asinkron
- Penyangga(buffer) otomatis atau eksplisit

Kami melihat masalah yang terkait dengan masing-masing fitur ini selanjutnya.

3.4.2.1 NAMING

Proses yang ingin berkomunikasi harus memiliki cara untuk merujuk satu sama lain. Mereka dapat menggunakan komunikasi langsung atau tidak langsung. Di bawah komunikasi langsung, setiap proses yang ingin berkomunikasi harus secara eksplisit memberi nama penerima atau pengirim komunikasi. Dalam skema ini, yang send () dan receive() primitif didefinisikan sebagai:

- send (P, message) —Kirim pesan untuk memproses P.
- receive (Q, massage) —Menerima pesan dari proses Q.

Communication Link dalam skema ini memiliki properti berikut:

- Tautan/link dibuat secara otomatis di antara setiap pasangan proses yang ingin berkomunikasi. Prosesnya hanya perlu mengetahui identitas satu sama lain untuk berkomunikasi.
- Tautan\link dikaitkan dengan dua proses.
- Di antara setiap pasangan proses, ada persis satu tautan\link.

Skema ini menunjukkan simetri dalam menyikapi; yaitu, proses pengirim dan proses penerima harus menyebutkan yang lain untuk berkomunikasi. Varian dari skema ini menggunakan asimetri dalam pengaturan. Di sini, hanya pengirim yang menamai penerima; penerima tidak perlu menyebutkan pengirimnya. Dalam skema ini, yang mengirim () dan menerima () primitif didefinisikan sebagai berikut:

- send (P, message) —Kirim pesan untuk memproses P.
- receive (id, message) —Menerima pesan dari proses apa pun. ID variabel diatur ke nama proses yang komunikasi telah terjadi/mengambil tempat.

Kerugian dalam kedua skema ini (simetris dan asimetris) adalah modularitas terbatas dari definisi proses yang dihasilkan. Mengubah pengenal suatu proses mungkin mengharuskan memeriksa semua definisi proses lainnya. Semua referensi ke pengidentifikasi lama harus ditemukan, sehingga mereka dapat dimodifikasi ke pengidentifikasi baru. Secara umum, seperti teknik-teknik **hard-coding**, di mana para pengidentifikasi harus secara eksplisit, tidak memiliki rancangan teknik-teknik yang melibatkan tipuan, seperti yang dijelaskan berikut.

Dengan komunikasi tidak langsung, pesan dikirim ke dan diterima dari kotak surat, atau port. Kotak surat dapat dilihat secara abstrak sebagai objek di mana pesan dapat dilacak dan diproses dari mana pesan dapat dipicu. Setiap kotak surat memiliki identifikasi unik. Sebagai contoh, antrian pesan POSIX menggunakan nilai integer untuk mengidentifikasi kotak surat. Suatu proses dapat berkomunikasi

dengan proses lain melalui sejumlah kotak surat yang berbeda, tetapi dua proses dapat berkomunikasi hanya jika mereka memiliki kotak surat bersama. Yang mengirim () dan menerima () primitif didefinisikan sebagai berikut:

- send (A, massage) —Kirim pesan ke kotak surat A.
- receive (A, massage) —Menerima pesan dari kotak surat A.

Dalam skema ini, tautan\link komunikasi memiliki properti berikut:

- Tautan\link dibuat antara sepasang proses hanya jika kedua anggota pasangan memiliki kotak surat bersama.
- Tautan\link mungkin terkait dengan lebih dari dua proses.
- Antara masing-masing pasang proses komunikasi, sejumlah tautan yang berbeda mungkin ada, dengan setiap tautan yang terkait dengan satu kotak surat.

Sekarang anggaplah bahwa proses P1, P2, dan P3 semua berbagi kotak surat A. Proses P1 mengirim pesan ke A, sementara P2 dan P3 mengeksekusi receive() dari A. Proses mana yang akan menerima pesan yang dikirim oleh P1? Jawabannya tergantung pada metode mana yang kita pilih:

- Izinkan tautan\link dikaitkan dengan dua proses paling banyak.
- Berikan paling banyak satu proses sekaligus untuk menjalankan operasi receive ().
- Biarkan sistem untuk memilih secara sewenang-wenang proses mana yang akan menerima pesan (yaitu, baik P2 atau P3, tetapi tidak keduanya, akan menerima pesannya). Sistem dapat menentukan analgoritma untuk memilih proses mana yang akan menerima pesan (misalnya, round robin, di mana proses bergantian menerima pesan). Sistem dapat mengidentifikasi penerima ke pengirim.

Kotak surat (mail box) dapat dimiliki baik oleh suatu proses atau oleh sistem operasi. Jika kotak surat dimiliki oleh suatu proses (yaitu kotak surat adalah bagian dari ruang alamat proses), maka kita membedakan antara pemilik (yang hanya dapat menerima pesan melalui kotak surat ini) dan pengguna (yang hanya dapat mengirim pesan ke kotak surat). Karena setiap kotak pesan memiliki pemilik yang unik, tidak akan ada kebingungan tentang proses mana yang harus menerima pesan yang dikirim ke kotak surat ini. Ketika sebuah proses yang memiliki kotak surat berakhir, kotak surat menghilang. Setiap proses yang kemudian mengirim pesan ke kotak surat ini harus diberi tahu bahwa kotak surat tidak ada lagi.

Sebaliknya, kotak surat yang dimiliki oleh sistem operasi memiliki keberadaannya sendiri. Ini independen dan tidak terikat pada proses tertentu. Sistem operasi

kemudian harus menyediakan mekanisme yang memungkinkan proses untuk melakukan hal berikut:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

Proses yang membuat kotak surat baru adalah pemilik kotak surat secara default. Awalnya, pemilik adalah satu-satunya proses yang dapat menerima pesan melalui kotak surat ini. Namun, kepemilikan dan hak istimewa penerima dapat diteruskan ke proses lain melalui panggilan sistem yang sesuai. Tentu saja, ketentuan ini bisa menghasilkan banyak penerima untuk setiap kotak surat.

3.4.2.2 SYNCHRONIZATION

Komunikasi antar proses terjadi melalui panggilan untuk send () dan receive () primitif. Ada berbagai pilihan desain untuk menerapkan setiap primitif. Melewati pesan dapat berupa blocking atau nonblocking — juga dikenal sebagai synchronous dan asynchronous. (Di sepanjang teks ini, Anda akan menemukan konsep perilaku synchronous dan asynchronous dalam kaitannya dengan berbagai algoritma sistem operasi.)

- Blocking Send. Proses pengiriman diblokir sampai pesan diterima oleh proses penerimaan atau oleh kotak surat.
- Nonblocking send. Proses pengiriman mengirim pesan dan melanjutkan operasi.
- Blocking Receive. Blok penerima sampai pesan tersedia.
- Nonblocking Receive. Penerima mengambil pesan yang valid atau null.

Berbagai kombinasi send() dan receive () dimungkinkan. Ketika keduanya send() dan receive () memblokir, kami memiliki pertemuan antara pengirim dan penerima. Solusi untuk masalah produsen-konsumen menjadi sepele ketika kita menggunakan pemblokiran pernyataan send () dan receive (). Produser hanya memanggil pengirim send () dan menunggu sampai pesan dikirim ke penerima atau kotak surat. Demikian juga, ketika konsumen memanggil send (), ia akan memblokir hingga sebuah pesan tersedia. Ini diilustrasikan dalam Gambar 3.15 dan 3.16.

3.4.2.3 BUFFERING

Apakah komunikasi itu langsung atau tidak langsung, pesan yang dipertukarkan dengan proses komunikasi berada dalam antrian sementara. Pada dasarnya, antrian seperti itu dapat diimplementasikan dengan tiga cara:

```
message next produced; while (true) {
    /* produce an item in next produced */ send(next
produced); }
```

Figure 3.15 The producer process using message passing.

- **Zero capacity.** Antrean memiliki panjang maksimum nol; dengan demikian, tautan tidak dapat memiliki pesan yang menunggu di dalamnya. Dalam hal ini, pengirim harus memblokir hingga penerima menerima pesan.
- **Bounded Capacity**. Antrian memiliki panjang n terbatas; dengan demikian, paling banyak n pesan dapat berada di dalamnya. Jika antrian tidak penuh ketika pesan baru dikirim, pesan ditempatkan dalam antrian (baik pesan disalin atau pointer ke pesan disimpan), dan pengirim dapat melanjutkan eksekusi tanpa menunggu. Kapasitas tautan terbatas, namun. Jika tautan penuh, pengirim harus memblokir hingga ruang tersedia di antrian.
- **Unbounded Capacity.** Panjang antrian berpotensi tidak terbatas; dengan demikian, sejumlah pesan dapat menunggu di dalamnya. Pengirim tidak pernah memblokir.

Kasus kapasitas-nol kadang-kadang disebut sebagai sistem pesan tanpa penyanggaan. Kasus-kasus lain disebut sebagai sistem dengan buffering otomatis.

3.5 Contoh Sistem IPC

Di bagian ini, kami mengeksplorasi tiga sistem IPC yang berbeda. Pertama-tama, kami membahas POSIX API untuk memori bersama dan kemudian diskusikan pesan yang lewat di Machoperating sistem. Kami menyimpulkan dengan Windows, yang menarik menggunakan memori bersama sebagai mekanisme untuk menyediakan jenis pengiriman pesan tertentu.

3.5.1 Contoh: Memori Bersama POSIX

Beberapa mekanisme IPC tersedia untuk sistem POSIX, termasuk yang dibagikan memori dan pengiriman pesan. Di sini, kami menjelajahi API POSIX untuk dibagika ingatan. Memori bersama POSIX diatur menggunakan file yang dipetakan memori, yang mengaitkan wilayah memori bersama dengan file. Proses pertama harus dibuat

```
message next consumed;
while (true) { receive(next consumed);
/* consume the item in next consumed */
}
```

Figure 3.16 The consumer process using message passing.

objek memori bersama menggunakan shm open () system call, sebagai berikut:

```
shm fd = shm open(name, O CREAT | O RDRW, 0666);
```

Parameter pertama menentukan nama objek memori bersama. Proses yang ingin mengakses memori bersama ini harus mengacu pada objek dengan nama ini. Parameter selanjutnya menentukan bahwa objek memori bersama harus dibuat jika belum ada (O CREAT) dan objek terbuka untuk dibaca dan menulis (O RDRW). Parameter terakhir menetapkan izin direktori dari objek memori bersama. Panggilan yang berhasil untuk membuka shm () mengembalikan deskripsi file integer untuk objek memori bersama. Setelah objek ditetapkan, fungsi ftruncate () digunakan untuk mengonfigurasi ukuran objek dalam byte. Panggilan

```
ftruncate(shm fd, 4096);
```

set ukuran objek menjadi 4,096 byte. Akhirnya, fungsi mmap () menetapkan file memory-mapped yang berisi shared-memory object. Ini juga mengembalikan pointer ke file memory-mapped yang digunakan untuk mengakses shared-memory object.

Program yang ditunjukkan pada Gambar 3.17 dan 3.18 menggunakan model produsen-konsumen dalam mengimplementasikan shared memory. Produser menetapkan suatu shared memory object dan menulis ke shared memory, dan konsumen membaca dari shared memory.

Produser, ditunjukkan pada Gambar 3.17, membuat shared-memory bernama OS dan menulis string terkenal "Hello World!" untuk berbagi memori. memori program memetakan shared memory object dari ukuran yang ditentukan dan memungkinkan menulis ke objek. (Jelas, hanya menulis diperlukan untuk produser). Bendera MAP SHARED menetapkan bahwa perubahan pada shared memory objek akan terlihat oleh semua proses berbagi objek. Perhatikan bahwa kami menulis untuk shared memory objects dengan memanggil fungsi sprintf () dan menulis string diformat ke ptr pointer. Setelah setiap menulis, kita harus menaikkan penunjuk dengan jumlah byte yang ditulis. Proses konsumen, ditunjukkan pada Gambar 3.18, membaca dan mengeluarkan isinya dari shared memory.

Konsumen juga memanggil fungsi shm unlink (), yang menghapus segmen shared memory setelah konsumen mengakses saya t. Kami menyediakan latihan lebih lanjut menggunakan API shared memory POSIX di latihan pemrograman di akhir bab ini. Selain itu, kami menyediakan cakupan lebih rinci dari pemetaan memori di Bagian 9.7.

3.5.2 Contoh: Mach

Sebagai contoh pengiriman pesan, kita selanjutnya mempertimbangkan operasi Mach sistem. Anda mungkin ingat bahwa kami memperkenalkan Mach di Bab 2 sebagai bagian dari sistem operasi Mac OS X. Kernel Mach mendukung pembuatan

dan penghancuran beberapa tugas, yang mirip dengan proses tetapi memiliki beberapa utas kontrol dan sumber daya terkait yang lebih sedikit. Sebagian besar komunikasi dalam Mach — termasuk semua informasi intertask — dilakukan oleh pesan. Pesan dikirim ke dan diterima dari kotak surat, yang disebut port di Mach.

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main()
{/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message 0 = "Hello";
const char *message 1 = "World!";
/* shared memory file descriptor */
int shm fd;
/* pointer to shared memory obect */
void *ptr;
/* create the shared memory object */
shm fd = shm open(name, O CREAT | O RDRW, 0666);
/* configure the size of the shared memory object */
ftruncate(shm fd, SIZE);
/* memory map the shared memory object */
ptr = mmap(0, SIZE, PROT WRITE, MAP SHARED, shm fd, 0);
/* write to the shared memory object */
sprintf(ptr,"%s",message 0);
ptr += strlen(message 0);
sprintf(ptr,"%s",message 1);
ptr += strlen(message 1);
return 0;
}
```

Gambar 3.17 Proses produser yang menggambarkan POSIX shared-memory API.

Bahkan panggilan sistem dibuat oleh pesan. Ketika sebuah tugas dibuat, duakotak surat khusus — kotak surat Kernel dan kotak surat Beritahu — jugadibuat. Kernel menggunakan kotak surat Kernel untuk berkomunikasi dengan tugas dan mengirimkan pemberitahuan kejadian kejadian ke port Beritahu. Hanya tiga sistempanggilan diperlukan untuk transfer pesan. Panggilan pesan send () mengirim pesanke kotak pesan. Pesan diterima melalui pesan terima (). Prosedur jarak jauhpanggilan (RPCs) dieksekusi melalui msg rpc (), yang mengirim amessage dan menunggu tepat untuk satu pesan balasan dari pengirim. Dengan cara ini, model RPC

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main()
{/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm fd;
/* pointer to shared memory obect */
void *ptr;
/* open the shared memory object */
shm fd = shm open(name, O RDONLY, 0666);
/* memory map the shared memory object */
ptr = mmap(0, SIZE, PROT READ, MAP SHARED, shm fd, 0);
/* read from the shared memory object */
printf("%s",(char *)ptr);
/* remove the shared memory object */
shm unlink(name);
return 0;
}
```

Gambar 3.18 Proses konsumen yang menggambarkan POSIX shared-memory API.

tipe panggilan prosedur subrutin yang khas tetapi dapat bekerja di antara systems — hense istilah jarak jauh. Panggilan prosedur jarak jauh dibahas secara rinci di Bagian 3.6.2.

Port mengalokasikan () system call membuat kotak surat baru dan mengalokasikan ruang untuk antrean pesannya. Ukuran maksimum antrean pesan menjadi delapan pesan. Tugas yang membuat kotak surat adalah pemilik kotak surat itu. Pemilik juga diizinkan untuk menerima dari kotak surat. Hanya satu tugas pada satu waktu yang dapat memiliki atau menerima dari kotak surat, tetapi hak ini dapat dikirim ke tugas lain.

Antrian pesan mailbox pada awalnya kosong. Saat pesan dikirim ke kotak surat, pesan disalin ke kotak surat. Semua pesan memiliki prioritas yang sama. Mach menjamin bahwa beberapa pesan dari pengirim yang sama diantrkan dalam urutan First In, First Out (FIFO) tetapi tidak menjamin pemesanan mutlak. Misalnya, pesan dari dua pengirim dapat diantrekan dalam urutan apa pun.

Pesan itu sendiri terdiri dari header dengan panjang tetap diikuti oleh bagian data variabel-panjang. Header menunjukkan panjang pesan dan mencakup dua nama kotak surat. Satu nama kotak surat menentukan kotak surat ke mana pesan itu dikirim. Umumnya, utas pengiriman mengharapkan balas, sehingga nama kotak surat pengirim diteruskan ke tugas penerima, yang dapat menggunakannya sebagai "alamat pengirim".

Bagian variabel dari pesan adalah daftar item data yang diketik. Setiap entri dalam daftar memiliki tipe, ukuran, dan nilai. Jenis objek yang ditentukan dalam pesan adalah penting, karena objek yang ditentukan oleh sistem operasi — seperti kepemilikan atau menerima hak akses, status tugas, dan segmen memori — dapat dikirim dalam pesan.

Operasi kirim dan terima sendiri fleksibel. Misalnya, ketika pesan dikirim ke kotak pesan, kotak surat mungkin penuh. Jika kotak pesan tidak penuh, pesan akan disalin ke kotak surat, dan utas pengiriman berlanjut. Jika kotak pesan penuh, utas pengiriman memiliki empat opsi:

1. Tunggu tanpa batas sampai ada ruang di kotak surat.

2. Tunggu paling banyak milidetik.

3. Jangan menunggu sama sekali melainkan segera kembali.

4. Untuk sementara cache pesan. Di sini, pesan diberikan kepada operasi sistem untuk menyimpan, meskipun kotak surat tempat pesan itu berada terkirim sudah penuh. Ketika pesan dapat dimasukkan ke dalam kotak surat, pesan dikirim kembali ke pengirim. Hanya satu pesan ke kotak surat lengkap yang dapat ditunda kapan saja untuk utas pengiriman yang diberikan.

Pilihan terakhir dimaksudkan untuk tugas server, seperti driver printer garis. Setelah

menyelesaikan permintaan, tugas tersebut mungkin perlu mengirim balasan satu kali ke tugas yang meminta layanan, tetapi mereka juga harus melanjutkan permintaan layanan lainnya,bahkan jika kotak surat balasan untuk klien penuh.

Operasi penerimaan harus menentukan kotak surat atau set kotak pesan dari mana pesan harus diterima. Set kotak surat adalah kumpulan kotak surat,

sebagaimana dinyatakan oleh tugas, yang dapat dikelompokkan bersama dan diperlakukan sebagai satu kotak surat untuk tujuan dari tugas tersebut. Thread dalam tugas hanya dapat menerima dari kotak surat atau kotak surat yang tugasnya telah menerima akses. Sistem status port () panggilan mengembalikan jumlah pesan dalam kotak pesan yang diberikan. Operasi penerimaan mencoba menerima dari (1) kotak pesan apa pun dalam satu set kotak pesan atau (2) spesifik kotak surat (bernama). Jika tidak ada pesan yang menunggu untuk diterima, untaian penerima dapat menunggu paling banyak milidetik atau tidak menunggu sama sekali.

Sistem Mach dirancang khusus untuk sistem terdistribusi, yang kita bahas pada Bab 17, tetapi Mach terbukti cocok untuk sistem dengan inti pemrosesan yang lebih sedikit, sebagaimana dibuktikan oleh penyertaannya di Mac OS X sistem. Masalah utama dengan sistem pesan umumnya buruk kinerja yang disebabkan oleh penyalinan pesan ganda: pesan disalin pertama dari pengirim ke kotak surat dan kemudian dari kotak surat ke penerima. Sistem pesan Mach mencoba untuk menghindari operasi duplikasi ganda dengan menggunakan teknik manajemen virtual-memori (Bab 9). Pada dasarnya, peta Mach ruang alamat yang berisi pesan pengirim ke alamat penerima ruang. Pesan itu sendiri tidak pernah benar-benar disalin. Manajemen pesan ini teknik memberikan dorongan kinerja yang besar tetapi bekerja hanya untuk intrasistem pesan. Sistem operasi Mach dibahas secara lebih rinci di online Lampiran B.

3.5.3 Contoh: Windows

Sistem operasi Windows adalah contoh desain modern yang menggunakan modularitas untuk meningkatkan fungsionalitas dan mengurangi waktu yang diperlukan untuk menerapkan fitur baru. Windows menyediakan dukungan untuk berbagai lingkungan operasi, atau subsistem. Program aplikasi berkomunikasi dengan subsistem ini melalui mekanisme pengiriman pesan. Dengan demikian, program aplikasi dapat dianggap sebagai klien dari server subsistem.

Fasilitas message-passing di Windows disebut fasilitas advanced procedure call (ALPC) tingkat lanjut. Ini digunakan untuk komunikasi antara dua proses pada mesin yang sama. Hal ini mirip dengan mekanisme panggilan prosedur jarak jauh standar (RPC) yang banyak digunakan, tetapi dioptimalkan untuk dan khusus untuk Windows. (Panggilan prosedur jarak jauh dibahas secara rinci di Bagian 3.6.2.) Seperti Mach, Windows menggunakan objek port untuk menetapkan dan memelihara koneksi antara dua proses. Windows menggunakan dua jenis port: port koneksi dan port komunikasi.

Proses server mempublikasikan objek port koneksi yang terlihat oleh semua proses. Ketika seorang klien menginginkan layanan dari subsistem, ia membuka pegangan ke objek port sambungan server dan mengirim permintaan koneksi ke port tersebut. Server kemudian membuat saluran dan mengembalikan pegangan ke klien. Saluran ini terdiri dari sepasang port komunikasi pribadi: satu untuk klien — pesan server, yang lainnya untuk server — pesan klien. Selain itu, saluran komunikasi

mendukung mekanisme panggilan balik yang memungkinkan klien dan server untuk menerima permintaan saat mereka biasanya mengharapkan balasan.

Ketika saluran ALPC dibuat, salah satu dari tiga teknik pengiriman pesan

terpilih:

1. Untuk pesan kecil (hingga 256 byte), antrean pesan port digunakan sebagai penyimpanan perantara, dan pesan disalin dari satu proses ke yang lain.

2. Pesan yang lebih besar harus dilewatkan melalui objek bagian, yang merupakan suatu wilayah memori bersama yang terkait dengan saluran.

3. Ketika jumlah data terlalu besar untuk masuk ke objek bagian, API adalah tersedia yang memungkinkan proses server untuk membaca dan menulis langsung ke dalam ruang alamat klien.

Klien harus memutuskan kapan akan menyiapkan saluran apakah perlu mengirim pesan besar. Jika klien menentukan bahwa ia ingin mengirim pesan besar, ia meminta objek bagian yang akan dibuat. Demikian pula, jika server memutuskan bahwa balasan akan besar, itu membuat objek bagian. Agar objek bagian dapat digunakan, pesan kecil dikirim yang berisi informasi penunjuk dan ukuran tentang objek bagian. Metode ini lebih rumit daripada metode pertama yang tercantum di atas, tetapi metode ini menghindari penyalinan data. Struktur panggilan prosedur lokal lanjutan di Windows ditunjukkan pada Gambar 3.19.

Penting untuk dicatat bahwa fasilitas ALPC di Windows bukan bagian dari Windows API dan karenanya tidak terlihat oleh programmer aplikasi. Sebaliknya, aplikasi yang menggunakan Windows API memanggil panggilan prosedur jarak jauh standar. Ketika RPC sedang dipanggil pada proses pada sistem yang sama, RPC ditangani secara tidak langsung melalui ALPC. panggilan prosedur. Selain itu, banyak layanan kernel menggunakan ALPC untuk berkomunikasi dengan proses klien.

3.6 KOMUNIKASI DALAM SISTEM CLIENT-SERVER

Pada bagian 3.4, kami menjelaskan bagaimana proses dapat berkomunikasi menggunakan memori bersama dan pengiriman pesan. Teknik-teknik ini dapat digunakan untuk komunikasi pada sistem client-server (Bagian 1.11.4). Dalam bagian ini, kami memaparkan tiga strategi lain untuk komunikasi dalam sistem client-server : *socket*, *remote procedure calls* (RPCs), dan *pipes.*

3.6.1 Sockets

Sebuah *sockets* didefinisikan sebagai titik akhir untuk komunikasi. Sepasang proses komunikasi melalui jaringan menggunakan sepasang *sockets*-satu pada setiap proses. Sebuah *sockets* teridentifikasi dengan gabungan alamat IP dengan nomor port. Biasanya, *sockets* digunakan pada arsitektur client-server. Server menunggu perintah dari client dengan mendengar ke port tertentu. Sekali perinyah diterima,

server akan menerima koneksi dari *client socket* untuk melakukan koneksi. Server melakukan beberapa *services* (seperti telnet, FTP, dan HTTP) mendengar ke port yang diketahui (sebuah server telnet diarahkan ke port 23; server FTP diarahkan ke port 21; dan web, atau HTTP server diarahkan ke port 80). Seluruh port dibawah 1024 akan dianggap ***dikenal***; kita dapat menggunakan ini untuk implementasi *services* standar.

Ketika client memproses perintah untuk koneksi, ini akan dikirimkan ke port dari computer host. Port ini memiliki beberapa nomor *arbitrary* lebih dari 1024. Sebagai contoh, jika client pada host X dengan alamat IP 146.86.5.20 ingin melakukan koneksi ke server web (dimana diarahkan ke port 80) ke alamat 161.25.19.8, host X akan mengirimkan ke port 1625, koneksi ini memerlukan kedua *sockets*: (146.86.5.20:1625) pada host X dan (161.25.19.8:80) pada server web. Situasi ini diilustrasikan pada Figure 3.20. Paket *traveling* antara host dikirimkan agar proses berbasis pada destinasi nomor port.

Seluruh koneksi harus unik. Karena itu, jika proses lain pada host X juga ingin melakukan koneksi dengan server web yang sama, ini akan dikirimkan ke nomor port yang lebih dari 1024 dan tidak sama dengan 1625. Ini memastikan bahwa seluruh koneksi termasuk keunikan kedua *sockets.*

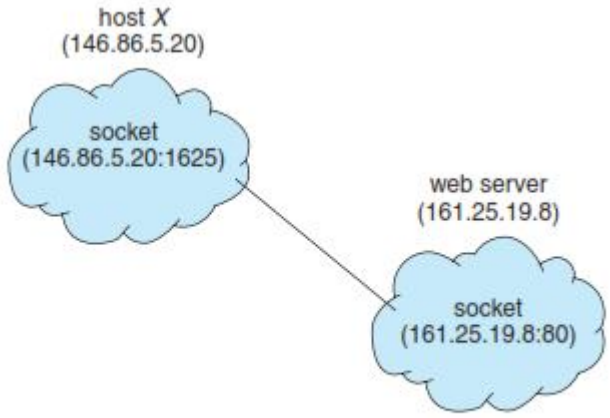


Figure 3.20 Communication using sockets.

Meskipun biasanya contoh program pada teks ini menggunakan C, kita akan mengilustrasikannya menggunakan Java, dimana itu menyediakan antarmuka yang lebih mudah untuk *sockets* dan memiliki *library* yang kaya untuk keperluan jaringan. Bagi yang tertarik menggunakan pemograman C atau C++ pada *sockets* harus melihat catatan bibliografi pada akhir chapter.
Java menyediakan tiga tipe *sockets* yang berbeda. *Connection-oriented (TCP) sockets* diimplementasikan dengan kelas *socket. Connectionless (UDP) sockets* menggunakan kelas *DatagramSocket.* Akhirnya, kelas *MulticastSocket* merupakan bagian dari kelas *DatagramSocket. Multicast socket* memungkinkan data untuk dikirim ke beberapa pengguna.
Contoh kami yang menjelaskan tanggal server yang menggnakan *connection-oriented TCP sockets.* Operasi memungkinkan client untuk meminta tanggal dan waktu saat ini dari server. Server akan diarahkan ke port 6013, meskipun port dapat memiliki nomor

arbitrary yang lebih dari 1024. Ketika koneksi diterima, server akan mengirimkan kembali tanggal dan waktu ke client.
Tanggal server di tampilkan pada Figure 3.21. Server membuat *ServerSocket* yang merujuk ke port 6013. Server kemudian memulai mendengar pada portnya dengan metode *accept()*. Server akan block metode *accept()* menunggu permintaan client untuk koneksi. Ketika koneksi diterima, *accept()* akan kembali ke *socket* yang server gunakan untuk komunikasi dengan client.
Rincian tentang bagaimana server komunikasi dengan *socket* adalah sebaga berikut. Pertama server akan melakukan objek *PrintWriter* yang akan digunakan untuk berkomunikasi dengan client. Objek *PrintWriter* memungkinkan server untuk menulis ke *socket* menggunakan metode *print() dan PrintIn()* rutin untuk output. Server akan memproses kiriman tanggal ke *socket,* server menutup *socket* ke client dan lanjut mendegarkan perintah lainnya.
Sebuah client berkomunikasi dengan server dengan membuat *socket* dan mengkoneksikan ke port yang dimana server dengarkan. Kami mengimplementasikan client pada pemograman Java yang ditunjukkan pada Figure 3.22. Client akan membuat *Socket* dan meminta koneksi ke server dengan IP 127.0.0.1 pada port 6013. Ketika koneksi dilakukan, client dapat membaca dari *socket* menggunakan statement *stream I/O* normal. Setelah menerima tanggal dari server, client menutup

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figure 3.21 Date server.

*Socket*nya dan keluar. Alamat IP 127.0.0.1 adalah alamat IP special yang diketahui juga sebagai ***loopback*****.** Ketika komputer mengarahkan ke alamat IP 127.0.0.1, ini

diarahkan ke komputer itu sendiri. Mekanisme ini memungkinkan client dan server berada pada host yang sama untuk berkomunikasi menggunakan protocol TCP/IP. Alamat IP 127.0.0.1 dapat digantikan dengan alamat IP lain yang digunakan host untuk menjalankan tanggal server. Sebagai tambahan untuk alamat IP, nama host asli, seperti www.westminstercollege.edu, dapat digunakan juga.

Komunikasi menggunakan *sockets*—meskipun umum dan efisien—ini dianggap sebagai form level-rendah komunikasi proses distribusi. Satu alasan mengapa *sockets* memungkinkan sebuah *bytes unstructured stream* untuk digantikan untuk komunikasi. Ini bertanggungjawab pada client atau aplikasi server untuk memaksa struktur data. Pada dua bagian selanjutnya, kami melihat pada dua metode level-tinggi pada komunikasi : *remote procedure calls (RPCs) dan pipes*

3.6.2 Remote Procedure Callls

Sebuah form yang umum untuk layanan remote pada paradigm RPC, dimana kita telah bahas sebelumnya pada bagian 3.5.2. RPC di desain untuk

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args) {
      try {
         /* make connection to server socket */
         Socket sock = new Socket("127.0.0.1",6013);

         InputStream in = sock.getInputStream();
         BufferedReader bin = new
            BufferedReader(new InputStreamReader(in));

         /* read the date from the socket */
         String line;
         while ( (line = bin.readLine()) != null)
            System.out.println(line);

         /* close the socket connection*/
         sock.close();
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Figure 3.22 Date client.

Mekanisme panggilan prosedur abstrak untuk menggunakan kedua sistem dengan koneksi jaringan. Hal ini mirip dalam banyak hal dengan mekanisme IPC yang dijelaskan dalam Bagian 3.4, dan biasanya dibangun di atas sistem semacam itu. Namun, di sini, karena kita berurusan dengan lingkungan di mana proses sedang dieksekusi pada sistem yang terpisah, kita harus menggunakan skema komunikasi berbasis pesan untuk menyediakan remote service.

Berbeda dengan pesan IPC, pesan dipertukarkan dalam komunikasi RPCterstruktur dengan baik dan dengan demikian tidak lagi hanya paket data. Setiap

pesan adalah ditujukan ke daemon RPC mendengarkan port pada sistem remote, dan masing-masing berisi identifier yang menentukan fungsi yang akan dieksekusi dan parameter untuk meneruskan ke fungsi itu. Fungsi ini kemudian dieksekusi seperti yang diminta, dan apa saja output dikirim kembali ke pemohon dalam pesan terpisah.

Port hanyalah nomor yang disertakan di awal paket pesan. Sedangkan sistem biasanya memiliki satu alamat jaringan, ia dapat memiliki banyak port dalam alamat itu untuk membedakan banyak layanan jaringan yang didukungnya. Jika sebuah proses jarak jauh membutuhkan layanan, ia alamat pesan ke port yang tepat. Untuk Misalnya, jika suatu sistem ingin memungkinkan sistem lain untuk dapat mencantumkan arusnya pengguna, itu akan memiliki daemon yang mendukung RPC seperti itu yang melekat pada sebuah port—katakanlah, port 3027. Setiap sistem remote dapat memperoleh informasi yang dibutuhkan (yaitu, daftar pengguna saat ini) dengan mengirimkan pesan RPC ke port 3027 pada server Data akan diterima dalam pesan balasan.

Semantik RPC memungkinkan klien untuk menjalankan prosedur pada host jarak jauh karena akan memanggil prosedur secara lokal. Sistem RPC menyembunyikan detail yang memungkinkan komunikasi berlangsung dengan menyediakan ***stub*** di sisi klien. Biasanya, *stub* yang terpisah untuk setiap prosedur remote yang terpisah. Ketika klien memanggil prosedur jarak jauh, sistem RPC memanggil stub yang sesuai, memberikannya parameter yang diberikan kepada prosedur jarak jauh. Rintisan ini menempatkan port pada server dan marsekal parameter. Parameter marshalling melibatkan pengemasan parameter ke dalam bentuk yang dapat ditransmisikan melalui jaringan. Stub kemudian mengirimkan pesan ke server menggunakan pengiriman pesan. Sebuah rintisan serupa di sisi server menerima pesan ini dan memanggil prosedur di server. Jika perlu, nilai kembali dikirimkan kembali ke klien menggunakan teknik yang sama. Pada sistem Windows, kode rintisan dikompilasi dari spesifikasi yang ditulis dalam Bahasa Definisi Antarmuka Microsoft (MIDL), yang digunakan untuk mendefinisikan antarmuka antara klien dan program server.

Salah satu masalah yang dihadapi dengan berbagai perbedaan adalah presentasi pada mesin klien dan server. Pertimbangkan representasi bilangan bulat 32-bit. Beberapa sistem (dikenal sebagai big-endian) menyimpan byte yang paling signifikan terlebih dahulu, sementara sistem lain (dikenal sebagai little-endian) menyimpan byte yang paling tidak penting pertama. Tidak satu pun pesan yang "lebih baik" melainkan, pilihan ini berbeda dengan arsitektur komputer. Untuk mengatasi perbedaan seperti ini, banyak sistem RPC mendefinisikan presentasi independen - data dari data. Presentasi-presentasi yang lain diketahui lebih lanjut dari presentasi internal (XDR) .Ontheclientside, parameter marshallinginmengonversimenghubungkan -menghubungkandatainto XDR sebelum mereka dikirim ke server. Di sisi server, data XDR adalah unmarshalled dan dikonversi ke server yang bergantung padanya untuk server.

Isu penting lainnya melibatkan semantik panggilan. Sedangkan panggilan prosedur lokal gagal hanya dalam keadaan ekstrim, RPC dapat gagal, atau diduplikasi dan dieksekusi lebih dari sekali, sebagai akibat dari kesalahan jaringan umum. Salah satu cara untuk mengatasi masalah ini adalah untuk sistem operasi untuk memastikan bahwa pesan ditindak tepat satu kali, bukan paling banyak sekali. Sebagian besar prosedur lokal menunjukkan fungsionalitas “tepat”, tetapi perlu diterapkan.

Pertama, pertimbangkan "paling banyak sekali." Semantik ini dapat diimplementasikan dengan melampirkan stempel waktu untuk setiap pesan. Server harus menyimpan riwayat semua cap waktu pesan yang telah diproses atau sejarah cukup besar untuk memastikan bahwa pesan yang berulang terdeteksi. Pesan masuk yang memiliki stempel waktu dalam riwayat diabaikan. Klien kemudian dapat mengirim pesan kepada orang lain atau kadang-kadang dan memastikan bahwa itu terjadi begitu saja.

Untuk “tepat satu kali,” kita perlu menghapus risiko bahwa server tidak akan pernah menerima permintaan. Untuk mencapai hal ini, server harus mengimplementasikan protokol "paling banyak satu kali" yang dijelaskan di atas tetapi juga harus mengakui kepada klien bahwa theRPC panggilan telah tereksekusi di bawah standar. Ini dapat direalisasi di seluruh jaringan. Pelanggan harusmembuat RPCcall secara periodik hingga ia menerima ACK untuk panggilan balik.

Walaupun masalah penting lainnya muncul pada komunikasi antara server dan klien. Dengan prosedur panggilan standar, beberapa bentuk binding berlangsung selama link, load, atau eksekusi waktu (Bagian 8)

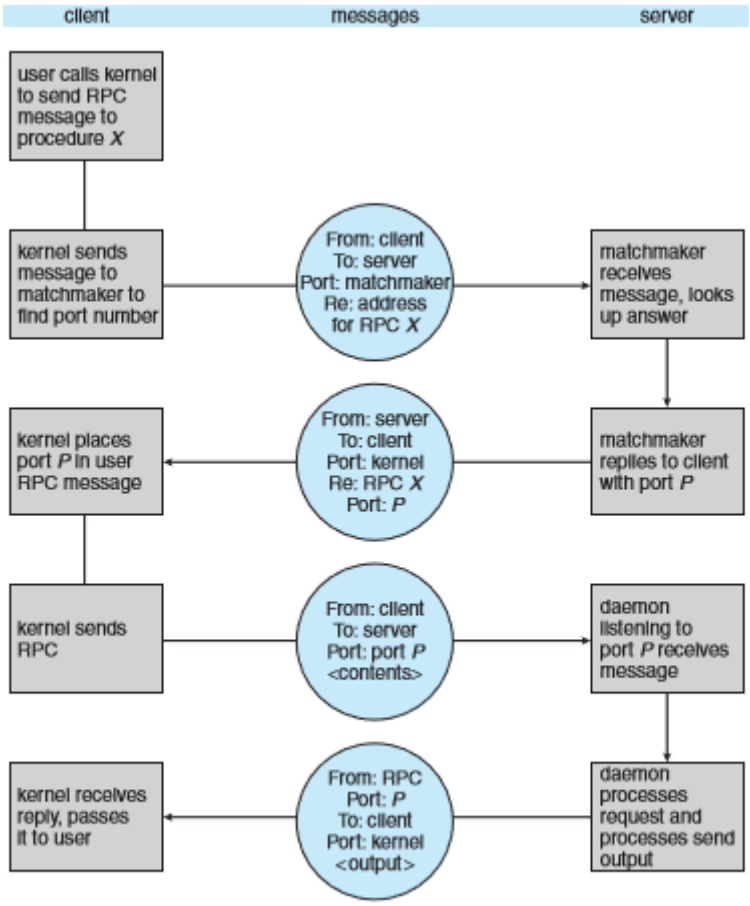


Figure 3.23 Execution of a remote procedure call (RPC).

jadi nama procudr panggilan digantikan dengan alamat memori panggilan prosedur. Skema RPC membutuhkan binding yang mirip dengan klien dan port server, tetapi bagaimana klien tau nomor port pada server? Bahkan sistem memiliki informasi penuh dengan yang lainnya, karena mereka tidak share memory.

Ada dua pendekatan yang umum. Pertama, informasi yang mengikat dapat ditentukan sebelumnya, dalam bentuk alamat port tetap. Pada waktu kompilasi, panggilan RPC memiliki nomor port tetap yang terkait dengannya. Setelah sebuah program dikompilasi, server tidak dapat mengubah nomor port dari layanan yang diminta. Kedua, pengikatan dapat dilakukan secara dinamis dengan mekanisme rendezvous. Biasanya, sistem operasi menyediakan daemon rendezvous (juga disebut matchmaker) pada port RPC tetap. Klien kemudian mengirim pesan yang berisi nama RPC ke daemon rendezvous yang meminta alamat port RPC yang harus dijalankannya. Nomor port dikembalikan, dan panggilan RPC dapat dikirim ke port tersebut hingga proses berakhir (atau server mogok). Metode ini membutuhkan overhead tambahan dari permintaan awal tetapi lebih fleksibel daripada pendekatan pertama. Figure3.23menunjukkan sebagai contoh interaksi.

Skema RPC berguna dalam mengimplementasikan sistem file terdistribusi (Bab 17). Sistem seperti itu dapat diimplementasikan sebagai satu set daemon RPC dan klien. Pesan tersebut ditujukan ke port sistem file terdistribusi pada sebuah server tempat operasi file akan berlangsung. Pesan itu berisi disk operasi yang harus dilakukan. Operasi disk mungkin membaca, menulis, mengganti nama, Hapus, atau status, yang terkait dengan file-file sistem yang terkait kembali pesan berisi data apa pun yang dihasilkan dari panggilan itu, yang dieksekusi oleh DFS daemon atas nama klien. Misalnya, pesan mungkin berisi Permintaan kirim pemindahan beberapa filter jadi dibagi selalu untuk memperbaiki permintaan. Dalam kasus terakhir, beberapa permintaan mungkin diperlukan jika file secara keseluruhan adalah untuk dipindahkan.

3.6.3 Pipes

Sebuah pipa berfungsi sebagai saluran yang memungkinkan dua proses untuk berkomunikasi. Pipa-pipa itu merupakan salah satu mekanisme IPC pertama dalam sistem UNIX awal. Mereka biasanya menyediakan salah satu cara sederhana untuk proses berkomunikasi satu sama lain, meskipun mereka juga memiliki beberapa keterbatasan. Dalam menerapkan pipa, empat masalah harus diperhatikan:

1. Apakah pipa memungkinkan komunikasi dua arah, atau komunikasi searah?
2. Jika komunikasi dua arah diperbolehkan, apakah itu setengah dupleks (data bisa bepergian hanya satu arah pada satu waktu) atau dupleks penuh (data dapat melakukan perjalanan di kedua arah pada waktu bersamaan)?
3. Harus ada hubungan (seperti orang tua-anak) yang ada antara proses komunikasi?
4. Dapatkah pipa berkomunikasi melalui jaringan, atau harus berkomunikasi proses berada di mesin yang sama?

Pada bagian berikut, kami mengeksplorasi dua jenis pipa yang umum digunakan pada keduanya sistem UNIX dan Windows: pipa biasa dan pipa bernama.

3.6.3.1 Ordinary Pipes

Pipes biasa memungkinkan dua proses untuk berkomunikasi dalam standar produsen–fashion konsumen: produser menulis ke salah satu ujung pipa (ujung tulis) dan konsumen membaca dari ujung yang lain (baca-akhir). Akibatnya, biasa saja pipa bersifat unidirectional, memungkinkan hanya komunikasi satu arah. Jika dua arah komunikasi diperlukan, dua pipa harus digunakan, dengan setiap pengiriman pipa data dalam arah yang berbeda. Kami selanjutnya menggambarkan membangun pipa biasa pada sistem UNIX dan Windows. Dalam kedua contoh program, satu proses menulis pesan Sambutan ke pipa, sementara proses lainnya

membaca ini pesan dari pipa. Pada sistem UNIX, pipa biasa dibangun menggunakan fungsi

Pipe(int fd[])

Fungsi ini menciptakan sebuah pipa yang diakses melalui file int fd [] deskriptor: fd [0] adalah ujung-baca dari pipa, dan fd [1] adalah akhir tulis.

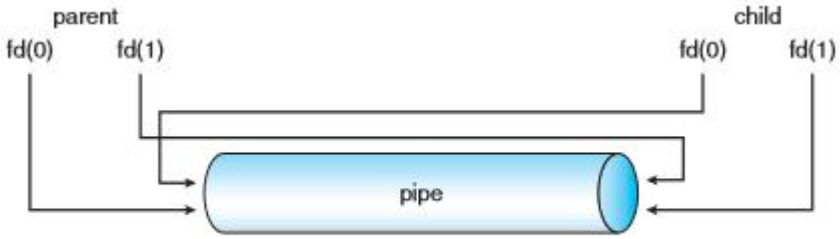


Figure 3.24 File descriptors for an ordinary pipe.

UNIX memperlakukan pipa sebagai jenis file khusus. Dengan demikian, pipa dapat diakses menggunakan panggilan sistem read() dan write().

Masalah biasa tidak bisa dipahami dari berbagai proses yang menyebabkannya. Biasanya, proses orang tua menciptakan sebuah pipa dan menggunakannya untuk berkomunikasi dengan proses achild yang menghasilkan gelombang (). RecallfromSection3.3.1thachachproses prosesmenjalankanperbedaandarifahnya. Karena apipe adalah jenis file khusus, anak mewarisi pipa dari proses induknya. Figure 3.24 mengilustrasikan hubungan deskriptor fisik dan proses anak-anak.

Dalam program UNIX yang ditunjukkan pada Gambar 3.25, proses induk menciptakan pipa dan kemudian mengirimkan sebagai panggilan fork() bergantung pada bagaimana data berjalan melalui pipa. Dalam hal ini, orang tua menulis ke pipa, dan anak membaca darinya. Penting untuk diperhatikan bahwa proses yang berbeda dan proses kecil pada awalnya menutup ujung pipa yang tidak digunakan. Meskipun program yang ditunjukkan pada Gambar 3.25tidak perlu melakukan tindakan, ini merupakan langkah awal yang penting untuk mengukur pembacaan hasil dari ujung dan bagian akhir file (baca () kembali0) dimana penulis telah menutup bagian dari pipa.

Pipa biasa pada sistem Windows disebut pipa anonim, dan mereka berperilaku serupa dengan rekan UNIX mereka: mereka searah dan

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

      /* Program continues in Figure 3.26 */
```

Figure 3.25 Ordinary pipe in UNIX.

```
   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr,"Pipe failed");
      return 1;
   }

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }

   if (pid > 0) { /* parent process */
      /* close the unused end of the pipe */
      close(fd[READ_END]);

      /* write to the pipe */
      write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

      /* close the write end of the pipe */
      close(fd[WRITE_END]);
   }
   else { /* child process */
      /* close the unused end of the pipe */
      close(fd[WRITE_END]);

      /* read from the pipe */
      read(fd[READ_END], read_msg, BUFFER_SIZE);
      printf("read %s",read_msg);

      /* close the write end of the pipe */
      close(fd[READ_END]);
   }

   return 0;
}
```

Figure 3.26 Figure 3.25, continued.

menggunakan hubungan orangtua-anak antara proses komunikasi. Selain itu, membaca dan menulis ke pipa dapat dilakukan dengan fungsi ReadFile () dan WriteFile () biasa. API Windows untuk membuat fungsi tipeistheCreatePipe (), yang dilewatifourparameters. Theparametersmenyediakanpasatpasutanganuntuk (1) membaca dan (2) menulispipa, dan juga (3) anstesi struktur STARTUPINFO, yang digunakan untuk mendeskripsikan proses untuk mengalilkan dengan pipa. Selanjutnya, (4) ukurantipe dari (inbytes) mungkin dispesifikasikan.

Gambar 3.27 mengilustrasikan proses pembuatan saluran anonim untuk berkomunikasi dengan anaknya. Tidak seperti sistem UNIX, di mana proses anak

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;

        /* Program continues in Figure 3.28 */
```

Figure 3.27 Windows anonymous pipe — parent process.

secara otomatis mewarisi pipa yang dibuat oleh induknya, Windows mengharuskan programmer untuk menentukan atribut mana yang akan diwariskan oleh proses turunan. Hal ini dicapai dengan terlebih dahulu menginisialisasi struktur ATTRIBUT KEAMANAN untuk memungkinkan pegangan diwariskan dan kemudian mengarahkan proses tangan anak untuk input standar atau keluaran standar ke pegangan baca atau tulis pada pipa. Karena anak akan membaca dari pipa, orang tua harus mengalihkan input standar anak ke pegangan baca dari pipa. Selain itu, karena pipa-pipa setengah dupleks, maka perlu untuk melarang anak mewarisi ujung-menulis resep. Program ini mem-proteksi proses mekanik termasuk program dalam Angka 3.11, kecuali yang kelima, meteran perancah, yang menunjukkan bahwa proses anak adalah mewarisi gagang yang ditunjuk dari induknya. Sebelum menulis ke pipa, orang tua terlebih dahulu menutup bagian yang tidak digunakan dari pipa. Proses anak yang membaca dari pipa ditunjukkan pada Gambar 3.29. Sebelum membaca dari pipa, program ini mendapatkan pegangan baca ke pipa dengan memohon GetStdHandle ().

Perhatikan bahwa pipa biasa memerlukan hubungan orangtua-anak antara proses komunikasi pada sistem UNIX dan Windows. Ini berarti bahwa alat-alat tersebut dapat digunakan hanya untuk komunikasi di antara proses mesin yang sama.

3.6.3.2 Named Pipes

Pipes biasa menyediakan mekanisme sederhana untuk memungkinkan sepasang proses
untuk berkomunikasi. Namun, pipa biasa hanya ada saat prosesnya berkomunikasi satu sama lain. Pada sistem UNIX dan Windows, satu kali proses telah selesai berkomunikasi dan telah dihentikan, pipa yang biasa tidak ada lagi.

Pipa bernama memberikan alat komunikasi yang jauh lebih kuat. Komunikasi bisa dua arah, dan tidak ada hubungan orangtua-anak yang diperlukan. Setelah pipa bernama didirikan, beberapa proses dapat menggunakannya untuk

komunikasi. Bahkan, dalam skenario yang khas, sebuah pipa bernama memiliki beberapa penulis. Selain itu,pipa bernama terus ada setelah proses berkomunikasi

```
/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
   fprintf(stderr, "Create Pipe Failed");
   return 1;
}

/* establish the START_INFO structure for the child process */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
 TRUE, /* inherit handles */
 0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
   fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figure 3.28 Figure 3.27, continued.

selesai. Baik UNIX dan sistem Windows mendukung pipa bernama, meskipun detail Implementasinya sangat berbeda. Selanjutnya, kami akan uraikan pipa bernama pada setiap system ini.

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

   /* get the read handle of the pipe */
   ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

   /* the child reads from the pipe */
   if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
     printf("child read %s",buffer);
   else
     fprintf(stderr, "Error reading from pipe");

   return 0;
}
```

Figure 3.29 Windows anonymous pipes—child process.

Pipa bernama disebut sebagai sistem inUNIX FIFOs. Setelah dibuat, mereka muncul sebagai file khas dalam sistem file. FIFO dibuat dengan mkfifo () system call dan dimanipulasi dengan panggilan sistem open (), read (), write (), dan close () biasa. Ini akan terus ada sampai secara eksplisit dihapus dari sistem file. Meskipun FIFO mengijinkan komunikasi dua arah, hanya transmisi half-duplex yang diizinkan. Jika data harus berjalan di kedua arah, dua perangkat elektronik digunakan secara khusus. Selain itu, proses komunikasi akan berada di mesin yang sama. Jika komunikasi intermachine diperlukan, soket (Bagian 3.6.1) harus digunakan.

Pipa bernama pada sistem Windows menyediakan mekanisme komunikasi yang lebih kaya manisme dari rekan UNIX mereka. Komunikasi full-duplex diperbolehkan, dan proses berkomunikasi dapat berada pada yang sama atau berbeda mesin. Selain itu, hanya data berorientasi byte yang dapat dikirimkan melalui UNIX FIFO, sedangkan Windows sistem memungkinkan kedua data *byte-orm message-oriented.* Pipa bernama dibuat dengan fungsi CreateNamedPipe (), dan klien dapat terhubung ke pipa bernama menggunakan ConnectNamedPipe (). Komunikasi di atas pipa bernama dapat diselesaikan menggunakan fungsu ReadFile () dan WriteFile ().

3.7 SUMMARY

Proses adalah program yang sedang dieksekusi. Saat proses dijalankan, ia mengubah status. Keadaan suatu proses ditentukan oleh aktivitas proses saat ini. Setiap proses dapat berada di salah satu kondisi berikut: baru, siap, berjalan, menunggu, atau dihentikan.

PIPES IN PRACTICE

Pipes digunakan cukup sering dalam lingkungan baris perintah UNIX untuk situasi di mana output dari satu perintah berfungsi sebagai input ke yang lain. Sebagai contoh, perintah UNIX ls menghasilkan daftar direktori. Untuk daftar direktori yang sangat panjang, output mungkin bergulir melalui beberapa layar. Perintah lebih mengatur output dengan hanya menampilkan satu layar output pada satu waktu; pengguna harus menekan tombol spasi untuk berpindah dari satu layar ke layar berikutnya. Menyiapkan pipes antara ls dan lebih banyak perintah (yang berjalan sebagai proses individual) memungkinkan output dari ls untuk dikirim sebagai input ke lebih banyak, memungkinkan pengguna untuk menampilkan direktori besar yang menampilkan layar pada suatu waktu. Sebuah pipes dapat dibangun pada baris perintah menggunakan | character. Perintah lengkapnya

ls | more

Dalam skenario ini, perintah ls berfungsi sebagai produser, dan outputnya dikonsumsi oleh lebih banyak perintah.
Sistem Windows menyediakan perintah yang lebih untuk shell DOS dengan fungsionalitas yang serupa dengan yang dari rekan UNIX. Shell DOS juga menggunakan | karakter untuk membangun pipa. Satu-satunya perbedaan adalah bahwa untuk mendapatkan daftar direktori, DOS menggunakan perintah dir daripada ls, seperti yang ditunjukkan di bawah ini:

dir | more

Setiap proses diwakili dalam sistem operasi oleh Process Control Blocknya sendiri (PCB).

Suatu proses, ketika tidak dieksekusi, ditempatkan dalam antrean menunggu. Ada dua kelas utama antrian dalam sistem operasi: Antrian permintaan / I dan antrian siap. Antrean siap berisi semua proses yang siap dijalankan dan menunggu CPU. Setiap proses diwakili oleh PCB.

Sistem operasi harus memilih proses dari berbagai antrian penjadwalan. Penjadwalan jangka panjang (pekerjaan) adalah pemilihan proses yang akan diizinkan untuk bersaing untuk CPU. Biasanya, penjadwalan jangka panjang sangat dipengaruhi oleh pertimbangan alokasi sumber daya, terutama manajemen memori. Penjadwalan jangka pendek (CPU) adalah pemilihan satu proses dari antrean siap.

Sistem operasi harus menyediakan mekanisme untuk proses induk untuk membuat proses child baru. Parent dapat menunggu childnya berhenti sebelum melanjutkan, atau Parent dan child dapat melakukan secara bersamaan. Ada beberapa alasan untuk memungkinkan eksekusi bersamaan: berbagi informasi, kecepatan komputasi, modularitas, dan kenyamanan.

Proses yang dijalankan dalam sistem operasi dapat berupa proses independen atau proses kerja sama. Proses kerjasama membutuhkan mekanisme komunikasi interprocess untuk berkomunikasi satu sama lain. Pada prinsipnya, komunikasi dicapai melalui dua skema: memori bersama dan pengiriman pesan. Metode memori bersama membutuhkan proses berkomunikasi.

```
#include <sys/types.h>
#include <stdio.h> #include
<unistd.h> int value = 5; int main()
{
pid t pid; pid = fork();
    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */ wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */ return 0;
    }
}
```

Figure 3.30 What output will be at Line A?

untuk berbagi beberapa variabel. Proses ini diharapkan dapat bertukar informasi melalui penggunaan variabel-variabel bersama ini. Dalam sistem memori bersama, tanggung jawab untuk menyediakan komunikasi terletak pada pemrogram aplikasi; sistem operasi hanya perlu menyediakan memori bersama. Metode pengiriman pesan memungkinkan proses untuk bertukar pesan. Tanggung jawab untuk menyediakan komunikasi dapat beristirahat dengan sistem operasi itu sendiri. Kedua skema ini tidak saling eksklusif dan dapat digunakan bersamaan dalam satu sistem operasi.

Komunikasi dalam sistem client-server dapat menggunakan (1) soket, (2) panggilan prosedur jarak jauh- Remote Procedure Calls (RPCs), atau (3) pipes. Soket didefinisikan sebagai titik akhir untuk komunikasi. Koneksi antara sepasang aplikasi terdiri dari sepasang soket, satu di setiap ujung saluran komunikasi. RPC adalah bentuk lain dari komunikasi terdistribusi. RPC terjadi ketika proses (atau rangkaian) memanggil prosedur pada aplikasi jarak jauh. Pipes menyediakan cara yang relatif sederhana untuk proses berkomunikasi satu sama lain. Pipes biasa memungkinkan komunikasi antara proses parent dan child, sementara penyaringan nama pipes tidak berhubungan untuk berkomunikasi.

LATIHAN PRAKTIK

3.1 Menggunakan program yang ditunjukkan pada Gambar 3.30, jelaskan apa outputnya di LINE A.
3.2 Termasuk proses induk awal, berapa banyak proses yang dibuat oleh program yang ditunjukkan pada Gambar 3.31?

```
#include <stdio.h> #include <unistd.h>
int main()
{
    /* fork a child process */ fork();

    /* fork another child process */ fork();

    /* and fork another */ fork(); return 0; }
```

Figure 3.31 How many processes are created?

3.3 Versi asli dari sistem operasi seluler iOS Apple tidak menyediakan sarana pemrosesan bersamaan. Diskusikan tiga komplikasi utama yang proses konkuren menambah sistem operasi.
3.4 Prosesor Sun UltraSPARC memiliki beberapa set register. Jelaskan apa yang terjadi ketika pergantian konteks terjadi jika konteks baru sudah dimuat ke salah satu set register. Apa yang terjadi jika konteks baru ada dalam memori daripada di set register dan semua set register sedang digunakan?
3.5 Ketika proses membuat proses baru menggunakan operasi fork (), yang mana dari status berikut ini dibagi antara proses induk dan proses anak? Sebuah.

- Stack
- Heap
- Shared Memory Segments

3.6 Pertimbangkan semantik “exactly one” sehubungan dengan mekanisme RPC. Apakah algoritma untuk mengimplementasikan semantik ini berjalan dengan benar bahkan jika pesan ACK yang dikirim kembali ke klien hilang karena masalah jaringan? Jelaskan urutan pesan, dan diskusikan apakah "exactly one" masih dipertahankan.
3.7 Asumsikan bahwa sistem terdistribusi rentan terhadap kegagalan server. Mekanisme apa yang diperlukan untuk menjamin semantik “exactly one” untuk pelaksanaan RPCs?

LATIHAN

3.8 Jelaskan perbedaan antara penjadwalan jangka pendek, jangka menengah, dan jangka panjang.

```
#include <stdio.h> #include <unistd.h> int
main() {
    int i;
```

```
for (i = 0; i < 4; i++) fork(); return 0; }
```

Figure 3.32 How many processes are created?

3.9 Jelaskan tindakan yang diambil oleh kernel untuk beralih konteks antar proses.
3.10 Buatlah pohon proses serupa dengan Gambar 3.8. Untuk mendapatkan informasi proses untuk sistem UNIX atau Linux, gunakan perintah ps -ael.

```
#include <sys/types.h>
#include <stdio.h> #include <unistd.h> int
main()
{
pid t pid;
    /* fork a child process */ pid = fork();
    if (pid < 0) { /* error occurred */ fprintf(stderr, "Fork Failed"); return
        1;
    }
    else if (pid == 0) { /* child process */ execlp("/bin/ls","ls",NULL);
        printf("LINE J");
    }
    else { /* parent process */
        /* parent will wait for the child to complete */ wait(NULL);
        printf("Child Complete");
    } return 0; }
```

Figure 3.33 When will LINE J be reached?

Gunakan command man ps untuk mendapatkan informasi lebih lanjut tentang perintah ps. Task manager pada sistem Windows tidak memberikan ID proses induk, tetapi alat process monitor, tersedia dari technet.microsoft.com, menyediakan alat process-tree

3.11 Jelaskan peran proses init pada sistem UNIX dan Linux dalam hal penghentian proses.
3.12 Termasuk proses induk awal, berapa banyak proses yang dibuat oleh program yang ditunjukkan pada Gambar 3.32?
3.13 Jelaskan keadaan di mana garis kode yang ditandai printf ("LINE J") pada Gambar 3.33 akan tercapai.

3.14 Dengan menggunakan program pada Gambar 3.34, identifikasikan nilai pid pada garis A, B, C, dan D. (Asumsikan bahwa pid aktual dari parent dan child adalah 2600 dan 2603, masing-masing.)

```
#include <sys/types.h>
#include <stdio.h> #include <unistd.h> int
main()
{
```

```
pid t pid, pid1;

    /* fork a child process */ pid = fork();

    if (pid < 0) { /* error occurred */ fprintf(stderr, "Fork Failed"); return
        1;
    }
    else if (pid == 0) { /* child process */ pid1 = getpid(); printf("child:
        pid = %d",pid); /* A */
        printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid(); printf("parent: pid = %d",pid); /* C */ printf("parent:
        pid1 = %d",pid1); /* D */ wait(NULL);
    } return 0; }
```

Figure 3.34 What are the pid values?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h> #define SIZE 5 int nums[SIZE] =

{0,1,2,3,4}; int main()

{int i; pid t pid; pid = fork();


    if (pid == 0)for (i = 0; i < SIZE; i++)nums[i] *= -i;{    {


        printf("CHILD: %d ",nums[i]); /* LINE X */
     }


       }else if (pid > 0)wait(NULL);        {

     for (i = 0; i < SIZE; i++)
        printf("PARENT: %d ",nums[i]); /* LINE Y */
  } return 0; }
```

Figure 3.35 What output will be at Line X and Line Y?

3.15 Berikan contoh situasi di mana pipes biasa lebih cocok daripada pipes bernama dan contoh situasi di mana pipes bernama lebih cocok daripada pipes biasa.

3.16 Pertimbangkan mekanisme RPC. Jelaskan konsekuensi yang tidak diinginkan yang dapat timbul dari tidak menegakkan semantik “at most once” atau “exactly one”. Jelaskan kemungkinan penggunaan untuk mekanisme yang tidak memiliki jaminan ini.

3.17 Dengan menggunakan program yang ditunjukkan pada Gambar 3.35, jelaskan apa outputnya di garis X dan Y.

3.18 Apa manfaat dan kerugian masing-masing yang berikut ini? Pertimbangkan baik level sistem dan level programmer.
a. Komunikasi sinkron dan asinkron (synchronous and asynchronous communication)
b. Penyangga otomatis dan eksplisit (Automatic and explicit buffering)
c. Kirim dengan menyalin dan kirim dengan referensi (send by copy and send by reference)
d. Pesan berukuran tetap dan berukuran variabel (fixed-sized and variable-sized messages)

MASALAH PROGRAM

3.19 Menggunakan sistem UNIX atau Linux, tulislah program C yang menfork proses child yang pada akhirnya menjadi proses zombie. Proses zombie ini harus tetap berada di sistem paling tidak selama 10 detik. Status proses dapat diperoleh dari perintah

Ps -- l

Status proses ditunjukkan di bawah kolom S; proses dengan keadaan Z adalah zombi. Proses identifier (pid) dari proses child tercantum dalam kolom PID, dan yang dari parent tercantum dalam kolom PPID.

Mungkin cara termudah untuk menentukan bahwa proses anak memang zombie adalah untuk menjalankan program yang telah Anda tulis di latar belakang (menggunakan &) dan kemudian jalankan perintah ps-l untuk menentukan apakah anak itu adalah proses zombie. Karena Anda tidak ingin terlalu banyak proses zombie yang ada dalam sistem, Anda harus menghapus salah satu yang telah Anda buat. Cara termudah untuk melakukannya adalah dengan menghentikan proses induk menggunakan perintah kill. Misalnya, jika id proses dari parent adalah 4884, Anda akan memasukkan

kill -9 4884

3.20 Sistem operasi Pid Manager bertanggung jawab untuk mengelola pengidentifikasi proses. Ketika sebuah proses pertama kali dibuat, ia diberi pid unik oleh manajer pid. Pid dikembalikan ke manajer pid ketika proses selesai eksekusi, dan manajer dapat menetapkan kembali pid ini. Identifikasi proses dibahas lebih lengkap di Bagian 3.3.1. Apa yang paling penting di sini adalah mengenali bahwa pengidentifikasi proses harus unik; tidak ada dua proses aktif yang dapat memiliki pid yang sama.

Gunakan konstanta berikut untuk mengidentifikasi kisaran nilai pid yang mungkin:

#define MIN PID 300
#define MAX PID 5000

Anda dapat menggunakan struktur data pilihan Anda untuk mewakili ketersediaan pengidentifikasi proses. Salah satu strategi adalah mengadopsi apa yang telah dilakukan Linux dan menggunakan bitmap di mana nilai 0 pada posisi i menunjukkan bahwa id proses dari nilai i tersedia dan nilai 1 menunjukkan bahwa id proses sedang digunakan. Terapkan API berikut untuk mendapatkan dan merilis pid:

- int allocate_map (void) —Membuat dan menginisialisasi infrastruktur untuk merepresentasikan pids; mengembalikan — 1 jika tidak berhasil, 1 jika berhasil
- int allocate_pid (void) —Mengalokasikan dan mengembalikan pid; kembali—1 jika tidak dapat mengalokasikan pid (semua pids sedang digunakan)
- void release_pid (int pid) —Mengganti pid

Masalah pemrograman ini akan dimodifikasi nanti di Chpaters 4 dan 5.

3.21 The Collatz conjecture menyangkut apa yang terjadi ketika kita mengambil bilangan bulat positif n dan menerapkan algoritma berikut:

$$n = \begin{cases} n/2, & \text{if n is even} \\ 3 \times n + 1, & \text{if n is odd} \end{cases}$$

Conjecture menyatakan bahwa ketika algoritma ini terus diterapkan, semua bilangan bulat positif akhirnya akan mencapai 1. Sebagai contoh, jika n = 35, urutannya adalah

35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Tulis program C menggunakan pemanggilan sistem fork () yang menghasilkan urutan ini dalam proses child. Nomor awal akan diberikan dari baris perintah. Sebagai contoh, jika 8 dilewatkan sebagai parameter pada baris perintah, proses child akan menghasilkan 8, 4, 2, 1. Karena proses parent dan child memiliki salinan data sendiri, maka perlu bagi child untuk output urutannya. Mintalah parent memanggil panggilan wait () untuk menunggu proses child selesai sebelum keluar dari program. Lakukan pemeriksaan kesalahan yang diperlukan untuk memastikan bahwa bilangan bulat positif dilewatkan pada baris perintah.

3.22 Dalam Latihan 3.21, proses child harus mengeluarkan urutan angka yang dihasilkan dari algoritma yang ditentukan oleh dugaan Collatz karena parent dan child memiliki salinan data mereka sendiri. Pendekatan lain untuk merancang program ini adalah membuat objek memori bersama antara proses induk dan proses child. Teknik ini memungkinkan child

untuk menulis isi urutan ke objek memori bersama. Parent kemudian dapat mengeluarkan urutan ketika child selesai. Karena memori dibagikan, perubahan apa pun yang dibuat child akan tercermin dalam proses induk juga.

Program ini akan disusun menggunakan memori bersama POSIX seperti yang dijelaskan dalam Bagian 3.5.1. Proses induk akan berlanjut melalui langkah-langkah berikut:

a) Buat objek shared-memory (shm open (), ftruncate (), dan mmap ()).
b) Buat proses anak dan tunggu sampai selesai.
c) Keluaran isi shared-memory
d) Hapus shared-memory object.

Satu bidang perhatian dengan cooperating process melibatkan masalah sinkronisasi. Dalam latihan ini, proses parent dan child harus dikoordinasikan sehingga parent tidak mengeluarkan urutan sampai child menyelesaikan eksekusi. Kedua proses ini akan disinkronkan menggunakan panggilan sistem wait(): proses induk akan meminta wait (), yang akan menangguhkannya sampai proses child keluar.

3.23 Bagian 3.6.1 mendeskripsikan nomor port di bawah 1024 yang sudah dikenal — artinya, mereka menyediakan layanan standar. Port 17 dikenal sebagai layanan Quote-of-the-day. Ketika klien terhubung ke port 17 di server, server merespons dengan kutipan untuk hari itu.

Ubah server tanggal yang ditunjukkan pada Gambar 3.21 sehingga memberikan kutipan hari daripada tanggal saat ini. Tanda kutip harus dapat dicetak karakter ASCII dan harus berisi kurang dari 512 karakter, meskipun beberapa baris diperbolehkan. Karena port 17 sudah dikenal dan karena itu tidak tersedia, minta server Anda mendengarkan port 6017. Klien tanggal yang ditunjukkan pada Gambar 3.22 dapat digunakan untuk membaca tanda kutip yang dikembalikan oleh server Anda.

3.24 Haiku adalah puisi tiga baris di mana baris pertama berisi lima suku kata, baris kedua berisi tujuh suku kata, dan baris ketiga berisi lima suku kata. Tulis server haiku yang mendengarkan port 5575. Ketika klien terhubung ke port ini, server merespons dengan haiku. Klien tanggal yang ditunjukkan pada Gambar 3.22 dapat digunakan untuk membaca tanda kutip yang dikembalikan oleh server haiku Anda.

3.25 Server echo menggemakan kembali apa pun yang diterimanya dari klien. Misalnya, jika klien mengirim server string Halo di sana !, server akan merespons dengan Hello di sana!
Tuliskan server echo menggunakan API jaringan Java yang dijelaskan di Bagian 3.6.1. Server ini akan menunggu koneksi klien menggunakan metode accept (). Ketika koneksi klien diterima, server akan berputar, melakukan langkah-langkah berikut:

- Baca data dari soket ke buffer.
- Tulis isi buffer kembali ke klien.

Server akan keluar dari loop hanya ketika telah menentukan bahwa klien telah menutup koneksi.

Server tanggal yang ditunjukkan pada Gambar 3.21 menggunakan kelas java.io.Buffered Reader. BufferedReader memperluas kelas java.io.Reader, yang digunakan untuk membaca aliran karakter. Namun, server echo tidak dapat menjamin akan membaca karakter dari klien; mungkin menerima data biner juga. Kelas java.io.InputStream berhubungan dengan data pada level byte daripada level karakter. Dengan demikian, server echo Anda harus menggunakan objek yang memperluas java.io.Input Stream. Metode read () di kelas java.io.InputStream mengembalikan −1 ketika klien menutup ujung sambungan soketnya.

3.26 Rancang sebuah program menggunakan pipes biasa di mana satu proses mengirim pesan string ke proses kedua, dan proses kedua membalikkan kasus dari setiap karakter dalam pesan dan mengirimkannya kembali ke proses pertama. Contohnya, jika proses pertama adalah proses akhir Hi Ada, proses kedua akan kembali ke sini. Ini akan membutuhkan penggunaan dua pipes, satu untuk mengirim pesan asli dari proses pertama ke proses kedua dan yang lainnya untuk mengirim pesan yang telah dimodifikasi dari proses kedua ke proses pertama. Anda dapat menulis program ini menggunakan pipa UNIX atau Windows.

3.27 Rancang program penyalinan file bernama filecopy menggunakan pipes biasa. Program ini akan melewati dua parameter: nama file yang akan disalin dan nama file yang disalin. Program ini kemudian akan membuat pipes biasa dan menulis isi file yang akan disalin ke pipes. Proses child akan membaca file ini dari pipes dan menulisnya ke file tujuan. Sebagai contoh, jika kita menjalankan program sebagai berikut:

filecopy input.txt copy.txt

file input.txt akan ditulis ke pipes. Proses child akan membaca isi file ini dan menulisnya ke file tujuan copy.txt. Anda dapat menulis program ini menggunakan UNIX atau pipes Windows.

PROGRAMMING PROJECT

Project 1 — UNIX Shell dan Fitur History

Proyek ini terdiri dari merancang program C untuk melayani sebagai antarmuka shell yang menerima perintah pengguna dan kemudian mengeksekusi setiap perintah dalam proses terpisah. Proyek ini dapat diselesaikan pada sistem Linux, UNIX, atau Mac OS X.

Antarmuka shell memberi pengguna prompt, setelah itu perintah berikutnya dimasukkan. Contoh di bawah ini menggambarkan prompt osh> dan perintah pengguna berikutnya: cat prog.c. (Perintah ini menampilkan file prog.c pada terminal menggunakan perintah UNIX cat.)

Osh> cat prog.c

Salah satu teknik untuk mengimplementasikan antarmuka shell adalah memiliki proses induk terlebih dahulu membaca apa yang pengguna masukkan pada baris perintah (dalam hal ini, catprog.c), dan kemudian membuat proses turunan terpisah yang melakukan perintah. Kecuali ditentukan lain, proses induk menunggu child untuk keluar sebelum melanjutkan. Ini mirip dalam fungsionalitas untuk penciptaan proses baru yang diilustrasikan pada Gambar 3.10. Namun, shell UNIX biasanya juga memungkinkan proses child untuk berjalan di latar belakang, atau secara bersamaan. Untuk mencapai hal ini, kami menambahkan ampersand (&) di akhir perintah. Jadi, jika kita menulis ulang perintah di atas sebagai

osh> cat prog.c &

proses parent dan child akan berjalan secara bersamaan.

Proses child terpisah dibuat menggunakan panggilan sistem fork (), dan perintah pengguna dijalankan menggunakan salah satu panggilan sistem dalam keluarga exec () (seperti yang dijelaskan dalam Bagian 3.3.1).

Program C yang menyediakan operasi umum shell baris perintah diberikan pada Gambar 3.36. Fungsi main () menampilkan prompt osh-> dan menguraikan langkah-langkah yang harus diambil setelah input dari pengguna telah dibaca. Fungsi main () terus-menerus loop sepanjang harus berjalan sama dengan 1; ketika pengguna masuk keluar pada prompt, program Anda akan diatur harus berjalan ke 0 dan berakhir. Proyek ini disusun menjadi dua bagian: (1) membuat proses child dan mengeksekusi perintah pada child, dan (2) memodifikasi shell untuk memungkinkan fitur riwayat.

```
#include <unistd.h>

#define MAX LINE 80 /* The maximum length command */ int main(void)

{
char *args[MAXLINE/2 + 1]; /* command line arguments */ int shouldrun = 1; /* flag to
determine when to exit program */

    while (shouldrun) { printf("osh>"); fflush(stdout);

        /**
                * After reading user input, the steps are:
                * (1) fork a child process using fork()
                * (2) the child process will invoke execvp()
```

```
*        (3) if command included &, parent will invoke wait() */ } return 0; }
```

Figure 3.36 Outline of simple shell.

Bagian I - Membuat Proses child

Tugas pertama adalah memodifikasi fungsi main () pada Gambar 3.36 sehingga proses child di-fork dan mengeksekusi perintah yang ditentukan oleh pengguna. Ini akan membutuhkan penguraian apa yang telah dimasukkan oleh pengguna ke dalam token terpisah dan menyimpan token dalam rangkaian string karakter (args pada Gambar 3.36). Sebagai contoh, jika pengguna memasukkan perintah ps -ael pada prompt osh>, nilai yang disimpan dalam larik args adalah:

```
args[0] = "ps"

args[1] = "-ael"

args[2] = NULL
```

Array argumen ini akan diteruskan ke fungsi execvp (), yang memiliki prototipe berikut:

```
execvp(char *command, char *params[]);
```

Di sini, perintah mewakili perintah yang harus dilakukan dan params menyimpan parameter ke perintah ini. Untuk proyek ini, fungsi execvp () harus dipanggil sebagai execvp (args [0], args). Pastikan untuk memeriksa apakah pengguna memasukkan & untuk menentukan apakah proses parent harus menunggu child untuk keluar.

Bagian II — Membuat History Features

Tugas selanjutnya adalah memodifikasi program antarmuka shell sehingga menyediakan fitur riwayat yang memungkinkan pengguna untuk mengakses perintah yang paling baru dimasukkan. Pengguna akan dapat mengakses hingga 10 perintah dengan menggunakan fitur. Perintah akan diberi nomor berurutan mulai dari 1, dan penomoran akan berlanjut sampai 10. Sebagai contoh, jika pengguna telah memasukkan 35 perintah, 10 perintah terakhir akan berjumlah 26 hingga 35.

Pengguna akan dapat daftar sejarah perintah dengan memasukkan sejarah perintah pada prompt osh>. Sebagai contoh, asumsikan bahwa sejarah terdiri dari perintah (dari yang paling hingga yang paling baru):

```
ps, ls -l, top, cal, who, date
```

The command history akan menampilkan:

```
6 ps
5 ls -l
4 top
3 cal
```

2 who
1 date

Program Anda harus mendukung dua teknik untuk mengambil perintah dari command history:
1. Saat pengguna enter !!, perintah terbaru dalam history dieksekusi.
2. Saat pengguna memasukkan sendiri(single)! diikuti oleh integer N, perintah N dalam history dieksekusi.
Melanjutkan contoh kita dari atas, jika pengguna enter !!, perintah ps akan dilakukan; jika pengguna enter! 3, kal perintah akan dieksekusi. Setiap perintah yang dijalankan dengan cara ini harus digemakan di layar pengguna. Perintah juga harus ditempatkan di buffer history sebagai perintah selanjutnya.
Program ini juga harus mengelola penanganan kesalahan dasar. Jika tidak ada perintah dalam history, enter !! harus menghasilkan pesan : “No commands in history.” Jika tidak ada perintah yang sesuai dengan nomor yang dimasukkan dengan single!, "No such command in history."

Project 2 — Modul Kernel Linux untuk Tugas Pencatatan (Listing Tasks)
Dalam proyek ini, Anda akan menulis modul kernel yang mencantumkan semua tugas saat ini dalam sistem Linux. Pastikan untuk meninjau proyek pemrograman di Bab 2, yang berkaitan dengan pembuatan modul kernel Linux, sebelum Anda memulai proyek ini. Proyek ini dapat diselesaikan menggunakan mesin virtual Linux yang disediakan dengan teks ini.

Bagian I — Iterasi atas Tugas secara Linear
Seperti yang diilustrasikan dalam Bagian 3.1, PCB di Linux diwakili oleh struct tugas struktur, yang ditemukan dalam file include <linux / sched.h>. Di Linux, untuk setiap proses () makro dengan mudah memungkinkan iterasi atas semua tugas saat ini dalam sistem:

```
#include <linux/sched.h> struct task struct *task;

for each process(task){
                    /* on each iteration task points to the next task */
}
```

Berbagai bidang dalam struct tugas kemudian dapat ditampilkan sebagai program loop melalui untuk setiap process () makro.

Bagian I Tugas
Rancang sebuah modul kernel yang melakukan iterasi melalui semua tugas dalam sistem menggunakan untuk setiap process () makro. Secara khusus, output nama tugas (dikenal sebagai nama yang dapat dieksekusi), status, dan proses id dari setiap tugas. (Anda mungkin harus membaca struktur struct tugas di <linux / sched.h> untuk mendapatkan nama-nama bidang ini.) Tulis kode ini di titik masuk modul sehingga isinya akan muncul di buffer log kernel, yang dapat dilihat menggunakan perintah dmesg. Untuk memverifikasi bahwa kode Anda berfungsi dengan benar, bandingkan isi buffer log kernel dengan output dari perintah berikut, yang mencantumkan semua tugas dalam sistem:

```
ps -el
```

Kedua nilai harus sangat mirip. Karena tugas bersifat dinamis, ada kemungkinan bahwa beberapa tugas dapat muncul dalam satu daftar tetapi tidak yang lain.

Bagian II — Iterasi Tugas-Tugas dengan Pohon Pencarian Kedalaman-Pertama (Depth-First Search Tree)

Bagian kedua dari proyek ini melibatkan iterasi atas semua tugas dalam sistem menggunakan Depth-First Search Tree (DFS). (Sebagai contoh: iterasi DFS dari proses pada Gambar 3.8 adalah 1, 8415, 8416, 9298, 9204, 2, 6, 200, 3028, 3610, 4005.) Linux mempertahankan pohon prosesnya sebagai serangkaian daftar. Memeriksa struct tugas di <linux / sched.h>, kita melihat dua objek kepala daftar struct:

children

and

sibling

Objek-objek ini adalah penunjuk ke daftar child tugas itu, serta siblingnya. Linux juga mempertahankan referensi ke tugas init (tugas struct struct tugas init). Dengan menggunakan informasi ini serta operasi makro pada daftar, kita dapat mengulang ulang child dari init sebagai berikut:

```
struct task struct *task; struct list head *list;

list for each(list, &init task->children) {
    task = list entry(list, struct task struct, sibling);
                    /* task points to the next child in the list */
}
```

Daftar untuk each () makro dilewatkan dua parameter, kedua tipe struct struct head:

- Penunjuk ke head daftar yang akan dilalui
- Penunjuk ke simpul head dari daftar yang akan dilalui

Pada setiap iterasi daftar untuk masing-masing (), parameter pertama diatur ke struktur daftar child berikutnya. Kami kemudian menggunakan nilai ini untuk mendapatkan setiap struktur dalam daftar menggunakan daftar entry() makro.

Bagian II Penugasan (assignment)

Dimulai dari tugas init, rancang modul kernel yang mengulang tugas keseluruhan dalam sistem menggunakan pohon DFS. Sama seperti pada bagian pertama dari proyek ini, tampilkan nama, status, dan pid dari setiap tugas. Lakukan iterasi ini di modul entri kernel sehingga outputnya muncul di buffer log kernel.

Jika Anda menampilkan semua tugas dalam sistem, Anda dapat melihat lebih banyak tugas daripada yang muncul dengan perintah ps -ael. Ini karena beberapa rangkaian muncul

sebagai child tetapi tidak muncul sebagai proses biasa. Oleh karena itu, untuk memeriksa output dari pohon DFS, gunakan perintah

ps-lF

Perintah ini mencantumkan semua tugas — termasuk rangkaian — dalam sistem. Untuk memverifikasi bahwa Anda memang telah melakukan iterasi DFS yang tepat, Anda harus memeriksa hubungan di antara berbagai tugas keluaran dengan perintah ps.

PERTANYAAN DAN JAWABAN

Contoh Pertanyaan Seputar Process Management

1. Jelaskan Konsep Dasar dan Definisi Proses!

   Jawab:
   Konsep Dasar dan Definisi Proses:

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

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

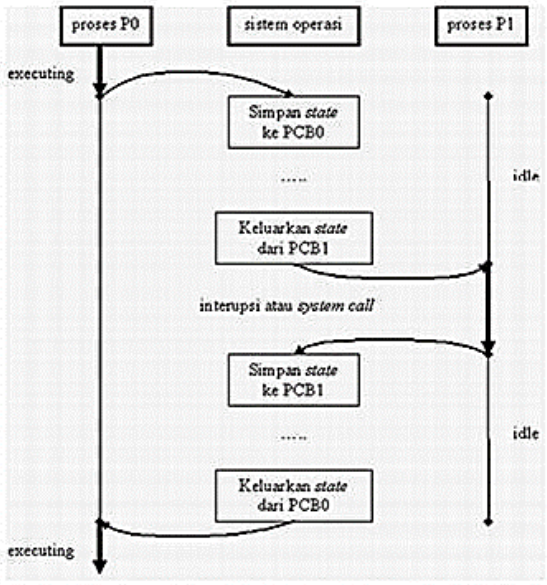
   Walau dua proses dapat dihubungkan dengan program yang sama, program tersebut dianggap dua urutan eksekusi yang berbeda. Sebagai contoh, beberapa pengguna dapat menjalankan copy yang berbeda pada mail program, atau pengguna yang sama dapat meminta banyak copy dari program editor. Tiap-tiap proses ini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

2. Apa yang dimaksud Process Control Block? Jelaskan!

Jawab:

Process Control Block

Gambar 2.
Proses Control Block



Gambar 3. CPU Register

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

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

3. Jelaskan dan sebutkan Kelebihan dari Process Scheduling

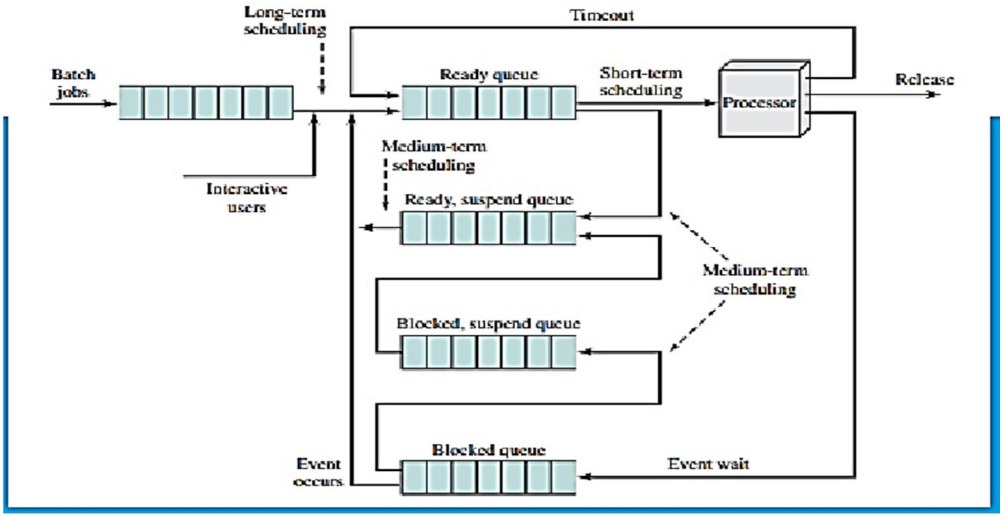
Jawab:

Scheduling atau Penjadwalan adalah metode pengaturan dalam penggunaan waktu prosesor dari sejumlah proses yang akan membutuhkan resorce atau yang saling berkompetisi merebutkan suatu resource.
Kelebihan Scheduling:

- Setiap proses yang meminta sesuatu dapat dilayani secara adil *
- Penggunaan waktu prosesor yang efisien
- Meminimalkan overhead, agar tidak terjadi starvation
- Dapat memaksimalkan throughput
- Response time dapat terpenuhi

4. Sebutkan Jenis – Jenis Process Scheduling dan jelaskan perbedaannya!

Jawab:



Jenis – Jenis Process Shceduling:

- Long-Term Scheduling / Penjadwalan jangka panjang
- Medium-term Scheduling / Penjadwalan Jangka Menengah
- Short-term Scheduling / Penjadwalan Jangka Pendek

Perbedaan Short term, Medium term, Long term:

- Short term = Untuk memaksimumkan kinerja sistem yang memilih kriteria kinerja yang di harapkan. Dan dijalankan pada waktu ada pengalihan proses untuk memilih proses yang akan diteruskan.
- Medium term = digunakan untuk menangani proses swapping. Dan juga mengendalikan suspended ke ready.

- Long term = yang bekerja pada antrian batch dan memilih antrian berikutnya untuk melakukan pengeksekusian oleh sistem. Batch merupakan proses dengan penggunaan sumber daya yang intensif.

5. Jelaskan Pengertian Interprocess Communication! Sebutkan dan Jelaskan cara-caranya!

Jawab:

IPC (Inter-Process Communication) adalah komunikasi antar proses untuk mengirim data dari satu proses ke proses yang lain, baik antar proses dalam satu komputer maupun proses-proses dalam komputer yang berbeda. IPC dapat dilakukan dengan berbagai cara yaitu Shared memory, Pipe, Messages passing, dan sebagainya.

Berikut penjelasan mengenai cara-cara tersebut:

1. Shared memory

Sistem Berbagi Memori atau yang disebut juga sebagai Shared Memory System merupakan salah satu cara komunikasi antar proses dengan cara mengalokasikan suatu alamat memori untuk dipakai berkomunikasi antar proses. Alamat dan besar alokasi memori yang digunakan biasanya ditentukan oleh pembuat program. Pada metode ini, sistem akan mengatur proses mana yang akan memakai memori pada waktu tertentu sehingga pekerjaan dapat dilakukan secara efektif.

2. Pipe

Pipe merupakan komunikasi sequensial antar proses yang saling terelasi, namun pipe memiliki kelemahan yaitu hanya bisa digunakan untuk komunikasi antar proses yang saling berhubungan, dan komunikasinya yang dilakukan adalah secara sequensial. Urutan informasi yang ada dalam sebuah pipe ada yang mirip dengan antrian queue. Jika komunikasi yang diinginkan adalah komunikasi dua arah maka kita harus membuat dua pipe, karena sebuah pipe hanya bisa digunakan untuk komunikasi satu arah saja.

3. Messages passing

Sistem berkirim pesan adalah proses komunikasi antar bagian sistem untuk membagi variabel yang dibutuhkan. Proses ini menyediakan dua operasi yaitu mengirim pesan dan menerima pesan. Ketika dua bagian sistem ingin berkomunikasi satu sama lain, yang harus dilakukan pertama kali adalah membuat sebuah link komunikasi antara keduanya. Setelah itu, kedua bagian itu dapat saling bertukar pesan melalui link komunikasi tersebut. Sistem berkirim pesan sangat penting dalam sistem operasi. Karena dapat diimplementasikan dalam banyak hal seperti pembagian memori, pembagian bus, dan melaksanakan proses yang membutuhkan pengerjaan bersama antara beberapa bagian sistem operasi.

Terdapat dua macam cara berkomunikasi antar proses, yaitu:

- Komunikasi langsung
- Komunikasi tidak langsung

6. Jelaskan apa itu API (Application Programming Interface) dan sebutkan keuntungan memprogram dengan menggunakan API!

Jawab:

API adalah sebuah gabungan perintah dan urutan yang bisa digunakan pengembang software untuk mendesain games atau aplikasi. API menyediakan bahan yang biasa diolah seperti perintah dasar, contohnya membuat windows atau tombol. Selain itu, API juga biasa digunakan untuk perintah lanjutan, misalnya mengaplikasikan bentuk timbul dalam sebuah poligon.

Keuntungan memprogram dengan menggunakan API adalah:

- Portabilitas. Programmer yang menggunakan API dapat menjalankan programnya dalam sistem operasi mana saja asalkan sudah ter- install API tersebut. Sedangkan system call berbeda antar sistem operasi, dengan catatan dalam implementasinya mungkin saja berbeda.

- Lebih Mudah Dimengerti. API menggunakan bahasa yang lebih terstruktur dan mudah dimengerti daripada bahasa system call. Hal ini sangat penting dalam hal editing dan pengembangan.

Ada tiga jenis Bahasa Pemrograman Java Application Programming Interface (API):

- inti resmi Java API, yang terdapat dalam JDK atau JRE, dari salah satu edisi dari Java Platform. Tiga edisi dari Java Platform adalah Java ME (Micro edition), Java SE (Standard edition), dan Java EE (Enterprise edition).

- Resmi opsional API yang dapat didownload secara terpisah. Spesifikasi API ini didefinisikan sesuai dengan Spesifikasi Jawa Request (JSR), dan kadang-kadang beberapa API ini kemudian dimasukkan dalam API inti dari platform (contoh yang paling terkenal dari jenis ini adalah swing).

- API tidak resmi, yang dikembangkan oleh pihak ketiga, tetapi tidak berkaitan dengan JSRs apapun.

7. Apa yang kalian ketahui tentang Multicast Communication?

Jawab:

multicast merupakan transmisi yang dimaksudkan untuk banyak tujuan, tetapi tidak harus semua host. Oleh karena itu, multicast dikenal sebagai metode tranmisi one to many (satu kebanyak). Multicast digunakan dalam kasus-kasus tertentu, misalnya ketika sekelompok computer perlu menerima transmisi tertentu.

Alamat IP Multicast (Multicast IP Address) adalah alamat yang digunakan untuk menyampaikan satu paket kepada banyak penerima. Dalam sebuah intranet yang memiliki alamat multicast IPv4, sebuah paket yang ditujukan ke sebuah alamat multicast akan diteruskan oleh router ke subjaringan di mana terdapat host-host yang sedang berada dalam kondisi “listening” terhadap lalu lintas jaringan yang dikirimkan ke alamat multicast tersebut. Dengan cara ini, alamat multicast pun menjadi cara yang efisien untuk mengirimkan paket data dari satu sumber ke beberapa tujuan untuk beberapa jenis komunikasi. Alamat multicast didefinisikan dalam RFC 1112.

Alamat-alamat multicast IPv4 didefinisikan dalam ruang alamat kelas D, yakni 224.0.0.0/4, yang berkisar dari 224.0.0.0 hingga 239.255.255.255. Prefiks alamat 224.0.0.0/24 (dari alamat 224.0.0.0 hingga 224.0.0.255) tidak dapat digunakan karena dicadangkan untuk digunakan oleh lalu lintas multicast dalam subnet lokal.

Protokol-protokol tertentu menggunakan range alamat khusus untuk multicast. Sebagai contoh, alamat ip dalam kelas D telah direservasi untuk keperluan multicast. Jika semua host perlu menerima data video, mereka akan menggunakan alamat ip multicast yang sama. Ketika mereka menerima paket yang ditujukan ke alamat tersebut, mereka akan memprosesnya. Ingatlah bahwa system masih tetapi memiliki alamt ip mereka sendiri-mereka juga mendengarkan alamat multicast mereka.

Salah satu contohnya adalah streaming audio atau video. Misalkan banyak computer ingin menerima transmisi video pada waktu yang bersamaan. Jika data tersebut dikirimkan ke setiap computer secara individu, maka diperlukan beberapa aliran data. Jika data tersebut dikirimkan sebagai broadcast, maka tidak perlu lagi proses untuk semua system. Dengan multicast data tersebut hanya dikirim sekali, tetapi diterima oleh banyak system.

Bibliographical Notes

Process creation, management, and IPC in UNIX and Windows systems, respectively, are discussed in [Robbins and Robbins (2003)] and [Russinovich and Solomon (2009)]. [Love (2010)] covers support for processes in the Linux kernel, and [Hart (2005)] covers Windows systems programming in detail. Coverage of the multiprocess model used in Google's Chrome can be found at http://blog.chromium.org/2008/09/multi-process-architecture.html.

Message passing for multicore systems is discussed in [Holland and Seltzer (2011)]. [Baumann et al. (2009)] describe performance issues in sharedmemory and message-passing systems. [Vahalia (1996)] describes interprocess communication in the Mach system.

The implementation of RPCs is discussed by [Birrell and Nelson (1984)]. [Staunstrup (1982)] discusses procedure calls versus message-passing communication. [Harold (2005)] provides coverage of socket programming in
Java.
[Hart (2005)] and [Robbins and Robbins (2003)] cover pipes in Windows and UNIX systems, respectively.

Bibliography

[Baumann et al. (2009)] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, P. Simon, T. Roscoe,A. Schupbach,and A. Singhania,¨ "Themultikernel: a new OS architecture for scalable multicore systems" (2009), pages 29–44.

[Birrell and Nelson (1984)] A. D. Birrell and B. J. Nelson, “Implementing Remote Procedure Calls”, *ACM Transactions on Computer Systems*, Volume 2, Number 1 (1984), pages 39–59.

[Harold (2005)] E. R. Harold, *Java Network Programming,* Third Edition, O’Reilly & Associates (2005).

[Hart (2005)] J. M. Hart, *Windows System Programming,* Third Edition, AddisonWesley (2005).

[Holland and Seltzer (2011)] D. Holland and M. Seltzer, “Multicore OSes: looking forward from 1991, er, 2011”, *Proceedings of the 13th USENIX conference on Hot topics in operating systems* (2011), pages 33–33.

[Love (2010)] R. Love, *Linux Kernel Development,* Third Edition, Developer’s Library (2010).

[Robbins and Robbins (2003)] K. Robbins and S. Robbins, *Unix Systems Programming: Communication, Concurrency and Threads,* Second Edition, Prentice Hall (2003).

[Russinovich and Solomon (2009)] M. E. Russinovich and D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista,* Fifth Edition, Microsoft Press (2009).

[Staunstrup (1982)] J. Staunstrup, “Message Passing Communication Versus Procedure Call Communication”, *Software—Practice and Experience*, Volume 12, Number 3 (1982), pages 223–234.

[Vahalia (1996)]U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall (1996).