Operating Systems Training Module

A Comprehensive Guide for Beginners

Prepared by xAI Training Team

May 07, 2025

Contents

1 Introduction to Operating Systems

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

1.1 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.

1.2 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).

2 Processes and Threads

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

2.1 Process Creation Example

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

```c
#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;
}
```

Explanation:

- `fork()`: Creates a child process.

- `pid == 0`: Child process executes.
- `pid > 0`: Parent process executes.

3 Process Scheduling

Scheduling determines which process runs next.

3.1 Round-Robin Scheduling Concept

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

```c
#include <stdio.h>

struct Process {
    int id;
    int burstTime;
};

void roundRobin(struct Process proc[], int n, int quantum) {
    int remainingTime[n];
    for (int i = 0; i < n; i++) {
        remainingTime[i] = proc[i].burstTime;
    }
    int time = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remainingTime[i] > 0) {
                done = 0;
                if (remainingTime[i] > quantum) {
                    time += quantum;
                    remainingTime[i] -= quantum;
                } else {
                    time += remainingTime[i];
                    printf("Process %d completed at time %d\n", proc
                       [i].id, time);
                    remainingTime[i] = 0;
                }
            }
        }
        if (done == 1) break;
    }
}

int main() {
    struct Process proc[] = {{1, 10}, {2, 5}, {3, 8}};
    int quantum = 4;
    roundRobin(proc, 3, quantum);
    return 0;
}
```

4 Memory Management

Memory management allocates and deallocates memory for processes.

4.1 Dynamic Memory Allocation

Use `malloc` and `free` in C.

```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;
}
```

5 File Systems

File systems organize and store data on storage devices.

5.1 File Operations in C

Read and write files using standard I/O.

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    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;
}
```

6 Synchronization

Synchronization prevents race conditions in concurrent processes.

6.1 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;
}
```

7 Deadlock and Avoidance

Deadlock occurs when processes block each other's resources.

7.1 Deadlock Detection Concept

Monitor resource allocation to detect cycles (conceptual, no code due to complexity).

8 Inter-Process Communication

Processes communicate via mechanisms like pipes or shared memory.

8.1 Pipe Example

Use a pipe for parent-child communication.

```c
#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;
}
```

9 Virtualization and Containers

Virtualization emulates hardware; containers share the OS kernel.

9.1 Virtual Memory Concept

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

10 Conclusion

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

11 References

- Operating System Concepts, Silberschatz, Galvin, Gagne
- Linux Kernel Documentation: `https://www.kernel.org/doc`