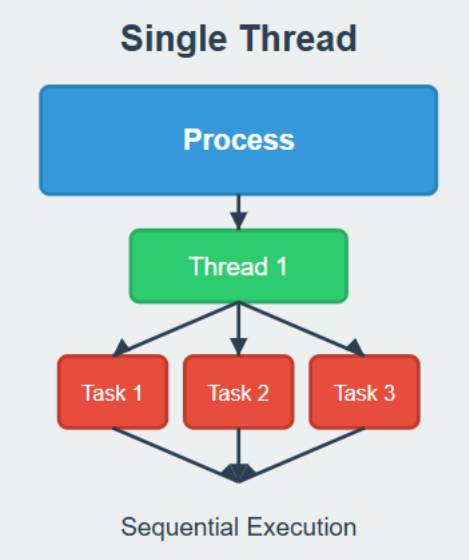
1. **Jelaskan dalam 2 paragraph disertai dengan gambar tentang konsep single thread dan multithread ! ada pada materi ch4.**
   Jawab :

   a. **Single Thread** adalah model eksekusi di mana sebuah proses hanya memiliki satu alur eksekusi (thread) yang menjalankan instruksi secara berurutan. Dalam single thread, hanya satu tugas yang dapat diproses pada satu waktu, sehingga jika terjadi blocking (misalnya menunggu I/O), seluruh proses akan terhenti hingga tugas tersebut selesai. Contohnya adalah program sederhana yang membaca input, memprosesnya, lalu menampilkan output tanpa kemampuan untuk menangani tugas lain secara bersamaan. Single thread cocok untuk aplikasi yang tidak membutuhkan paralelisme, seperti kalkulator atau skrip kecil.

      Contoh ilustrasi :
      [Proses] → [Thread 1] → Tugas 1 → Tugas 2 → Tugas 3

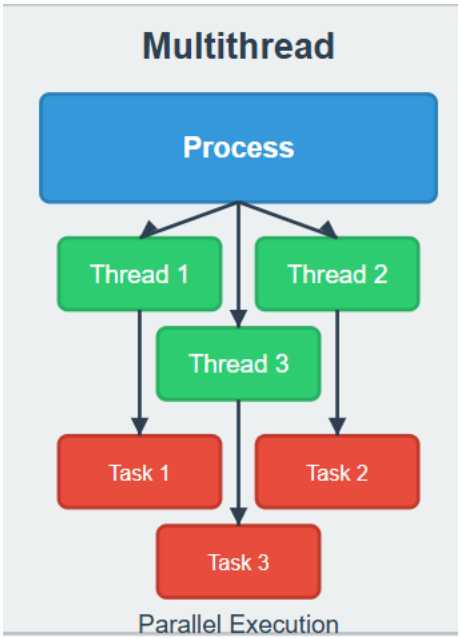


   b. **Multithread** memungkinkan sebuah proses memiliki beberapa thread yang berjalan secara bersamaan dalam ruang alamat memori yang sama. Setiap thread dapat mengeksekusi tugas berbeda secara independen, sehingga meningkatkan responsivitas dan efisiensi. Misalnya, dalam aplikasi web browser, satu thread dapat menangani antarmuka pengguna, sementara thread lain mengunduh data atau memproses konten. Multithread sangat bermanfaat dalam sistem multicore karena memungkinkan paralelisme sejati. Namun, tantangannya adalah sinkronisasi dan manajemen sumber daya bersama untuk menghindari race condition atau deadlock.

      Contoh ilustrasi :
      [Proses] → [Thread 1] → Tugas 1

[Thread 2] → Tugas 2

[Thread 3] → Tugas 3



---

2. Implementasi Threading

Java Fork/Join Framework (SumTask.java)

```
/**
 * Fork/join parallelism in Java
 *
 * Figure 4.18
 *
 * @author Gagne, Galvin, Silberschatz
 * Operating System Concepts  - Tenth Edition
 * Copyright John Wiley & Sons - 2018
 */

import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int SIZE = 10000;
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
```

```
private int[] array;

public SumTask(int begin, int end, int[] array) {
   this.begin = begin;
   this.end = end;
   this.array = array;
}

protected Integer compute() {
   if (end - begin < THRESHOLD) {
      // conquer stage
      int sum = 0;
      for (int i = begin; i <= end; i++)
         sum += array[i];

      return sum;
   }
   else {
      // divide stage
      int mid = begin + (end - begin) / 2;

      SumTask leftTask = new SumTask(begin, mid, array);
      SumTask rightTask = new SumTask(mid + 1, end, array);

      leftTask.fork();
      rightTask.fork();

      return rightTask.join() + leftTask.join();
   }
}

public static void main(String[] args) {
   ForkJoinPool pool = new ForkJoinPool();
   int[] array = new int[SIZE];

   // create SIZE random integers between 0 and 9
   java.util.Random rand = new java.util.Random();

   for (int i = 0; i < SIZE; i++) {
      array[i] = rand.nextInt(10);
   }

   // use fork-join parallelism to sum the array
```

```
        SumTask task = new SumTask(0, SIZE-1, array);

        int sum = pool.invoke(task);

        System.out.println("The sum is " + sum);
    }
}
```

Penjelasan Fork/Join Parallelism pada SumTask.java

Kode SumTask.java menggunakan Fork/Join Framework untuk menerapkan multithreading dalam menghitung jumlah seluruh elemen dari sebuah array. Pendekatan ini membagi tugas besar menjadi beberapa tugas kecil (divide and conquer), lalu menjalankannya secara paralel menggunakan thread.

Tahapan Kerja:

Divide:

Jika panjang array dari indeks begin ke end lebih besar dari THRESHOLD (1000), tugas dibagi dua: bagian kiri (begin ke mid) dan kanan (mid+1 ke end).
Dua sub-tugas baru dibuat: leftTask dan rightTask.
Fork:

Kedua tugas tersebut dijalankan secara paralel dengan memanggil .fork(). Ini memberi tahu ForkJoinPool agar mengeksekusi sub-task di thread terpisah.
Join:

Setelah fork, hasil sub-tugas diambil dengan .join() dan dijumlahkan untuk memberikan hasil akhir.
Conquer:

Jika ukuran array lebih kecil dari THRESHOLD, penjumlahan dilakukan secara langsung (tanpa pembagian lebih lanjut).
Konteks Threading:

Kelas SumTask mewarisi RecursiveTask<Integer> — artinya, ini adalah tugas yang akan mengembalikan nilai bertipe Integer.
ForkJoinPool secara otomatis mengatur thread pool dan membagi tugas ke berbagai worker thread di balik layar.
Kesimpulan: Dengan ForkJoinPool, program ini dapat memanfaatkan banyak inti CPU untuk memproses array besar secara efisien. Ini adalah bentuk multithreaded programming tingkat tinggi yang mudah dikelola dan scalable di Java.

POSIX Threading di Linux (thrd-posix.c)

```
/**
 * A pthread program illustrating how to
 * create a simple thread and some of the pthread API
 * This program implements the summation function where
 * the summation operation is run as a separate thread.
 *
 * Most Unix/Linux/OS X users
 * gcc thrd.c -lpthread
 *
 * Figure 4.11
 *
 * @author Gagne, Galvin, Silberschatz
 * Operating System Concepts  - Tenth Edition
 * Copyright John Wiley & Sons - 2018
 */

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

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of attributes for the thread */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      /*exit(1);*/
      return -1;
   }

   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
      /*exit(1);*/
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
```

```
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);

   /* now 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;

   if (upper > 0) {
      for (i = 1; i <= upper; i++)
         sum += i;
   }

   pthread_exit(0);
}
```

Penjelasan Penerapan Thread di Linux (thrd-posix.c)
Kode ini merupakan contoh sederhana penggunaan POSIX thread untuk menjalankan operasi penjumlahan angka dari 1 hingga n di dalam thread terpisah. Tujuannya adalah untuk menjalankan fungsi tertentu di thread yang berbeda dari thread utama (main thread).

Struktur Program:

Deklarasi Thread & Atribut:

pthread_t tid; digunakan untuk menyimpan ID dari thread yang akan dibuat.
pthread_attr_t attr; menyimpan atribut default dari thread.
Validasi Input:

Program menerima satu argumen berupa angka.
Dicek apakah angka valid (bukan negatif), lalu dikonversi ke integer.
Inisialisasi & Pembuatan Thread:

pthread_attr_init(&attr); mengambil atribut default.

pthread_create(&tid, &attr, runner, argv[1]); membuat thread baru yang menjalankan fungsi runner().
Sinkronisasi:

pthread_join(tid, NULL); menunggu sampai thread selesai dieksekusi sebelum melanjutkan ke bagian printf.
Fungsi runner():

Fungsi inilah yang dijalankan dalam thread terpisah.
Mengonversi parameter menjadi int, lalu menjumlahkan bilangan dari 1 sampai n dan menyimpannya di variabel global sum.
Konteks Threading:

Ini adalah multithreading manual: programmer harus menangani atribut, membuat thread, dan menyinkronkannya dengan join.
Variabel sum bersifat shared memory, artinya dapat diakses oleh thread manapun — karena itu, dalam kasus yang kompleks perlu proteksi (misalnya mutex), tapi tidak diperlukan di contoh ini karena hanya satu thread bekerja dengan sum.
Kesimpulan: Program ini menunjukkan bagaimana thread dapat digunakan untuk menjalankan tugas tambahan secara paralel dengan thread utama, meskipun dalam contoh ini thread utama hanya menunggu hasil dari thread runner(). Ini adalah dasar dari multithreading berbasis POSIX standard di Linux.

Win32 Threading di Windows (thrd-win32.c)

```
/**
 * This program creates a separate thread using the CreateThread() system call.
 *
 * Figure 4.13
 *
 * @author Gagne, Galvin, Silberschatz
 * Operating System Concepts  - Tenth Edition
 * Copyright John Wiley & Sons - 2018
 */

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


DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(PVOID 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;

    // do some basic error checking
    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, 0, Summation, &Param, 0, &ThreadId);

    if (ThreadHandle != NULL) {
        WaitForSingleObject(ThreadHandle, INFINITE);
        CloseHandle(ThreadHandle);
        printf("sum = %d\n",Sum);
    }
}
```

Penjelasan Penerapan Thread di Microsoft Windows (thrd-win32.c)
Kode ini menunjukkan bagaimana sistem operasi Windows menangani pembuatan thread menggunakan API khusus Windows, yaitu CreateThread(). Fungsinya mirip dengan pthread_create() di Linux, namun dengan sintaks dan pendekatan yang berbeda.

Struktur Program:

Deklarasi Variabel:

DWORD Sum; adalah variabel global yang menyimpan hasil penjumlahan, bisa diakses oleh semua thread.
DWORD ThreadId; menyimpan ID thread yang dibuat.
HANDLE ThreadHandle; digunakan untuk mengelola thread yang sedang berjalan.
int Param; menyimpan batas atas penjumlahan.
Validasi Input:

Program menerima satu parameter input dari command line.
Dicek apakah input valid dan ≥ 0.
Membuat Thread:

CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId); membuat sebuah thread baru.
Summation() adalah fungsi yang akan dijalankan secara paralel oleh thread ini.
Parameter &Param dikirim ke fungsi Summation() sebagai nilai batas penjumlahan.
Sinkronisasi dan Output:

WaitForSingleObject(ThreadHandle, INFINITE); memastikan thread utama menunggu hingga thread Summation selesai dieksekusi.
Setelah thread selesai, CloseHandle(ThreadHandle); menutup handle untuk melepaskan resource.
Hasil Sum ditampilkan ke layar.
Fungsi Thread Summation():

Fungsi ini akan dijalankan oleh thread baru.
Melakukan penjumlahan dari 0 sampai Upper dan menyimpan hasilnya di variabel Sum.
Konteks Threading:

Pendekatan Windows menggunakan CreateThread() dari Windows API, berbeda dari POSIX.
Fungsi Summation() bertipe DWORD WINAPI dan menerima parameter bertipe PVOID (generic pointer), sesuai konvensi thread di Windows.
Tidak digunakan mutex atau synchronization primitive karena hanya satu thread menulis ke Sum, dan tidak ada race condition.
Kesimpulan: Program ini adalah contoh dasar bagaimana menjalankan sebuah fungsi dalam thread terpisah di Windows. Sistem operasi Windows menyediakan kontrol lebih rendah (low-level) melalui CreateThread(), memungkinkan pemrograman multithread dengan cara manual, dibandingkan pendekatan high-level seperti di Java.

Latihan Soal (PRACTISE EXERCISE)

4.1 Berikan tiga contoh pemrograman di mana multithreading memberikan kinerja yang lebih baik daripada solusi single-threaded.

Jawab :

- **Aplikasi pengolahan gambar** : Ketika mengedit gambar dengan resolusi tinggi, operasi seperti penerapan filter dapat dibagi ke beberapa thread. Setiap thread dapat memproses bagian gambar yang berbeda secara paralel, sehingga mempercepat waktu pemrosesan secara signifikan.
- **Server web** : Server web multithreaded dapat menangani banyak permintaan klien secara bersamaan. Setiap thread dapat memproses permintaan terpisah, sehingga server dapat melayani banyak klien tanpa harus menunggu permintaan sebelumnya selesai diproses.
- **Aplikasi pencarian dan pengindeksan** : Saat mengindeks database besar atau melakukan pencarian di dalamnya, pekerjaan dapat dibagi ke beberapa thread. Misalnya, dalam mesin pencari, satu set thread dapat mencari di bagian tertentu dari indeks, mempercepat waktu respons secara keseluruhan.

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

Jawab :

Menggunakan Hukum Amdahl, menghitung peningkatan kecepatan:

Hukum Amdahl: $S = 1/((1-p) + p/n)$

Dimana:

S = speedup (peningkatan kecepatan)

p = komponen paralel (60% = 0,6)

n = jumlah core

a) Untuk dua core pemrosesan:

$S = 1/((1-0,6) + 0,6/2)$

$S = 1/(0,4 + 0,3)$

$S = 1/0,7$

$S = 1,43x$

b) Untuk empat core pemrosesan:

$S = 1/((1-0,6) + 0,6/4)$

$S = 1/(0,4 + 0,15)$

$S = 1/0,55$

$S = 1,82x$

4.3 Apakah server web multithread yang dijelaskan dalam Bagian 4.1 menunjukkan paralelisme tugas atau data?

Jawab : Web server multithreaded yang dijelaskan di bagian 4.1 menunjukkan paralelisme task. Ini karena setiap thread menangani permintaan klien yang berbeda dan independen (tugas berbeda), bukan memproses bagian-bagian dari permintaan yang sama secara paralel. Setiap thread menjalankan tugas lengkap dan terpisah dari awal hingga akhir.

4.4 Apa dua perbedaan antara thread tingkat pengguna dan thread tingkat kernel? Dalam keadaan apa salah satu jenis thread lebih baik daripada thread lainnya?

Jawab :

Perbedaan antara thread tingkat pengguna dan thread tingkat kernel:

- **Manajemen** : Thread tingkat pengguna dikelola oleh pustaka thread di ruang pengguna, sedangkan thread tingkat kernel dikelola langsung oleh sistem operasi.
- **Overhead** : Thread tingkat pengguna memiliki overhead lebih rendah karena tidak memerlukan mode kernel untuk operasi thread, sementara thread tingkat kernel memiliki overhead lebih tinggi karena membutuhkan perubahan ke mode kernel.

Thread tingkat pengguna lebih baik saat:

- Aplikasi membutuhkan banyak thread dengan switching yang sering
- Portabilitas antar sistem operasi diperlukan

Thread tingkat kernel lebih baik saat:

- Aplikasi membutuhkan paralelisme nyata pada sistem multiprosesor
- Thread perlu melakukan banyak panggilan sistem yang memblokir
- Kebutuhan akan responsivitas real-time tinggi

4.5 Jelaskan tindakan yang diambil oleh kernel untuk beralih konteks antara thread tingkat kernel.

Jawab :

1. Menyimpan state thread saat ini (register, program counter, stack pointer)
2. Memperbarui blok kontrol thread (TCB) thread saat ini
3. Menempatkan thread saat ini pada antrian yang sesuai (siap, terblokir, dll)
4. Memilih thread berikutnya untuk dijalankan berdasarkan algoritma penjadwalan
5. Memperbarui status thread terpilih dalam TCB-nya
6. Memuat konteks (state) thread terpilih ke dalam CPU
7. Melanjutkan eksekusi thread terpilih

4.6 Sumber daya apa yang digunakan saat thread dibuat? Apa perbedaannya dengan sumber daya yang digunakan saat proses dibuat?

Jawab :

Sumber daya untuk thread:

- Thread Control Block (TCB)
- Stack terpisah
- Program counter dan register set
- Ruang penyimpanan thread-spesifik

Sumber daya untuk proses:

- Process Control Block (PCB)
- Ruang alamat memori terpisah

- Stack, heap, dan segmen data
- Deskriptor file dan handle sumber daya lainnya
- Informasi akuntansi dan keamanan

Perbedaan utama: Thread berbagi ruang alamat, kode program, data, file terbuka, dan sumber daya sistem lainnya dengan thread lain dalam proses yang sama. Proses memiliki sumber daya tersebut secara terpisah dan independen. Oleh karena itu, pembuatan proses membutuhkan lebih banyak sumber daya dan waktu dibandingkan pembuatan thread.

4.7 Asumsikan bahwa sistem operasi memetakan thread tingkat pengguna ke kernel menggunakan model many-to-many dan pemetaan dilakukan melalui LWP. Lebih jauh, sistem memungkinkan pengembang untuk membuat thread real-time untuk digunakan dalam sistem real-time. Apakah perlu mengikat thread real-time ke LWP? Jelaskan.
Jawab : Ya, dalam kasus ini, perlu untuk mengikat thread real-time ke Lightweight Process (LWP) tertentu. Alasannya:

1. **Prediktabilitas**: Sistem real-time membutuhkan perilaku yang dapat diprediksi dan deterministik. Tanpa pengikatan, thread real-time dapat dipindahkan antar LWP yang berbeda oleh penjadwal, mengganggu jaminan waktu.
2. **Prioritas**: Thread real-time biasanya berjalan dengan prioritas tinggi. Mengikatnya ke LWP tertentu memastikan prioritas ini ditegakkan secara konsisten.
3. **Isolasi sumber daya**: Pengikatan memungkinkan thread real-time untuk menghindari persaingan untuk LWP dengan thread non-real-time, mengurangi jitter dan variasi latensi.
4. **Afinitas CPU**: Dalam sistem multiprosesor, mengikat thread real-time ke LWP juga dapat memungkinkan afinitas CPU, yang dapat meningkatkan kinerja cache dan mengurangi overhead komunikasi.

Dengan model pemetaan many-to-many, tanpa pengikatan khusus, tidak ada jaminan bahwa thread real-time akan selalu memiliki LWP tersedia ketika dibutuhkan, yang bisa menjadi masalah kritis untuk aplikasi real-time.

4.8 Berikan dua contoh pemrograman di mana multithreading tidak memberikan kinerja yang lebih baik daripada solusi single-threaded.
Jawab :

1. **Algoritma sekuensial murni**: Program seperti perhitungan Fibonacci rekursif sederhana yang memiliki ketergantungan ketat pada hasil sebelumnya tidak dapat dibagi menjadi tugas paralel yang efektif. Overhead pembuatan dan manajemen thread justru akan menurunkan kinerja dibandingkan solusi single-thread.
2. **Aplikasi dengan I/O terbatas**: Program yang sebagian besar waktunya dihabiskan untuk menunggu operasi I/O (seperti membaca file kecil dari disk) tidak akan mendapatkan keuntungan signifikan dari multithreading, karena bottleneck utamanya bukan pada pemrosesan CPU, melainkan pada kecepatan perangkat I/O.

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

Jawab : Solusi multithreaded dengan thread kernel ganda dapat memberikan kinerja lebih baik pada sistem single-processor dalam kondisi:

1. **Operasi I/O intensif**: Ketika satu thread menunggu operasi I/O selesai, thread lain dapat menggunakan CPU, meningkatkan utilisasi CPU secara keseluruhan.
2. **Blocking system calls**: Saat satu thread terblokir oleh panggilan sistem, thread lain dapat tetap berjalan.
3. **Variasi dalam latensi cache**: Thread yang berbeda mungkin mengalami pola cache miss yang berbeda, sehingga memungkinkan CPU untuk tetap bekerja ketika satu thread mengalami cache miss.
4. **Prioritas yang berbeda**: Thread dengan prioritas tinggi untuk tugas kritis dapat berjalan sesuai kebutuhan, sementara thread dengan prioritas lebih rendah berjalan ketika CPU tersedia.

4.10 Komponen status program berikut manakah yang dibagikan di seluruh thread dalam proses multithreaded?

a. Nilai register
b. Memori heap
c. Variabel global
d. Memori stack

Jawab :

b. Heap memory - Thread berbagi akses ke heap memory dalam proses yang sama

c. Global variables - Variable global dapat diakses dan dimodifikasi oleh semua thread dalam proses

(Register values dan stack memory biasanya unik untuk setiap thread)

4.11 Dapatkah solusi multithreaded yang menggunakan beberapa thread tingkat pengguna mencapai kinerja yang lebih baik pada sistem multiprosesor daripada pada sistem prosesor tunggal? Jelaskan.

Jawab : Tidak, solusi multithreaded yang hanya menggunakan thread tingkat pengguna tidak dapat mencapai kinerja yang lebih baik pada sistem multiprosesor dibandingkan pada sistem single-processor. Alasannya:

1. Thread tingkat pengguna semuanya dipetakan ke satu thread kernel (dalam model many-to-one), sehingga hanya dapat berjalan pada satu prosesor pada satu waktu.
2. Sistem operasi hanya melihat satu thread kernel dan menjadwalkannya untuk berjalan pada satu prosesor, tidak peduli berapa banyak thread tingkat pengguna yang ada di dalamnya.
3. Untuk memanfaatkan beberapa prosesor secara simultan, dibutuhkan beberapa thread kernel yang dapat dijadwalkan pada prosesor yang berbeda secara bersamaan (model one-to-one atau many-to-many).

4.12 Dalam Bab 3, kita membahas peramban Google Chrome dan praktiknya membuka setiap tab baru dalam proses terpisah. Apakah manfaat yang sama akan diperoleh jika, sebaliknya, Chrome dirancang untuk membuka setiap tab baru dalam thread terpisah? Jelaskan.
Jawab : Jika Chrome dirancang untuk membuka setiap tab baru dalam thread terpisah (bukan proses terpisah), manfaat berikut tidak akan tercapai:

1. **Isolasi keamanan**: Thread dalam proses yang sama berbagi ruang alamat dan sumber daya, sehingga eksploitasi keamanan di satu tab berpotensi memengaruhi tab lain.
2. **Stabilitas sistem**: Jika satu thread mengalami crash, kemungkinan besar akan menyebabkan seluruh proses browser crash, tidak hanya tab yang bermasalah.
3. **Pengelolaan sumber daya**: Sulit untuk melacak dan membatasi penggunaan memori dan CPU oleh masing-masing tab jika semuanya berjalan sebagai thread dalam proses yang sama.
4. **Pembersihan memori**: Menutup tab tidak akan memastikan semua sumber daya yang terkait dibersihkan dengan benar, karena adanya potensi kebocoran memori dari thread yang berbagi heap.

Pendekatan proses terpisah Chrome menghasilkan isolasi yang jauh lebih baik dengan biaya sedikit peningkatan penggunaan memori, yang menghasilkan browser yang lebih stabil dan aman.

4.13 Apakah mungkin untuk memiliki konkurensi tetapi tidak paralelisme? Jelaskan.

Jawab : Ya, sangat mungkin untuk memiliki konkurensi tanpa paralelisme. Konkurensi mengacu pada kemampuan untuk menangani beberapa tugas yang tumpang tindih dalam waktu, sementara paralelisme adalah eksekusi tugas secara bersamaan.

Contoh konkurensi tanpa paralelisme:

1. **Time-sharing pada sistem single-core**: Sistem operasi dapat menjalankan banyak thread secara bergantian dengan cepat, memberikan ilusi konkurensi, meskipun pada kenyataannya hanya satu thread yang berjalan pada satu waktu.
2. **Event loop**: Dalam pemrograman berbasis peristiwa (seperti JavaScript di browser), satu thread dapat menangani beberapa tugas asinkron secara bergantian tanpa paralelisme sejati.
3. **Cooperative multitasking**: Thread secara sukarela melepaskan kontrol CPU, memungkinkan konkurensi tetapi tanpa eksekusi paralel.

Konkurensi berfokus pada struktur program dan kemampuan untuk menangani beberapa tugas, sedangkan paralelisme berfokus pada eksekusi simultan.

4.14 Dengan menggunakan Hukum Amdahl, hitung perolehan percepatan untuk aplikasi berikut:

- 40 persen paralel dengan (a) delapan inti pemrosesan dan (b) enam belas inti pemrosesan

- 67 persen paralel dengan (a) dua inti pemrosesan dan (b) empat inti pemrosesan
- 90 persen paralel dengan (a) empat inti pemrosesan dan (b) delapan inti pemrosesan

Jawab : Menggunakan rumus: S = 1/((1-p) + p/n), di mana p adalah bagian paralel dan n adalah jumlah core.

- **40 persen paralel:**

  a) Dengan delapan core: S = 1/((1-0,4) + 0,4/8) S = 1/(0,6 + 0,05) S = 1/0,65 S = 1,54x
  b) Dengan enam belas core: S = 1/((1-0,4) + 0,4/16) S = 1/(0,6 + 0,025) S = 1/0,625 S = 1,6x

- **67 persen paralel:**

  a) Dengan dua core: S = 1/((1-0,67) + 0,67/2) S = 1/(0,33 + 0,335) S = 1/0,665 S = 1,5x
  b) Dengan empat core: S = 1/((1-0,67) + 0,67/4) S = 1/(0,33 + 0,1675) S = 1/0,4975 S = 2,01x

- **90 persen paralel:**

  a) Dengan empat core: S = 1/((1-0,9) + 0,9/4) S = 1/(0,1 + 0,225) S = 1/0,325 S = 3,08x
  b) Dengan delapan core: S = 1/((1-0,9) + 0,9/8) S = 1/(0,1 + 0,1125) S = 1/0,2125 S = 4,71x

4.15 Tentukan apakah masalah berikut menunjukkan paralelisme tugas atau data :

- Menggunakan utas terpisah untuk membuat gambar mini untuk setiap foto dalam koleksi
- Mentransposisi matriks secara paralel
- Aplikasi jaringan tempat satu utas membaca dari jaringan dan utas lainnya menulis ke jaringan
- Aplikasi penjumlahan array fork-join yang dijelaskan dalam Bagian 4.5.2
- Sistem Grand Central Dispatch

  Jawab :

  1. Menggunakan utas terpisah untuk membuat gambar mini (thumbnail) untuk setiap foto dalam koleksi

     Paralelisme data: Ini adalah contoh paralelisme data karena operasi yang sama (pembuatan thumbnail) diterapkan secara paralel pada elemen data yang berbeda (foto yang berbeda). Setiap thread melakukan fungsi identik tetapi pada bagian data yang berbeda dari koleksi foto.
  2. Mentransposisi matriks secara parallel

     Paralelisme data: Dalam transposi matriks paralel, operasi yang sama (pertukaran elemen) diterapkan pada bagian-bagian matriks yang berbeda secara bersamaan. Setiap thread melakukan proses yang identik pada bagian data yang berbeda.

3. Aplikasi jaringan tempat satu utas membaca dari jaringan dan utas lainnya menulis ke jaringan
   Paralelisme tugas: Ini adalah contoh paralelisme tugas karena setiap thread melakukan fungsi yang berbeda (membaca vs menulis). Thread-thread ini menjalankan operasi yang berbeda dan independen satu sama lain, meskipun mereka mungkin bekerja pada aliran data yang sama.
4. Aplikasi penjumlahan array fork-join yang dijelaskan dalam Bagian 4.5.2
   Paralelisme data: Dalam penjumlahan array menggunakan paradigma fork-join, array dibagi menjadi beberapa segmen dan setiap thread melakukan operasi penjumlahan yang sama pada segmen yang berbeda. Kemudian hasil dari setiap segmen digabungkan untuk mendapatkan hasil akhir.
5. Sistem Grand Central Dispatch
   Paralelisme tugas: Grand Central Dispatch (GCD) adalah teknologi yang mendistribusikan tugas-tugas berbeda ke thread yang tersedia dalam thread pool. GCD berfokus pada eksekusi berbagai tugas independen secara paralel, bukan pada pemrosesan paralel dari set data tunggal. Ini memungkinkan pengembang untuk menentukan tugas-tugas yang perlu dilakukan tanpa harus khawatir tentang pembuatan dan manajemen thread secara eksplisit.

4.16 Sistem dengan dua prosesor inti ganda memiliki empat prosesor yang tersedia untuk penjadwalan. Aplikasi yang menggunakan CPU secara intensif sedang berjalan pada sistem ini. Semua input dilakukan pada saat program dimulai, ketika satu file harus dibuka. Demikian pula, semua keluaran dilakukan sebelum program berakhir, saat hasil program harus ditulis ke satu berkas. Antara awal dan akhir, program sepenuhnya terikat CPU. Tugas Anda adalah meningkatkan kinerja aplikasi ini dengan melakukan multithreading. Aplikasi berjalan pada sistem yang menggunakan model threading satu-ke-satu (setiap thread pengguna memetakan ke thread kernel).

- Berapa banyak thread yang akan Anda buat untuk melakukan input dan output? Jelaskan.

  Jawab : Saya akan membuat 1 thread untuk menangani operasi input dan output. Alasannya:

  - Operasi I/O hanya terjadi di awal (membaca file input) dan di akhir program (menulis file output)
  - Operasi ini terjadi secara sekuensial dan terpisah dari bagian pemrosesan CPU-intensif
  - Menggunakan thread terpisah untuk I/O akan memungkinkan thread komputasi utama untuk fokus pada pemrosesan tanpa interupsi

- Berapa banyak thread yang akan Anda buat untuk bagian aplikasi yang menggunakan CPU secara intensif? Jelaskan.

Jawab : Saya akan membuat 4 thread untuk bagian CPU-intensif, sesuai dengan jumlah core yang tersedia (2 prosesor dual-core = 4 core). Alasannya:

- Aplikasi ini CPU-intensif, sehingga akan mendapatkan manfaat dari paralelisme sejati
- Dengan model threading one-to-one, setiap thread pengguna akan dipetakan ke thread kernel
- Membuat thread sebanyak jumlah core akan mengoptimalkan penggunaan sumber daya tanpa overhead konteks switching berlebihan
- Dengan 4 thread komputasi dan 1 thread I/O, semua core akan dimanfaatkan secara efisien selama fase komputasi

4.17 Pertimbangkan segmen kode berikut:

```
apid_t pid;

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

a. Berapa banyak proses unik yang dibuat? Jawab :
   - Awalnya: 1
   - Setelah fork pertama: +1 → 2 proses
   - Di dalam if (pid == 0): fork lagi → +1 → 3 proses
   - fork terakhir dijalankan oleh ketiga proses tersebut → +3 → **Total: 6 proses unik**

b. Berapa banyak thread unik yang dibuat?
   Jawab :
   - Semua proses punya 1 thread default
   - Satu proses (yang menjalankan thread_create(...)) menambahkan 1 thread ekstra jadi, banyak thread yaitu 7 thread (6 proses × 1 thread + 1 thread tambahan)

4.18 Seperti yang dijelaskan dalam Bagian 4.7.2, Linux tidak membedakan antara proses dan thread. Sebaliknya, Linux memperlakukan keduanya dengan cara yang sama, memungkinkan sebuah task menjadi lebih mirip proses atau thread tergantung pada kumpulan flag yang diberikan ke system call clone(). Namun, sistem operasi lain, seperti Windows, memperlakukan proses dan thread secara berbeda. Biasanya, sistem seperti itu menggunakan notasi di mana struktur data untuk sebuah proses berisi pointer ke thread-thread terpisah yang dimiliki oleh proses tersebut. Bandingkan dua pendekatan ini untuk memodelkan proses dan thread di dalam kernel.

Jawab :

Pendekatan Linux:

- Linux menggunakan pendekatan unified atau terpadu, di mana proses dan thread diperlakukan sebagai *task* yang sama.
- Perbedaan antara proses dan thread ditentukan oleh flag yang diberikan ke system call clone().
- Karena itu, Linux tidak memiliki perbedaan mendasar antara proses dan thread pada level kernel.
- Fleksibilitas clone() memungkinkan pembentukan berbagai bentuk hubungan antara task, seperti thread berbagi memori atau proses terpisah.

Pendekatan Windows (dan sistem lainnya):

- Windows membedakan secara eksplisit antara proses dan thread.
- Sebuah proses memiliki ruang alamat sendiri dan sumber daya lainnya.
- Thread adalah unit eksekusi yang hidup di dalam proses, dan semua thread dalam proses yang sama berbagi ruang alamat dan sumber daya.
- Struktur data kernel untuk proses berisi pointer ke semua thread-nya.

Perbandingan:

- Linux lebih fleksibel dan sederhana di level implementasi kernel karena menyatukan keduanya, namun dapat membingungkan bagi pemula.

Windows memberikan pemisahan yang lebih jelas antara proses dan thread, memudahkan pemahaman konseptual, tapi implementasi kernel lebih kompleks.

4.19 Program yang ditunjukkan pada Gambar 4.23 menggunakan API Pthreads. Apa output dari program tersebut pada BARIS C dan BARIS P?

Jawab :

Gambar.4.23

```
int oldstate;

pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);

/* What operations would be performed here? */

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Penjelasan fungsi pthread_setcancelstate:

- Fungsi ini mengatur apakah sebuah thread dapat dibatalkan (cancelled) oleh thread lain.
- PTHREAD_CANCEL_DISABLE: Membuat thread tidak bisa dibatalkan.
- PTHREAD_CANCEL_ENABLE: Membuat thread bisa dibatalkan lagi.
- Argumen kedua (&oldstate) akan menyimpan nilai status pembatalan thread sebelum perubahan, misalnya apakah sebelumnya thread bisa dibatalkan atau tidak.

  Jadi :

- LINE C adalah baris pertama pemanggilan pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
- LINE P adalah baris kedua pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);

  Maka:

- Output di LINE C: nilai oldstate berisi status pembatalan sebelum DISABLE, yang biasanya ENABLE secara default. Jadi oldstate = PTHREAD_CANCEL_ENABLE.
- Output di LINE P: nilai oldstate sekarang adalah PTHREAD_CANCEL_DISABLE, karena status sebelumnya dinonaktifkan oleh LINE C.

4.20 Pertimbangkan sebuah sistem multicore dan program multithread yang ditulis menggunakan model many-to-many threading. Misalkan jumlah user-level thread dalam program lebih banyak daripada jumlah inti pemrosesan (core) dalam sistem. Diskusikan implikasi performa dari skenario berikut:

a. Jumlah kernel thread yang dialokasikan untuk program lebih sedikit daripada jumlah inti pemrosesan.
b. Jumlah kernel thread yang dialokasikan untuk program sama dengan jumlah inti pemrosesan.
c. Jumlah kernel thread yang dialokasikan untuk program lebih banyak dari jumlah inti pemrosesan tetapi masih lebih sedikit dari jumlah user-level thread.

Jawab :

a. Kernel thread < Jumlah core

Implikasi:

- Tidak semua core bisa digunakan secara optimal karena ada lebih sedikit kernel thread yang bisa dijadwalkan ke core.
- Underutilization (pemanfaatan kurang) dari hardware terjadi.
- Beberapa user-level thread harus menunggu lebih lama untuk mendapatkan akses ke kernel thread.

- Performa menurun karena tidak memanfaatkan seluruh kapasitas paralelisme dari prosesor.

b. Kernel thread = Jumlah core

Implikasi:

- Ini adalah situasi ideal dalam banyak kasus.
- Setiap kernel thread bisa berjalan pada satu core secara paralel tanpa kompetisi antar thread di level kernel.
- Penjadwalan menjadi lebih efisien dan meminimalisasi context switching.
- Performa optimal bisa dicapai asalkan tidak terlalu banyak blocking atau operasi I/O.

c. Kernel thread > Jumlah core, tetapi < jumlah user thread

Implikasi:

- Lebih banyak kernel thread daripada core berarti akan terjadi konteks switching antar kernel thread di core yang sama.
- Namun, karena masih lebih sedikit dari jumlah user thread, tetap saja ada beberapa user thread yang harus menunggu untuk bisa dieksekusi.
- Sistem jadi lebih fleksibel dibanding skenario (a), karena lebih banyak user thread bisa dijalankan secara paralel.
- Performa lebih baik dari (a), tapi tidak sebaik (b)—karena adanya overhead switching dan antrian thread.

4.21 Pthreads menyediakan API untuk mengelola pembatalan thread. Fungsi pthread_setcancelstate() digunakan untuk mengatur status pembatalan. Prototipe fungsi tersebut adalah sebagai berikut:
pthread_setcancelstate(int state, int *oldstate)
Dua nilai yang mungkin untuk parameter state adalah:
PTHREAD_CANCEL_ENABLE dan PTHREAD_CANCEL_DISABLE
Dengan menggunakan potongan kode yang ditunjukkan pada Gambar 4.24, berikan dua contoh operasi yang sesuai untuk dilakukan di antara pemanggilan fungsi untuk menonaktifkan dan mengaktifkan kembali pembatalan thread.
Jawab :

1. Penulisan ke File atau Database (I/O sensitif):

Misalnya, menyimpan data penting yang tidak boleh terganggu karena jika pembatalan terjadi di tengah proses, file atau database bisa rusak:

```
fwrite(data, sizeof(char), data_size, file_pointer);
```

2. Alokasi atau Dealokasi Memori Kritis:

Saat mengalokasikan atau membebaskan memori, kita tidak ingin thread dibatalkan di tengah proses, karena bisa menyebabkan memory leak atau kebocoran sumber daya:

```
free(buffer);
```

4.22 Tulis program multithreaded yang menghitung berbagai nilai statistik dari sebuah daftar angka. Program ini akan menerima sekumpulan angka melalui baris perintah (command line), lalu membuat tiga thread pekerja (worker threads) yang terpisah.Thread pertama menghitung rata-rata dari angka-angka tersebut.Thread kedua menentukan nilai maksimum.Thread ketiga menentukan nilai minimum.
90 81 78 95 79 72 85
Nilai rata-rata adalah 82
Nilai minimum adalah 72
Nilai maksimum adalah 95
Variabel yang menyimpan nilai rata-rata, minimum, dan maksimum disimpan secara global. Thread pekerja akan mengatur nilai-nilai ini, dan thread utama (parent thread) akan mencetak hasilnya setelah semua worker selesai dieksekusi.
Jawab :

- Gunakan array global untuk menyimpan angka dari input.
- Buat tiga thread:

Thread 1: menghitung rata-rata.
Thread 2: mencari nilai maksimum.
Thread 3: mencari nilai minimum.

- Setelah semua thread selesai (pthread_join), cetak semua nilai.
- Gunakan variabel global untuk hasilnya :

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

int numbers[100];
int count;
int average, min, max;

void* find_average(void* arg) {
    int sum = 0;
    for (int i = 0; i < count; i++) sum += numbers[i];
    average = sum / count;
    pthread_exit(0);
}
```

```
void* find_min(void* arg) {
    min = numbers[0];
    for (int i = 1; i < count; i++)
        if (numbers[i] < min) min = numbers[i];
    pthread_exit(0);
}

void* find_max(void* arg) {
    max = numbers[0];
    for (int i = 1; i < count; i++)
        if (numbers[i] > max) max = numbers[i];
    pthread_exit(0);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Usage: %s <list of integers>\n", argv[0]);
        return -1;
    }

    // Simpan angka dari command line ke array numbers
    count = argc - 1;
    for (int i = 0; i < count; i++) {
        numbers[i] = atoi(argv[i + 1]);
    }

    // Buat thread
    pthread_t tid1, tid2, tid3;

    pthread_create(&tid1, NULL, find_average, NULL);
    pthread_create(&tid2, NULL, find_min, NULL);
    pthread_create(&tid3, NULL, find_max, NULL);

    // Tunggu semua thread selesai
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    // Cetak hasil
    printf("The average value is %d\n", average);
    printf("The minimum value is %d\n", min);
    printf("The maximum value is %d\n", max);

    return 0;
}
```

4.23 Tulis program multithreaded yang mencetak bilangan prima. Cara kerjanya: Pengguna menjalankan program dan memasukkan sebuah angka lewat command line.Program akan membuat satu thread yang mencetak semua bilangan prima yang kurang dari atau sama dengan angka yang dimasukkan.
Jawab :

- Buat thread baru yang menerima angka sebagai parameter.
- Di dalam thread, loop dari 2 hingga angka tersebut dan cek apakah bilangan tersebut prima.
- Cetak semua bilangan prima yang ditemukan.

```
void* print_primes(void* arg) {
    int num = *(int*)arg;
    for (int i = 2; i <= num; i++) {
        int is_prime = 1;
        for (int j = 2; j <= i/2; j++) {
            if (i % j == 0) {
                is_prime = 0;
                break;
            }
        }
        if (is_prime)
            printf("%d ", i);
    }
    pthread_exit(0);
}
```

4.24 Teknik Monte Carlo untuk Menghitung π
Cara menarik untuk menghitung π adalah dengan menggunakan teknik yang dikenal sebagai *Monte Carlo*, yang memanfaatkan pengacakan. Teknik ini bekerja sebagai berikut: Misalkan Anda memiliki sebuah lingkaran yang terinskripsi di dalam sebuah persegi (lihat Gambar 4.25). Anggap jari-jari lingkaran ini = 1.

1. Hasilkan sejumlah titik acak sebagai koordinat (x, y). Titik-titik ini harus terletak di dalam batas persegi, yaitu $-1 \leq x \leq 1$ dan $-1 \leq y \leq 1$. Dari keseluruhan titik acak yang dihasilkan, sebagian akan jatuh di dalam lingkaran.

2. Taksir nilai π dengan rumus:

   π = 4x (jumlah titik dalam lingkaran) / (total titik)

   Buatlah versi *multithreaded* dari algoritma di atas, dengan ketentuan: Setiap *thread* bertanggung-jawab menghasilkan sejumlah titik acak dan menghitung berapa banyak di antaranya yang jatuh di dalam lingkaran. Hasil hitung tiap *thread* (jumlah titik dalam lingkaran) disimpan dalam variabel global bersama. Setelah semua *thread* selesai, *thread*

induk (“parent”) akan mengumpulkan hasil-hasil tersebut, menghitung taksiran π, dan menampilkannya. Silakan bereksperimen dengan berbagai jumlah titik acak: secara umum, semakin banyak titik, semakin dekat taksiran nilai π.

Gambar 4.25. Teknik Monte Carlo untuk menghitung π
(Lingkaran biru berjari-jari 1 di dalam persegi berkoordinat (–1,–1) sampai (1,1). Titik pusat di (0,0).)

Jawab :

contoh implementasi C menggunakan pthread untuk menghitung π dengan metode Monte Carlo secara multithreaded.
Program menerima dua argumen:

1. Jumlah thread
2. Total titik (akan dibagi merata ke tiap thread)

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

// Variabel global
long long total_in_circle = 0;
long long points_per_thread;
pthread_mutex_t lock;

// Fungsi yang dijalankan tiap thread
void* worker(void* arg) {
    unsigned int seed = time(NULL) ^ (uintptr_t)&arg;  // seed untuk rand_r
    long long local_count = 0;
    for (long long i = 0; i < points_per_thread; i++) {
        // generate titik (x, y) di [-1,1] × [-1,1]
        double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
        double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
        if (x*x + y*y <= 1.0)
            local_count++;
    }
    // tambahkan ke total global secara aman
    pthread_mutex_lock(&lock);
    total_in_circle += local_count;
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <num_threads> <total_points>\n", argv[0]);
        return 1;
    }

    int num_threads = atoi(argv[1]);
    long long total_points = atoll(argv[2]);
    if (num_threads <= 0 || total_points <= 0) {
        fprintf(stderr, "Argumen harus positif.\n");
        return 1;
    }

    // Titik dibagi rata ke tiap thread
    points_per_thread = total_points / num_threads;

    pthread_t *threads = malloc(num_threads * sizeof(pthread_t));
    pthread_mutex_init(&lock, NULL);

    // Buat dan jalankan thread
    for (int i = 0; i < num_threads; i++) {
        if (pthread_create(&threads[i], NULL, worker, NULL) != 0) {
            perror("pthread_create");
            return 1;
        }
    }

    // Tunggu semua thread selesai
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    // Estimasi π
    double pi_estimate = 4.0 * (double)total_in_circle / (points_per_thread * num_threads);
    printf("Estimasi π = %.6f\n", pi_estimate);

    // Bersih-bersih
    pthread_mutex_destroy(&lock);
    free(threads);
    return 0;
}
```

4.25 Ulangi soal 4.24, tetapi alih-alih menggunakan thread terpisah untuk menghasilkan titik acak, gunakan OpenMP untuk melakukan paralelisasi proses pembangkitan titik-titik tersebut.

Berhati-hatilah agar perhitungan nilai π tidak ditempatkan dalam bagian paralel, karena Anda hanya ingin menghitung π satu kali saja

Jawab :

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

int main(int argc, char *argv[]) {
   if (argc != 2) {
      fprintf(stderr, "Usage: %s <total_points>\n", argv[0]);
      return 1;
   }

   long long total_points = atoll(argv[1]);
   if (total_points <= 0) {
      fprintf(stderr, "Total titik harus positif.\n");
      return 1;
   }

   long long in_circle = 0;

   // Atur banyak thread (opsional)
   // omp_set_num_threads(4);

   #pragma omp parallel
   {
      unsigned int seed = (unsigned int)time(NULL) ^ omp_get_thread_num();
      long long local_count = 0;

      #pragma omp for
      for (long long i = 0; i < total_points; i++) {
         double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
         double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
         if (x*x + y*y <= 1.0) {
            local_count++;
         }
      }

      // Reduksi manual ke variabel global
      #pragma omp atomic
      in_circle += local_count;
   }  // Semua thread bergabung di sini

   // Hitung π di luar region paralel
   double pi_estimate = 4.0 * (double)in_circle / (double)total_points;
```

```
    printf("Estimasi π = %.6f\n", pi_estimate);

    return 0;
}
```

Penjelasan :

- Pembacaan Input
  Program menerima satu argumen: total_points (jumlah total titik acak).
- Variabel Reduksi
  - in_circle dihitung lewat #pragma omp atomic agar penambahan dari tiap thread aman.
  - Tiap thread punya local_count sendiri untuk mengurangi kontensi.
- Pengacakan
  - Gunakan rand_r(&seed) agar setiap thread punya seed berbeda (seed diturunkan dari time(NULL) dan ID thread).
- Paralelisasi
  - #pragma omp parallel memulai region paralel.
  - #pragma omp for membagi loop i = 0 … total_points-1 ke setiap thread.
- Perhitungan π
  Dilakukan di luar region paralel, sekali saja, setelah semua thread selesai (} otomatis melakukan barrier).

4.26 Modifikasilah server tanggal berbasis socket (Gambar 3.27 pada Bab 3) sehingga server tersebut melayani setiap permintaan klien dalam thread yang terpisah.

Jawab : Berikut adalah modifikasi dari server tanggal berbasis socket agar melayani setiap permintaan klien dalam thread terpisah:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>
```

```
#include <netinet/in.h>
#include <pthread.h>
#include <time.h>

#define PORT 8080
#define MAX_CONNECTIONS 100

// Struktur untuk meneruskan data ke thread
typedef struct {
    int client_socket;
} thread_args;

// Fungsi yang akan dijalankan oleh thread untuk menangani klien
void *handle_client(void *args) {
    // Ambil data dari struktur
    int client_socket = ((thread_args *)args)->client_socket;
    free(args); // Bebaskan memori untuk struktur yang dialokasikan

    time_t current_time;
    char time_string[100];
    char buffer[1024] = {0};

    // Baca pesan dari klien (opsional)
```

```
    read(client_socket, buffer, 1024);

    // Dapatkan waktu saat ini
    current_time = time(NULL);

    // Ubah waktu ke format string
    strcpy(time_string, ctime(&current_time));

    // Kirim data tanggal dan waktu ke klien
    write(client_socket, time_string, strlen(time_string));

    // Tutup koneksi dengan klien
    close(client_socket);

    pthread_exit(NULL);
}

int main() {
    int server_fd, client_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    pthread_t thread_id;
```

```
// Buat socket file descriptor
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    perror("Socket gagal dibuat");
    exit(EXIT_FAILURE);
}

// Atur opsi socket
int opt = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))) {
    perror("setsockopt");
    exit(EXIT_FAILURE);
}

// Tentukan alamat dan port
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);

// Bind socket ke alamat dan port
if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    perror("Bind gagal");
    exit(EXIT_FAILURE);
}
```

```
    // Siap menerima koneksi
    if (listen(server_fd, MAX_CONNECTIONS) < 0) {
        perror("Listen gagal");
        exit(EXIT_FAILURE);
    }

    printf("Server tanggal berjalan. Menunggu koneksi...\n");

    // Loop utama untuk menerima koneksi
    while (1) {
        // Terima koneksi klien baru
        if ((client_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0) {
            perror("Accept gagal");
            continue;
        }

        printf("Klien baru terhubung\n");

        // Alokasi memori untuk argumen thread
        thread_args *args = malloc(sizeof(thread_args));
        args->client_socket = client_socket;
```

```
        // Buat thread baru untuk menangani klien
        if (pthread_create(&thread_id, NULL, handle_client, args) != 0) {
            perror("Thread creation failed");
            close(client_socket);
            free(args);
            continue;
        }

        // Lepaskan thread agar dapat dihancurkan secara otomatis setelah selesai
        pthread_detach(thread_id);
    }

    return 0;
}
```

Server ini akan:

1. Membuat socket dan mendengarkan koneksi masuk
2. Untuk setiap koneksi klien baru, membuat thread terpisah untuk menangani permintaan
3. Thread akan mengirimkan tanggal dan waktu saat ini ke klien
4. Menggunakan pthread_detach() agar resource thread dibebaskan otomatis setelah selesai

4.27 Deret Fibonacci adalah rangkaian bilangan 0, 1, 1, 2, 3, 5, ... yang secara formal dapat diekspresikan sebagai:

- $fib_0 = 0$
- $fib_1 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$

Buatlah program multithreaded yang menghasilkan deret Fibonacci. Program ini harus bekerja sebagai berikut:

- Pada command line, pengguna akan memasukkan jumlah bilangan Fibonacci yang harus dihasilkan program.
- Program akan membuat thread terpisah yang akan menghasilkan bilangan Fibonacci, menempatkan deret tersebut dalam data yang dapat dibagikan antar thread (array kemungkinan besar adalah struktur data yang paling tepat).
- Ketika thread selesai dieksekusi, thread induk akan menampilkan deret yang dihasilkan oleh thread anak.
- Karena thread induk tidak dapat mulai menampilkan deret Fibonacci sampai thread anak selesai, thread induk harus menunggu thread anak selesai.
- Gunakan teknik yang dijelaskan di Bagian 4.4 untuk memenuhi persyaratan ini.

Jawab : Untuk membuat program multithreaded yang menghasilkan deret Fibonacci sesuai persyaratan, saya akan membuat solusi menggunakan bahasa C dengan POSIX threads. Program ini akan:

1. Menerima input dari pengguna berupa jumlah bilangan Fibonacci yang diinginkan
2. Membuat thread anak untuk menghasilkan deret Fibonacci
3. Thread induk menunggu thread anak selesai menggunakan pthread_join()
4. Setelah thread anak selesai, thread induk menampilkan hasil deret Fibonacci

Berikut adalah implementasi programnya:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>



// Struktur data untuk menyimpan parameter dan hasil

typedef struct {

    int count;

    unsigned long long *sequence;

} fibonacci_data;
```

```
// Fungsi untuk thread anak yang menghasilkan deret Fibonacci
void *fibonacci_thread(void *param) {
    fibonacci_data *data = (fibonacci_data *)param;
    int n = data->count;

    // Alokasi memori untuk deret
    data->sequence = (unsigned long long *)malloc(n * sizeof(unsigned long long));

    if (data->sequence == NULL) {
        printf("Gagal mengalokasikan memori\n");
        pthread_exit(NULL);
    }

    // Menghasilkan deret Fibonacci
    if (n >= 1)
        data->sequence[0] = 0;
    if (n >= 2)
        data->sequence[1] = 1;

    for (int i = 2; i < n; i++) {
        data->sequence[i] = data->sequence[i-1] + data->sequence[i-2];
    }
```

```
    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    fibonacci_data data;
    int count;

    printf("Masukkan jumlah bilangan Fibonacci yang ingin dihasilkan: ");
    scanf("%d", &count);

    if (count <= 0) {
        printf("Jumlah harus positif\n");
        return 1;
    }

    data.count = count;

    // Membuat thread anak
    pthread_create(&tid, NULL, fibonacci_thread, &data);

    // Thread induk menunggu thread anak selesai
    pthread_join(tid, NULL);
```

```
    // Menampilkan deret Fibonacci
    printf("Deret Fibonacci: ");
    for (int i = 0; i < count; i++) {
        printf("%llu ", data.sequence[i]);
    }
    printf("\n");

    // Membebaskan memori
    free(data.sequence);

    return 0;
}
```

Program ini menggunakan teknik yang dijelaskan di Bagian 4.4, yaitu:

- Membuat struktur data bersama (fibonacci_data) untuk komunikasi antar thread
- Menggunakan pthread_create() untuk membuat thread anak
- Menggunakan pthread_join() untuk menunggu thread anak selesai

Thread anak akan menghasilkan deret Fibonacci dan menyimpannya dalam array, kemudian thread induk akan menampilkan hasil setelah thread anak selesai.

4.28 Modifikasi masalah pemrograman Latihan 3.20 dari Bab 3, yang meminta Anda untuk merancang manajer pid. Modifikasi ini akan terdiri dari menulis program multithreaded yang menguji solusi Anda untuk Latihan 3.20. Anda akan membuat sejumlah thread—misalnya, 100—dan setiap thread akan meminta pid, tidur selama periode waktu acak, dan kemudian melepaskan pid. (Tidur selama periode waktu acak mensimulasikan penggunaan tipikal di mana pid ditetapkan ke proses baru, proses berjalan, dan kemudian berakhir, dan pid dilepaskan pada terminasi proses.) Pada UNIX dan Linux, tidur dicapai melalui fungsi `sleep()`, yang diberikan integer yang mewakili jumlah detik untuk tidur.
Jawab : Berikut implementasi program multithreaded untuk menguji solusi manajer PID:

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

```
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define MIN_PID 300
#define MAX_PID 5000
#define NUM_THREADS 100

// Mutex untuk sinkronisasi akses ke manajer PID
pthread_mutex_t pid_mutex;

// Struktur data untuk pelacakan PID (bitmap)
int *pid_map;
int pid_range;

// Inisialisasi struktur data untuk PID
int allocate_map(void) {
   pid_range = MAX_PID - MIN_PID + 1;
   pid_map = (int *)calloc(pid_range, sizeof(int));

   if (pid_map == NULL)
      return -1;  // Alokasi gagal

   return 1;  // Alokasi berhasil
}

// Alokasikan PID
int allocate_pid(void) {
   pthread_mutex_lock(&pid_mutex);

   for (int i = 0; i < pid_range; i++) {
      if (pid_map[i] == 0) {
         pid_map[i] = 1;  // Tandai PID sebagai digunakan
         pthread_mutex_unlock(&pid_mutex);
         return i + MIN_PID;
      }
   }

   pthread_mutex_unlock(&pid_mutex);
   return -1;  // Tidak ada PID yang tersedia
}

// Bebaskan PID
void release_pid(int pid) {
   if (pid < MIN_PID || pid > MAX_PID)
```

```
        return;

    pthread_mutex_lock(&pid_mutex);
    pid_map[pid - MIN_PID] = 0;  // Tandai PID sebagai bebas
    pthread_mutex_unlock(&pid_mutex);
}

// Fungsi thread
void *thread_function(void *arg) {
    int thread_id = *((int *)arg);
    int pid, sleep_time;

    // Alokasikan PID
    pid = allocate_pid();

    if (pid == -1) {
        printf("Thread %d: Gagal mengalokasikan PID\n", thread_id);
        pthread_exit(NULL);
    }

    printf("Thread %d: Mendapatkan PID %d\n", thread_id, pid);

    // Tidur selama waktu acak (1-5 detik)
    sleep_time = rand() % 5 + 1;
    printf("Thread %d dengan PID %d: Tidur selama %d detik\n", thread_id, pid,
sleep_time);
    sleep(sleep_time);

    // Bebaskan PID
    printf("Thread %d: Melepaskan PID %d\n", thread_id, pid);
    release_pid(pid);

    free(arg);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int i, status;

    // Inisialisasi generator angka acak
    srand(time(NULL));

    // Inisialisasi mutex
    pthread_mutex_init(&pid_mutex, NULL);
```

```
    // Inisialisasi map PID
    if (allocate_map() == -1) {
        printf("Gagal mengalokasikan map PID\n");
        return 1;
    }

    // Buat thread
    for (i = 0; i < NUM_THREADS; i++) {
        int *thread_id = malloc(sizeof(int));
        *thread_id = i;

        status = pthread_create(&threads[i], NULL, thread_function, (void
*)thread_id);

        if (status != 0) {
            printf("Gagal membuat thread %d\n", i);
            free(thread_id);
        }
    }

    // Tunggu semua thread selesai
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // Bersihkan
    pthread_mutex_destroy(&pid_mutex);
    free(pid_map);

    printf("Semua thread selesai\n");

    return 0;
}
```

Program ini:

1. Mengimplementasikan manajer PID dari Latihan 3.20 dengan fungsi `allocate_map()`, `allocate_pid()`, dan `release_pid()`
2. Membuat 100 thread yang masing-masing akan:
    - Meminta PID
    - Tidur selama periode acak (1-5 detik)
    - Melepaskan PID
3. Menggunakan mutex untuk mencegah race condition saat mengakses manajer PID

4.29 Latihan 3.25 di Bab 3 melibatkan perancangan server echo menggunakan API threading Java. Server ini single-threaded, artinya server tidak dapat merespons klien echo lain sampai klien saat ini keluar. Modifikasi solusi untuk Latihan 3.25 sehingga server echo melayani setiap klien dalam permintaan terpisah.

Jawab **:** Berikut adalah implementasi server echo multithreaded menggunakan threading Java:

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

public class MultithreadedEchoServer {
   private static final int PORT = 6789;
   private static final int POOL_SIZE = 10;

   public static void main(String[] args) {
      ExecutorService pool =
Executors.newFixedThreadPool(POOL_SIZE);

      try {
         ServerSocket serverSocket = new ServerSocket(PORT);
         System.out.println("Server echo multithreaded berjalan pada
port " + PORT);

         while (true) {
            // Menerima koneksi dari klien
            Socket clientSocket = serverSocket.accept();
            System.out.println("Klien baru terhubung: " +
clientSocket.getInetAddress());

            // Membuat handler dan menjalankannya di thread pool
            ClientHandler handler = new ClientHandler(clientSocket);
            pool.execute(handler);
         }
      } catch (IOException e) {
         System.err.println("Kesalahan server: " + e.getMessage());
         pool.shutdown();
      }
   }

   // Kelas untuk menangani setiap klien dalam thread terpisah
```

```
    private static class ClientHandler implements Runnable {
        private final Socket clientSocket;

        public ClientHandler(Socket socket) {
            this.clientSocket = socket;
        }

        @Override
        public void run() {
            try {
                // Siapkan input dan output stream
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));
                PrintWriter out = new PrintWriter(
                    clientSocket.getOutputStream(), true);

                String inputLine;

                // Loop echo - baca input dari klien dan kirim kembali
                while ((inputLine = in.readLine()) != null) {
                    System.out.println("Server menerima: " + inputLine);

                    // Jika klien mengirim "exit", keluar dari loop
                    if (inputLine.equals("exit")) {
                        break;
                    }

                    // Echo kembali ke klien
                    out.println("ECHO: " + inputLine);
                }

                System.out.println("Klien terputus: " +
clientSocket.getInetAddress());

                // Tutup koneksi
                in.close();
                out.close();
                clientSocket.close();

            } catch (IOException e) {
```

```
                    System.err.println("Kesalahan handler: " + e.getMessage());
                }
            }
        }
}
```

Penjelasan implementasi:

1. Server menggunakan `ExecutorService` untuk mengelola thread pool dengan ukuran tetap
2. Untuk setiap koneksi klien baru, server membuat instance `ClientHandler`
3. Handler dieksekusi dalam thread dari pool
4. Setiap handler menangani komunikasi dengan klien secara terpisah:
    - Membaca pesan dari klien
    - Mengembalikan pesan yang sama sebagai echo
    - Memproses sampai klien mengirim "exit" atau terputus
5. Dengan pendekatan ini, server dapat melayani banyak klien secara bersamaan

Server ini mengatasi keterbatasan server single-threaded dengan memungkinkan banyak klien terhubung dan mendapatkan layanan secara bersamaan.