Rangkaian / Utas

Model proses yang diperkenalkan pada Bab 3 mengasumsikan bahwa suatu proses adalah sebuah program eksekusi dengan satu rangkaian kontrol. Hampir semua sistem operasi modern, bagaimanapun, menyediakan fitur – fitur yang memungkinkan suatu proses untuk memuat beberapa rangkaian kontrol. Dalam bab ini, kami memperkenalkan banyak konsep terkait dengan sistem komputer multithread, termasuk diskusi tentang API untuk Pthread, Windows, dan Java thread. Kami melihat sejumlah masalah yang terkait dengan pemrograman multithread dan pengaruhnya pada desain sistem operasi. Akhirnya, kami mengeksplorasi bagaimana Windows dan Linux sistem operasi mendukung rangkaian pada tingkat kernel.

TUJUAN BAB

- Untuk memperkenalkan gagasan rangkaian — unit dasar penggunaan CPU yang membentuk dasar sistem komputer multithreaded
- Untuk mediskusikan APIs untuk Pthreads, Windows, and library Java thread
- Untuk mengesklporasi beberapa strategi yang menyediakan implisit threading
- Untuk memeriksa isu – isu yang berkaitan dengan multithreaded programming
- Untuk mengcover sistem operasi bagi threads dalam Windows dan Linux

4.1 IKHTISAR

Sebuah rangkaian adalah unit dasar atau awal dalam penggunaan CPU. Dia membandingkan ID Thread, penghitung program, register set, dan tumpukan. Dia membagikan dengan rangkaian yang memiliki proses yang sama yaitu kode bagian, data section, dan sumber daya sistem operasi lainnya, seperti open file dan signals. Proses tradisional (atau kelas berat) memiliki satu rangkaian kontrol. Jika sebuah proses memiliki banyak Threads control, dia dapat melakukan tugas lebih dari satu waktu. Gambar 4.1 megilustrasikan perbedaan antara traditional single-threaded process dan multithreaded process.

4.1.1 Motivasi

Hampir semua software aplikasi / perangkat lunak yang aktif/berjalan/ada pada komputer modern adalah multithreaded. Sebuah aplikasi biasanya dilaksanakan sebagai proses terpisah dengan beberapa

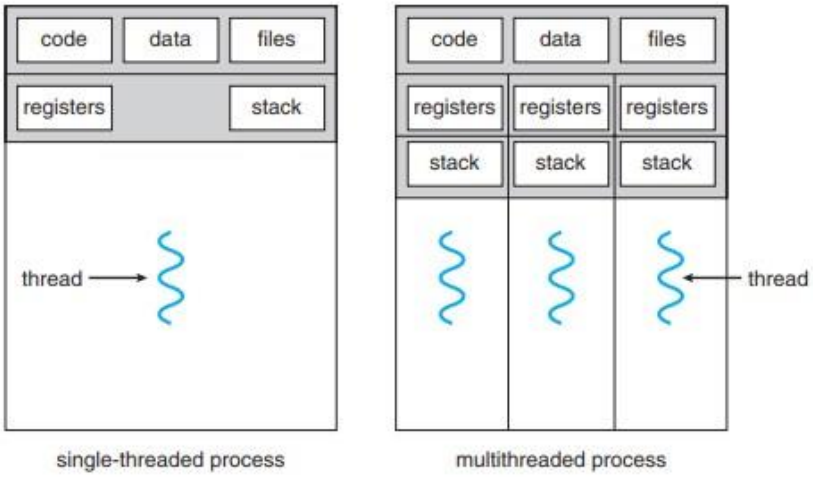


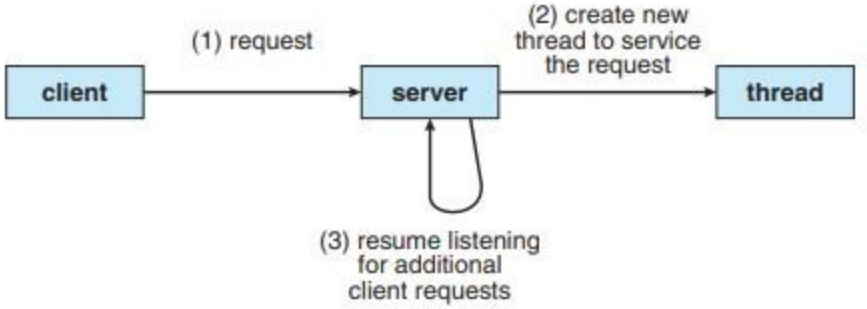
Figure 4.1 Single-threaded and multithreaded processes.

threads kontrol. Sebuah browser web bisa jadi mempunyai satu gambar tampilan rangkaian atau teks sementara rangkaian lain mengambil data dari jaringan, misalnya. Sebuah pengolah kata mungkin memiliki rangkaian untuk menampilkan grafik, rangkaian lain untuk menanggapi penekanan tombol dari pengguna, dan rangkaian ketiga untuk melakukan pemeriksaan ejaan dan tatabahasa di latar belakang. Aplikasi juga bisa dirancang untuk memanfaatkan kemampuan pemrosesan pada sistem multicore. Seperti itu aplikasi dapat melakukan beberapa tugas intensif CPU secara paralel di seluruh beberapa core computer

Dalam situasi tertentu, satu aplikasi mungkin diperlukan untuk melakukan beberapa tugas serupa. Misalnya, server web menerima permintaan klien untuk halaman web, gambar, suara, dan sebagainya. Server web yang sibuk mungkin memiliki beberapa (mungkin ribuan) klien secara bersamaan mengaksesnya. Jika server web berjalan sebagai proses single-threaded tradisional, dia akan dapat melayani hanya satu klien pada suatu waktu, dan klien mungkin harus menunggu waktu yang sangat lama untuk permintaannya untuk dilayani.

Salah satu solusinya adalah dengan menjalankan server sebagai satu proses yang dapat diterima permintaan. Ketika server menerima permintaan, dia menciptakan proses yang terpisah untuk melayani permintaan itu. Faktanya, metode penciptaan proses ini umum digunakan sebelum rangkaian menjadi populer. Proses pembuatan memakan waktu dan intensif sumber daya, namun. Jika proses baru akan melakukan tugas yang sama seperti proses yang ada, mengapa harus menanggung semua itu? Umumnya lebih efisien menggunakan satu proses yang berisi beberapa rangkaian. Jika proses webserver multithreaded, server akan membuat rangkaian terpisah yang mendengarkan klien permintaan. Ketika permintaan dibuat, daripada membuat proses lain, server itu membuat rangkaian baru untuk melayani permintaan dan melanjutkan mendengarkan permintaan tambahan. Ini diilustrasikan pada Gambar 4.2.

Thread/rangkaian juga memainkan peran penting dalam sistem remote procedure call (RPC). Merujuk dari Bab 3 bahwa RPC memungkinkan komunikasi interprocess dengan menyediakan mekanisme komunikasi mirip dengan fungsi biasa atau panggilan prosedur. Biasanya, server RPC adalah multithreaded. Ketika server mendapatkan pesan, dia melayani pesan menggunakan rangaian yang terpisah. Ini memungkinkan server untuk melayani beberapa permintaan bersamaan



Akhirnya, kebanyakan kernel sistem operasi sekarang multithread. Beberapa thread beroperasi di kernel, dan setiap thread melakukan tugas tertentu, seperti itu sebagai mengelola perangkat, mengelola memori, atau menangani gangguan. Sebagai contoh, Solaris memiliki serangkaian rangkaian di kernel khusus untuk penanganan interupsi; Linux menggunakan rangkaian kernel untuk mengatur jumlah memori bebas dalam sistem.

4.1.2 Keuntungan

Manfaat dari pemrograman multithread dapat dipecah menjadi empat kategori utama:

1. **Ketanggapan**. Multithreading aplikasi interaktif memungkinkan sebuah program untuk terus berjalan bahkan jika sebagian diblokir atau tidak melakukan operasi yang panjang, sehingga meningkatkan respons terhadap pengguna. Kualitas ini sangat berguna dalam merancang antarmuka pengguna. Untuk misalnya, pertimbangkan apa yang terjadi ketika pengguna mengklik tombol yang dihasilkan dalam kinerja operasi yang memakan waktu. Sebuah single-threaded aplikasi akan menjadi tidak responsif kepada pengguna sampai operasi dilakukan lengkap. Sebaliknya, jika operasi yang memakan waktu dilakukan dirangkaian yang terpisah, aplikasi tetap responsif terhadap pengguna.

2. **Berbagi sumber daya**. Proses hanya dapat membagi sumber daya melalui teknik seperti memori bersama dan pengiriman pesan. Teknik semacam itu harus secara eksplisit diatur oleh programmer. Namun, rangkaian berbagi memori dan sumber daya dari proses yang menjadi milik mereka secara default. Manfaat dari berbagi kode dan data adalah memungkinkan aplikasi memiliki beberapa rangkaian aktivitas yang berbeda dalam ruang alamat yang sama.

3. **Ekonomi**. Mengalokasikan memori dan sumber daya untuk pembuatan proses sangat mahal. Karena rangkaian berbagi sumber daya dari proses yang menjadi milik mereka, lebih ekonomis untuk membuat dan beralih konteks rangkaian. Secara empiris mengukur perbedaan dalam overhead bisa sulit, tetapi secara umum secara signifikan lebih memakan waktu untuk membuat dan mengelola proses daripada rangkaian. Di Solaris, sebagai contoh, membuat proses adalah sekitar tiga puluh kali

Figure 4.3 Concurrent execution on a single-core system.

lebih lambat daripada membuat rangkaian, dan pergantian konteks sekitar lima kali lebih lambat.

4. **Skalabilitas**. Manfaat multithreading dapat lebih besar dalam arsitektur multiprosesor, di mana rangkaian dapat berjalan secara paralel pada core pemrosesan yang berbeda. Proses single-threaded hanya dapat berjalan satu prosesor, tidak peduli berapa banyak yang tersedia. Kami menjelajahi masalah ini lebih lanjut di bagian berikut.

4.2 Pemrograman Multicore

Sebelumnya dalam sejarah desain komputer, sebagai tanggapan terhadap kebutuhan akan lebih banyak kinerja komputasi, sistem satu-CPU berevolusi menjadi sistem multi-CPU. Tren yang lebih baru dan serupa dalam desain sistem adalah menempatkan banyak komputasi core pada satu chip. Setiap inti muncul sebagai prosesor terpisah ke sistem operasi (Bagian 1.3.2). Apakah inti muncul di chip CPU atau dalam chip CPU, kami menyebutnya sistem multicore atau sistem multiprosesor. Pemrograman multithread menyediakan mekanisme untuk penggunaan yang lebih efisien dari beberapa core komputasi dan peningkatan konkurensi. Pertimbangkan sebuah aplikasi dengan empat rangkaian. Pada sistem dengan inti komputasi tunggal, konkurensi hanya berarti bahwa pelaksanaan rangkaian akan disisipkan dari waktu ke waktu (Gambar 4.3), karena inti pemrosesan hanya mampu mengeksekusi satu rangkaian sekaligus. Pada sistem dengan beberapa inti, bagaimanapun, konkurensi berarti bahwa rangkaian dapat berjalan secara paralel, karena sistem dapat menetapkan rangkaian terpisah untuk masing-masing inti (Gambar 4.4).

Perhatikan perbedaan antara paralelisme dan konkurensi dalam diskusi ini. Suatu sistem sejajar jika dapat melakukan lebih dari satu tugas secara bersamaan. Sebaliknya, sistem konkuren mendukung lebih dari satu tugas dengan mengizinkan semua tugas untuk membuat kemajuan. Dengan demikian untuk memiliki konkurensi tanpa paralelisme. Sebelum munculnya arsitektur SMP dan multicore, kebanyakan computer sistem hanya memiliki prosesor tunggal. Penjadwal CPU dirancang untuk memberikan ilusi paralelisme dengan cepat beralih antar proses di sistem, sehingga memungkinkan setiap proses untuk membuat kemajuan.

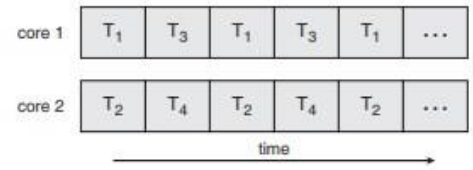


Figure 4.4 Parallel execution on a multicore system.

Hukum Amdahl

Hukum Amdahl adalah formula yang mengidentifikasi perolehan kinerja potensial dari menambahkan core komputasi tambahan ke aplikasi yang memiliki serial (tidak paralel) dan komponen paralel. Jika S adalah bagian dari aplikasi yang harus dilakukan secara serial pada suatu sistem dengan N processing core, rumus muncul sebagai berikut:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Sebagai contoh, asumsikan kita memiliki aplikasi yang 75 persen paralel dan 25 persen serial. Jika kita menjalankan aplikasi ini pada sistem dengan dua pemrosesan core, kita bisa mendapatkan kecepatan 1,6 kali. Jika kita menambahkan dua inti tambahan (untuk total empat), kecepatannya 2,28 kali.

Satu fakta menarik tentang Hukum Amdahl adalah saat N mendekati tak terhingga, kecepatan menyatu menjadi 1 / S. Misalnya, jika 40 persen dari aplikasi dilakukan secara serial, kecepatan maksimum adalah 2,5 kali, terlepas dari jumlah inti pemrosesan yang kami tambahkan. Ini adalah prinsip fundamental di belakang Hukum Amdahl: bagian serial aplikasi dapat memiliki efek yang tidak proporsional pada kinerja yang kita peroleh dengan menambahkan tambahan core komputasi.

Beberapa berpendapat bahwa Hukum Amdahl tidak memperhitungkan perangkat keras peningkatan kinerja yang digunakan dalam desain multicore sistem kontemporer. Argumen semacam itu menyarankan Hukum Amdahl dapat berhenti berlaku karena jumlah inti pemrosesan terus meningkat pada sistem computer modern.

Proses seperti itu berjalan bersamaan, tetapi tidak secara paralel. Karena sistem telah berkembang dari puluhan rangkaian ke ribuan rangkaian, CPU desainer telah meningkatkan kinerja sistem dengan menambahkan perangkat keras untuk meningkatkan kinerja rangkaian. CPU Intel modern sering mendukung dua rangkaian perinti, sedangkan Oracle T4 CPU mendukung delapan rangkaian per inti. Dukungan ini berarti beberapa rangkaian dapat dimuat ke inti untuk beralih cepat. Komputer multicore tidak diragukan lagi akan terus meningkat dalam jumlah inti dan dukungan untaian perangkat keras.

4.2.1 Tantangan Pemrograman

Tren ke arah sistem multicore terus menempatkan tekanan pada sistem desainer dan programer aplikasi untuk memanfaatkan lebih baik dari beberapa core komputasi. Desainer sistem operasi harus menulis penjadwalan algoritme yang menggunakan

beberapa inti pemrosesan untuk memungkinkan eksekusi parallel yang ditunjukkan pada Gambar 4.4. Untuk programmer aplikasi, tantangannya adalah untuk memodifikasi program yang ada serta merancang program baru yang multithread. Secara umum, terdapat lima area yang mempunyai tantangan dalam pemrograman untuk sistem multicore:

1. **Mengidentifikasi tugas**. Ini melibatkan pemeriksaan aplikasi untuk menemukan area yang dapat dibagi menjadi tugas yang terpisah dan bersamaan. Idealnya, tugasnya independen satu sama lain dan dengan demikian dapat berjalan secara paralel pada core individu.

2. **Keseimbangan**. Saat mengidentifikasi tugas yang dapat berjalan secara paralel, programmer juga harus memastikan bahwa semua tugas melakukan pekerjaan yang sama dengan nilai yang sama. Di beberapa contoh, tugas tertentu mungkin tidak berkontribusi sebanyak nilai proses keseluruhan sebagai tugas lain. Menggunakan inti eksekusi terpisah untuk menjalankannya tugas mungkin tidak sebanding dengan biayanya.

3. **Pemisahan data**. Sama seperti aplikasi dibagi menjadi tugas yang terpisah, yang data yang diakses dan dimanipulasi oleh tugas harus dibagi untuk dijalankan inti yang terpisah.

4. **Ketergantungan data**. Data yang diakses oleh tugas harus diperiksa ketergantungan antara dua atau lebih tugas. Ketika satu tugas bergantung pada data dari yang lain, programmer harus memastikan bahwa eksekusi dari tugas disinkronkan untuk mengakomodasi ketergantungan data. Kami memeriksa strategi seperti itu di Bab 5.

5. **Pengujian dan debugging**. Ketika sebuah program berjalan secara parallel beberapa core, banyak jalur eksekusi yang berbeda dimungkinkan. Pengujian dan debugging program konkuren semacam itu secara inheren lebih sulit daripada menguji dan debugging aplikasi single-threaded.

Karena tantangan ini, banyak pengembang perangkat lunak berpendapat bahwa munculnya sistem multicore akan membutuhkan pendekatan yang sama sekali baru untuk merancang perangkat lunak sistem di masa depan. (Demikian pula, banyak pendidik ilmu komputer percaya bahwa pengembangan perangkat lunak harus diajarkan dengan penekanan yang meningkat pada parallel pemrograman.)

4.2.2 Jenis Paralelisme

Secara umum, ada dua jenis paralelisme: paralelisme data dan tugas paralelisme. **Paralelisasi data** berfokus pada mendistribusikan subset dari data yang sama di beberapa core komputasi dan melakukan operasi yang sama pada masing-masing inti. Pertimbangkan, misalnya, menjumlahkan isi array ukuran N. Pada sistem single-core, satu thread hanya akan menjumlahkan elemen [0]. . . [N - 1]. Pada sistem dual-core, bagaimanapun, thread A, berjalan pada core 0, dapat menjumlahkan elemen [0]. . . [N / 2 - 1] ketika rangkaian B, berjalan di inti 1, dapat menjumlahkan elemen [N / 2]. . . [N - 1]. Kedua rangkaian akan berjalan secara parallel pada komputasi core terpisah.

Tugas paralelisme melibatkan distribusi bukan data tetapi tugas (thread) beberapa core komputasi. Setiap rangkaian melakukan operasi yang unik. Rangkaian yang berbeda mungkin beroperasi pada data yang sama, atau mereka mungkin beroperasi pada data

yang berbeda. Pertimbangkan lagi contoh kita di atas. Berbeda dengan situasi itu, contoh tugas paralelisme mungkin melibatkan dua rangkaian, masing – masing melakukan operasi statistik yang unik pada larik elemen. Rangkaiannya lagi beroperasi secara paralel pada inti komputasi terpisah, tetapi masing – masing melakukan operasi yang unik.

Pada dasarnya, kemudian, paralelisme data melibatkan distribusi data di beberapa inti dan tugas paralelisme pada distribusi tugas di seluruh beberapa core. Namun, dalam prakteknya, beberapa aplikasi secara ketat mengikuti salah satu data atau tugas paralelisme. Dalam banyak kasus, aplikasi menggunakan gabungan strategi keduanya.

4.3 Model Multithreading

Diskusi kami sejauh ini telah memperlakukan rangkaian dalam pengertian umum. Namun, dukungan
rangkaian rangkaian dapat disediakan baik di tingkat pengguna, untuk rangkaian pengguna, atau oleh kernel, untuk rangkaian kernel. Untaian pengguna didukung di atas kernel dan dikelola tanpa dukungan kernel, sedangkan rangkaian kernel didukung dan dikelola langsung oleh sistem operasi. Hampir semua kontemporer sistem operasi — termasuk Windows, Linux, Mac OS X, dan Solaris—mendukung rangkaian kernel.

Pada akhirnya, hubungan harus ada di antara rangkaian pengguna dan kernel rangkaian. Pada bagian ini, kami melihat tiga cara umum untuk menetapkan semacam hubungan: model banyak-ke-satu, model satu-ke-satu, dan banyak-ke-banyak model.

4.3.1 Model Banyak-ke-Satu

Model banyak-ke-satu (Gambar 4.5) memetakan banyak thread tingkat pengguna menjadi satu rangkaian kernel. Manajemen thread dilakukan oleh perpustakaan thread di ruang pengguna, jadi ini efisien (kami membahas pustaka rangkaian di Bagian 4.4). Namun, keseluruhannya proses akan memblokir jika rangkaian membuat panggilan sistem pemblokiran. Juga, karena hanya satu rangkaian dapat mengakses kernel pada satu waktu, beberapa rangkaian tidak dapat dijalankan paralel pada sistem multicore. Untaian hijau — pustaka untaian tersedia untuk Sistem Solaris dan diadopsi dalam versi awal Java — menggunakan banyak-ke-satu model. Namun, sangat sedikit sistem yang terus menggunakan model karena itu ketidakmampuan untuk mengambil keuntungan dari beberapa inti pemrosesan.

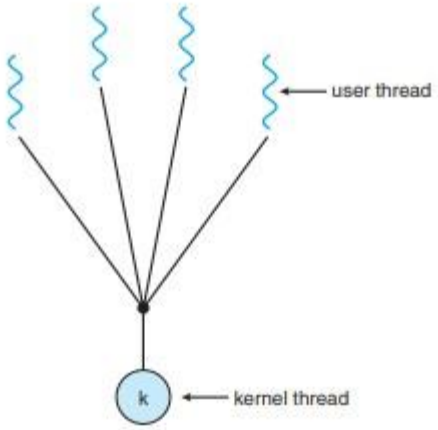


Figure 4.5 Many-to-one model.

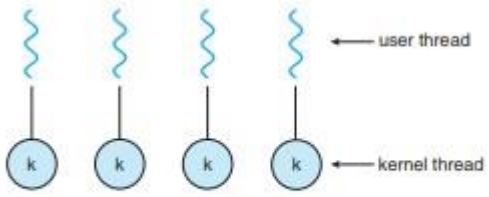


Figure 4.6 One-to-one model.

4.3.2 Model Satu-ke-Satu

Model satu-ke-satu (Gambar 4.6) memetakan setiap pengguna rangkaian ke rangkaian kernel. Memberikan lebih banyak konkurensi daripada model banyak-ke-satu dengan mengizinkan yang lain rangkaian untuk berjalan ketika rangkaian membuat panggilan sistem pemblokiran. Ini juga memungkinkan beberapa rangkaian untuk berjalan secara paralel pada multiprocessors. Satu-satunya kelemahan model ini adalah bahwa membuat rangkaian pengguna membutuhkan pembuatan yang sesuai rangkaian kernel. Karena biaya overhead membuat rangkaian kernel dapat membebani kinerja suatu aplikasi, sebagian besar implementasi dari model ini membatasi jumlah rangkaian yang didukung oleh sistem. Linux, bersama dengan keluarga Sistem operasi Windows, terapkan model satu-ke-satu.

4.3.3 Banyak-ke-Banyak Model

Model banyak-ke-banyak (Gambar 4.7) multiplexes banyak thread tingkat pengguna untuk jumlah yang lebih kecil atau sama dengan rangkaian kernel. Jumlah rangkaian kernel dapat spesifik untuk aplikasi tertentu atau mesin tertentu (sebuah aplikasi dapat mengalokasikan lebih banyak rangkaian kernel pada multiprosesor dari pada satu prosesor).

Mari kita pertimbangkan efek dari desain ini pada konkurensi. Padahal banyak – ke banyak model memungkinkan pengembang untuk membuat sebanyak mungkin thread

pengguna sesuai keinginannya, itu tidak menghasilkan konkurensi yang benar, karena kernel dapat menjadwalkan satu rangkaian sekaligus. Model satu-ke-satu memungkinkan konkurensi yang lebih besar, tetapi pengembang harus berhati-hati untuk tidak membuat terlalu banyak rangkaian dalam aplikasi (dan dalam beberapa kasus mungkin terbatas pada jumlah rangkaian yang dia bias membuat). Model banyak-ke-banyak tidak menderita dari kekurangan-kekurangan ini: pengembang dapat membuat sebanyak mungkin rangkaian pengguna, dan yang sesuai rangkaian kernel dapat berjalan secara paralel pada multiprosesor. Juga, ketika sebuah thread melakukan panggilan sistem pemblokiran, kernel dapat menjadwalkan thread lain untuk eksekusi.

Satu variasi pada model banyak-ke-banyak masih mengalikan banyak tingkat pengguna rangkaian ke sejumlah rangkaian kernel yang lebih kecil atau sama tetapi juga memungkinkan thread tingkat pengguna untuk terikat ke rangkaian kernel. Variasi ini kadang-kadang disebut sebagai model dua tingkat (Gambar 4.8). Sistem operasi Solaris mendukung model dua tingkat dalam versi yang lebih tua dari Solaris 9. Namun, dimulai dengan Solaris 9, sistem ini menggunakan model satu-ke-satu.

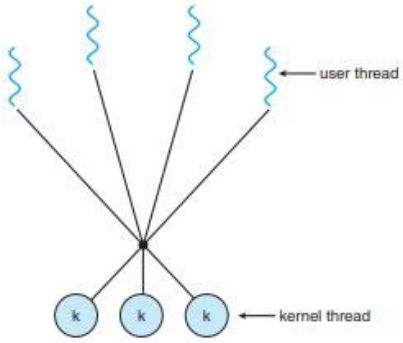


Figure 4.7 Many-to-many model.

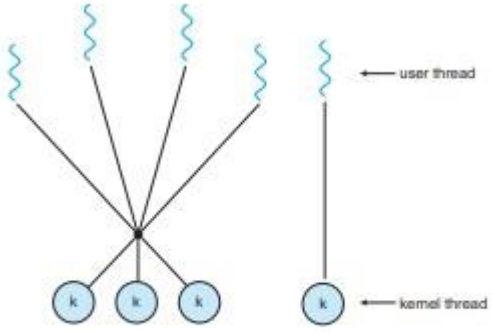


Figure 4.8 Two-level model.

4.4 Pustaka Rangkaian

Pustaka thread menyediakan programmer dengan API untuk membuat dan mengelola untaian. Ada dua cara utama menerapkan sebuah thread Perpustakaan. Pendekatan pertama adalah menyediakan perpustakaan sepenuhnya di ruang pengguna tanpa dukungan kernel. Semua kode dan struktur data untuk perpustakaan ada di ruang pengguna. Ini berarti bahwa

memohon fungsi dalam hasil perpustakaan dalam fungsi local panggilan di ruang pengguna dan bukan panggilan sistem.

Pendekatan kedua adalah untuk mengimplementasikan perpustakaan tingkat kernel yang didukung langsung oleh sistem operasi. Dalam hal ini, kode dan struktur data untuk perpustakaan ada di ruang kernel. Meminjam fungsi di API untuk pustaka biasanya menghasilkan panggilan sistem ke kernel.

Tiga perpustakaan thread utama sedang digunakan hari ini: POSIX Pthreads, Windows, dan Java. Pthreads, ekstensi rangkaian dari standar POSIX, dapat disediakan baik sebagai tingkat pengguna atau pustaka tingkat kernel. Perpustakaan thread Windows adalah pustaka tingkat kernel yang tersedia di sistem Windows. Java thread API memungkinkan rangkaian dibuat dan dikelola langsung di program Java. Namun, karena dalam banyak kasus, JVM berjalan di atas sistem operasi host, Java thread API umumnya diimplementasikan menggunakan pustaka thread yang tersedia pada sistem host. Ini berarti bahwa pada sistem Windows, Java thread adalah biasanya diimplementasikan menggunakan Windows API; Sistem UNIX dan Linux sering gunakan Pthreads.

Untuk POSIX dan Windows threading, data apa pun yang dinyatakan secara global — yang dinyatakan di luar fungsi apa pun — dibagikan di antara semua untaian yang termasuk dalam proses yang sama. Karena Java tidak memiliki gagasan tentang data global, akses ke data bersama harus secara eksplisit diatur di antara untaian. Data yang dinyatakan lokal ke suatu fungsi biasanya disimpan di stack. Karena setiap utas memiliki tumpukannya sendiri, setiap utas memiliki salinan data lokal sendiri.

Di sisa bagian ini, kami mendeskripsikan pembuatan thread dasar menggunakan tiga pustaka thread ini. Sebagai contoh ilustratif, kami merancang program multithread yang melakukan penjumlahan bilangan bulat non-negatif dalam utas yang terpisah menggunakan fungsi penjumlahan yang terkenal: sum $=\sum_{i=0}^{n} \mathrm{i}$
Sebagai contoh, jika N adalah 5, fungsi ini akan mewakili penjumlahan bilangan bulat dari 0 hingga 5, yaitu 15. Masing-masing dari tiga program akan dijalankan dengan batas atas dari penjumlahan yang dimasukkan pada baris perintah. Jadi, jika pengguna memasuki 8, penjumlahan nilai integer dari 0 hingga 8 akan menjadi output.

Sebelum kita melanjutkan dengan contoh-contoh pembuatan thread kami, kami memperkenalkan dua strategi umum untuk membuat beberapa utas: asynchronous threading dan sinkron threading. Dengan threading asynchronous, setelah orang tua membuat thread anak, induknya melanjutkan kembali eksekusi, sehingga orang tua dan anak melakukan eksekusi secara bersamaan. Setiap utas berjalan sendiri-sendiri dari setiap utas lainnya, dan utas induk tidak perlu tahu ketika meruntuhkan. Karena utasnya independen, biasanya ada sedikit pembagian data antar utas. Asynchronous threading adalah strategi yang digunakan pada server multithread yang diilustrasikan pada Gambar 4.2.

Threading sinkron terjadi ketika benang induk menciptakan satu atau lebih anak dan kemudian harus menunggu semua anaknya berhenti sebelum melanjutkan — strategi yang disebut fork-join. Di sini, untaian yang dibuat oleh induk bekerja secara bersamaan, tetapi induk tidak dapat melanjutkan hingga pekerjaan ini selesai. Setelah setiap thread selesai bekerja, itu berakhir dan bergabung dengan induknya. Hanya setelah semua anak bergabung, orang tua melanjutkan eksekusi. Biasanya, pengeditan sinkron melibatkan pembagian data yang signifikan di antara untaian. Misalnya, utas induk dapat menggabungkan hasil yang dihitung oleh berbagai turunannya. Semua contoh berikut menggunakan threading sinkron.

4.4.1 Pthreads

Pthreads mengacu pada standar POSIX (IEEE 1003.1c) yang mendefinisikan API untuk pembuatan dan sinkronisasi thread. Ini adalah spesifikasi untuk perilaku ulir, bukan

implementasi. Desainer sistem operasi dapat menerapkan spesifikasi dengan cara apa pun yang mereka inginkan. Banyak sistem menerapkan spesifikasi Pthreads; kebanyakan adalah sistem tipe UNIX, termasuk Linux, Mac OS X, dan Solaris. Meskipun Windows tidak mendukung Pthreads secara asli, beberapa implementasi pihak ketiga untuk Windows tersedia.

Program C ditunjukkan pada Gambar 4.9 menunjukkan Pthreads API dasar untuk membangun program amultithread yang menghitung penjumlahan dari bilangan bulat non-negatif dalam utas yang terpisah. Dalam program Pthreads, utas yang terpisah

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

int sum; /* this data is shared by the thread(s) */ void
*runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{  pthread t tid; /* the thread identifier */ pthread attr
          t attr; /* set of thread attributes */
 if (argc != 2) {  fprintf(stderr,"usage: a.out <integer
          value>\n"); return -1;
          }
          if (atoi(argv[1]) < 0) {  fprintf(stderr,"%d must be >=
                    0\n",atoi(argv[1])); return -1;
          }
          /* get the default attributes
          */ pthread attr init(&attr); /*
          create the thread */ pthread
          create(&tid,&attr,runner,arg
          v[1]);
          /* wait for the thread to exit */
          pthread join(tid,NULL);
          printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
          int i, upper = atoi(param);
          sum = 0;
          for (i = 1; i <= upper; i++)
          sum += i; pthread
          exit(0);
}
```

Gambar 4.9 Program C multithreaded menggunakan Pthreads API.

mulai eksekusi dalam fungsi yang ditentukan. Pada Gambar 4.9, ini adalah fungsi runner (). Ketika program ini dimulai, satu utas kontrol dimulai di main (). Setelah beberapa inisialisasi, main () membuat utas kedua yang mulai mengontrol fungsi runner (). Kedua utas berbagi jumlah data global

Mari kita lihat lebih dekat pada program ini. Semua program Pthreads harus menyertakan file header pthread.h. Pernyataan itu disampaikan

```
#define NUM THREADS 10

/* an array of threads to be joined upon */
pthread t workers[NUM THREADS];

for (int i = 0; i < NUM THREADS; i++) pthread
        join(workers[i], NULL);
```

Gambar 4.10 Pthread code untuk bergabung dengan sepuluh utas.

pengidentifikasi untuk utas yang akan kita buat. Setiap utas memiliki seperangkat atribut, termasuk ukuran tumpukan dan informasi penjadwalan. Deklarasi attr ttr pthread mewakili atribut untuk thread. Kita atur atribut dalam fungsi panggil pthread attr init (& attr). Karena kami tidak secara eksplisit mengatur atribut apa pun, kami menggunakan atribut default yang disediakan. (InChapter 6, kami membahas beberapa atribut penjadwalan yang disediakan oleh Pthreads API.) Sebuah utas terpisah dibuat dengan pthread create () function call. Selain meneruskan pengenal untaian dan atribut untuk utas, kami juga meneruskan nama fungsi tempat **utas** baru akan memulai eksekusi — dalam hal ini, fungsi runner (). Terakhir, kami meneruskan parameter integer yang disediakan pada baris perintah, argv [1]

Pada titik ini, program memiliki dua utas: utas awal (atau induk) di utas utama () dan penjumlahan (atau anak) yang melakukan operasi penjumlahan dalam fungsi runner (). Program ini mengikuti strategi fork-join yang dijelaskan sebelumnya: setelah membuat simpul penjumlahan, utas induk akan menunggu untuk mengakhiri dengan memanggil fungsi pthread join (). Benang penjumlahan akan berakhir saat memanggil fungsi pthread exit (). Setelah benang penjumlahan kembali, utas induk akan menampilkan nilai jumlah data bersama.

Program contoh ini hanya membuat satu utas. Dengan meningkatnya dominasi sistem multicore, program penulisan yang berisi beberapa utas menjadi semakin umum. Metode sederhana untuk menunggu pada beberapa utas menggunakan fungsi pthread join () adalah untuk melampirkan operasi dalam loop sederhana. Misalnya, Anda dapat bergabung pada sepuluh utas menggunakan kode Pthread yang ditunjukkan pada Gambar 4.10

4.4.2 Windows Threads

Teknik untuk membuat untaian menggunakan pustaka thread Windows mirip dengan teknik Pthreads dalam beberapa cara. Kami menggambarkan Windows thread API dalam program C yang ditunjukkan pada Gambar 4.11. Perhatikan bahwa kita harus menyertakan file header windows.h ketika menggunakan API Windows.

Sama seperti pada versi Pthreads yang ditunjukkan pada Gambar 4.9, data yang dibagikan oleh utas yang terpisah — dalam hal ini, Jumlah — dideklarasikan secara global (tipe data DWORD adalah bilangan bulat 32-bit yang tidak ditandatangani). Kami juga mendefinisikan fungsi Summation () yang harus dilakukan dalam utas yang terpisah. Fungsi ini dilewatkan pointer ke void, yang didefinisikan Windows sebagai LPVOID. Benang yang melakukan fungsi ini mengatur jumlah data global ke nilai penjumlahan dari 0 ke parameter yang diteruskan ke Summation ()

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

```
        DWORD Sum; /* data is shared by the thread(s) */
        /* the thread runs in this separate function */
        DWORD WINAPI Summation(LPVOID Param)
        {
DWORD Upper = *(DWORD*)Param; for
(DWORD i = 0; i <= Upper; i++)
Sum += i; return
0;
        }
        int main(int argc, char *argv[])
        {
DWORD         ThreadId;
HANDLE ThreadHandle;
int Param;
if (argc != 2) {
fprintf(stderr,"An integer parameter is required\n"); return
        -1;
        }Param = atoi(argv[1]); if
        (Param < 0) {
fprintf(stderr,"An integer >= 0 is required\n"); return
-1;
        }
        /* create the thread */
        ThreadHandle = CreateThread(
NULL, /* default security attributes */
0, /* default stack size */
Summation, /* thread function */
&Param, /* parameter to thread function */
0, /* default creation flags */
&ThreadId); /* returns the thread identifier */ if
        (ThreadHandle != NULL) {
/* now wait for the thread to finish */
WaitForSingleObject(ThreadHandle,INFINITE);
/* close the thread handle */
CloseHandle(ThreadHandle);
printf("sum = %d\n",Sum);
}
        }
```

Gambar 4.11 Program C multithreaded menggunakan Windows API.

Thread dibuat di Windows API menggunakan fungsi CreateThread (), dan — sama seperti di Pthreads — seperangkat atribut untuk utas dilewatkan ke fungsi ini. Atributatribut ini termasuk informasi keamanan, ukuran tumpukan, dan bendera yang dapat diatur untuk menunjukkan jika utas dimulai dalam keadaan ditangguhkan. Dalam program ini, kami menggunakan nilai default untuk atribut ini. (Nilai default awalnya tidak mengatur utas ke status ditangguhkan dan menjadikannya memenuhi syarat untuk dijalankan oleh penjadwal CPU.) Setelah utas penjumlahan dibuat, induk harus menunggunya selesai sebelum mengeluarkan nilai Jumlah, karena nilai ditetapkan oleh utas penjumlahan. Ingat

bahwa program Pthread (Gambar 4.9) memiliki utas induk menunggu utas penjumlahan menggunakan pthread join () pernyataan.
Kami melakukan hal yang sama dengan ini di Windows API menggunakan fungsi WaitForSingleObject (), yang menyebabkan pembuatan thread untuk memblokir sampai benang penjumlahan telah keluar.

Dalam situasi yang memerlukan menunggu beberapa utas untuk diselesaikan, fungsi WaitForMultipleObjects () digunakan. Fungsi ini dilewatkan empat parameter:

1. Jumlah objek yang harus ditunggu
2. Penunjuk ke array objek
3. Bendera yang menunjukkan apakah semua benda telah diberi tanda
4. Durasi batas waktu (atau INFINITE)

Misalnya, jika THandles adalah larik untaian HANDLE objek dengan ukuran N, utas induk dapat menunggu semua utas anak untuk melengkapi dengan pernyataan ini:
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);

4.4.3 Java Threads

Thread adalah model dasar dari eksekusi program dalam program Java, dan bahasa Java dan API-nya menyediakan serangkaian fitur yang kaya untuk pembuatan dan pengelolaan untaian. Semua program Java terdiri dari setidaknya satu utas kontrol — bahkan program Java sederhana yang hanya terdiri dari metode main () berjalan sebagai satu utas dalam JVM. Java thread tersedia di semua sistem yang menyediakan JVM termasuk Windows, Linux, dan Mac OS X. Java thread API juga tersedia untuk aplikasi Android.

Ada dua teknik untuk membuat utas dalam program Java. Salah satu pendekatan adalah membuat kelas baru yang diturunkan dari kelas Thread dan mengesampingkan metode run () -nya. Teknik alternatif — dan yang lebih umum digunakan — adalah mendefinisikan kelas yang mengimplementasikan antarmuka Runnable. Antarmuka Runnable didefinisikan sebagai berikut:

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

Ketika kelas mengimplementasikan Runnable, itust akan menentukan metode run (). Kode yang menerapkan metode run () adalah apa yang berjalan sebagai utas yang terpisah.

Gambar 4.12 menunjukkan versi Java dari program multithread yang menentukan penjumlahan bilangan bulat non-negatif. Kelas Penjumlahan mengimplementasikan antarmuka Runnable. Pembuatan thread dilakukan dengan membuat instance objek dari kelas Thread dan meneruskan konstruktor objek Runnable.

Membuat objek Thread tidak secara khusus membuat utas baru; melainkan, metode start () membuat utas baru. Memanggil metode start () untuk objek baru melakukan dua hal:

1. Ini mengalokasikan memori dan menginisialisasi utas baru di JVM.
2. Ini memanggil metode run (), membuat thread memenuhi syarat untuk dijalankan oleh JVM. (Perhatikan lagi bahwa kita tidak pernah memanggil run () method secara langsung. Sebaliknya, kita memanggil method start (), dan memanggil method run () atas nama kita.)

Ketika program penjumlahan berjalan, JVM membuat dua utas. Yang pertama adalah thread induk, yang memulai eksekusi dalam metode main (). Thread kedua dibuat ketika start () metode pada objek Thread dipanggil. Benang anak ini memulai eksekusi

dalam metode run () dari kelas Penjumlahan. Setelah mengeluarkan nilai dari penjumlahan, utas ini berakhir ketika keluar dari metode run ()

Berbagi data antara utas terjadi dengan mudah di Windows dan Pthreads, karena data yang dibagikan secara sederhana dideklarasikan secara global. Sebagai bahasa berorientasi objek murni, Java tidak memiliki gagasan tentang data global. Jika dua atau lebih untaian membagi data dalam program Java, pembagian terjadi dengan meneruskan referensi ke objek yang dibagikan ke untaian yang sesuai. Dalam program Java yang ditunjukkan pada Gambar 4.12, utas utama dan simpul penjumlahan berbagi contoh objek dari kelas Sum. Objek yang dibagikan ini direferensikan melalui metode getSum () dan setSum () yang sesuai. (Anda mungkin bertanya-tanya mengapa kita tidak menggunakan objek Integer daripada mendesain kelas penjumlahan baru. Alasannya adalah bahwa kelas Integer tidak dapat diubah — yaitu, begitu nilainya diatur, ia tidak dapat berubah.)

Ingat bahwa orangtua thread di Pthreads dan pustaka Windows menggunakan pthread join () dan WaitForSingleObject () (masing-masing) untuk menunggu utas penjumlahan selesai sebelum melanjutkan. Metode join () di Java menyediakan fungsionalitas yang serupa. (Perhatikan bahwa join () dapat membuang InterruptedException, yang kita pilih untuk diabaikan.) Jika parentmustwait untuk beberapa thread selesai, join () method dapat diapit dalam for loop yang mirip dengan yang ditunjukkan untuk Pthreads pada Gambar 4.10.

4.5 Implicit Threading

Dengan terus berkembangnya pemrosesan multicore, aplikasi yang berisi ratusan atau bahkan ribuan benang muncul di cakrawala. Merancang aplikasi semacam itu bukanlah pekerjaan yang sepele: programmer tidak hanya harus menjawab tantangan yang diuraikan dalam Bagian 4.2 tetapi juga mengalami kesulitan tambahan. Kesulitan-kesulitan ini, yang berhubungan dengan kebenaran program, dibahas dalam Bab 5 dan 7.

Salah satu cara untuk mengatasi kesulitan ini dan lebih mendukung desain aplikasi multithread adalah untuk mentransfer pembuatan dan pengelolaan

```
class Sum
{
private int sum; public
int getSum() {  return
sum;
}
public void setSum(int sum) {
this.sum = sum;
}
}
class Summation implements Runnable
{
private int upper; private
Sum sumValue;
public Summation(int upper, Sum sumValue) {
this.upper = upper;
this.sumValue = sumValue;
}
public void run() {  int
sum = 0;
for (int i = 0; i <= upper; i++)
```

```
sum += i;
sumValue.setSum(sum);
}
        }
        public class Driver
        {
public static void main(String[] args) {
 if  (args.length  >  0)  {      if
(Integer.parseInt(args[0]) < 0)
System.err.println(args[0] + " must be >= 0."); else
{
Sum sumObject = new Sum();
int upper = Integer.parseInt(args[0]);
Thread thrd = new Thread(new Summation(upper, sumObject));
thrd.start(); try {  thrd.join();
System.out.println
("The sum of "+upper+" is "+sumObject.getSum());
} catch (InterruptedException ie) { }
}
}
else
System.err.println("Usage: Summation <integer value>"); }
        }
```

Gambar 4.12 Program Java untuk penjumlahan bilangan bulat non-negatif. threading dari pengembang aplikasi ke pustaka kompilasi dan run-time. Strategi ini,

JVM DAN SISTEM OPERASI HOST

JVM biasanya diimplementasikan di atas sistem operasi host (lihat Gambar 16.10). Pengaturan ini memungkinkan JVM untuk menyembunyikan rincian implementasi sistem operasi yang mendasarinya dan untuk menyediakan lingkungan abstrak yang konsisten yang memungkinkan program Java untuk beroperasi pada platform apa pun yang mendukung JVM. Spesifikasi untuk JVM tidak menunjukkan bagaimana thread Java dipetakan ke sistem operasi yang mendasarinya, daripada membiarkan keputusan itu untuk implementasi khusus JVM. Sebagai contoh, sistem operasi Windows XP menggunakan model satu-ke-satu; Oleh karena itu, setiap utas Java untuk JVM berjalan pada peta sistem seperti itu ke utas kernel. Pada sistem operasi yang menggunakan banyak-ke-banyak model (seperti Tru64 UNIX), sebuah thread Java dipetakan sesuai dengan banyak-ke-banyak model. Solaris awalnya mengimplementasikan JVM menggunakan model banyak-ke-satu
(pustaka benang hijau, yang disebutkan sebelumnya). Kemudian rilis dari JVM diimplementasikan menggunakan model banyak-ke-banyak. Dimulai dengan Solaris 9, Java thread dipetakan menggunakan model satu-ke-satu. Selain itu, mungkin ada hubungan antara pustaka thread Java dan pustaka thread pada sistem operasi host. Sebagai contoh, implementasi dari JVM untuk keluarga Windows dari sistem operasi mungkin menggunakan API Windows ketika membuat untaian Java; Sistem Linux, Solaris, dan Mac OS X mungkin menggunakan Pthreads API.

disebut threading implisit, adalah tren yang populer saat ini. Pada bagian ini, kami mengeksplorasi tiga pendekatan alternatif untuk merancang program multithread yang dapat memanfaatkan prosesor multicore melalui threading implisit.

4.5.1 Thread Pools

Dalam Bagian 4.1, kami mendeskripsikan server web multithread. Dalam situasi ini, setiap kali server menerima permintaan, itu menciptakan utas yang terpisah untuk melayani permintaan. Sedangkan menciptakan utas yang terpisah tentu lebih unggul untuk menciptakan proses yang terpisah, tetapi masih memiliki potensi masalah. Masalah pertama menyangkut jumlah waktu yang dibutuhkan untuk membuat utas, bersama dengan fakta bahwa utas akan dibuang begitu selesai. Isu kedua lebih merepotkan. Jika kami mengizinkan semua permintaan bersamaan untuk dilayani di utas baru, kami belum menempatkan batasan pada jumlah utas yang aktif secara bersamaan dalam sistem. Untaian tak terbatas dapat menghabiskan sumber daya sistem, seperti waktu atau memori CPU. Salah satu solusi untuk masalah ini adalah menggunakan kolam thread.

Ide umum di balik kolam thread adalah membuat sejumlah utas pada proses startup dan menempatkannya ke dalam kolam, di mana mereka duduk dan menunggu pekerjaan. Ketika server menerima permintaan, ia membangkitkan sebuah utas dari kumpulan ini — jika ada yang tersedia — dan memberikannya permintaan untuk layanan. Setelah benang menyelesaikan layanannya, ia kembali ke kolam dan menunggu lebih banyak pekerjaan. Jika pool tidak berisi thread yang tersedia, server menunggu sampai salah satu menjadi gratis.

Pool thread menawarkan manfaat ini:

1. Menyervis permintaan dengan utas yang ada lebih cepat daripada menunggu untuk membuat utas.
2. Sebuah thread pool membatasi jumlah thread yang ada pada satu titik. Ini sangat penting pada sistem yang tidak dapat mendukung sejumlah besar rangkaian konkuren.
3. Memisahkan tugas yang harus dilakukan dari mekanisme pembuatan tugas memungkinkan kita menggunakan berbagai strategi untuk menjalankan tugas. Misalnya, tugas dapat dijadwalkan untuk dijalankan setelah penundaan waktu atau dilakukan secara berkala.

Jumlah utas di kumpulan dapat diatur secara heuristik berdasarkan faktor-faktor seperti jumlah CPU dalam sistem, jumlah memori fisik, dan jumlah permintaan klien konkuren yang diharapkan. Arsitektur thread-pool yang lebih canggih dapat secara dinamis menyesuaikan jumlah utas di kolam sesuai dengan pola penggunaan. Arsitektur semacam itu memberikan manfaat lebih lanjut dari memiliki kolam yang lebih kecil — dengan demikian memakan lebih sedikit memori — ketika beban pada sistem rendah. Kami membahas satu arsitektur seperti itu, Grand Central Dispatch Apple, di bagian ini nanti.

TheWindows API menyediakan beberapa fungsi yang berhubungan dengan kumpulan thread. Menggunakan thread pool API mirip dengan membuat thread dengan fungsi Thread Create (), seperti yang dibahas dalam Bagian 4.4.2. Di sini, fungsi yang dijalankan sebagai thread terpisah didefinisikan. Fungsi seperti itu dapat muncul sebagai berikut:

```
DWORD WINAPI PoolFunction(AVOID Param) {
/*
* this function runs as a separate thread.
```

```
*/
}
```

Pointer ke PoolFunction () dilewatkan ke salah satu fungsi di thread pool API, dan sebuah thread dari pool mengeksekusi fungsi ini. Salah satunya di thread pool API adalah fungsi QueueUserWorkItem (), yang melewati tiga parameter:

- LPTHREAD START ROUTINE Function — sebuah penunjuk ke fungsi yang dijalankan sebagai utas yang terpisah
- Parameter PVOID — parameter yang diteruskan ke Fungsi
- Bendera ULONG — tanda yang menunjukkan bagaimana rangkaian ulir adalah untuk membuat dan mengelola eksekusi utas

Contoh pemanggilan fungsi adalah sebagai berikut: *QueueUserWorkItem(&PoolFunction, NULL, 0);*

Ini menyebabkan utas dari utas ulir untuk mengaktifkan PoolFunction () atas nama pemrogram. Dalam contoh ini, kami tidak mengirimkan parameter ke PoolFunction (). Karena kami menetapkan 0 sebagai bendera, kami menyediakan rangkaian untaian tanpa petunjuk khusus untuk pembuatan thread.

Anggota lain dalam Windows kolam renang API termasuk utilitas yang memanggil fungsi pada interval periodik atau ketika permintaan I / O asynchronous selesai. Paket java.util.concurrent di Java API menyediakan utilitas thread-pool juga.

4.5.2 OpenMP

OpenMP adalah seperangkat arahan kompilator serta API untuk program yang ditulis dalam C, C ++, atau FORTRAN yang menyediakan dukungan untuk pemrograman paralel dalam lingkungan memori bersama. OpenMP mengidentifikasi wilayah paralel sebagai blok kode yang dapat berjalan secara paralel. Pengembang aplikasi memasukkan arahan compiler ke kode mereka di wilayah paralel, dan arahan ini menginstruksikan perpustakaan run-time OpenMP untuk mengeksekusi wilayah secara paralel. Program C berikut mengilustrasikan direktif kompilator di atas wilayah paralel yang berisi pernyataan printf ():

```
#include <omp.h> #include
<stdio.h>
int main(int argc, char *argv[])
{
/* sequential code */
#pragma omp parallel
{
printf("I am a parallel region.");
}
/* sequential code */ return
0;
}
```

Ketika OpenMP menemukan arahan

#pragma omp parallel

itu menciptakan sebanyak mungkin benang ada core yang sedang diproses dalam sistem. Jadi, untuk sistem dual-core, dua utas dibuat, untuk sistem quad-core, empat diciptakan; Dan seterusnya. Semua utas kemudian secara bersamaan menjalankan wilayah paralel. Karena setiap utas keluar dari wilayah paralel, itu diakhiri.

OpenMP menyediakan beberapa arahan tambahan untuk menjalankan wilayah kode secara paralel, termasuk paralel loop. Sebagai contoh, asumsikan kita memiliki dua array a dan b dari ukuran N. Kami ingin menjumlahkan isinya dan menempatkan hasilnya dalam larik c. Kita dapat menjalankan tugas ini secara paralel dengan menggunakan segmen kode berikut, yang berisi direktif kompilator untuk memparalelkan untuk loop: #pragma omp parallel untuk

```
untuk ( i = 0; i <N; i ++) {  c
[ i ] = a [ i ] + b [ i ];

}
```

Open MP membagi pekerjaan yang terkandung dalam for loop di antara thread yang dibuat sebagai respons terhadap direktif

#pragma omp parallel untuk

Selain menyediakan arahan untuk paralelisasi, Open MP memungkinkan para pengembang untuk memilih di antara beberapa level paralelisme. Misalnya, mereka dapat mengatur jumlah utas secara manual. Ini juga memungkinkan pengembang untuk mengidentifikasi apakah data dibagi antara utas atau pribadi ke utas. Terbuka MP tersedia di beberapa open-source dan komersial kompiler untuk Linux, Windows, dan sistem Mac OS X. Kami mendorong pembaca yang tertarik untuk mempelajari lebih lanjut tentang Open MP untuk berkonsultasi dengan bibliografi di akhir bab ini.

4.5.3 Grand Central Dispatch

Grand Central Dispatch (GCD) - teknologi untuk sistem operasi Apple Mac OS X dan i OS - adalah kombinasi ekstensi untuk bahasa C, API , dan pustaka run-time yang memungkinkan pengembang aplikasi untuk mengidentifikasi bagian kode untuk menjalankan sejajar. Seperti OpenMP, GCD mengelola sebagian besar detail penguliran.

GCD mengidentifikasi ekstensi ke bahasa C dan C ++ yang dikenal sebagai **blok** . SEBUAH blok hanyalah unit kerja yang berdiri sendiri. Ini ditentukan oleh tanda sisipan ^ disisipkan di depan sepasang tanda kurung *{}* . Contoh sederhana dari sebuah blok ditunjukkan di bawah ini:

```
{ printf ("I am a block"); }
```

Jadwal GCD memblokir eksekusi run-time dengan menempatkannya pada a **pengiriman antrian** . Ketika menghapus satu blok dari antrian, ia menugaskan blok ke suatu tersedia thread dari kolam thread yang dikelolanya. GCD mengidentifikasi dua jenis antrian pengiriman: ***serial*** dan ***bersamaan*** .

Blok yang ditempatkan pada antrian serial dihapus dalam urutan FIFO . Setelah blok dihapus dari antrean, ia harus menyelesaikan eksekusi sebelum blok lain dihapus. Setiap proses memiliki antrean serial sendiri (dikenal sebagai **antrean utamanya**). Pengembang dapat membuat antrian serial tambahan yang bersifat lokal ke proses tertentu. Antrian serial berguna untuk memastikan eksekusi berurutan dari beberapa tugas.

Blok yang ditempatkan pada antrean bersamaan juga dihapus dalam urutan FIFO , tetapi beberapa blok dapat dihapus pada satu waktu, sehingga memungkinkan beberapa blok untuk dijalankan secara paralel. Ada tiga antrian pengiriman bersamaan sistem, dan mereka dibedakan berdasarkan prioritas: rendah, default, dan tinggi. Prioritas mewakili perkiraan dari kepentingan relatif dari blok. Cukup sederhana, blok dengan prioritas yang lebih tinggi harus ditempatkan pada antrian pengiriman prioritas tinggi.

Segmen kode berikut mengilustrasikan mendapatkan prioritas-prioritas antrian bersamaan dan mengirimkan blok ke antrian menggunakan fungsi pengiriman dispatch_async ( ) function :

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Secara internal, kolam thread GCD terdiri dari benang POSIX . GCD secara aktif mengelola kolam, memungkinkan jumlah benang untuk tumbuh dan menyusut sesuai dengan permintaan aplikasi dan kapasitas sistem.

4.5.4 Pendekatan Lain

Pool thread, Open MP , dan Grand Central Dispatch hanyalah beberapa dari banyak teknologi yang muncul untuk mengelola aplikasi multithread. Pendekatan komersil com-lain termasuk perpustakaan paralel dan bersamaan, seperti Intel Threading Building Blocks (TBB) dan beberapa produk dari Microsoft. Bahasa Java dan API telah melihat gerakan yang signifikan untuk mendukung program sewa-menyewa juga. Contoh penting adalah paket java.util .concurrent , yang mendukung pembuatan dan manajemen thread implisit.

4.6 Masalah Threading

Pada bagian ini, kami membahas beberapa masalah yang perlu dipertimbangkan dalam merancang program multithread.

4.6.1 Garpu () dan exec () Sistem Panggilan

Di Bab 3, kami menjelaskan bagaimana panggilan sistem fork ( ) digunakan untuk membuat proses duplikasi terpisah. Semantik sistem fork ( ) dan exec () memanggil perubahan dalam program multithread.

Jika satu thread dalam program memanggil fork ( ) , apakah proses baru menduplikasi semua thread, atau apakah proses baru adalah single-threaded? Beberapa

sistem UNIX telah memilih untuk memiliki dua versi fork ( ) , yang menduplikasi semua thread dan yang lain yang hanya menggandakan utas yang memanggil call fork () system.

Exec () system call biasanya bekerja dengan cara yang sama seperti yang dijelaskan dalam Bab 3. Artinya, jika thread memanggil exec () system call, program ditentukan dalam parameter exec () akan mengganti seluruh proses - termasuk semua utas.

Yang mana dari dua versi fork ( ) yang digunakan tergantung pada aplikasi. Jika exec ( ) dipanggil segera setelah forking, maka duplikasi semua thread tidak diperlukan, karena program yang ditentukan dalam parameter ke exec () akan menggantikan proses. Dalam hal ini, menduplikat hanya benang panggilan yang sesuai. Namun, jika proses terpisah tidak memanggil exec ( ) setelah forking, proses yang terpisah harus menduplikasi semua utas.

4.6.2 Penanganan Sinyal

Sebuah **sinyal** digunakan dalam sistem UNIX untuk memberi tahu proses bahwa suatu peristiwa tertentu telah terjadi. Suatu sinyal dapat diterima secara sinkron atau asinkron, tergantung pada sumber dan alasan untuk acara yang ditandai. Semua sinyal, baik sinkron atau asinkron, mengikuti pola yang sama:

1. Suatu sinyal dihasilkan oleh terjadinya suatu kejadian tertentu.
2. Sinyal dikirimkan ke suatu proses.
3. Setelah dikirimkan, sinyal harus ditangani.

Contoh sinyal sinkron termasuk akses memori ilegal dan divi-sion oleh 0. Jika program berjalan melakukan salah satu dari tindakan ini, sinyal dihasilkan. Sinyal sinkron dikirim ke proses yang sama yang melakukan operasi yang menyebabkan sinyal (itulah alasannya mereka dianggap sinkron).

Ketika suatu sinyal dihasilkan oleh suatu kejadian di luar proses yang sedang berjalan, proses itu menerima sinyal secara asynchronous. Contoh sinyal tersebut termasuk mengakhiri proses dengan penekanan tombol spesifik (seperti < control > < C >) dan memiliki waktu berakhir. Biasanya, sinyal asynchronous dikirim ke proses lain.

Sinyal dapat ***ditangani*** oleh salah satu dari dua penangan yang mungkin:

1. Penangan sinyal default
2. Penangan sinyal yang ditentukan pengguna

Setiap sinyal memiliki **penangan sinyal default** yang dijalankan kernel saat menangani sinyal itu. Tindakan standar ini dapat ditimpa oleh **pengguna yang ditentukan penangan sinyal** yang dipanggil untuk menangani sinyal. Sinyal ditangani di cara berbeda. Beberapa sinyal (seperti mengubah ukuran jendela) diabaikan begitu saja; orang lain (seperti akses memori ilegal) ditangani dengan mengakhiri program.

Penanganan sinyal dalam program single-threaded sangat mudah: sinyal selalu dikirimkan ke suatu proses. Namun, mengirimkan sinyal lebih rumit dalam program multithread, di mana suatu proses mungkin memiliki beberapa utas. Di mana, kemudian, haruskah sinyal dikirimkan?

Secara umum, opsi berikut ada:

1. Kirim sinyal ke benang tempat sinyal berlaku.
2. Kirim sinyal ke setiap utas dalam proses.
3. Kirim sinyal ke utas tertentu dalam proses.
4. Tetapkan utas tertentu untuk menerima semua sinyal untuk proses tersebut.

Metode pengiriman sinyal tergantung pada jenis sinyal yang dihasilkan. Sebagai contoh, sinyal sinkron perlu dikirimkan ke thread yang menyebabkan sinyal dan tidak ke thread lain dalam proses. Namun, situasi dengan sinyal asynchronous tidak begitu jelas. Beberapa sinyal asynchronous - seperti sinyal yang mengakhiri proses (< control > < C > , misalnya) - harus dikirim ke semua utas.

Fungsi UNIX standar untuk mengirimkan sinyal adalah

kill (pid t pid , int signal)

Fungsi ini menentukan proses (pid) ke mana sinyal (sinyal) tertentu akan dikirimkan. Kebanyakan UNIX versi multithread memungkinkan sebuah untaian untuk menentukan sinyal mana yang akan diterima dan mana yang akan diblok. Oleh karena itu, dalam beberapa kasus, sinyal asinkron dapat dikirimkan hanya ke utas yang tidak memblokirnya. Namun, karena sinyal hanya perlu ditangani sekali, sinyal biasanya dikirimkan hanya ke utas pertama yang ditemukan yang tidak memblokirnya. POSIX Pthreads menyediakan fungsi berikut, yang memungkinkan sinyal dikirimkan ke thread tertentu (tid):

pthread kill (pthread t tid , int signal)

Meskipun Windows tidak secara eksplisit memberikan dukungan untuk sinyal, itu memungkinkan kita untuk meniru mereka menggunakan **panggilan prosedur asynchronous (APC s).** Fasilitas APC memungkinkan utas pengguna untuk menentukan fungsi yang akan dipanggil ketika utas pengguna menerima pemberitahuan acara tertentu. Seperti yang ditunjukkan oleh namanya, APC secara kasar setara dengan sinyal asynchronous di UNIX . Namun, sedangkan UNIX harus bersaing dengan bagaimana menangani sinyal dalam lingkungan multithread, fasilitas APC lebih mudah, karena APC dikirim ke thread tertentu daripada proses.

4.6.3 Pembatalan Thread

Pembatalan thread melibatkan mengakhiri thread sebelum selesai. Untuk Misalnya, jika beberapa utas secara bersamaan mencari melalui basis data dan satu utas mengembalikan hasilnya, utas yang tersisa mungkin dibatalkan. Situasi lain mungkin terjadi ketika pengguna menekan tombol pada browser web yang menghentikan halaman web dari memuat lebih jauh. Seringkali, halaman web memuat menggunakan beberapa utas - setiap gambar dimuat dalam utas yang terpisah. Ketika seorang pengguna menekan tombol stop pada browser, semua thread yang memuat halaman dibatalkan.

Untaian yang harus dibatalkan sering disebut sebagai **untaian target** .
Pembatalan thread target dapat terjadi dalam dua skenario berbeda:

1. **Pembatalan asinkron** . Satu utas segera mengakhiri utas target.
2. **Pembatalan dibatalkan** . Benang target secara berkala memeriksa apakah itu harus mengakhiri, memungkinkannya kesempatan untuk mengakhiri dirinya secara teratur.

Kesulitan dengan pembatalan terjadi dalam situasi di mana sumber daya telah dialokasikan ke utas yang dibatalkan atau di mana utas dibatalkan sementara di tengah memperbarui data yang dibagikan dengan utas lainnya. Ini menjadi sangat bermasalah dengan pembatalan asynchronous. Seringkali, sistem operasi akan memperoleh kembali sumber daya sistem dari utas yang dibatalkan tetapi tidak akan mendapatkan kembali semua sumber daya. Oleh karena itu, membatalkan utas secara asynchronous mungkin tidak membebaskan sumber daya sistem yang diperlukan secara luas.

Dengan pembatalan yang ditangguhkan, sebaliknya, satu utas menunjukkan bahwa utas target harus dibatalkan, tetapi pembatalan hanya terjadi setelah utas target mengecek bendera untuk menentukan apakah pembatalan itu harus dibatalkan atau tidak. Untaian dapat melakukan pemeriksaan ini pada titik yang dapat dibatalkan dengan aman. Di Pthreads , pembatalan thread dimulai dengan menggunakan pthread membatalkan ( ) fungsi. Identifier dari thread target dilewatkan sebagai parameter ke fungsi. Kode berikut mengilustrasikan pembuatan - dan kemudian membatalkan - utas:

```
pthread t tid ;

/ * buat utas * /  pthread buat ( & tid , 0,
pekerja, NULL);
. . .
/ * membatalkan utas * /  pthread
cancel ( tid );
```

Meminta pthread membatalkan ( ) hanya menunjukkan permintaan untuk membatalkan thread target, namun; pembatalan sebenarnya tergantung pada bagaimana thread target diatur untuk menangani permintaan. Pthreads mendukung tiga mode pembatalan. Setiap mode didefinisikan sebagai negara dan tipe, seperti yang diilustrasikan dalam tabel di bawah ini. Thread dapat mengatur status pembatalan dan mengetik menggunakan API .

Mode	Negara	Mengetik
Mati	Cacat	-
Tangguhan	Diaktifkan	Tangguhan
Asynchronous	Diaktifkan	Asynchronous

Seperti yang digambarkan oleh tabel, Pthreads memungkinkan utas untuk menonaktifkan atau memungkinkan pembatalan. Tentunya, utas tidak dapat dibatalkan jika pembatalan dinonaktifkan. Namun, permintaan pembatalan tetap tertunda, sehingga utas tersebut nantinya dapat mengaktifkan pembatalan dan menanggapi permintaan.

Jenis pembatalan default adalah pembatalan yang ditangguhkan. Di sini, pembatalan hanya terjadi ketika sebuah thread mencapai **titik pembatalan** . Salah satu teknik untuk menetapkan titik pembatalan adalah untuk memohon pthread testcancel ( ) fungsi. Jika permintaan pembatalan ditemukan tertunda, fungsi yang dikenal sebagai **pengendali pembersihan** akan dipanggil. Fungsi ini memungkinkan sumber daya apa pun yang mungkin telah diperoleh untaian untuk dirilis sebelum utas dihentikan.

Kode berikut mengilustrasikan bagaimana utas dapat menanggapi permintaan pembatalan menggunakan pembatalan yang ditangguhkan:

```
while (1) {
   /* do some work for awhile */
   /* . . . */

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

Karena masalah yang dijelaskan sebelumnya, pembatalan asinkron tidak dianjurkan dalam dokumentasi Pthreads . Jadi, kami tidak membahasnya di sini. Catatan yang menarik adalah bahwa pada sistem Linux, pembatalan thread menggunakan Pthreads API ditangani melalui sinyal (Bagian 4.6.2).

4.6.4 Penyimpanan Thread-Lokal

Thread milik proses berbagi data dari proses. Memang, berbagi data ini memberikan salah satu manfaat dari pemrograman multithread. Namun, dalam beberapa keadaan, setiap utas mungkin memerlukan salinan data tertentu. Kami akan memanggil **penyimpanan** data **-thread lokal seperti itu** (atau **TLS** .) Sebagai contoh, dalam sistem pemrosesan transaksi, kami mungkin melayani setiap transaksi dalam utas yang terpisah. Selanjutnya, setiap transaksi dapat diberi pengenal unik. Untuk mengaitkan setiap utas dengan pengenal uniknya, kita bisa menggunakan penyimpanan thread-lokal.

Sangat mudah untuk membingungkan TLS dengan variabel lokal. Namun, variabel lokal hanya terlihat selama satu pemanggilan fungsi tunggal, sedangkan data TLS terlihat di seluruh fungsi panggilan. Dalam beberapa hal, TLS mirip dengan data statis . Perbedaannya adalah data TLS unik untuk setiap utas. Kebanyakan pustaka thread - termasuk Windows dan Pthread - menyediakan beberapa bentuk dukungan untuk penyimpanan lokal-thread; Java juga menyediakan dukungan.

4.6.5 Aktivasi Penjadwal

Masalah terakhir yang harus dipertimbangkan dengan program multithread adalah komunikasi antara kernel dan pustaka thread, yang mungkin diperlukan oleh banyak-kebanyak dan dua-tingkat model yang dibahas dalam Bagian 4.3.3. Koordinasi seperti itu memungkinkan jumlah utas kernel disesuaikan secara dinamis untuk membantu memastikan kinerja terbaik.

Banyak sistem yang mengimplementasikan model banyak-ke-banyak atau dua tingkat menempatkan struktur data antara antara pengguna dan utas kernel. Struktur data ini - biasanya dikenal sebagai **proses ringan** , atau **LWP** - ditunjukkan pada Gambar 4.13. Untuk pustaka ulir pengguna, LWP tampaknya menjadi prosesor virtual tempat aplikasi dapat menjadwalkan thread pengguna untuk dijalankan. Setiap LWP dilampirkan ke utas kernel, dan itu adalah utas kernel yang jadwal sistem operasi untuk dijalankan pada prosesor fisik. Jika blok utas kernel (seperti ketika menunggu operasi I / O selesai), blok LWP juga. Sampai rantai, benang tingkat pengguna yang melekat pada LWP juga memblokir.

Aplikasi mungkin memerlukan sejumlah LWP untuk berjalan secara efisien. Pertimbangkan aplikasi berbasis CPU yang dijalankan pada satu prosesor. Dalam skenario ini, hanya satu utas yang dapat dijalankan pada satu waktu, jadi satu LWP sudah cukup. Aplikasi yang I / O- Intensive mungkin memerlukan beberapa LWP untuk dieksekusi, namun. Biasanya, LWP diperlukan untuk setiap panggilan sistem pemblokiran

bersamaan. Anggaplah, misalnya, bahwa lima permintaan file-read yang berbeda terjadi secara bersamaan. Diperlukan lima LWP, karena semua bisa menunggu penyelesaian I / O di kernel. Jika proses hanya memiliki empat LWP , maka permintaan kelima harus menunggu salah satu LWP untuk kembali dari kernel.

Satu skema untuk komunikasi antara pustaka pengguna-benang dan kernel dikenal sebagai **aktivasi penjadwal** . Ia bekerja sebagai berikut: Kernel menyediakan aplikasi dengan satu set prosesor virtual (LWP s), dan aplikasi dapat menjadwalkan thread pengguna ke prosesor virtual yang tersedia. Selanjutnya, kernel harus menginformasikan aplikasi tentang kejadian tertentu. Prosedur ini dikenal sebagai **upcall** . Upcall ditangani oleh pustaka thread dengan **handler upcall** , dan penangan upcall harus berjalan pada prosesor virtual. Satu peristiwa yang memicu upcall terjadi ketika utas aplikasi akan diblokir. Dalam skenario ini, kernel membuat upcall ke aplikasi yang memberitahukan bahwa sebuah thread akan memblokir dan mengidentifikasi thread khusus. Kernel kemudian mengalokasikan prosesor virtual baru ke aplikasi. Aplikasi ini menjalankan handler upcall pada prosesor virtual baru ini, yang menghemat keadaan dari blok pemblokiran dan melepaskan prosesor virtual tempat thread pemblokiran sedang berjalan. Handler upcall kemudian menjadwalkan thread lain yang memenuhi syarat untuk dijalankan pada prosesor virtual baru. Ketika peristiwa dimana thread pemblokiran sedang menunggu terjadi, kernel membuat lagi upcall ke thread library yang memberitahukan bahwa thread yang sebelumnya diblokir sekarang layak untuk dijalankan. Handler upcall untuk acara ini juga membutuhkan prosesor virtual, dan kernel dapat mengalokasikan prosesor virtual baru atau mendahului salah satu utas pengguna dan menjalankan handler upcall pada prosesor virtualnya. Setelah menandai blok yang tidak diblokir sebagai layak untuk dijalankan, aplikasi menjadwalkan thread yang memenuhi syarat untuk berjalan pada prosesor virtual yang tersedia.

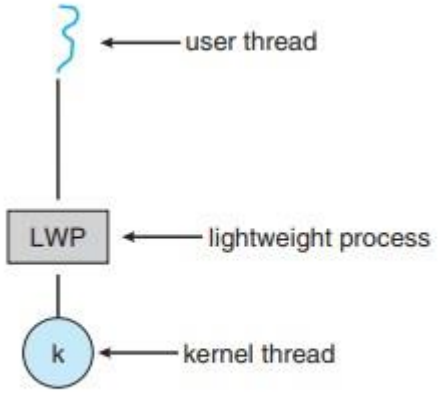


Figure 4.13 Lightweight process (LWP).

4.7 Contoh Sistem Operasi

Pada titik ini, kami telah memeriksa sejumlah konsep dan masalah terkait utas. Kami menyimpulkan bab ini dengan menjelajahi bagaimana benang diterapkan dalam sistem Windows dan Linux.

4.7.1 Thread Windows

Windows mengimplementasikan Windows API, yang merupakan API utama untuk keluarga sistem operasi Microsoft (Windows 98, NT, 2000, dan XP, juga seperti Windows

7). Memang, banyak dari apa yang disebutkan di bagian ini berlaku untuk seluruh keluarga sistem operasi ini.

Aplikasi Windows berjalan sebagai proses yang terpisah, dan setiap proses dapat dilakukan berisi satu atau beberapa utas. API Windows untuk membuat utas tercakup Bagian 4.4.2. Selain itu, Windows menggunakan pemetaan satu-ke-satu yang dijelaskan di Bagian 4.3.2, di mana setiap thread tingkat pengguna memetakan ke kernel yang terkait benang.

Komponen umum utas meliputi:

- ID thread secara unik mengidentifikasi utas
- Satu set register yang mewakili status prosesor
- Tumpukan pengguna, digunakan saat utas berjalan dalam mode pengguna, dan sebuah kernel stack, digunakan ketika thread berjalan dalam mode kernel
- Area penyimpanan pribadi yang digunakan oleh berbagai pustaka run-time dan tautan dinamis perpustakaan (DLL)

Set register, tumpukan, dan area penyimpanan pribadi dikenal sebagai konteks utas. Struktur data utama dari sebuah thread meliputi:

- ETHREAD — blok thread eksekutif
- KTHREAD — blok utas kernel
- TEB — blok lingkungan utas

Komponen utama ETHREAD termasuk pointer ke proses untuk mana benang itu berada dan alamat dari rutinitas di mana thread mulai mengontrol. ETHREAD juga berisi pointer ke yang sesuai KTHREAD.

The KTHREAD termasuk penjadwalan dan informasi sinkronisasi untuk utas. Selain itu, KTHREAD termasuk tumpukan kernel (digunakan ketika thread berjalan dalam mode kernel) dan pointer ke TEB.

ETHREAD dan KTHREAD sepenuhnya ada di ruang kernel; ini berarti hanya kernel yang dapat mengaksesnya. TEB adalah struktur data ruang pengguna yang diakses ketika utas berjalan dalam mode pengguna. Di antara bidang lainnya, TEB berisi pengenal untaian, tumpukan mode pengguna, dan larik untuk penyimpanan thread-lokal. Struktur utas Windows diilustrasikan pada Gambar 4.14.

4.7.2 Thread Linux

Linux menyediakan fork () system call dengan fungsi tradisional menduplikat proses, seperti yang dijelaskan di Bab 3. Linux juga menyediakan kemampuan untuk membuat thread menggunakan clone () system call. Namun, Linux tidak membedakan antara proses dan utas. Bahkan, Linux menggunakan istilah tugas—Ratus daripada proses atau utas — ketika mengacu pada aliran kontrol dalam sebuah program.

Ketika klon () dipanggil, dilewatkan satu set bendera yang menentukan bagaimana banyak berbagi terjadi antara tugas orang tua dan anak. Beberapa dari ini bendera tercantum pada Gambar 4.15. Misalnya, anggaplah klon () dilewatkan bendera CLONE FS, CLONE VM, CLONE SIGHAND, dan CLONE FILES. Orang tua dan tugas anak kemudian akan berbagi informasi file-sistem yang sama (seperti direktori kerja saat ini), ruang memori yang sama, penangan sinyal yang sama,

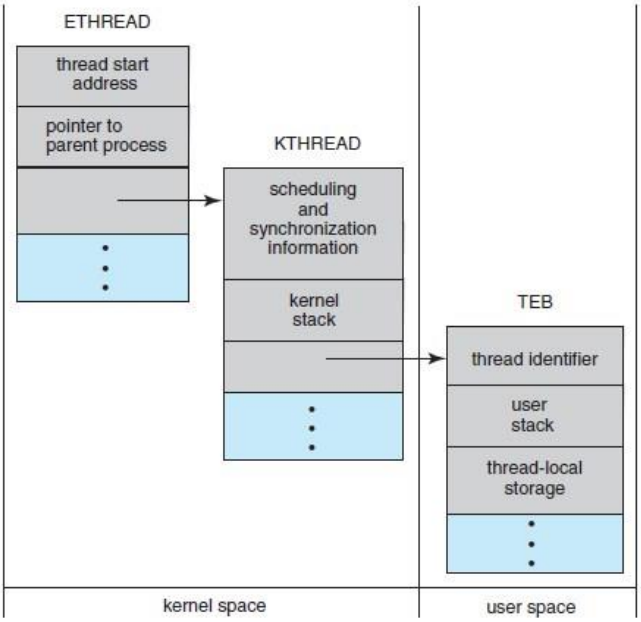


Figure 4.14 Data structures of a Windows thread.

Dan terbukalah set file yang sama. Menggunakan clone() dalam mode ini setara dengan membuat *thread* (thread) seperti yang dijelaskan dalam bab ini, karena *parent task* membagi sebagian besar sumber dayanya dengan *child task*. Namun, jika tidak ada *flag* yang ditetapkan ketika clone () dipanggil, tidak ada pembagian yang terjadi, menghasilkan fungsionalitas yang serupa dengan yang disediakan oleh sistem pemangillan fork () .

Level berbagi yang bervariasi dimungkinkan karena cara tugas ditunjukkan dalam Linux kernel. Struktur data kernel yang unik (khusus, struct task struct) ada untuk setiap tugas dalam sistem. Struktur data ini, alih-alih menyimpan data untuk tugas, berisi pointer ke struktur data lain di mana data ini disimpan — misalnya, struktur data yang mewakili daftar file yang terbuka, informasi penanganan sinyal, dan memori virtual. Ketika fork () dipanggil, tugas baru dibuat, bersama dengan salinan dari semua struktur data terkait dari *parent process* (proses induk). Tugas baru juga dibuat ketika panggilan sistem clone () dibuat. Namun, daripada menyalin semua struktur data, tugas baru menunjuk ke struktur data dari tugas induk, tergantung pada kumpulan *flag* yang dilewatkan ke clone ().

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figure 4.15 Some of the flags passed when `clone()` is invoked.

4.8 Rangkuman

Thread adalah aliran kontrol dalam suatu proses. Proses multi-threaded berisi beberapa aliran kontrol yang berbeda dalam ruang alamat yang sama. Manfaat multi threading termasuk peningkatan respons terhadap pengguna, berbagi sumber daya dalam proses, ekonomi, dan faktor skalabilitas, seperti penggunaan yang lebih efisien dari beberapa inti pemrosesan.

Thread tingkat pengguna adalah thread yang terlihat oleh programmer dan tidak diketahui oleh kernel. Sistem operasi kernel mendukung dan mengatur thread tingkat kernel. Secara umum, thread tingkat pengguna lebih cepat dibuat dan dikelola daripada thread kernel, karena tidak ada intervensi dari kernel yang diperlukan.

Tiga jenis model yang berbeda menghubungkan pengguna dan thread kernel. Model banyak-ke-satu memetakan banyak thread pengguna ke thread kernel tunggal. Model satu-kesatu memetakan setiap thread pengguna ke thread kernel terkait. Model banyak-ke-banyak mengalikan banyak thread pengguna ke sejumlah thread kernel yang lebih kecil atau sama banyaknya.

Sebagian besar sistem operasi modern menyediakan menunjang thread untuk kernel. Termasuk Windows, Mac OS X, Linux, dan Solaris.

Pustaka thread menyediakan programmer aplikasi dengan API untuk membuat dan mengelola thread. Tiga pustaka utama thread digunakan secara umum: POSIX Pthreads, Windows threads, dan Java threads.

Selain secara eksplisit membuat thread menggunakan API yang disediakan oleh pustaka, kita dapat menggunakan implisit threading, di mana pembuatan dan pengelolaan threading ditransfer ke penyusun dan run-time pustaka. Strategi untuk implisit threading termasuk thread pool, OpenMP, dan Grand Central Dispatch.

Program multi-thread memperkenalkan banyak tantangan untuk programmer, termasuk semantik panggilan sistem fork () dan exec (). Masalah lainnya termasuk penanganan sinyal, pembatalan thread, penyimpanan thread-lokal, dan aktivasi penjadwalan.

Latihan-latihan

4.1 Berikan dua contoh pemrograman di mana multi-threading memberikan kinerja yang lebih baik daripada single-threaded.

4.2 Apa dua perbedaan antara thread tingkat pengguna dan thread tingkat kernel? Dalam situasi apa satu jenis lebih baik dari yang lain?

4.3 Jelaskan tindakan yang dilakukan oleh kernel untuk beralih konteks antara thread level kernel.

4.4 Sumber apa yang digunakan saat thread dibuat? Bagaimana mereka berbeda dari yang digunakan ketika proses dibuat?

4.5 Asumsikan bahwa sistem operasi memetakan tingkat pengguna ke kernel menggunakan banyak-ke-banyak model dan pemetaan dilakukan melalui LWP. Selain itu, sistem ini memungkinkan pengembang untuk membuat thread real-time untuk digunakan dalam sistem real-time. Apakah perlu untuk mengikat thread realtime ke LWP? Menjelaskan.

Latihan

4.6 Berikan dua contoh pemrograman di mana multi-threading tidak memberikan kinerja yang lebih baik daripada solusi single-threaded.

4.7 Dalam keadaan apa multi-thread menggunakan beberapa thread kernel memberikan kinerja yang lebih baik daripada single-threaded pada sistem prosesor tunggal?

4.8 Manakah dari komponen program state berikut yang dibagikan di seluruh thread dalam proses multi-thread? a) Register values
b) Heap memory
c) Global variables
d) Stack memory

4.9 Dapatkah solusi multithread menggunakan beberapa tingkat pengguna threads mencapai kinerja yang lebih baik pada sistem multiprosesor daripada pada sistem singleprocessor? Jelaskan!

4.10 Di Bab 3, kami membahas peramban Chrome Google dan membuka setiap situs web baru dalam proses terpisah. Apakah manfaat yang sama telah tercapai jika Chrome dirancang untuk membuka setiap situs web baru dalam thread yang terpisah? Jelaskan!

4.11 Mungkinkah mempunyai konkurensi tetapi bukan paralelisme? Jelaskan!

4.12 Dengan menggunakan Hukum Amdahl, hitung pertambahan percepatan aplikasi yang memiliki komponen paralel 60 persen untuk (a) dua inti pemrosesan dan (b) empat inti pemrosesan.

4.13 Tentukan apakah masalah berikut menunjukkan tugas atau paralelisme data:
- Program statistik multithread yang dijelaskan dalam Latihan 4.21
- Validator Sudoku multithreaded yang dijelaskan dalam Proyek 1 dalam bab ini
- Program pengurutan multithread yang dijelaskan dalam Proyek 2 dalam bab ini
- Server web multithread yang dijelaskan dalam Bagian 4.1.

4.14 Sistem dengan dua prosesor dual-core memiliki empat prosesor yang tersedia untuk penjadwalan. Aplikasi intensif CPU berjalan di sistem ini. Semua input dilakukan saat memulai program, ketika satu file harus dibuka. Demikian pula, semua output dilakukan tepat sebelum program berakhir, ketika hasil program harus ditulis ke file tunggal. Antara startup dan terminasi, program ini sepenuhnya terikat pada CPU. Tugas Anda adalah meningkatkan kinerja aplikasi ini dengan cara multithreading. Aplikasi ini berjalan pada sistem yang menggunakan model threading satu-ke-satu (setiap thread pengguna memetakan ke theard kernel).
- Berapa banyak thread yang akan Anda buat untuk melakukan input dan output? Jelaskan!
- Berapa banyak thread yang akan Anda buat untuk bagian aplikasi intensif CPU? Jelaskan!

4.15 Perhatikan segmen kode berikut :

```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
  fork();
  thread_create( . . .);
}
fork();
```

a) Berapa banyak proses unik yang dibuat?
b) Berapa banyak thread unik yang dibuat?

4.16 Sebagaimana dijelaskan dalam Bagian 4.7.2, Linux tidak membedakan antara proses dan thread. Sebaliknya, Linux memperlakukan keduanya dengan cara yang sama, memungkinkan sebuah tugas menjadi lebih mirip dengan proses atau thread tergantung pada rangkaian flag yang dilewatkan ke sistem pemanggilan clone (). Namun, sistem operasi lain, seperti Windows, memperlakukan proses dan thread

dengan berbeda. Biasanya, sistem tersebut menggunakan notasi di mana struktur data untuk suatu proses berisi pointer ke thread-thread yang terpisah dari proses tersebut. Bandingkan kedua pendekatan ini untuk memodelkan proses dan thread di dalam kernel.

4.17 Program yang ditunjukkan pada Gambar 4.16 menggunakan Pthreads API. Apa yang akan menjadi output dari program di LINE C dan LINE P?

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

#include <types.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Figure 4.16 C program for Exercise 4.17.

4.18 Pertimbangkan sistem multicore dan program multithread yang ditulis menggunakan model threading banyak-ke-banyak. Biarkan jumlah thread tingkat pengguna dalam program lebih besar dari jumlah inti pemrosesan dalam sistem. Diskusikan implikasi kinerja dari skenario berikut.

a) Jumlah thread kernel yang dialokasikan ke program kurang dari jumlah inti pemrosesan.
b) Jumlah thread kernel yang dialokasikan ke program sama dengan jumlah inti pemrosesan.

c) Jumlah thread kernel yang dialokasikan untuk program lebih besar dari jumlah inti pemrosesan tetapi kurang dari jumlah thread tingkat pengguna.

4.19 Pthreads menyediakan API untuk mengelola pembatalan thread. Fungsi setcancelstate () Pthread digunakan untuk mengatur status pembatalan. Prototipe ini muncul sebagai berikut:

pthread setcancelstate (int state, int * oldstate)

Dua nilai yang mungkin untuk keadaan adalah PTHREAD CANCEL ENABLE dan PTHREAD CANCEL DISABLE. Menggunakan segmen kode yang ditunjukkan pada Gambar 4.17, berikan contoh dua operasi yang akan cocok untuk melakukan antara panggilan untuk menonaktifkan dan mengaktifkan pembatalan thread.

```
int oldstate;

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

/* What operations would be performed here? */

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Figure 4.17 C program for Exercise 4.19.

Programing Problem

4.20 Ubah masalah pemrograman Latihan 3.20 dari Bab 3, yang meminta Anda mendesain pengelola pid. Modifikasi ini akan terdiri dari penulisan program multithread yang menguji solusi Anda untuk Latihan 3.20. Anda akan membuat sejumlah thread — misalnya, 100 — dan setiap thread akan meminta pid, tidur untuk waktu randengan, lalu lepaskan pid. (Tidur untuk periode waktu acak mendekati penggunaan pid tertentu di mana pid ditugaskan untuk proses baru, proses mengeksekusi dan kemudian berakhir, dan pid dirilis pada pemutusan proses.) Pada sistem UNIX dan Linux, tidur adalah dicapai melalui fungsi sleep (), yang dilewatkan nilai integer yang mewakili jumlah detik untuk tidur. Masalah ini akan dimodifikasi di Bab 5.

4.21 Tulis program multithread yang menghitung berbagai nilai statistik untuk daftar nomor. Program ini akan dilewatkan serangkaian angka pada baris perintah dan kemudian akan membuat tiga thread pekerja yang terpisah. Satu thread akan menentukan rata-rata bilangan, yang kedua akan menentukan nilai maksimum, dan yang ketiga akan menentukan nilai minimum. Misalnya, program Anda dilewatkan bilangan bulat

90 81 78 95 79 72 85

Program ini akan melaporkan

Nilai rata-ratanya adalah 82
Nilai minimumnya adalah 72
Nilai maksimumnya adalah 95

Variabel yang mewakili nilai rata-rata, minimum, dan maksimum akan disimpan secara global. Benang pekerja akan menetapkan nilai-nilai ini, dan thread induk akan menghasilkan nilai setelah para pekerja keluar. (Kami jelas bisa memperluas program ini dengan membuat thread tambahan yang menentukan nilai statistik lainnya, seperti median dan standar deviasi.)

4.22 Cara menghitung yang menarik! adalah menggunakan teknik yang dikenal sebagai Monte Carlo, yang melibatkan pengacakan. Teknik ini berfungsi sebagai berikut: Misalkan Anda memiliki lingkaran yang tertulis dalam persegi, seperti yang ditunjukkan pada Gambar 196 Bab 4 Threads

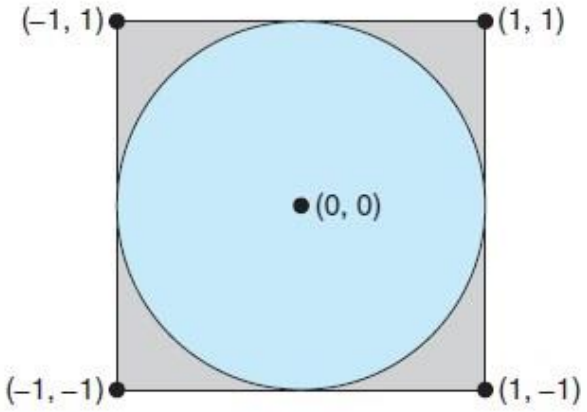


Figure 4.18 Monte Carlo technique for calculating pi.

4.18 (Assume that the radius of this circle is 1.) First, generate a series of random points as simple (x, y) coordinates.These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle. Next, estimate π by performing the following calculation:

π = 4× *(banyak titik di lingkaran)* / *(total banyak point)*

Tuliskan versi multithread dari algoritma ini yang membuat thread yang terpisah untuk menghasilkan sejumlah titik acak. Thread akan menghitung jumlah titik yang terjadi dalam lingkaran dan menyimpan yang menghasilkan variabel global. Ketika thread ini telah keluar, thread induk akan menghitung dan menampilkan nilai perkiraan π Perlu bereksperimen dengan jumlah titik acak yang dihasilkan. Sebagai aturan umum, semakin besar jumlah titik, semakin dekat pendekatan ke π. Dalam pengunduhan kode sumber untuk teks ini, kami menyediakan contoh program yang menyediakan teknik untuk menghasilkan angka acak, serta menentukan apakah titik acak (x, y) terjadi dalam lingkaran. Pembaca tertarik dengan detail metode Monte Carlo untuk memperkirakan π harus berkonsultasi dengan bibliografi di akhir bab ini. Dalam Bab 5, kami memodifikasi latihan ini menggunakan materi yang relevan dari bab itu.

4.23 Ulangi Latihan 4.22, tetapi alih-alih menggunakan thread yang terpisah untuk menghasilkan poin acak, gunakan OpenMP untuk memparalelkan pembangkitan poin. Berhati-hatilah agar tidak menempatkan perhitungan π di wilayah paralel, karena Anda ingin menghitung π hanya sekali.

4.24 Tulis program multithread yang menghasilkan bilangan prima. Program ini harus bekerja sebagai berikut: Pengguna akan menjalankan program dan akan memasukkan nomor pada baris perintah. Program kemudian akan membuat thread

terpisah yang menghasilkan semua bilangan prima kurang dari atau sama dengan nomor yang dimasukkan oleh pengguna.

4.25 Ubahlah server tanggal berbasis soket (Gambar 3.21) di Bab 3 sehingga server melayani setiap permintaan klien dalam thread yang terpisah.

4.26 Urutan Fibonacci adalah rangkaian angka 0, 1, 1, 2, 3, 5, 8, ... Secara formal, dapat dinyatakan sebagai:

$f\ ib0 = 0\ f$

$ib1 = 1$

$f\ ibn = f\ ibn-1 + f\ ibn-2$

Tulis program multithread yang menghasilkan deret Fibonacci. Program ini harus bekerja sebagai berikut: Pada baris perintah, pengguna akan memasukkan nomor Fibonacci yang akan dihasilkan oleh program. Program ini kemudian akan membuat thread yang terpisah yang akan menghasilkan angka Fibonacci, menempatkan urutan dalam data yang dapat dibagi oleh benang (array mungkin merupakan struktur data yang paling nyaman). Ketika thread selesai dieksekusi, thread induk akan menghasilkan urutan yang dihasilkan oleh child thread. Karena rangkaian induk tidak dapat memulai mengeluarkan urutan Fibonacci sampai thread anak selesai, thread induk harus menunggu thread anak selesai. Gunakan teknik yang dijelaskan di Bagian 4.4 untuk memenuhi persyaratan ini.

4.27 Latihan 3.25 di Bab 3 melibatkan perancangan server gema menggunakan Java threading API. Server ini single-threaded, yang berarti bahwa server tidak dapat menanggapi klien gema konkuren sampai klien saat ini keluar. Ubah solusi ke Latihan 3.25 sehingga layanan server echo setiap klien dalam permintaan terpisah.

Programing Project Proyek 1 — Sudoku Solution Validator

Sebuah teka-teki Sudoku menggunakan grid 9 × 9 di mana setiap kolom dan baris, serta masing-masing dari sembilan 3 × 3 subgrid, harus berisi semua digit 1 · · · 9. Gambar 4.19 menyajikan contoh teka-teki Sudoku yang valid . Proyek ini terdiri dari merancang aplikasi multithread yang menentukan apakah solusi teka-teki Sudoku valid.

Ada beberapa cara berbeda dari multithreading aplikasi ini. Salah satu strategi yang disarankan adalah membuat thread yang memeriksa kriteria berikut:

- Sebuah thread untuk memeriksa bahwa setiap kolom berisi angka 1 hingga 9
- Sebuah thread untuk memeriksa bahwa setiap baris berisi digit 1 hingga 9
- Sembilan thread untuk memeriksa bahwa masing-masing dari 3 × 3 subgrid berisi digit 1 hingga 9

Ini akan menghasilkan total sebelas thread terpisah untuk memvalidasi teka-teki Sudoku. Namun, Anda dipersilakan untuk membuat lebih banyak thread untuk proyek ini. Misalnya, daripada membuat satu thread yang memeriksa kesembilannya

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 4.19 Solution to a 9 × 9 Sudoku puzzle.

kolom, Anda dapat membuat sembilan thread terpisah dan meminta masing-masing memeriksanya satu kolom.

Passing Parameters to Each Thread

Thread induk akan membuat thread pekerja, memberikan setiap pekerja lokasi yang harus diperiksa di kisi Sudoku. Langkah ini memerlukan beberapa parameter untuk setiap thread. Pendekatan termudah adalah membuat struktur data menggunakan struct. Sebagai contoh, struktur untuk meneruskan kolom rowand di mana sebuah thread harus mulai memvalidasi akan muncul sebagai berikut: /* structure for passing data to threads */
typedef struct

{

int row; int

column;

} parameters;

Kedua program Pthreads dan Windows akan membuat thread pekerja menggunakan strategi yang serupa dengan yang ditunjukkan di bawah ini:

parameters *data = (parameters *) malloc(sizeof(parameters));

data->row = 1; data->column = 1;

/* Now create the thread passing it data as a parameter */

Penunjuk data akan diteruskan ke fungsi pthread create () (Pthreads) atau fungsi CreateThread () (Windows), yang pada gilirannya akan meneruskannya sebagai parameter ke fungsi yang dijalankan sebagai utas yang terpisah. Mengembalikan Hasil ke Thread Induk Setiap thread pekerja ditugaskan tugas menentukan validitas wilayah tertentu dari teka-teki Sudoku. Setelah seorang pekerja melakukan ini

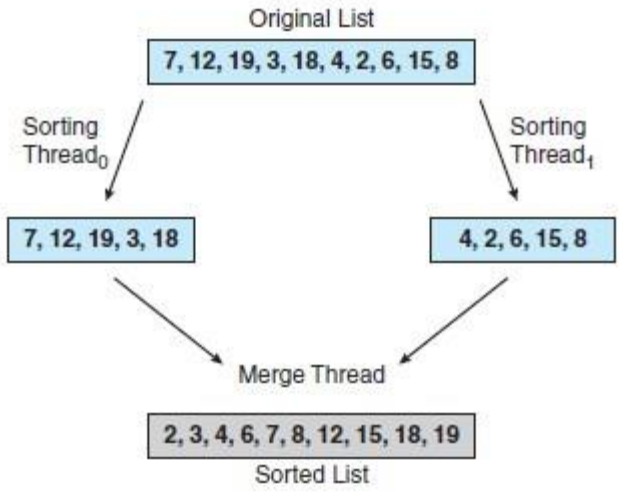


Figure 4.20 Multithreaded sorting.

periksa, ia harus meneruskan hasilnya kembali ke induknya. Salah satu cara yang baik untuk menangani ini adalah dengan membuat array nilai-nilai integer yang terlihat oleh setiap utas. Indeks in dalam array ini berhubungan dengan thread pekerja engan. Jika seorang pekerja menetapkan nilai yang sesuai ke 1, itu menunjukkan bahwa wilayahnya dari teka-teki Sudoku valid. Nilai 0 akan menunjukkan sebaliknya. Ketika semua utas pekerja telah selesai, utas induk memeriksa setiap entri dalam larik hasil untuk menentukan apakah teka-teki Sudoku valid.

Project 2 — Aplikasi Sorting Multithreaded

Tulis program sorting multithreaded yang berfungsi sebagai berikut: Daftar bilangan bulat dibagi menjadi dua daftar yang lebih kecil dengan ukuran yang sama. Dua utas yang terpisah (yang akan kita sebut sortir utas), sortir setiap sublist menggunakan algoritme pengurutan pilihan Anda. Kedua sublist kemudian digabung dengan thread ketiga — thread yang menggabungkan — yang menggabungkan dua sublist menjadi satu daftar yang diurutkan. Karena data global dibagi lintas semua utas, mungkin cara termudah untuk mengatur data adalah membuat larik global. Setiap thread penyortiran akan bekerja pada satu setengah dari array ini. Array global kedua dengan ukuran yang sama dengan array integer yang tidak disortir juga akan dibuat. Benang penggabungan kemudian akan menggabungkan dua subliter ke dalam larik kedua ini. Secara grafis, program ini terstruktur sesuai Gambar 4.20. Proyek pemrograman ini akan membutuhkan parameter yang melewati ke masing-masing thread penyortiran. Secara khusus, perlu untuk mengidentifikasi indeks awal dari mana setiap utas mulai memilah. Lihat instruksi di Project 1 untuk detail tentang mengirimkan parameter ke utas. Thread induk akan menampilkan array yang diurutkan setelah semua string penyortiran keluar.

Bibliographical Notes

Threads have had a long evolution, starting as “cheap concurrency” in programming languages and moving to “lightweight processes,” with early examples that included the Thoth system ([Cheriton et al. (1979)]) and the Pilot 200 Chapter 4 Threads system ([Redell et al. (1980)]). [Binding (1985)] described moving threads into the UNIX kernel. Mach ([Accetta et al. (1986)], [Tevanian et al. (1987)]), and V ([Cheriton (1988)]) made extensive use of threads, and eventually almost all major operating systems implemented them in some form or another.

[Vahalia (1996)] covers threading in several versions of UNIX. [McDougall and Mauro (2007)] describes developments in threading the Solaris kernel. [Russinovich and Solomon (2009)] discuss threading in the Windows operating system family. [Mauerer (2008)] and [Love (2010)] explain how Linux handles threading, and [Singh (2007)] covers threads in Mac OS X. Information on Pthreads programming is given in [Lewis and Berg (1998)] and [Butenhof (1997)]. [Oaks and Wong (1999)] and [Lewis and Berg (2000)] discuss multithreading in Java. [Goetz et al. (2006)] present a detailed discussion of concurrent programming in Java. [Hart (2005)] describes multithreading using Windows. Details on using OpenMP can be found at http://openmp.org. An analysis of an optimal thread-pool size can be found in [Ling et al. (2000)]. Scheduler activations were first presented in [Anderson et al. (1991)], and [Williams (2002)] discusses scheduler activations in the NetBSD system. [Breshears (2009)] and [Pacheco (2011)] cover parallel programming in detail. [Hill and Marty (2008)] examine Amdahl's Law with respect to multicore systems. The Monte Carlo technique for estimating ! is further discussed in http://math.fullerton.edu/mathews/n2003/montecarlopimod.html.

Bibliography

[Accetta et al. (1986)] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the Summer USENIX Conference (1986), pages 93–112.

[Anderson et al. (1991)] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", Proceedings of the ACM Symposium on Operating Systems Principles (1991), pages 95–109.
[Binding (1985)] C. Binding, "Cheap Concurrency in C", SIGPLAN Notices, Volume 20, Number 9 (1985), pages 21–27.

[Breshears (2009)] C. Breshears, The Art of Concurrency, O'Reilly & Associates (2009).

[Butenhof (1997)] D. Butenhof, Programming with POSIX Threads, AddisonWesley (1997).

[Cheriton (1988)] D. Cheriton, "The V Distributed System", Communications of the ACM, Volume 31, Number 3 (1988), pages 314–333.

[Cheriton et al. (1979)] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager, "Thoth, a Portable Real-Time Operating System", Communications of the ACM, Volume 22, Number 2 (1979), pages 105–115.

[Goetz et al. (2006)] B. Goetz, T. Peirls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, Java Concurrency in Practice, Addison-Wesley (2006).

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

[Hill and Marty (2008)] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era", IEEE Computer, Volume 41, Number 7 (2008), pages 33–38.

[Lewis and Berg (1998)] B. Lewis and D. Berg, Multithreaded Programming with Pthreads, Sun Microsystems Press (1998).

[Lewis and Berg (2000)] B. Lewis and D. Berg, Multithreaded Programming with Java Technology, Sun Microsystems Press (2000).

[Ling et al. (2000)] Y. Ling, T. Mullen, and X. Lin, “Analysis of Optimal Thread Pool Size”, Operating System Review, Volume 34, Number 2 (2000), pages 42–55.

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

[Mauerer (2008)] W. Mauerer, Professional Linux Kernel Architecture, John Wiley and Sons (2008).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, Solaris Internals, Second Edition, Prentice Hall (2007).

[Oaks and Wong (1999)] S. Oaks and H. Wong, Java Threads, Second Edition, O’Reilly & Associates (1999).

[Pacheco (2011)] P. S. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann (2011).

[Redell et al. (1980)] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. P. Purcell, “Pilot: An Operating System for a Personal Computer”, Communications of the ACM, Volume 23, Number 2 (1980), pages 81–92.
[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). [Singh (2007)] A. Singh, Mac OS X Internals: A Systems Approach, AddisonWesley (2007).

[Tevanian et al. (1987)] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, “Mach Threads and the Unix Kernel: The Battle for Control”, Proceedings of the Summer USENIX Conference (1987).

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

[Williams (2002)] N. Williams, “An Implementation of Scheduler Activations on the NetBSD Operating System”, 2002 USENIX Annual Technical Conference, FREENIX Track (2002).