Operating System Course Report - First Half of the Semester

A class

October 10, 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

3.7.1 Konsep Threads

3.7.2 Hubungan antara Threads dan Proses

3.7.3 Manfaat Penggunaan Threads

Beberapa manfaat utama dari *multithreading* meliputi:

- Peningkatan kinerja dan efisiensi:
  - *Multithreading* memungkinkan eksekusi beberapa tugas secara bersamaan dengan memanfaatkan sumber daya *CPU* secara lebih optimal. Ketika sebuah program berjalan dalam mode satu *thread* (dikenal sebagai *single-threading*), prosesor hanya dapat menangani satu tugas pada satu waktu. Namun, dengan *multithreading*, beberapa *threads* dapat berjalan dalam waktu yang sama, memanfaatkan kemampuan inti prosesor untuk menangani lebih dari satu operasi. Misalnya, dalam aplikasi grafis, satu *thread* dapat menangani rendering objek, sementara *thread* lain menangani input pengguna.
  - Pada sistem dengan *multiprosesor* atau *multi-core*, *threads* dapat berjalan secara paralel, yang artinya setiap inti prosesor dapat menangani *thread* yang berbeda. Misalnya, jika sebuah komputer memiliki empat inti prosesor, ia dapat menjalankan empat *threads* secara paralel, yang meningkatkan kemampuan komputer untuk menyelesaikan pekerjaan lebih cepat. Ini sangat berguna untuk aplikasi yang membutuhkan banyak komputasi seperti pemrosesan data besar, aplikasi *machine learning*, atau aplikasi *scientific computing*. Selain itu, *throughput*, atau jumlah tugas yang bisa diselesaikan dalam satu waktu, juga akan meningkat secara signifikan dengan adanya pemrosesan paralel ini.
- Responsivitas aplikasi yang lebih baik:
  - Aplikasi yang memanfaatkan *multithreading* cenderung lebih responsif terhadap input pengguna meskipun ada tugas berat yang sedang berjalan di latar belakang. Misalnya, dalam sebuah program pengeditan video, satu *thread* dapat terus menjalankan tugas rendering, sementara *thread* lain tetap menerima input pengguna seperti pemotongan video atau perubahan filter. Pengguna tidak akan merasa aplikasi "terkunci" atau "membeku", karena

proses rendering dan interaksi pengguna dapat berjalan secara bersamaan.

  - Dengan adanya *multithreading*, aplikasi dapat mencegah terjadinya *bottleneck* atau kemacetan dalam menjalankan berbagai tugas yang memakan waktu lama. Sebagai contoh, saat aplikasi sedang melakukan pengunduhan file besar, alih-alih harus menunggu hingga pengunduhan selesai, aplikasi masih dapat merespons input pengguna seperti membuka menu, melakukan tugas lainnya, atau menampilkan informasi progres pengunduhan dalam waktu nyata. Ini sangat membantu dalam memberikan pengalaman pengguna yang lebih baik.

- Penggunaan sumber daya yang lebih efisien:

  - *Threads* dalam satu proses berbagi memori dan sumber daya lainnya, membuat penggunaan memori menjadi lebih efisien dibandingkan menjalankan beberapa proses yang terpisah. Jika setiap tugas dilakukan dalam proses terpisah, sistem harus menyediakan ruang memori dan sumber daya lainnya untuk masing-masing proses, yang tentu saja membutuhkan lebih banyak overhead. Namun, dengan *threads*, overhead ini dapat dikurangi karena mereka berbagi ruang memori yang sama.
  - Selain itu, karena *threads* berbagi ruang memori, pertukaran informasi di antara mereka jauh lebih cepat dibandingkan jika menggunakan proses terpisah yang memerlukan mekanisme komunikasi antar-proses (*IPC*). Dalam aplikasi yang sering membutuhkan pembagian data antar-tugas, seperti aplikasi basis data atau sistem transaksi, efisiensi komunikasi antar-*threads* ini sangat penting untuk menjaga kinerja dan responsivitas yang optimal.
  - Sebagai contoh, bayangkan sebuah sistem pemrosesan transaksi di mana satu *thread* bertugas membaca data transaksi dari basis data, sementara *thread* lain memverifikasi transaksi tersebut. Karena keduanya berbagi ruang memori, data transaksi yang dibaca tidak perlu disalin dari satu *thread* ke *thread* lain, yang menghemat waktu dan memori. Dalam skala besar, penghematan ini bisa sangat signifikan.

- Penyederhanaan desain program:

  - Dengan menggunakan *multithreading*, pengembang dapat membagi tugas besar menjadi tugas-tugas kecil yang dijalankan dalam *threads* terpisah, menyederhanakan struktur program secara keseluruhan. Misalnya, dalam aplikasi permainan video, satu *thread* dapat bertugas untuk menangani fisika objek, sementara *thread* lain bertugas untuk mengelola AI karakter, dan *thread* lain mengurus rendering grafis. Dengan membagi tugas-tugas ini ke dalam *threads* terpisah, pengembangan dan pemeliharaan program menjadi lebih mudah karena setiap *thread* dapat difokuskan pada satu tugas spesifik.
  - Selain itu, *threads* juga memudahkan pengelolaan beberapa tugas secara bersamaan tanpa mempersulit desain program secara keseluruhan. Setiap *thread* dapat dikembangkan, diuji, dan dikelola secara independen dari *threads* lain, mengurangi kerumitan kode dan menghindari potensi kesalahan dalam sinkronisasi tugas.
  - Misalnya, dalam aplikasi e-commerce yang kompleks, satu *thread* dapat menangani pemrosesan pesanan, sementara yang lain menangani verifikasi pembayaran, dan *thread* ketiga menangani pengiriman. Desain terpisah semacam ini memudahkan pengembang untuk memisahkan logika bisnis yang berbeda dan memastikan setiap bagian program berfungsi dengan benar secara terisolasi.

- Komunikasi antar-*threads* yang efisien:

  - Dalam sebuah proses yang menjalankan banyak *threads*, komunikasi antar-*threads* dapat dilakukan lebih efisien karena mereka berbagi ruang memori yang sama. Misalnya, jika satu *thread* menghasilkan data yang dibutuhkan oleh *thread* lain, data tersebut dapat diakses langsung tanpa perlu disalin ke memori terpisah. Hal ini berbeda dengan proses yang terpisah, yang membutuhkan mekanisme seperti soket atau saluran komunikasi yang memperlambat interaksi antar proses.
  - Keuntungan dari komunikasi antar-*threads* yang lebih efisien ini sangat penting dalam aplikasi yang membutuhkan sinkronisasi cepat antar tugas, seperti aplikasi server yang menangani banyak permintaan pengguna secara bersamaan. Sebagai contoh, sebuah server web dapat memiliki satu *thread* yang menangani permintaan

pengguna, sementara *thread* lain mengumpulkan data dari basis data. Kedua *threads* dapat berkomunikasi dengan cepat untuk memberikan respons yang lebih cepat kepada pengguna.

- Skalabilitas yang lebih baik:
  - Program yang dirancang dengan *multithreading* dapat lebih mudah diskalakan untuk memanfaatkan sistem dengan lebih banyak inti prosesor. Misalnya, pada sistem dengan banyak inti prosesor, *threads* tambahan dapat dijalankan secara paralel untuk meningkatkan efisiensi pemrosesan. Ini sangat bermanfaat dalam server modern yang harus menangani ribuan pengguna sekaligus. Dengan menambahkan lebih banyak *threads*, server dapat menangani lebih banyak permintaan tanpa menurunkan kinerja.
  - Selain itu, program berbasis *multithreading* cenderung lebih siap untuk beradaptasi dengan teknologi perangkat keras baru. Dengan penambahan prosesor atau sumber daya lainnya, program tersebut dapat diubah dan dikembangkan tanpa harus mendesain ulang keseluruhan struktur program. Hal ini memungkinkan pengembangan aplikasi yang lebih berkelanjutan dan mudah ditingkatkan di masa depan.
- Pemanfaatan waktu tunggu *I/O*:
  - Dalam banyak kasus, operasi *I/O* (seperti membaca atau menulis file, atau berkomunikasi dengan jaringan) memakan waktu lebih lama dibandingkan operasi komputasi biasa. Dengan *multithreading*, satu *thread* dapat melanjutkan pemrosesan tugas lain sementara menunggu hasil dari operasi *I/O*, sehingga sistem tidak dibiarkan menganggur. Ini meningkatkan efisiensi penggunaan waktu dan sumber daya.
  - Sebagai contoh, dalam aplikasi berbasis web, sementara satu *thread* menunggu respons dari server jarak jauh, *threads* lain dapat melanjutkan pemrosesan data lokal atau menampilkan informasi kepada pengguna. Dengan demikian, pengguna tidak perlu menunggu hingga semua tugas selesai sebelum melanjutkan interaksi dengan aplikasi.
- Cocok untuk aplikasi *server*:

- Aplikasi server seperti server web atau server database sangat diuntungkan dari *multithreading* karena harus menangani banyak permintaan klien secara bersamaan. Alih-alih membuat satu proses terpisah untuk setiap klien (yang akan memakan banyak memori dan sumber daya sistem), server dapat menggunakan *threads* untuk menangani setiap koneksi klien secara lebih efisien. Setiap klien dapat dilayani oleh satu atau beberapa *threads*, memungkinkan server untuk memproses permintaan dengan cepat tanpa kelebihan beban.
- Selain itu, *multithreading* juga memungkinkan server untuk tetap responsif meskipun beban kerja meningkat. Misalnya, server dapat memulai *threads* baru untuk menangani permintaan klien tambahan tanpa memengaruhi kinerja *threads* yang sudah ada. Ini memastikan bahwa server dapat menangani lebih banyak klien sekaligus tanpa mengorbankan kualitas layanan.

References

[1] T. Z. Vitadiar, G. S. Permadi, *Sistem Operasi*, 2022.

[2] R. Watrianthos, I. Purnama, *Buku Ajar Sistem Operasi*, Uwais Inspirasi Indonesia, 2018.

3.7.4 Multithreading

3.7.5 Threads vs Process

3.7.6 Model Multithreading

3.7.7 Pengelolaan Threads

3.7.8 Penerapan Threads pada Sistem Operasi

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

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.