Operating Systems Training Module

A Comprehensive Guide for Beginners

Prepared by xAI Training Team May 07, 2025

Downloadable PDF version available for training purposes.

**Contents**

1. [Introduction to Operating Systems](#_bookmark0) 2
   1. [Why Study Operating Systems?](#_bookmark1) 2
   2. [Types of Operating Systems](#_bookmark2) 2
2. [Processes and Threads](#_bookmark3) 2
   1. [Process Creation Example](#_bookmark4) 2
3. [Process Scheduling](#_bookmark5) 3
   1. [Round-Robin Scheduling Concept](#_bookmark6) 3
4. [Memory Management](#_bookmark7) 4
   1. [Dynamic Memory Allocation](#_bookmark8) 4
5. [File Systems](#_bookmark9) 4
   1. [File Operations in C](#_bookmark10) 4
6. [Synchronization](#_bookmark11) 5
   1. [Mutex Example](#_bookmark12) 5
7. [Deadlock and Avoidance](#_bookmark13) 6
   1. [Deadlock Detection Concept](#_bookmark14) 6
8. [Inter-Process Communication](#_bookmark15) 6
   1. [Pipe Example](#_bookmark16) 6
9. [Virtualization and Containers](#_bookmark17) 6
   1. [Virtual Memory Concept](#_bookmark18) 6
10. [Conclusion](#_bookmark19) 7
11. [References](#_bookmark20) 7

1

# Introduction to Operating Systems

An Operating System (OS) is software that manages hardware resources and pro- vides services for applications. This module introduces OS concepts—processes, memory management, file systems, and more—with practical examples in C for system-level programming.

## Why Study Operating Systems?

* + - **Resource Management**: Optimizes CPU, memory, and I/O usage.
    - **System Understanding**: Foundation for software development and system design.
    - **Performance**: Enables efficient application execution.

## Types of Operating Systems

* + - Batch OS: Processes jobs in batches (e.g., early mainframes).
    - Time-Sharing OS: Supports multiple users (e.g., UNIX).
    - Real-Time OS: Ensures timely execution (e.g., embedded systems).

# Processes and Threads

A process is an executing program; threads are lightweight processes within a process.

## Process Creation Example

Below is a C program demonstrating process creation using fork().

**#include** <stdio.h>

**#include** <unistd.h>

**int** main() {

pid\_t pid = fork();

**if** (pid == 0) {

printf(”Child process: PID = %d\n”, getpid());

} **else if** (pid > 0) {

printf(”Parent process: PID = %d\n”, getpid());

} **else** {

printf(”Fork failed!\n”);

}

**return** 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

## Explanation:

* fork(): Creates a child process.
* pid == 0: Child process executes.
* pid > 0: Parent process executes.

# Process Scheduling

Scheduling determines which process runs next.

## Round-Robin Scheduling Concept

Round-robin assigns time slices to processes. Below is a simplified simulation.

|  |  |  |
| --- | --- | --- |
| 1 | **#include** <stdio.h> |  |
| 2 |  |
| 3 | **struct** Process { |
| 4 | **int** id; |
| 5 | **int** burstTime; |
| 6 | }; |
| 7 |  |
| 8 | **void** roundRobin(**struct** Process proc[], **int** n, **int** quantum) { |
| 9 | **int** remainingTime[n]; |
| 10  11 | **for** (**int** i = 0; i < n; i++) { remainingTime[i] = proc[i].burstTime; |
| 12 | } |
| 13 | **int** time = 0; |
| 14 | **while** (1) { |
| 15  16 | **int** done = 1;  **for** (**int** i = 0; i < n; i++) { |
| 17 | **if** (remainingTime[i] > 0) { |
| 18 | done = 0; |
| 19 | **if** (remainingTime[i] > quantum) { |
| 20 | time += quantum; |
| 21 | remainingTime[i] -= quantum; |
| 22 | } **else** { |
| 23 | time += remainingTime[i]; |
| 24  25  26 | printf(”Process %d completed at time %d\n”, [i].id, time);  remainingTime[i] = 0;  } | proc |
| 27  28 | }  } |  |
| 29  30 | **if** (done == 1) **break**;  } |  |
| 31 | } |  |
| 32 |  |  |
| 33 | **int** main() { |  |
| 34 | **struct** Process proc[] = {{1, 10}, {2, 5}, {3, 8}}; |  |
| 35 | **int** quantum = 4; |  |
| 36 | roundRobin(proc, 3, quantum); |  |
| 37 | **return** 0; |  |
| 38 | } |  |

# Memory Management

Memory management allocates and deallocates memory for processes.

## Dynamic Memory Allocation

Use malloc and free in C.

**#include** <stdio.h>

**#include** <stdlib.h>

**int** main() {

**int** \*arr;

**int** n = 5;

arr = (**int** \*)malloc(n \* **sizeof**(**int**)); **if** (arr == NULL) {

printf(”Memory allocation failed!\n”);

**return** 1;

}

**for** (**int** i = 0; i < n; i++) { arr[i] = i + 1;

}

printf(”Array: ”);

**for** (**int** i = 0; i < n; i++) { printf(”%d ”, arr[i]);

}

printf(”\n”); free(arr); **return** 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

# File Systems

File systems organize and store data on storage devices.

## File Operations in C

Read and write files using standard I/O.

**#include** <stdio.h>

**int** main() {

FILE \*file = fopen(”example.txt”, ”w”);

**if** (file == NULL) {

printf(”Error opening file!\n”);

**return** 1;

1

2

3

4

5

6

7

8

}

fprintf(file, ”Hello, Operating Systems!\n”); fclose(file);

file = fopen(”example.txt”, ”r”);

**char** buffer[100]; fgets(buffer, 100, file);

printf(”File content: %s”, buffer); fclose(file);

**return** 0;

}

9

10

11

12

13

14

15

16

17

18

# Synchronization

Synchronization prevents race conditions in concurrent processes.

## Mutex Example

Simulate a mutex using a simple lock variable (conceptual).

**#include** <stdio.h>

**#include** <pthread.h>

**int** counter = 0; pthread\_mutex\_t lock;

**void** \*increment(**void** \*arg) {

**for** (**int** i = 0; i < 1000; i++) { pthread\_mutex\_lock(&lock); counter++; pthread\_mutex\_unlock(&lock);

}

**return** NULL;

}

**int** main() {

pthread\_t t1, t2; pthread\_mutex\_init(&lock, NULL); pthread\_create(&t1, NULL, increment, NULL); pthread\_create(&t2, NULL, increment, NULL); pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

printf(”Final counter: %d\n”, counter); pthread\_mutex\_destroy(&lock);

**return** 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

# Deadlock and Avoidance

Deadlock occurs when processes block each other’s resources.

## Deadlock Detection Concept

Monitor resource allocation to detect cycles (conceptual, no code due to com- plexity).

# Inter-Process Communication

Processes communicate via mechanisms like pipes or shared memory.

## Pipe Example

Use a pipe for parent-child communication.

**#include** <stdio.h> **#include** <unistd.h> **#include** <string.h>

**int** main() {

**int** fd[2];

pipe(fd);

pid\_t pid = fork();

**if** (pid == 0) { close(fd[1]); **char** buffer[100];

read(fd[0], buffer, 100);

printf(”Child received: %s\n”, buffer); close(fd[0]);

} **else** {

close(fd[0]);

**char** \*msg = ”Hello from parent!”; write(fd[1], msg, strlen(msg) + 1); close(fd[1]);

}

**return** 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

# Virtualization and Containers

Virtualization emulates hardware; containers share the OS kernel.

## Virtual Memory Concept

Virtual memory maps process addresses to physical memory (conceptual, no code).

# Conclusion

This module covers OS fundamentals—processes, scheduling, memory, file sys- tems, synchronization, and communication. Practice these examples and ex- plore advanced topics like kernel programming and distributed systems.

# References

* Operating System Concepts, Silberschatz, Galvin, Gagne
* Linux Kernel Documentation: <https://www.kernel.org/doc>