Modul Praktikum

Pemrograman Lanjut

Laboratium Digital

Tim Penyusun Modul

Tim Asisten Lab Digital 2021

Program Studi S1 Teknik Komputer

Fakultas Teknik

Universitas Indonesia

2021

KATA PENGANTAR

Praktikum ini adalah bagian dari mata kuliah Pemrograman Lanjut dan Praktikum yang diberikan untuk mahasiswa Semester 2 Program Studi Teknik Komputer. Mata kuliah ini merupakan mata kuliah lanjutan yang diberikan kepada mahasiswa dari serangkaian mata kuliah untuk mendukung 2 CP Prodi, yaitu Mampu merancang algoritma untuk masalah tertentu dan mengimplementasikannya ke dalam pemrograman (C6) dan Mampu memanfaatkan teknologi informasi dan komunikasi (C3).

Capaian Pembelajaran Mata Kuliah (CPMK) dari mata kuliah ini adalah:

1. Mampu merancang program komputer prosedural kompleks dengan struktur data dinamis (C6)
2. Mampu menunjukkan sikap kritis, kreatif, dan inovatif dan menghargai orang lain dalam kelompok untuk memecahkan masalah bersama dan tugas kelompok Pemrograman Lanjut (C3, A3)
3. Mampu menggunakan software pemrogram komputer dengan mahir (C3)

Sedangkan Sub-CPMK yang akan dicapai adalah:

1.1. Mampu mengimplementasikan algoritma rekursif ke dalam pemrograman

1.2. Mampu mengimplementasikan algoritma searching dan sorting ke dalam pemrograman

1.3. Mampu membuat program komputer prosedural kompleks dengan linked list, stack dan queue

1.4. Mampu mengimplementasikan multi-threading dan parallel programming

1.5. Mampu merancang perangkat lunak sederhana dengan struktur data dinamis

2.1. Mampu menunjukkan proses berpikir kritis, kreatif dan inovatif dalam menyelesaikan permasalahan kelompok

2.2. Mampu berkomunikasi dengan sopan

2.3. Mampu menghargai pendapat orang lain

3.1. Mampu menggunakan software pemrogram komputer untuk program kompleks dengan mahir

Pada mata kuliah ini mahasiswa akan mengasah kemampuan cara berpikir dan penyelesaian masalah dengan membuat algoritma, kemudian menerjemahkan algoritma tersebut ke dalam bahasa pemrograman yang dapat dijalankan oleh komputer. Pada mata kuliah ini mahasiswa akan mempelajari Rekursif, Searching, Sorting, Linked list, Stack, Queue, Multi-threading, Parallel programming,, dan pada bagian akhir akan ditutup dengan proyek akhir pemrograman yang dibuat oleh mahasiswa.

Bahasa C merupakan bahasa pemrograman terstruktur, yang membagi program dalam sejumlah blok. Tujuannya adalah untuk mempermudah dalam pembuatan dan pengembangan program. Bahasa C menggunakan standarisasi ANSI (American National Standardization Institute) yang dijadikan acuan oleh para pembuat compiler C. Bahasa C terdiri dari fungsi-fungsi dan setiap program C memiliki fungsi utama yang disebut main. Program akan dieksekusi dimulai dari statement pertama pada fungsi main tersebut.

Akhir kata, diharapkan modul praktikum ini akan dapat menjadi referensi untuk membuat program dalam Bahasa C secara umum, dan menjadi panduan dalam menjalankan praktikum mata kuliah Pemrograman Lanjut, secara khusus.

Depok, Februari 2021

Tim Penyusun Modul

MODUL 8: MULTI THREADING

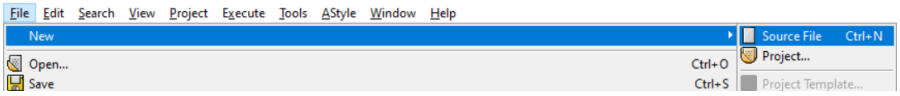
Definisi

Multithreading dapat didefinisikan sebagai kemampuan sebuah Processor atau CPU untuk melakukan pekerjaanya secara **paralel**. Prinsip kerja secara paralel ini dapat dijalankan dengan menggunakan **thread-thread** pada processor untuk menjalankan proses yang berbeda secara bersamaan.

Perbedaan utama antara **Multithreading** dengan **Multiprocessing** terdapat pada cara kerjanya. Dalam sebuah proses Multiprocessing digunakan lebih dari 2 *core* processor pada sebuah sistem operasi. Sedangkan Multithreading merupakan sebuah metode untuk menjalankan sebuah proses pada thread yang berbeda secara *concurrent* (bersamaan) sehingga ekseukusi program dapat berjalan lebih cepat.

Pada praktikum kali ini akan digunakan Library **OpenMP** untuk menjalankan program multithread dalam compiler DevC++. Berikut merupakan langkah instalasi yang dapat diikuti:

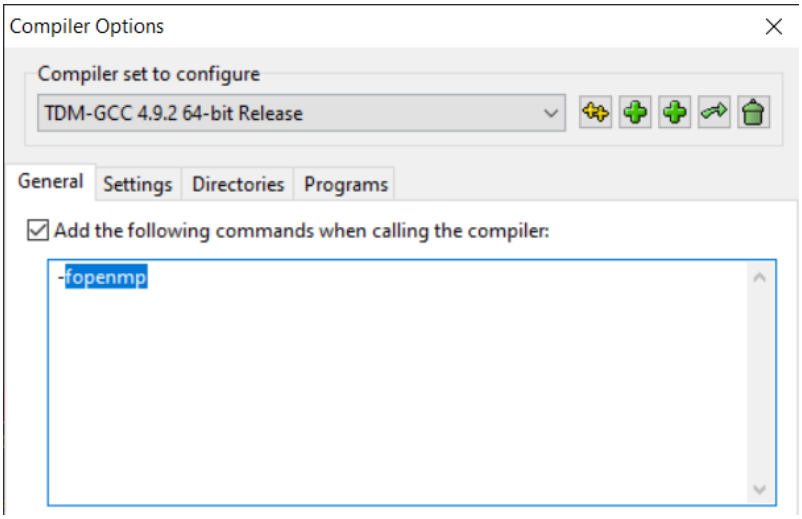
1. Membuat sebuah projek *standalone* (file .c tunggal yang bukan dalam sebuah projek)



2. Mendefinisikan penggunaan library OpenMP pada compiler

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

3. Mengaktifkan OpenMP dengan memberikan flag -fopenmp



Fungsi Multithreading

Dalam pengimplementasiannya Multithreading sangat umum digunakan karena konsep parallel yang dapat dilakukan dan operasinya yang sangat cepat. Beberapa fungsi dari Multithreading antara lain:

1. Menjalankan suatu proses lebih cepat

   Dengan pembentukan dan terminasi thread yang cepat maka proses juga dapat dijalankan lebih cepat. Selain itu, proses komunikasi dan *context switching* antar thread dalam sebuah processor dapat dilakukan dengan waktu yang singkat

2. Thread-thread memiliki sebuah Address Space yang sama

   Dengan memiliki Address Space yang sama pada beberapa Thread maka akan mempermudah melakukan *resource sharing* agar pertukaran data dapat terjadi lebih cepat.

3. Dapat menjalankan program dengan skala yang lebih besar

   Sebuah program dengan skala yang besar dapat dijalankan dalam berbagai thread pada masing-masing processor saat dijalankan secara multiprocessing. Sehingga akan meningkatkan kemampuan paralel lebih baik ketika multithreading dijalankan pada beberapa processor.

Implementasi Program

Prinsip kerja dari OpenMP yaitu menggunakan sebuah thread yang dijalankan pada awal sampai akhir secara sekuensial atau *master thread* dan menggunakan thread lainnya untuk dijalankan pada bagian paralel atau disebut sebagai *slave* thread. Ketika bagian paralel selesai dijanlankan, maka *slave thread* akan bergabung dengan master hingga akhir program.

Untuk menjalankan sebuah program secara paralel perlu didefinisikan sebuah **Paralel Region** dengan syntax sebagai berikut:

```
#pragma omp parallel
{
  //Program akan dijalankan secara parallel
  //Dalam region ini
}
```

Contoh implementasi multithreading menggunakan OpenMP adalah sebagai berikut::

```c
#include <stdio.h>
#include <omp.h>  //Library untuk menggunakan OpenMP

int main(){
    #pragma omp parallel //OMP compiler directive
    {
        //Blok ini akan secra otomatis dijalankan dalam parallel thread
        printf("Hello, world.\n");
    }

  return 0;
}
```

Dengan output sebagai berikut:

```
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.

--------------------------------
Process exited after 0.03884 seconds with return value 0
Press any key to continue . . .
```

Pada program diatas, kalimat “Hello, world,” akan di print sebanyak 12 kali. Dimana jumlah tersebut bergantung pada jumlah thread yang terdapat pada komputer / device kita. Dalam contoh tersebut digunakan *device* yang memiliki 6 Core dengan 12 Thread

Race Condition

Pada pengimplementasian *multithread*, dapat terjadi suatu masalah yang disebut dengan ***Race Condition***. *Race Condition* adalah kondisi dimana output yang dikeluarkan dari program *multithread* tidak konsisten dan tidak sesuai dengan yang kita inginkan.

Masalah ini dapat terjadi akibat adanya penggunaan ***Shared variable*** dimana seluruh thread dapat mengakses variabel tersebut. Berbeda dengan ***Private variable*** yang hanya dapat diakses sebuah thread saja. Hal ini membuat terjadinya akses terhadap suatu *resource* yang sama pada waktu yang bersamaan, sehingga nilai yang akan dioperasikan tidak sesuai.

Akibatnya, urutan instruksi yang dijalankan akan menjadi berantakan dan dengan demikian hasil output tidak tepat.

Lebih jelasnya, contoh program yang memungkinkan terdapat race condition adalah sebagai berikut:

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

int main()
{
      int n, i;
      int tmpsum = 0,sum = 0;

      printf("Banyak elemen untuk array: ");
      scanf("%d", &n);

      int* input = (int*) malloc (sizeof(int)*n);

      for(i=0;i<n;i++){
        //Memasukkan nilai sebanyak elemen pada array
        input[i] = i+1;
      }
      #pragma omp parallel shared(sum)
      {
            for(i=0;i<n;i++)
            {
                  tmpsum = *(input+i);
                  //Menjumlahkan seluruh isi array
                  //secara multithreading
                  sum += tmpsum;
            }
      }
      printf("%d", sum);
      return 0
```

Jika program tersebut dijalankan, maka Hasil output yang didapatkan tidak menentu. Hal tersebut dikarenakan pada suatu waktu terdapat beberapa thread yang mengakses variabel sum yang menyebabkan penjumlahan tidak berlangsung secara berurutan. Berikut merupakan hasil eksekusi program tersebut secara paralel:

```
Banyak elemen untuk array: 100
30053
```

```
Banyak elemen untuk array: 100
35444
```

```
Banyak elemen untuk array: 100
40400
```

Solusi

Race condition pada program multithread dapat dicegah dengan memberlakukan jeda atau delay pada tiap thread, supaya thread yang satu tidak akan mengganggu thread lainnya yang sedang berjalan. Sehingga, akses kepada shared variable-nya akan dilakukan secara teratur dan tidak terjadi secara rebutan.

Metode tersebut dikenal dengan Sinkronisasi pada thread agar proses yang berjalan dapat dieksekusi sesuai urutan dan menghasilkan output yang tepat. Beberapa metode Sinkronisasi yang dapat digunakan pada OpenMP:

1. **critical**
   Mendefinsikan sebuah critical region, dimana bagian tertentu pada program hanya dapat diakses oleh sebuah thread dalam satu waktu. Sehingga sebuah shared variabel dapat terhindar dari race condition
2. **atomic**
   Mendefinisikan bahwa proses memory update pada instruksi selanjutnya akan dijalankan secara atomik.
3. **ordered**
   Mendefinisikan bahwa program pada blok tertentu ataupun sebuah looping akan dijalankan secara berurutan atau sekuensial
4. **barrier**
   Mendefinisikan bahwa setiap thread akan menunggu thread lainnya sampai bagian tertentu sebelum melanjutkan proses selanjutnya
5. **nowait**
   Mendefinisikan thread dengan proses tertentu dapat melanjutkan eksekusinya tanpa menunggu thread lainnya

Selain itu, Dalam melakukan looping pada program multithreading perlu dilakukan yang dinamakan **Loop Scheduling**. Hal tersebut penting untuk dilakukan untuk program yang menggunakan looping dalam proses parallel. Untuk lebih jelas dapat dilihat pada hasil eksekusi program berikut:

```
Banyak elemen untuk array: 1
8
```

Program tersebut seharusnya menjumlahkan isi sebuah array dengan nilai 1 yang menghasilkan output bernilai 1 juga. Namun, dalam hal ini dijumlahkan sebanyak 8 kali karena dijalankan pada device dengan 8 thread.

Oleh karena itu, perlu dilakukan **Loop Scheduling** agar setiap thread dapat terbagi secara rata untuk setiap proses looping atau dalam bentuk *round robin*. Syntax yang digunakan untuk mendefinsikannya:

```
#pragma omp for schedule(static)
```

Contoh penggunaan sinkronisasi **atomic** untuk menghindari terjadinya race condition:

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

int main()
{
	int n, i;
	int tmpsum = 0,sum = 0;

	printf("Banyak elemen untuk array: ");
	scanf("%d", &n);

	int* input = (int*) malloc (sizeof(int)*n);

	for(i=0;i<n;i++){
	  //Memasukkan nilai sebanyak elemen pada array
	  input[i] = i+1;
	}
	#pragma omp parallel for schedule(static)
		for(i=0;i<n;i++)
		{
			tmpsum = *(input+i);
			//Penjumlahan array diberikan sinkronisasi
			#pragma omp atomic
			sum += tmpsum;
		}

	printf("%d", sum);
```

```
    return 0;
}
```

Maka, Dengan modifikasi tersebut hasil output yang diperoleh akan sesuai dengan hasil penjumlahannya yaitu:

```
Banyak elemen untuk array: 1
1
```

```
Banyak elemen untuk array: 100
5050
```