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* dan Proses
- 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 *(GeeksforGeeks, n.d)*.

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*, yang terjadi ketika 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.

GeeksforGeeks. (n.d.). *Model multi-threading dalam manajemen proses*. GeeksforGeeks. Diakses pada 3 Oktober 2024, dari `https://www.geeksforgeeks.org/multi-threading-models-in-process-management/`

3.7.5 *Threads* and Proses

3.7.6 Model *Multithreading*

Multithreading adalah fitur dari sistem operasi yang memungkinkan beberapa *thread* berjalan secara bersamaan dalam satu proses (Tutorialspoint, n.d).

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.

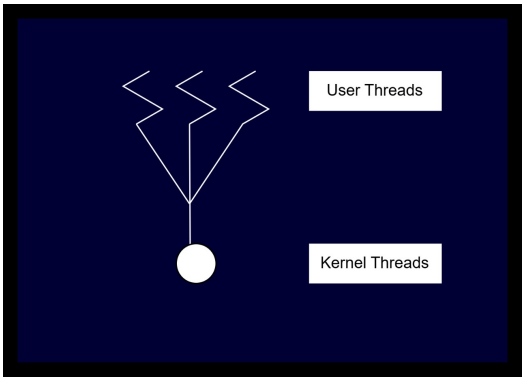

Figure 1: Gambar Model *Many-to-one*

- 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.

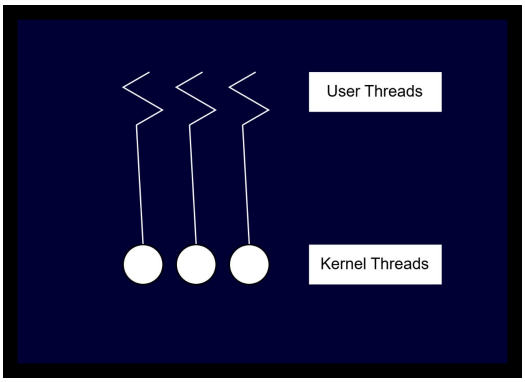

Figure 2: Gambar Model *One-to-one*

- Keuntungan

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

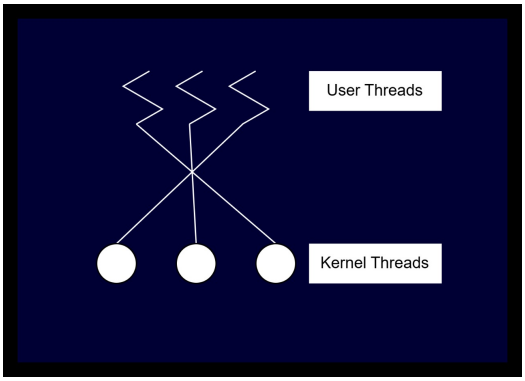

Figure 3: Gambar Model *Many-to-Many*

- 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.

Tutorialspoint. (n.d.). *Multithreading models in operating system.* Tutorialspoint. Diakses pada 3 Oktober 2024, dari `https://www.tutorialspoint.com/operating_system/os_multi_threading_models.htm`

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 *(Central Processing Unit)*, 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 (Application Programming Interface)* 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.

Studytonight. (n.d.). *Thread management in operating system*. Studytonight. Diakses pada 3 Oktober 2024, dari `https://www.studytonight.com/operating-system/thread-management`

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 melihat

halaman atau mengetik alamat baru karena ada *thread* yang khusus menangani *I/O* dan *thread* lainnya yang tetap aktif untuk menerima interaksi pengguna.

Scaler. (2023). *Introduction to multithreading in operating systems.* Scaler. Diakses pada 3 Oktober 2024, dari
`https://www.scaler.com/topics/multithreading-in-operating-system/`

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

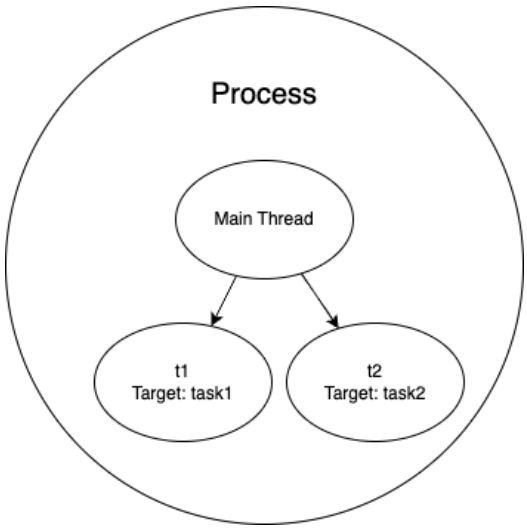

Figure 4: Ini adalah gambar contoh dari multithreading.

Seperti yang terlihat pada Gambar 4, inilah cara menambahkan gambar dengan keterangan.

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.

4.1.1 Group 1

```
    class Process:
    def __init__(self, pid, arrival_time, burst_time):
        self.pid = pid
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.completion_time = 0
        self.turnaround_time = 0
        self.waiting_time = 0
```

Header 1	Header 2	Header 3
Row 1, Column 1	Row 1, Column 2	Row 1, Column 3
Row 2, Column 1	Row 2, Column 2	Row 2, Column 3

Table 1: Your table caption

4.1.2 Kelompok 7

```
    import random

class Process:
    def __init__(self, pid, arrival_time, burst_time):
        self.id = pid
        self.arrival_time = arrival_time
        self.burst_time = burst_time

def round_robin_scheduling(processes, quantum):
    time = 0
    completed = []
    queue = []
    remaining_time = {process.id: process.burst_time for
                                    process in processes}
```

```
    while remaining_time:
        for process in processes:
            if process.id in remaining_time and process.
                                            arrival_time <=
                                            time:
                queue.append(process)

        if queue:
            current_process = queue.pop(0)
            time_spent = min(quantum, remaining_time[
                                            current_process.id
                                            ])
            time += time_spent
            remaining_time[current_process.id] -= time_spent

            if remaining_time[current_process.id] == 0:
                completed.append((current_process.id, time))
                del remaining_time[current_process.id]
            else:
                queue.append(current_process)
        else:
            time += 1
    return completed
    processes = [
        Process(1, 0, 7),
        Process(2, 2, 4),
        Process(3, 4, 1),
        Process(4, 5, 4)]

quantum = 3 #Quantum time ini untuk algoritma Round Robin
result = round_robin_scheduling(processes, quantum)
```

Soal :
Dari kode Python di atas, tuliskan output yang akan dihasilkan serta coba gambarkan eksekusi proses menggunakan Gantt Chart!
Jawaban :

```
    Waktu saat ini: 0
    Menjalankan process 1 selama 3 waktu
    Waktu saat ini: 3

    Waktu saat ini: 3
    Menjalankan process 2 selama 3 waktu
```

```
Waktu saat ini: 6

Waktu saat ini: 6
Menjalankan process 1 selama 3 waktu
Waktu saat ini: 9

Waktu saat ini: 9
Menjalankan process 3 selama 1 waktu
Waktu saat ini: 10

Waktu saat ini: 10
Menjalankan process 4 selama 3 waktu
Waktu saat ini: 13

Waktu saat ini: 13
Menjalankan process 2 selama 1 waktu
Waktu saat ini: 14

Waktu saat ini: 14
Menjalankan process 4 selama 1 waktu
Waktu saat ini: 15

Waktu saat ini: 15
Menjalankan process 1 selama 1 waktu
Waktu saat ini: 16

Waktu saat ini: 16
Menjalankan process 4 selama 1 waktu
Waktu saat ini: 17

Waktu saat ini: 17
Menjalankan process 4 selama 1 waktu
Waktu saat ini: 18
```

	P1	P2	P1	P3	P4	P2	P4	P1	P4
0	3	6	9	10	13	14	15	16	18

Figure 5: Gantt Chart.

4.2 Assignment 2: Deadlock Handling

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

4.2.1 Kelompok 7

```
import threading
import time
import random

class Resource:
    def __init__(self, name):
        self.name = name
        self.lock = threading.Lock()

class Process(threading.Thread):
    def __init__(self, name, resource1, resource2):
        threading.Thread.__init__(self)
        self.name = name
        self.resource1 = resource1
        self.resource2 = resource2

    def run(self):
        print(f"{self.name} mencoba untuk mengunci {self.
                                        resource1.name}")
        with self.resource1.lock:
            print(f"{self.name} mengunci {self.resource1.name
                                            }")
            time.sleep(random.uniform(1, 2))
            print(f"{self.name} mencoba untuk mengunci {self.
                                            resource2.name}")
            with self.resource2.lock:
                print(f"{self.name} mengunci {self.resource2.
                                                name}")
        print(f"{self.name} selesai.")

# Membuat dua sumber daya
resource_A = Resource("Resource A")
resource_B = Resource("\textbf{Resource B}")

# Membuat dua proses yang saling menunggu satu sama lain
process1 = Process("Process 1", resource_A, resource_B)
process2 = Process("Process 2", resource_B, resource_A)
```

```
# Menjalankan proses
process1.start()
process2.start()

process1.join()
process2.join()
```

Soal : Dari kode Python di atas, jelaskan skenario *deadlock* yang terjadi ketika dua proses berusaha mengunci dua sumber daya. Apa metode pencegahan yang bisa digunakan untuk menghindari *deadlock* dalam situasi ini?

Jawaban : Penjelasan Skenario *Deadlock*: Dalam kode ini, terdapat dua proses, yaitu **Process 1** dan **Process 2**. Keduanya mencoba untuk mengunci dua sumber daya, **Resource A** dan **Resource B**.

- **Process 1** mengunci **Resource A** dan kemudian mencoba mengunci **Resource B**.
- **Process 2** mengunci **Resource B** dan kemudian mencoba mengunci **Resource A**.

Jika **Process 1** sudah mengunci **Resource A** dan **Process 2** sudah mengunci **Resource B**, keduanya akan saling menunggu satu sama lain, menyebabkan *deadlock*. **Process 1** menunggu **Resource B** yang dipegang oleh **Process 2**, sedangkan **Process 2** menunggu **Resource A** yang dipegang oleh **Process 1**. Metode Pencegahan Deadlock:

1. Menghindari Siklus: Menjaga agar tidak ada siklus dalam alokasi sumber daya. Dalam kasus ini, bisa mengatur urutan penguncian sumber daya, misalnya semua proses harus mengunci **Resource A** terlebih dahulu sebelum **Resource B**.
2. Penggunaan Waktu Tunggu (*Timeout*): Jika suatu proses tidak dapat mengunci sumber daya dalam waktu tertentu, maka proses tersebut akan dibatalkan dan di-*restart*. Dengan cara ini, sumber daya akan tersedia untuk proses lain.

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.

4.3.1 Kelompok 7

```
    import threading
import time

# Fungsi untuk melakukan komputasi intensif
def computational_task(task_id):
    print(f"Thread {task_id} mulai.")
    # Simulasi kerja komputasi intensif
    time.sleep(2)  # Menggunakan waktu 2 detik sebagai
                                    simulasi kerja, bisa
                                    diganti
    print(f"Thread {task_id} selesai.")

def main():
    num_threads = 4  # Jumlah thread yang akan digunakan
    threads = []

    # Memulai thread
    for i in range(num_threads):
        thread = threading.Thread(target=computational_task,
                                        args=(i,))
        threads.append(thread)
        thread.start()

    # Menunggu semua thread selesai
    for thread in threads:
        thread.join()

if __name__ == "__main__":
    start_time = time.time()
    main()
    end_time = time.time()
    print(f"Waktu eksekusi total: {end_time - start_time:.2f}
                                     detik.")
```

Soal : Dari kode Python di atas, jelaskan bagaimana skenario *multithreading* diimplementasikan untuk menyelesaikan masalah komputasi intensif. Selain itu, terapkan Hukum Amdahl untuk menghitung percepatan teoretis program saat jumlah *thread* meningkat dari satu ke empat.

Jawaban :

- Penjelasan Skenario *Multithreading*

  1. Kita menentukan jumlah *thread* yang akan digunakan, yaitu empat buah *thread*.
  2. Kita memulai setiap *thread* menggunakan `threading.Thread`, dan setiap *thread* menjalankan `computational_task`.
  3. Setelah semua *thread* dimulai, kita menunggu sampai semua thread selesai menggunakan `join()`, sehingga program tidak akan melanjutkan eksekusi sampai semua *thread* selesai.

- Penerapan Hukum Amdahl
  Hukum Amdahl digunakan untuk menghitung percepatan teoretis yang diharapkan dari peningkatan jumlah prosesor (atau *thread*) untuk sebuah program. Hukum ini dinyatakan sebagai:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

  Di mana:

  - S = percepatan total.
  - P = proporsi waktu yang dapat diparalelkan.
  - N = jumlah *thread*.

  Mari kita asumsikan:

  - Waktu total tanpa paralelisasi (dengan 1 *thread*) = Dua detik (hanya ada satu *thread*).
  - Proporsi waktu yang dapat diparalelkan $P = 1$ (100% dari tugas dapat diparalelkan).
  - Jumlah *thread* $N = 4$.

  Maka, kita dapat menghitung percepatannya:

$$S = \frac{1}{(1-1) + \frac{1}{4}} = \frac{1}{0 + 0.25} = 4 \tag{2}$$

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.

4.4.1 Kelompok 7

```
import os
import sys
import subprocess

def create_file(filename):
    with open(filename, 'w') as f:
        f.write('')  # Create an empty file
    print(f'File "{filename}" created.')

def list_files():
    files = os.listdir('.')
    print("Files in current directory:")
    for file in files:
        print(file)

def delete_file(filename):
    try:
        os.remove(filename)
        print(f'File "{filename}" deleted.')
    except FileNotFoundError:
        print(f'File "{filename}" not found.')

def list_processes():
    try:
        processes = subprocess.check_output(['ps', 'aux']).
                                            decode('utf-8')
        print("Current running processes:")
        print(processes)
    except Exception as e:
        print(f'Error retrieving processes: {e}')

def system_status():
    print("System Status:")
    print(f"Current working directory: {os.getcwd()}")
```

```
    print(f"Number of files: {len(os.listdir('.'))}")

def main():
    print("Welcome to Simple CLI!")
    while True:
        command = input("Enter command (create, list, delete,
                                        processes, status,
                                       exit): ").strip().
                                       lower()

        if command == 'create':
            filename = input("Enter filename to create: ")
            create_file(filename)
        elif command == 'list':
            list_files()
        elif command == 'delete':
            filename = input("Enter filename to delete: ")
            delete_file(filename)
        elif command == 'processes':
            list_processes()
        elif command == 'status':
            system_status()
        elif command == 'exit':
            print("Exiting CLI. Goodbye!")
            sys.exit()
        else:
            print("Unknown command. Please try again.")

if __name__ == "__main__":
    main()
```

Soal :

Jelaskan bagaimana cara kerja perintah *create* dalam CLI yang telah Anda buat. Apa yang terjadi jika *file* dengan nama yang sama sudah ada di direktori saat ini?

Jawaban :

Perintah *create* dalam CLI berfungsi untuk membuat *file* baru dengan nama yang ditentukan oleh pengguna. Berikut adalah langkah-langkah yang terjadi saat perintah `create` dijalankan:

1. **Input Nama *File***: Pengguna diminta untuk memasukkan nama *file* yang ingin dibuat, misalnya `data.txt`.

2. **Membuat *File* Baru**: Aplikasi membuka *file* tersebut dalam mode

penulisan (`'w'`). Jika *file* dengan nama yang sama sudah ada, mode `'w'` akan menghapus isi *file* yang ada dan membuat *file* baru. Oleh karena itu, jika `data.txt` sudah ada, kontennya akan hilang dan *file* tersebut menjadi kosong.

3. **Menulis ke *File***: Setelah *file* dibuka, aplikasi menulis konten kosong ke dalam *file* dengan menggunakan `f.write('')`.

4. **Menampilkan Pesan Konfirmasi**: Setelah *file* berhasil dibuat, aplikasi menampilkan pesan konfirmasi kepada pengguna, mengonfirmasi bahwa *file* dengan nama yang ditentukan telah dibuat.

Jika pengguna mencoba membuat *file* dengan nama yang sama, isi dari *file* tersebut akan hilang, dan *file* baru (kosong) akan dibuat. Ini dapat menjadi risiko jika pengguna tidak menyadari bahwa *file* tersebut sudah ada, karena data yang sebelumnya tersimpan akan terhapus secara permanen.

Untuk mengatasi masalah ini, sebaiknya menambahkan logika untuk memeriksa apakah *file* sudah ada sebelum membuat *file* baru. Jika sudah ada, aplikasi dapat memberikan peringatan kepada pengguna dan meminta konfirmasi sebelum menghapus isi *file* yang ada.

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

4.5.1 Kelompok 7

```
import os
import sys

def create_file(filename):
    with open(filename, 'w') as f:
        f.write('')  # Membuat file kosong
    print(f'File "{filename}" created.')
```

```
def delete_file(filename):
    try:
        os.remove(filename)
        print(f'File "{filename}" deleted.')
    except FileNotFoundError:
        print(f'File "{filename}" not found.')

def read_file(filename):
    try:
        with open(filename, 'r') as f:
            content = f.read()
        print(f'Content of "{filename}":\n{content}')
    except FileNotFoundError:
        print(f'File "{filename}" not found.')

def write_to_file(filename, content):
    with open(filename, 'a') as f:
        f.write(content + '\n')  # Menambahkan konten ke
                                    dalam file
    print(f'Content written to "{filename}".')

def list_files():
    files = os.listdir('.')
    print("Files in current directory:")
    for file in files:
        print(file)

def main():
    print("Welcome to File System CLI!")
    while True:
        command = input("Enter command (create, delete, read,
                                     write, list, exit): "
                                    ).strip().lower()

        if command == 'create':
            filename = input("Enter filename to create: ")
            create_file(filename)
        elif command == 'delete':
            filename = input("Enter filename to delete: ")
            delete_file(filename)
        elif command == 'read':
            filename = input("Enter filename to read: ")
            read_file(filename)
        elif command == 'write':
```

```
            filename = input("Enter filename to write to: ")
            content = input("Enter content to write: ")
            write_to_file(filename, content)
        elif command == 'list':
            list_files()
        elif command == 'exit':
            print("Exiting CLI. Goodbye!")
            sys.exit()
        else:
            print("Unknown command. Please try again.")

if __name__ == "__main__":
    main()
```

Soal

Deskripsikan fungsi dari setiap perintah yang terdapat dalam kode *CLI* ini *(create, delete, read, write, list)*. Apa yang terjadi jika pengguna mencoba untuk membaca *file* yang tidak ada?

Jawaban

Dalam kode *CLI* di atas, terdapat beberapa perintah yang masing-masing memiliki fungsinya sendiri:

1. ***create***: Perintah ini digunakan untuk membuat *file* baru. Pengguna diminta untuk memasukkan nama *file* yang ingin dibuat. Jika *file* dengan nama yang sama sudah ada, isi dari *file* tersebut akan dihapus, dan *file* baru (kosong) akan dibuat.

2. ***delete***: Perintah ini digunakan untuk menghapus *file* yang ada. Pengguna memasukkan nama *file* yang ingin dihapus. Jika *file* tersebut tidak ditemukan, sistem akan memberikan pesan kesalahan yang menyatakan bahwa *file* tidak ada.

3. ***read***: Perintah ini digunakan untuk membaca isi dari *file* yang ada. Pengguna memasukkan nama *file* yang ingin dibaca. Jika *file* ditemukan, kontennya akan ditampilkan. Namun, jika *file* tidak ada, sistem akan menampilkan pesan kesalahan.

4. ***write***: Perintah ini digunakan untuk menambahkan konten baru ke dalam *file* yang ada. Pengguna diminta untuk memasukkan nama *file* dan konten yang ingin ditulis. Konten akan ditambahkan di akhir *file*, sehingga tidak menghapus data yang sudah ada.

5. ***list***: Perintah ini digunakan untuk menampilkan semua *file* yang ada di direktori saat ini. Sistem akan mencetak daftar nama *file* yang ada di dalam direktori tersebut.

Jika pengguna mencoba untuk membaca *file* yang tidak ada dengan menggunakan perintah `read`, aplikasi akan menangkap kesalahan `FileNotFoundError` dan menampilkan pesan kepada pengguna yang menyatakan bahwa *file* tersebut tidak ditemukan. Ini memastikan bahwa pengguna mendapatkan umpan balik yang jelas ketika mencoba mengakses *file* yang tidak ada, tanpa menyebabkan program *crash* (gagal berjalan).

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.