Operating System Course Report - First Half of the Semester

A class

October 2, 2024

Contents

1 Introduction

This report summarizes the topics covered during the first half of the Operating System course. It includes theoretical concepts, practical implementations, and assignments. The course focuses on the fundamentals of operating systems, including system architecture, process management, CPU scheduling, and deadlock handling.

2 Course Overview

2.1 Objectives

The main objectives of this course are:

- To understand the basic components and architecture of a computer system.
- To learn process management, scheduling, and inter-process communication.
- To explore file systems, input/output management, and virtualization.
- To study the prevention and handling of deadlocks in operating systems.

2.2 Course Structure

The course is divided into two halves. This report focuses on the first half, which covers:

- Basic Concepts and Components of Computer Systems
- System Performance and Metrics
- System Architecture of Computer Systems
- Process Description and Control
- Scheduling Algorithms
- Process Creation and Termination

- Introduction to Threads
- File Systems
- Input and Output Management
- Deadlock Introduction and Prevention
- User Interface Management
- Virtualization in Operating Systems

3 Topics Covered

3.1 Basic Concepts and Components of Computer Systems

This section explains the fundamental components that make up a computer system, including the CPU, memory, storage, and input/output devices.

3.2 System Performance and Metrics

This section introduces various system performance metrics used to measure the efficiency of a computer system, including throughput, response time, and utilization.

3.3 System Architecture of Computer Systems

Describes the architecture of modern computer systems, focusing on the interaction between hardware and the operating system.

3.4 Process Description and Control

Processes are a central concept in operating systems. This section covers:

- Process states and state transitions
- Process control block (PCB)
- Context switching

3.5 Scheduling Algorithms

This section covers:

- First-Come, First-Served (FCFS)
- Shortest Job Next (SJN)
- Round Robin (RR)

It explains how these algorithms are used to allocate CPU time to processes.

3.6 Process Creation and Termination

Details how processes are created and terminated by the operating system, including:

- Process spawning
- Process termination conditions

3.7 Introduction to Threads

This section introduces the concept of threads and their relation to processes, covering:

- Konsep Threads
- Hubungan antara Threads dan Proses
- Manfaat Penggunaan Threads
- Multithreading
- Threads vs Process
- Model Multithreading
- Pengelolaan Threads
- Penerapan Threads pada Sistem Operasi

Penjelasannya dibawah ini:

3.7.1 Konsep Threads

3.7.2 Hubungan antara Threads dan Proses

3.7.3 Manfaat Penggunaan Threads

3.7.4 *Multithreading*

Multithreading adalah teknik yang memungkinkan sebuah proses menjalankan beberapa tugas secara paralel dalam satu waktu. Setiap tugas ini disebut *thread*, dan masing-masing *thread* dapat berjalan secara independen, tetapi tetap berbagi sumber daya seperti memori dan file yang sama. Multithreading sangat berguna dalam meningkatkan efisiensi dan performa sebuah aplikasi, terutama untuk program yang perlu menangani banyak tugas bersamaan.

Dalam sistem operasi modern, multithreading digunakan untuk menjalankan proses-proses berat seperti pengolahan data atau menjalankan banyak aplikasi sekaligus, dengan memanfaatkan kemampuan prosesor untuk menangani beberapa pekerjaan secara simultan. Berikut beberapa konsep penting dalam multithreading:

- **Keuntungan Multithreading**: Salah satu keuntungan utama dari multithreading adalah efisiensi. Dengan membagi tugas menjadi beberapa *threads*, kita bisa memanfaatkan waktu tunggu saat satu *thread* menunggu hasil dari operasi I/O (seperti membaca file) dengan menjalankan *thread* lain. Hal ini membuat aplikasi lebih responsif dan cepat.

- **Konteks Berbagi Sumber Daya**: *Threads* dalam satu proses berbagi memori yang sama, sehingga mereka dapat bekerja sama dengan lebih mudah dibandingkan proses-proses terpisah. Namun, karena berbagi sumber daya, kita juga harus memastikan tidak ada *thread* yang saling mengganggu saat mengakses data yang sama. Ini bisa menyebabkan masalah seperti *race conditions*, di mana hasil akhir dari suatu proses tergantung pada urutan eksekusi *threads*.

- **Sinkronisasi *Threads***: Untuk mengatasi masalah dalam berbagi sumber daya, diperlukan mekanisme sinkronisasi. Contohnya, *mutex* dan *semaphore* digunakan untuk memastikan bahwa hanya satu *thread* yang dapat mengakses bagian tertentu dari memori atau data pada satu

waktu. Dengan cara ini, kita bisa mencegah konflik dan memastikan data tetap konsisten meskipun diakses oleh beberapa *threads*.

- **Skalabilitas Multithreading**: Dalam sistem multithreaded, kita bisa memanfaatkan sepenuhnya kemampuan prosesor multi-core. Misalnya, jika sebuah program memiliki empat *threads* dan dijalankan di komputer dengan prosesor empat inti, setiap *thread* bisa berjalan di intinya masing-masing secara bersamaan, yang meningkatkan performa secara signifikan. Namun, manajemen *threads* juga memerlukan pengelolaan yang baik untuk memastikan tidak terjadi *overhead* yang malah memperlambat kinerja aplikasi.

- **Tantangan dalam Multithreading**: Meskipun multithreading dapat meningkatkan performa, teknik ini juga menimbulkan tantangan. Salah satunya adalah debugging, karena perilaku aplikasi multithreaded bisa menjadi sulit diprediksi. Kesalahan seperti *deadlock* atau *race conditions* sering kali sulit ditemukan, karena mereka mungkin tidak selalu muncul pada setiap eksekusi program. Oleh karena itu, pengembangan aplikasi multithreaded memerlukan perhatian khusus untuk memastikan bahwa semua *threads* berjalan dengan aman dan efisien.

Dengan mengimplementasikan multithreading, kita bisa menciptakan aplikasi yang lebih efisien dan responsif, terutama ketika menangani tugas-tugas yang kompleks dan memerlukan pemrosesan paralel. Namun, seperti halnya teknologi lainnya, multithreading juga memerlukan perencanaan dan pengelolaan yang matang untuk memastikan hasil yang optimal.

3.7.5 Threads vs Process

3.7.6 Model *Multithreading*

Multithreading adalah fitur dari sistem operasi yang memungkinkan beberapa thread berjalan secara bersamaan dalam satu proses. Ada beberapa model *Multithreading* umumnya dibagi menjadi tiga, yaitu:

1. Many-to-One Model (Banyak ke Satu)
   Pada model ini, banyak *thread* yang dibuat oleh aplikasi *(user-level)* dipetakan ke satu *thread* di *kernel*. Artinya, sistem operasi hanya melihat satu *thread kernel* meskipun ada banyak *thread* di tingkat aplikasi.

Bayangkan kamu punya banyak tugas *(thread user)* yang dikerjakan secara bergantian oleh satu orang *(thread kernel)*. Orang tersebut harus menyelesaikan satu tugas sebelum beralih ke tugas lainnya, jadi jika satu tugas terhenti, semua tugas lain ikut tertunda.

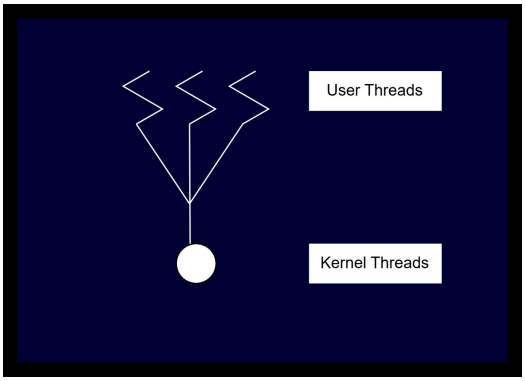

Figure 1: Gambar Model *Many-to-one*

- Keuntungan

  (a) Lebih cepat dan efisien
  Karena semua manajemen *thread* terjadi di tingkat aplikasi, tidak perlu berkomunikasi dengan *kernel*. Ini mengurangi waktu yang dibutuhkan untuk membuat, menghancurkan, atau mengubah *thread*.

  (b) Sederhana
  Mudah diimplementasikan karena *kernel* hanya perlu mengelola satu *thread*, terlepas dari berapa banyak *thread* yang dibuat oleh aplikasi.

- Kekurangan

  (a) Pemblokiran total
  Jika satu *thread* terblokir (misalnya saat menunggu input), seluruh proses akan berhenti, karena *kernel* hanya bisa menangani satu *thread*.

  (b) Tidak cocok untuk multiprosesor
  Semua *thread* dijalankan oleh satu *thread kernel*, jadi meskipun

ada banyak prosesor, hanya satu yang bisa digunakan. Ini membuat model ini kurang efisien di sistem dengan banyak prosesor.

2. One-to-One Model (Satu ke Satu)
   Dalam model ini, setiap *thread* yang dibuat di tingkat aplikasi dipetakan langsung ke satu *thread kernel.* Jadi, untuk setiap *thread user-level*, *kernel* membuat satu *thread* untuk mengelolanya. Bayangkan kamu punya banyak tugas *(thread user)*, dan setiap tugas dikerjakan oleh orang yang berbeda *(thread kernel).* Jika satu orang terhenti, yang lain tetap bisa bekerja, sehingga tugas-tugas lainnya tidak terganggu.

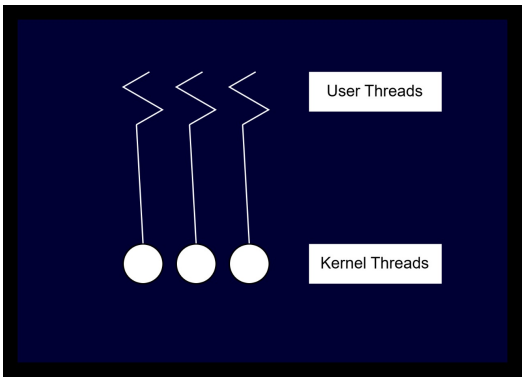

Figure 2: Gambar Model *One-to-one*

- Keuntungan

  (a) Paralelisme lebih baik
  Setiap *thread user-level* bisa berjalan secara paralel di prosesor yang berbeda, sehingga sistem multiprosesor bisa dimanfaatkan lebih baik.

  (b) Tidak ada pemblokiran total
  Jika satu *thread* terblokir, *thread* lainnya masih bisa berjalan, karena masing-masing *thread user-lever* dikelola oleh *thread kernel* yang terpisah.

- Kekurangan

(a) Overhead lebih besar
Membuat *thread kernel* untuk setiap *thread user-level* membutuhkan lebih banyak memori dan sumber daya sistem. Jika ada banyak *thread*, beban kerja untuk sistem operasi bisa meningkat drastis.

(b) Switching antar *thread* lebih lambat
Karena setiap perpindahan *(switching)* antar *thread* harus melibatkan *kernel*, ini membuat waktu *switching* lebih lambat dibanding model *many-to-one*.

3. Many-to-Many Model (Banyak ke Banyak)
Model ini mengizinkan banyak *thread user-level* untuk dipetakan ke sejumlah *thread kernel-level*. Dengan kata lain, beberapa *thread user-level* dapat dijalankan oleh beberapa *thread kernel-level*.

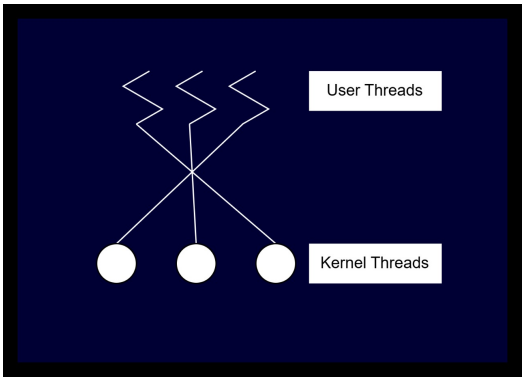

Figure 3: Gambar Model *Many-to-Many*

- Keuntungan

(a) Efisiensi dan fleksibilitas
Kamu bisa memiliki lebih banyak *thread user-level* daripada *thread kernel-level*. Jadi, *thread-thread user-level* tidak harus selalu dihubungkan satu-satu ke *thread kernel*, memungkinkan penggunaan sumber daya yang lebih efisien

(b) Pemanfaatan multiprosesor
Seperti model *one-to-one, thread user-level* dapat berjalan secara paralel di banyak prosesor, namun lebih hemat sumber

daya karena tidak semua *thread user* harus memiliki *thread kernel* masing-masing.

- Kekurangan

  (a) Kompleksitas manajemen
  Model ini membutuhkan mekanisme yang lebih rumit untuk memetakan *thread user-level* ke *thread kernel-level*, dan bisa jadi lebih sulit diimplementasikan.

  (b) Potensi *bottleneck*
  Jika terlalu banyak *thread user* yang dipetakan ke *thread kernel* yang sama, ini bisa menimbulkan kemacetan *(bottleneck)* yang memperlambat performa.

3.7.7 Pengelolaan *Threads*

Pengelolaan *threads* merupakan salah satu aspek penting dalam sistem operasi untuk memastikan bahwa *threads* dapat berjalan secara efisien dan tidak saling mengganggu. Sistem operasi bertanggung jawab untuk mengalokasikan waktu CPU, memori, serta sumber daya lain bagi setiap *thread*. Berikut adalah beberapa elemen utama dalam pengelolaan *threads*:

- **Pembuatan *Threads***: Sistem operasi memungkinkan aplikasi untuk membuat *threads* baru saat dibutuhkan. Misalnya, saat sebuah aplikasi memulai tugas paralel, sistem operasi akan membuat satu atau lebih *threads* untuk menangani tugas-tugas tersebut. Setiap *thread* memiliki identitas unik dan dikaitkan dengan proses induknya. Pembuatan *threads* biasanya dilakukan melalui API atau fungsi-fungsi khusus yang disediakan oleh sistem operasi, seperti `pthread_create()` pada *Linux* atau `CreateThread()` pada *Windows*.

- **Penjadwalan *Threads***: Sistem operasi menggunakan algoritma penjadwalan untuk menentukan urutan eksekusi *threads*. Ini memastikan bahwa *threads* dari berbagai proses mendapatkan waktu CPU secara adil dan efisien. Beberapa algoritma penjadwalan yang digunakan dalam pengelolaan *threads* antara lain *Round Robin* dan *Priority Scheduling*. Penjadwalan yang baik sangat penting untuk menjaga kinerja sistem secara keseluruhan, terutama dalam lingkungan dengan banyak *threads* yang berjalan bersamaan.

- **Sinkronisasi *Threads***: Ketika beberapa *threads* berbagi sumber daya yang sama, sistem operasi harus memastikan bahwa tidak terjadi konflik saat *threads* mengakses sumber daya tersebut. Mekanisme sinkronisasi seperti *mutexes*, *semaphores*, dan *condition variables* digunakan untuk mengelola akses bersamaan dan mencegah kondisi balapan (*race conditions*). Sinkronisasi memastikan bahwa *threads* tidak saling mengganggu dan data yang digunakan tetap konsisten.

- **Penghentian *Threads***: Setelah *threads* selesai menjalankan tugasnya, sistem operasi harus menghentikan *threads* tersebut dan membersihkan sumber daya yang digunakan. Penghentian *threads* dapat dilakukan secara manual oleh aplikasi, atau secara otomatis ketika tugas telah selesai. Fungsi-fungsi seperti `pthread_exit()` pada *Linux* atau `ExitThread()` pada *Windows* digunakan untuk mengakhiri *threads*.

- **Manajemen Konteks *Threads***: Ketika sistem operasi berpindah dari satu *thread* ke *thread* lainnya, perlu dilakukan penyimpanan dan pemulihan konteks eksekusi (*context switching*). Konteks ini mencakup informasi tentang status CPU, register, dan memori yang sedang digunakan oleh *thread*. Sistem operasi harus memastikan bahwa ketika sebuah *thread* dilanjutkan, ia bisa melanjutkan eksekusinya tanpa kehilangan data atau instruksi penting.

Pengelolaan *threads* yang efektif sangat penting untuk meningkatkan kinerja sistem operasi, terutama pada sistem dengan banyak tugas yang berjalan secara bersamaan. Dengan teknik yang tepat, *threads* dapat membantu mempercepat penyelesaian tugas dan memastikan pemanfaatan sumber daya secara optimal.

3.7.8 Penerapan *Threads* pada Sistem Operasi

Sistem operasi modern seperti *Windows*, *Linux*, dan *macOS* menerapkan konsep *threads* untuk meningkatkan efisiensi dan performa ketika menjalankan berbagai tugas secara bersamaan. Berikut adalah beberapa contoh penerapan *threads* pada sistem operasi:

- ***Multitasking***: Dalam sistem operasi, setiap proses atau aplikasi dapat terdiri dari beberapa *threads* yang bekerja bersamaan. Misalnya,

ketika kamu menjalankan *browser*, mengetik dokumen, dan mendengarkan musik, sistem operasi menggunakan *threads* untuk mengatur agar semua aplikasi tersebut bisa berjalan tanpa saling mengganggu. Setiap *thread* diatur secara independen sehingga *multitasking* dapat tercapai dengan lancar.

- **Pembagian Tugas dalam Aplikasi**: Banyak aplikasi modern menggunakan *threads* untuk membagi tugas-tugas yang lebih kecil. Misalnya, dalam aplikasi peramban (*browser*), satu *thread* bisa bertanggung jawab untuk menampilkan halaman web, sementara *thread* lain mengunduh gambar, dan *thread* lainnya lagi mengelola animasi. Penggunaan *threads* ini membuat aplikasi lebih responsif dan tidak bergantung pada satu proses untuk menyelesaikan semua tugas.

- **Penggunaan Sumber Daya yang Efisien**: Pada sistem dengan prosesor multi-core, sistem operasi dapat mendistribusikan *threads* ke berbagai inti (*core*). Ini memungkinkan proses berjalan secara paralel dan mempercepat kinerja sistem. Bayangkan sebuah tugas besar yang dibagi ke beberapa pekerja (*thread*), masing-masing menyelesaikan bagian mereka secara bersamaan, sehingga tugas selesai lebih cepat.

- ***Input/Output* (*I/O* Operations)**: Ketika aplikasi menunggu input dari pengguna atau operasi *I/O* seperti pembacaan file dari disk atau pengunduhan data dari internet, *threads* lain masih bisa berjalan. Misalnya, saat menunggu halaman web dimuat, kamu tetap bisa menggulir halaman atau mengetik alamat baru karena ada *thread* yang khusus menangani *I/O* dan *thread* lainnya yang tetap aktif untuk menerima interaksi pengguna.

3.8 File Systems

File systems provide a way for the operating system to store, retrieve, and manage data. This section explains:

- File system structure
- File access methods
- Directory management

3.9 Input and Output Management

Input and output management is key for handling the interaction between the system and external devices. This section includes:

- Device drivers
- I/O scheduling

3.10 Deadlock Introduction and Prevention

Explores the concept of deadlocks and methods for preventing them:

- Deadlock conditions
- Deadlock prevention techniques

3.11 User Interface Management

This section discusses the role of the operating system in managing the user interface. Topics covered include:

- Graphical User Interface (GUI)
- Command-Line Interface (CLI)
- Interaction between the user and the operating system

3.12 Virtualization in Operating Systems

Virtualization allows multiple operating systems to run concurrently on a single physical machine. This section explores:

- Concept of virtualization
- Hypervisors and their types
- Benefits of virtualization in modern computing

Seperti yang terlihat pada Gambar **??**, inilah cara menambahkan gambar dengan keterangan.

Figure 4: Ini adalah gambar contoh dari multithreading.

4 Assignments and Practical Work

4.1 Assignment 1: Process Scheduling

Students were tasked with implementing various process scheduling algorithms (e.g., FCFS, SJN, and RR) and comparing their performance under different conditions.

4.2 Assignment 2: Deadlock Handling

In this assignment, students were asked to simulate different deadlock scenarios and explore various prevention methods.

4.3 Assignment 3: Multithreading and Amdahl's Law

This assignment involved designing a multithreading scenario to solve a computationally intensive problem. Students then applied **Amdahl's Law** to calculate the theoretical speedup of the program as the number of threads increased.

4.4 Assignment 4: Simple Command-Line Interface (CLI) for User Interface Management

Students were tasked with creating a simple **CLI** for user interface management. The CLI should support basic commands such as file manipulation (creating, listing, and deleting files), process management, and system status reporting.

4.5 Assignment 5: File System Access

In this assignment, students implemented file system access routines, including:

- File creation and deletion
- Reading from and writing to files
- Navigating directories and managing file permissions

5 Conclusion

The first half of the course introduced core operating system concepts, including process management, scheduling, multithreading, and file system access. These topics provided a foundation for more advanced topics to be covered in the second half of the course.