SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 1st semester

OPERATING SYSTEM (PCA20C02J)

Lab Manual

Lab 1: Understanding the Booting Process of Linux

Title

Understanding the Booting Process of Linux

Aim

To observe and understand the various stages involved in the Linux booting process, from powering on the system to the display of the login prompt.

Procedure

1. **Start the System:** Power on a Linux-based system (e.g., a virtual machine running Ubuntu or Fedora).
2. **Observe BIOS/UEFI:** Pay attention to the initial messages displayed by the BIOS/UEFI firmware, which performs POST (Power-On Self-Test) and identifies bootable devices.
3. **GRUB/Bootloader Stage:** Observe the GRUB (Grand Unified Bootloader) menu (if configured), which allows selecting the operating system kernel.
4. **Kernel Loading and Initialization:** Watch for messages indicating the loading of the Linux kernel into memory and its initial setup.
5. **Init/Systemd Process:** Observe the messages related to the `init` process (or `systemd` in modern Linux distributions) taking over, mounting file systems, and starting essential services.
6. **Service Startup:** Note the various system services and daemons being started.
7. **Login Prompt:** Identify the final stage where the system presents the login prompt.
8. **Review Boot Logs (Optional):** After successful boot, you can examine system logs (e.g., `dmesg`, `/var/log/boot.log`, `journalctl`) to review the boot sequence in detail.

Source Code

```
# No specific source code for observation.
# Commands to review boot logs after system is up:
dmesg | less
cat /var/log/boot.log
journalctl -b
```

Input

N/A (System power-on)

Expected Output

Observation of sequential boot messages on the console, leading to a functional login prompt. Output from `dmesg` or `journalctl` showing kernel and system startup messages.

Lab 2: Understand the Behaviour of the OS and Get the CPU Type and Model

Title

Understanding OS Behavior and Retrieving CPU Information

Aim

To explore basic operating system behavior through system commands and to identify the CPU type and model of the underlying hardware.

Procedure

1. **Open Terminal:** Open a terminal window in your Linux environment.
2. **Check OS Version:** Use commands to identify the Linux distribution and kernel version.
3. **Monitor System Load:** Use commands to observe CPU utilization, memory usage, and running processes.
4. **Identify CPU Information:** Use specific commands to extract detailed information about the CPU, including its vendor, model name, core count, and architecture.
5. **Observe Process Behavior:** Start a simple background process and observe its entry in the process list.

Source Code

```
# Commands to execute in the terminal:

# Check OS version
cat /etc/os-release
uname -a

# Monitor system load (exit with 'q')
top
htop # If installed

# Get CPU information
cat /proc/cpuinfo | grep "model name"
cat /proc/cpuinfo | grep "vendor_id"
lscpu

# Observe process behavior (example: a simple sleep process)
sleep 60 &
ps aux | grep sleep
```

Input

N/A (Commands executed directly)

Expected Output

- Output showing the Linux distribution name and version.
- Output from `top`/`htop` displaying real-time system resource usage.

- Lines from `/proc/cpuinfo` or `lscpu` detailing the CPU model name, vendor, and other specifications.
- The `ps aux` command showing the `sleep 60` process running in the background.

Lab 3: Understanding Various Phases of Compilation and System Admin Commands - Simple Task Automations

Title

Compilation Phases and Simple System Task Automations

Aim

To understand the distinct phases involved in compiling a C program and to implement basic system administration task automations using shell scripting.

Procedure

1. **Create a Simple C Program:** Write a basic "Hello, World!" C program.
2. **Compilation Phases (Manual):**
   - **Preprocessing:** Use `gcc -E` to see the preprocessed output.
   - **Compilation:** Use `gcc -S` to generate assembly code.
   - **Assembly:** Use `gcc -c` to generate object code.
   - **Linking:** Use `gcc` (default) to create the executable.
3. **Automate a Task:**
   - **Scenario:** Create a shell script to regularly clean up temporary files in a specific directory.
   - **Scripting:** Write a shell script that identifies and deletes files older than a certain number of days.
   - **Execution:** Run the script and verify its functionality.

Source Code

````
// hello.c
#include <stdio.h>

int main() {
    printf("Hello, Lab Manual!\n");
    return 0;
}
```bash
#!/bin/bash
# cleanup.sh - A simple script to clean up old files

TARGET_DIR="/tmp/my_temp_files" # Change as needed for testing
DAYS_OLD=7

echo "Starting cleanup in $TARGET_DIR..."

# Create a dummy directory and some files for testing
mkdir -p "$TARGET_DIR"
touch "$TARGET_DIR/file1.txt"
touch -d "2 weeks ago" "$TARGET_DIR/old_file.log"
touch -d "3 days ago" "$TARGET_DIR/recent_file.tmp"

# Find and delete files older than DAYS_OLD
find "$TARGET_DIR" -type f -mtime +$DAYS_OLD -delete -print

echo "Cleanup complete."
````
````

Input

- For compilation: `hello.c`
- For automation: N/A (script runs with predefined parameters)

Expected Output

- **Compilation:**
    - `hello.i` (preprocessed output)
    - `hello.s` (assembly code)
    - `hello.o` (object code)
    - `a.out` (executable)
    - Running `a.out` should print "Hello, Lab Manual!"
- **Automation:**
    - Output from `cleanup.sh` indicating which files were deleted (e.g., `deleting ./tmp/my_temp_files/old_file.log`).
    - Verification that `old_file.log` is removed from `$TARGET_DIR` while other files remain.

Lab 4: System Admin Commands - Basics

Title

Basic System Administration Commands

Aim

To familiarize with essential system administration commands for managing files, directories, processes, and users in a Linux environment.

Procedure

1. **File and Directory Management:**
    - Create, copy, move, and delete files and directories.
    - Change file permissions and ownership.
    - List directory contents with various options.
2. **Process Management:**
    - List running processes.
    - Send signals to processes (e.g., kill).
    - Run processes in the background.
3. **User Management (Conceptual/Observation):**
    - List existing users.
    - Understand the basic structure of user accounts.
4. **System Information:**
    - Check disk space usage.
    - Check memory usage.
    - View network configuration.

Source Code

```
# Commands to execute in the terminal:

# File and Directory Management
mkdir my_dir
touch my_dir/file.txt
echo "Hello content" > my_dir/another_file.txt
ls -l my_dir
cp my_dir/file.txt my_dir/file_copy.txt
mv my_dir/another_file.txt my_dir/renamed_file.txt
chmod 755 my_dir/file_copy.txt
# sudo chown user:group my_dir/file.txt # Requires sudo, observe only
rm my_dir/file.txt
rmdir my_dir # Will fail if not empty, use rm -r my_dir for non-empty

# Process Management
sleep 100 & # Run in background
ps aux | grep sleep
kill <PID_of_sleep> # Replace <PID_of_sleep> with actual PID from ps aux
jobs # List background jobs

# User Management (Observation)
cat /etc/passwd | head -n 5 # View first 5 user entries
whoami
id
```

```
# System Information
df -h # Disk Free (human readable)
free -h # Memory Free (human readable)
ip a # Network interfaces
```

Input

N/A (Commands executed directly)

Expected Output

- Successful creation, manipulation, and deletion of files/directories.
- Output from `ls -l` showing permissions and ownership.
- `ps aux` listing the `sleep` process, and its termination after `kill`.
- Output from `cat /etc/passwd`, `whoami`, `id` showing user information.
- Outputs from `df -h`, `free -h`, `ip a` showing system resource and network details.

Lab 5: Shell Programs - Basic Level

Title

Basic Shell Scripting

Aim

To write and execute simple shell scripts to automate repetitive tasks and demonstrate fundamental scripting concepts like variables, conditional statements, and loops.

Procedure

1. **Create a Script File:** Create a new file with a `.sh` extension (e.g., `myscript.sh`).
2. **Add Shebang:** Start the script with `#!/bin/bash`.
3. **Variables:** Declare and use variables.
4. **User Input:** Prompt the user for input and store it in a variable.
5. **Conditional Statements:** Use `if-else` to perform different actions based on conditions.
6. **Loops:** Use `for` or `while` loops for repetitive tasks.
7. **Execute Script:** Make the script executable (`chmod +x myscript.sh`) and run it (`./myscript.sh`).

Source Code

```
#!/bin/bash
# myscript.sh - A basic shell script

echo "Hello, this is a basic shell script."

# Variable example
GREETING="Welcome"
NAME="User"
echo "$GREETING, $NAME!"

# User input example
read -p "Enter your favorite color: " COLOR
echo "You entered: $COLOR"

# Conditional statement example
if [ "$COLOR" == "blue" ]; then
    echo "Blue is a great color!"
else
    echo "That's an interesting color."
fi

# Loop example
echo "Counting from 1 to 3:"
for i in 1 2 3; do
    echo "Count: $i"
done

echo "Script finished."
```

Input

User will be prompted to enter their favorite color.

Expected Output

```
Hello, this is a basic shell script.
Welcome, User!
Enter your favorite color: [User enters a color, e.g., red]
You entered: red
That's an interesting color.
Counting from 1 to 3:
Count: 1
Count: 2
Count: 3
Script finished.
```

(Output for the conditional statement will vary based on user input.)

Lab 6: Process Creation and Overlay Concept

Title

Process Creation and Overlay using `fork()` and `exec()`

Aim

To understand how new processes are created in Linux using the `fork()` system call and how a new program can be loaded into an existing process's address space using the `exec()` family of system calls (overlay concept).

Procedure

1. **Process Creation (`fork()`):**
   - Write a C program that uses `fork()` to create a child process.
   - In the parent process, print its PID and the child's PID.
   - In the child process, print its PID and its parent's PID.
   - Demonstrate that both processes run concurrently.
2. **Overlay (`exec()`):**
   - Modify the child process in the above program to call one of the `exec()` functions (e.g., `execlp`).
   - Have the `exec()` function load and execute another simple program (e.g., `ls` or a custom "Hello" program).
   - Observe that the child process's original code is replaced by the new program.

Source Code

```
// fork_exec_demo.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    printf("Parent process (PID: %d) starting...\n", getpid());

    pid = fork(); // Create a child process

    if (pid < 0) {
        // Error occurred
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process (PID: %d, Parent PID: %d) created.\n", getpid(),
getppid());
        printf("Child process is now overlaying with 'ls -l /tmp'...\n");
        // Overlay the child process with the 'ls -l /tmp' command
        execlp("ls", "ls", "-l", "/tmp", NULL);
        // If execlp returns, it means an error occurred
        perror("execlp failed");
        exit(1); // Exit child process if exec fails
    } else {
```

```
        // Parent process
        printf("Parent process (PID: %d) waiting for child (PID: %d)...\n",
getpid(), pid);
        wait(NULL); // Wait for the child process to complete
        printf("Child process finished. Parent process (PID: %d) exiting.\n",
getpid());
    }

    return 0;
}
```

Input

N/A

Expected Output

```
Parent process (PID: [parent_pid]) starting...
Parent process (PID: [parent_pid]) waiting for child (PID: [child_pid])...
Child process (PID: [child_pid], Parent PID: [parent_pid]) created.
Child process is now overlaying with 'ls -l /tmp'...
# Output of 'ls -l /tmp' will appear here, e.g.:
total 4
drwxrwxrwt 2 root root 4096 May 21 09:00 .
drwxr-xr-x 1 root root 4096 May 21 09:00 ..
-rw-r--r-- 1 user user    0 May 21 09:00 my_temp_file.txt
Child process finished. Parent process (PID: [parent_pid]) exiting.
```

(The order of "Parent waiting" and "Child created" might vary slightly due to scheduling.)

Lab 7: File System and Working with Test Programs

Title

File System Basics and File I/O Programs

Aim

To understand the fundamental concepts of a file system and to write C programs for basic file input/output operations like creating, writing to, and reading from files.

Procedure

1. **File System Navigation (Conceptual):** Discuss the hierarchical structure of the Linux file system.
2. **C Program - File Creation and Writing:**
   - Write a C program that opens a file in write mode (`"w"`).
   - Write some text content into the file.
   - Close the file.
   - Verify the file content using `cat`.
3. **C Program - File Reading:**
   - Write a C program that opens an existing file in read mode (`"r"`).
   - Read content character by character or line by line.
   - Print the read content to the console.
   - Close the file.

Source Code

```
// write_file.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fp;
    char data[] = "This is a test line.\nAnother line of text.\n";

    fp = fopen("output.txt", "w"); // Open file in write mode

    if (fp == NULL) {
        perror("Error opening file for writing");
        return 1;
    }

    fprintf(fp, "%s", data); // Write data to file
    printf("Data written to output.txt\n");

    fclose(fp); // Close the file
    return 0;
}
```
```c
// read_file.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fp;
```

```
    char buffer[255]; // Buffer to store read data

    fp = fopen("output.txt", "r"); // Open file in read mode

    if (fp == NULL) {
        perror("Error opening file for reading");
        return 1;
    }

    printf("Content of output.txt:\n");
    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("%s", buffer); // Print content to console
    }

    fclose(fp); // Close the file
    return 0;
}
```

Input

- For `write_file.c`: N/A (content is hardcoded)
- For `read_file.c`: `output.txt` (created by `write_file.c`)

Expected Output

- **`write_file.c`:**
- `Data written to output.txt`

  And a file named `output.txt` created in the same directory with:

  ```
  This is a test line.
  Another line of text.
  ```

- **`read_file.c`:**
- `Content of output.txt:`
- `This is a test line.`
- `Another line of text.`

Lab 8: Programs Using File System

Title

Advanced File System Operations with C Programs

Aim

To implement C programs that interact with the file system beyond basic read/write, including operations like listing directory contents, checking file types, and manipulating file permissions.

Procedure

1. **C Program - Directory Listing:**
   - Write a C program that takes a directory path as a command-line argument.
   - Open the directory and read its entries.
   - Print the names of all files and subdirectories within it.
2. **C Program - File Information:**
   - Write a C program that takes a file path as a command-line argument.
   - Use `stat()` to retrieve file information (e.g., size, permissions, last modified time, file type).
   - Print the retrieved information.

Source Code

```
// list_dir.c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h> // For directory operations
#include <sys/types.h> // For opendir, readdir

int main(int argc, char *argv[]) {
    DIR *dp;
    struct dirent *entry;
    char *dir_path;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <directory_path>\n", argv[0]);
        return 1;
    }

    dir_path = argv[1];
    dp = opendir(dir_path); // Open the directory

    if (dp == NULL) {
        perror("Error opening directory");
        return 1;
    }

    printf("Contents of directory '%s':\n", dir_path);
    while ((entry = readdir(dp)) != NULL) {
        printf("%s\n", entry->d_name); // Print entry name
    }

    closedir(dp); // Close the directory
    return 0;
}
```

```c
// file_info.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h> // For stat()
#include <time.h>     // For ctime()

int main(int argc, char *argv[]) {
    struct stat file_stat;
    char *file_path;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file_path>\n", argv[0]);
        return 1;
    }

    file_path = argv[1];

    if (stat(file_path, &file_stat) == -1) {
        perror("Error getting file status");
        return 1;
    }

    printf("File Information for '%s':\n", file_path);
    printf("  Size: %ld bytes\n", file_stat.st_size);
    printf("  Permissions: %o\n", file_stat.st_mode & 0777); // Octal
permissions
    printf("  Last Modified: %s", ctime(&file_stat.st_mtime));
    printf("  File Type: ");
    if (S_ISREG(file_stat.st_mode)) {
        printf("Regular File\n");
    } else if (S_ISDIR(file_stat.st_mode)) {
        printf("Directory\n");
    } else if (S_ISLNK(file_stat.st_mode)) {
        printf("Symbolic Link\n");
    } else {
        printf("Other\n");
    }

    return 0;
}
```

Input

- For `list_dir.c`: A directory path (e.g., `/home/user` or `/tmp`)
- For `file_info.c`: A file path (e.g., `output.txt` from Lab 9)

Expected Output

- **`list_dir.c`:**
- `Contents of directory '/tmp':`
- `.`
- `..`
- `file1.txt`
- `my_temp_files`

  (Actual output will depend on the contents of the specified directory)

- **`file_info.c`:**
- `File Information for 'output.txt':`

- `Size: 40 bytes`
- `Permissions: 644`
- `Last Modified: Wed May 21 09:00:00 2025`
- `File Type: Regular File`

(Values will vary based on the actual file)

Lab 9: Programs to Implement Shared Memory

Title

Shared Memory Implementation for Inter-Process Communication (IPC)

Aim

To understand and implement inter-process communication (IPC) using shared memory, allowing multiple processes to access and modify the same region of memory.

Procedure

1. **Shared Memory Writer Program:**
   - Write a C program that creates a shared memory segment using `shmget()`.
   - Attach the shared memory segment to its address space using `shmat()`.
   - Write some data into the shared memory.
   - Detach the shared memory using `shmdt()`.
   - (Optional) Mark the shared memory for deletion using `shmctl()`.
2. **Shared Memory Reader Program:**
   - Write a C program that accesses the same shared memory segment (using the same key as the writer).
   - Attach the shared memory segment.
   - Read the data written by the writer program.
   - Print the read data.
   - Detach the shared memory.

Source Code

```
// shm_writer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define SHM_KEY 1234 // A unique key for the shared memory segment
#define SHM_SIZE 1024 // Size of the shared memory segment

int main() {
    int shm_id;
    char *shm_ptr;
    const char *message = "Hello from shared memory!";

    // 1. Create a shared memory segment
    shm_id = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
    if (shm_id == -1) {
        perror("shmget failed");
        return 1;
    }
    printf("Shared memory segment created with ID: %d\n", shm_id);

    // 2. Attach the shared memory segment
    shm_ptr = (char *)shmat(shm_id, NULL, 0);
    if (shm_ptr == (char *)-1) {
```

```
        perror("shmat failed");
        return 1;
    }
    printf("Shared memory attached at address: %p\n", shm_ptr);

    // 3. Write data to shared memory
    strncpy(shm_ptr, message, SHM_SIZE - 1);
    shm_ptr[SHM_SIZE - 1] = '\0'; // Ensure null termination
    printf("Data written to shared memory: '%s'\n", shm_ptr);

    printf("Waiting for reader to read (press Enter to detach and
delete)...\n");
    getchar(); // Wait for user input

    // 4. Detach the shared memory
    if (shmdt(shm_ptr) == -1) {
        perror("shmdt failed");
        return 1;
    }
    printf("Shared memory detached.\n");

    // 5. Mark the shared memory for deletion
    if (shmctl(shm_id, IPC_RMID, NULL) == -1) {
        perror("shmctl (IPC_RMID) failed");
        return 1;
    }
    printf("Shared memory marked for deletion.\n");

    return 0;
}
```
```c
// shm_reader.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

#define SHM_KEY 1234 // Must be the same key as the writer
#define SHM_SIZE 1024

int main() {
    int shm_id;
    char *shm_ptr;

    // 1. Get the shared memory segment (do not create)
    shm_id = shmget(SHM_KEY, SHM_SIZE, 0666);
    if (shm_id == -1) {
        perror("shmget failed (ensure writer is running first)");
        return 1;
    }
    printf("Shared memory segment found with ID: %d\n", shm_id);

    // 2. Attach the shared memory segment
    shm_ptr = (char *)shmat(shm_id, NULL, 0);
    if (shm_ptr == (char *)-1) {
        perror("shmat failed");
        return 1;
    }
    printf("Shared memory attached at address: %p\n", shm_ptr);

    // 3. Read data from shared memory
    printf("Data read from shared memory: '%s'\n", shm_ptr);

    // 4. Detach the shared memory
    if (shmdt(shm_ptr) == -1) {
```

```
        perror("shmdt failed");
        return 1;
    }
    printf("Shared memory detached.\n");

    return 0;
}
```

Input

- For `shm_writer.c`: Press Enter to proceed after data is written.
- For `shm_reader.c`: N/A

Expected Output

- **Run `shm_writer.c` first:**
- `Shared memory segment created with ID: [shm_id]`
- `Shared memory attached at address: [memory_address]`
- `Data written to shared memory: 'Hello from shared memory!'`
- `Waiting for reader to read (press Enter to detach and delete)...`

- **While `shm_writer.c` is waiting, run `shm_reader.c` in another terminal:**
- `Shared memory segment found with ID: [shm_id]`
- `Shared memory attached at address: [memory_address]`
- `Data read from shared memory: 'Hello from shared memory!'`
- `Shared memory detached.`

- **Then, press Enter in the `shm_writer.c` terminal:**
- `Shared memory detached.`
- `Shared memory marked for deletion.`

Lab 10: Understand the Paging Operations

Title

Understanding and Simulating Paging Operations

Aim

To understand the concept of paging in memory management, including virtual addresses, physical addresses, page tables, and page faults, and to simulate basic paging operations.

Procedure

1. **Conceptual Understanding:** Review the principles of paging, including the translation of virtual addresses to physical addresses using a page table.
2. **Simulation Program:**
   - Write a C program to simulate a simple paging system.
   - Define a small virtual memory space and a physical memory space.
   - Implement a simplified page table.
   - Simulate memory access requests (virtual addresses).
   - For each access, determine if it's a page hit or a page fault.
   - If a page fault occurs, simulate loading the page into a free frame (using a simple page replacement policy like FIFO or LRU if time permits, otherwise just allocate if space is available).
   - Translate the virtual address to its corresponding physical address.

Source Code

```
// paging_simulation.c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define VIRTUAL_PAGES 8   // Number of virtual pages
#define PHYSICAL_FRAMES 4 // Number of physical frames (main memory)
#define PAGE_SIZE 1024    // Size of each page/frame in bytes

// Simplified Page Table Entry
typedef struct {
    int frame_number; // -1 if not in memory
    bool valid;       // True if page is in memory
    // Add other fields like dirty bit, reference bit if needed for advanced
policies
} PageTableEntry;

PageTableEntry page_table[VIRTUAL_PAGES];
int physical_memory[PHYSICAL_FRAMES]; // Represents physical frames
int next_free_frame = 0;              // For simple allocation

void initialize_page_table() {
    for (int i = 0; i < VIRTUAL_PAGES; i++) {
        page_table[i].frame_number = -1;
        page_table[i].valid = false;
    }
    printf("Page table initialized.\n");
}
```

```
void simulate_memory_access(int virtual_address) {
    int page_number = virtual_address / PAGE_SIZE;
    int offset = virtual_address % PAGE_SIZE;

    printf("\nAccessing Virtual Address: %d (Page: %d, Offset: %d)\n",
           virtual_address, page_number, offset);

    if (page_number >= VIRTUAL_PAGES || page_number < 0) {
        printf("Error: Invalid virtual address (page number out of bounds).\n");
        return;
    }

    if (page_table[page_number].valid) {
        // Page Hit
        int physical_address = page_table[page_number].frame_number * PAGE_SIZE
+ offset;
        printf("  PAGE HIT! Page %d is in Frame %d.\n", page_number,
page_table[page_number].frame_number);
        printf("  Translated to Physical Address: %d\n", physical_address);
    } else {
        // Page Fault
        printf("  PAGE FAULT! Page %d not in memory.\n", page_number);

        if (next_free_frame < PHYSICAL_FRAMES) {
            // Allocate a free frame (simple FIFO/next available)
            page_table[page_number].frame_number = next_free_frame;
            page_table[page_number].valid = true;
            printf("  Allocated Page %d to Frame %d.\n", page_number,
next_free_frame);
            next_free_frame++;
            int physical_address = page_table[page_number].frame_number *
PAGE_SIZE + offset;
            printf("  Translated to Physical Address: %d\n", physical_address);
        } else {
            // No free frames, need a replacement policy (simplified: just
report full)
            printf("  Physical memory is full. Cannot load page %d (requires
replacement policy).\n", page_number);
        }
    }
}

void print_page_table() {
    printf("\n--- Current Page Table ---\n");
    printf("Page | Frame | Valid\n");
    printf("-----|-------|------\n");
    for (int i = 0; i < VIRTUAL_PAGES; i++) {
        printf("%4d | %5d | %5s\n", i, page_table[i].frame_number,
page_table[i].valid ? "Yes" : "No");
    }
    printf("--------------------------\n");
}

int main() {
    initialize_page_table();
    print_page_table();

    // Simulate a sequence of memory accesses
    simulate_memory_access(1000); // Page 0, Offset 1000
    simulate_memory_access(2500); // Page 2, Offset 400
    simulate_memory_access(1500); // Page 1, Offset 450
    simulate_memory_access(500);  // Page 0, Offset 500 (Hit)
    simulate_memory_access(3000); // Page 2, Offset 952 (Hit)
    simulate_memory_access(4000); // Page 3, Offset 904
    simulate_memory_access(6000); // Page 5, Offset 808 (Page Fault, no free
frame)
```

```
    print_page_table();

    return 0;
}
```

Input

N/A (Memory access requests are hardcoded in the `main` function)

Expected Output

```
Page table initialized.

--- Current Page Table ---
Page | Frame | Valid
-----|-------|------
   0 |    -1 |    No
   1 |    -1 |    No
   2 |    -1 |    No
   3 |    -1 |    No
   4 |    -1 |    No
   5 |    -1 |    No
   6 |    -1 |    No
   7 |    -1 |    No
--------------------------

Accessing Virtual Address: 1000 (Page: 0, Offset: 1000)
  PAGE FAULT! Page 0 not in memory.
  Allocated Page 0 to Frame 0.
  Translated to Physical Address: 1000

Accessing Virtual Address: 2500 (Page: 2, Offset: 400)
  PAGE FAULT! Page 2 not in memory.
  Allocated Page 2 to Frame 1.
  Translated to Physical Address: 1424

Accessing Virtual Address: 1500 (Page: 1, Offset: 450)
  PAGE FAULT! Page 1 not in memory.
  Allocated Page 1 to Frame 2.
  Translated to Physical Address: 2500

Accessing Virtual Address: 500 (Page: 0, Offset: 500)
  PAGE HIT! Page 0 is in Frame 0.
  Translated to Physical Address: 500

Accessing Virtual Address: 3000 (Page: 2, Offset: 952)
  PAGE HIT! Page 2 is in Frame 1.
  Translated to Physical Address: 2000

Accessing Virtual Address: 4000 (Page: 3, Offset: 904)
  PAGE FAULT! Page 3 not in memory.
  Allocated Page 3 to Frame 3.
  Translated to Physical Address: 3904

Accessing Virtual Address: 6000 (Page: 5, Offset: 808)
  PAGE FAULT! Page 5 not in memory.
  Physical memory is full. Cannot load page 5 (requires replacement policy).

--- Current Page Table ---
Page | Frame | Valid
-----|-------|------
   0 |     0 |   Yes
   1 |     2 |   Yes
```

```
   2 |      1 |   Yes
   3 |      3 |   Yes
   4 |     -1 |    No
   5 |     -1 |    No
   6 |     -1 |    No
   7 |     -1 |    No
--------------------------
```

Lab 11: Program to Implement File System Interface

Title

Implementing a Simplified File System Interface

Aim

To understand the conceptual design of a file system and to implement a simplified interface that mimics basic file system operations like creating, opening, reading, writing, and closing files, without directly using the operating system's file I/O calls. This will involve managing blocks and inodes.

Procedure

1. **Conceptual Design:** Design a simple file system structure. This might involve:
   - A simulated disk (e.g., a large array or a file).
   - A superblock to store file system metadata.
   - An inode table to store file metadata (size, block pointers, permissions).
   - A data block area for file content.
   - A free block list/bitmap.
2. **Implement Basic Functions:**
   - `my_mkfs()`: Format the simulated disk (initialize superblock, inode table, free list).
   - `my_open(filename, mode)`: Find/create inode, return a file descriptor.
   - `my_read(fd, buffer, size)`: Read data from blocks pointed to by inode.
   - `my_write(fd, buffer, size)`: Write data, allocate new blocks if needed.
   - `my_close(fd)`: Close the file.
   - `my_ls()`: List files in the root directory.

Source Code

```
// simple_fs.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define DISK_SIZE (1024 * 10) // 10 KB simulated disk
#define BLOCK_SIZE 128        // 128 bytes per block
#define NUM_BLOCKS (DISK_SIZE / BLOCK_SIZE)
#define NUM_INODES 16         // Max 16 files
#define MAX_FILENAME_LEN 16
#define MAX_FILE_BLOCKS 8     // Max blocks per file (128 * 8 = 1KB max file
size)

// Simulated Disk
unsigned char simulated_disk[DISK_SIZE];

// Superblock (simplified)
typedef struct {
    int num_inodes;
    int num_blocks;
    int free_blocks_count;
    int free_inode_count;
    // Pointers/offsets to inode table, data blocks, free list
} Superblock;
```

```
// Inode (simplified)
typedef struct {
    char filename[MAX_FILENAME_LEN];
    int file_size;
    int block_pointers[MAX_FILE_BLOCKS]; // Pointers to data blocks
    bool in_use;
} Inode;

Superblock sb;
Inode inode_table[NUM_INODES];
bool block_bitmap[NUM_BLOCKS]; // true if block is free

// --- File System Functions ---

void init_disk() {
    memset(simulated_disk, 0, DISK_SIZE);
    printf("Simulated disk initialized.\n");
}

void my_mkfs() {
    init_disk();

    // Initialize Superblock
    sb.num_inodes = NUM_INODES;
    sb.num_blocks = NUM_BLOCKS;
    sb.free_blocks_count = NUM_BLOCKS;
    sb.free_inode_count = NUM_INODES;
    printf("Superblock initialized.\n");

    // Initialize Inode Table
    for (int i = 0; i < NUM_INODES; i++) {
        inode_table[i].in_use = false;
        inode_table[i].file_size = 0;
        memset(inode_table[i].filename, 0, MAX_FILENAME_LEN);
        for (int j = 0; j < MAX_FILE_BLOCKS; j++) {
            inode_table[i].block_pointers[j] = -1; // -1 indicates no block
        }
    }
    printf("Inode table initialized.\n");

    // Initialize Block Bitmap (all blocks free initially)
    for (int i = 0; i < NUM_BLOCKS; i++) {
        block_bitmap[i] = true; // true means free
    }
    printf("Block bitmap initialized (all blocks free).\n");

    printf("File system formatted successfully.\n");
}

// Find a free inode
int find_free_inode() {
    for (int i = 0; i < NUM_INODES; i++) {
        if (!inode_table[i].in_use) {
            return i;
        }
    }
    return -1; // No free inode
}

// Find a free data block
int find_free_block() {
    for (int i = 0; i < NUM_BLOCKS; i++) {
        if (block_bitmap[i]) {
            block_bitmap[i] = false; // Mark as used
            sb.free_blocks_count--;
            return i;
```

```
        }
    }
    return -1; // No free block
}

// Get inode by filename
int get_inode_by_name(const char *filename) {
    for (int i = 0; i < NUM_INODES; i++) {
        if (inode_table[i].in_use && strcmp(inode_table[i].filename, filename)
== 0) {
            return i;
        }
    }
    return -1; // Not found
}

// Simplified file descriptor (just inode index for now)
typedef int file_descriptor;

file_descriptor my_open(const char *filename, const char *mode) {
    int inode_idx = get_inode_by_name(filename);

    if (strcmp(mode, "w") == 0) {
        if (inode_idx != -1) {
            // File exists, truncate it (for simplicity, just reset size and
blocks)
            inode_table[inode_idx].file_size = 0;
            for (int i = 0; i < MAX_FILE_BLOCKS; i++) {
                if (inode_table[inode_idx].block_pointers[i] != -1) {
                    block_bitmap[inode_table[inode_idx].block_pointers[i]] =
true; // Free block
                    sb.free_blocks_count++;
                    inode_table[inode_idx].block_pointers[i] = -1;
                }
            }
            printf("File '%s' truncated.\n", filename);
            return inode_idx;
        } else {
            // File does not exist, create new
            int new_inode_idx = find_free_inode();
            if (new_inode_idx == -1) {
                printf("Error: No free inodes to create file '%s'.\n",
filename);
                return -1;
            }
            strncpy(inode_table[new_inode_idx].filename, filename,
MAX_FILENAME_LEN - 1);
            inode_table[new_inode_idx].filename[MAX_FILENAME_LEN - 1] = '\0';
            inode_table[new_inode_idx].in_use = true;
            sb.free_inode_count--;
            printf("File '%s' created (inode %d).\n", filename, new_inode_idx);
            return new_inode_idx;
        }
    } else if (strcmp(mode, "r") == 0) {
        if (inode_idx == -1) {
            printf("Error: File '%s' not found for reading.\n", filename);
            return -1;
        }
        printf("File '%s' opened for reading.\n", filename);
        return inode_idx;
    } else {
        printf("Error: Unsupported mode '%s'. Use 'r' or 'w'.\n", mode);
        return -1;
    }
}

int my_write(file_descriptor fd, const char *buffer, int size) {
```

```
    if (fd == -1 || !inode_table[fd].in_use) {
        printf("Error: Invalid file descriptor.\n");
        return -1;
    }

    int bytes_written = 0;
    int remaining_size = size;
    int current_block_idx = inode_table[fd].file_size / BLOCK_SIZE;
    int offset_in_block = inode_table[fd].file_size % BLOCK_SIZE;

    while (remaining_size > 0 && current_block_idx < MAX_FILE_BLOCKS) {
        int block_num = inode_table[fd].block_pointers[current_block_idx];
        if (block_num == -1) {
            // Need to allocate a new block
            block_num = find_free_block();
            if (block_num == -1) {
                printf("Error: No free blocks left to write all data.\n");
                break;
            }
            inode_table[fd].block_pointers[current_block_idx] = block_num;
        }

        int write_this_block = BLOCK_SIZE - offset_in_block;
        if (write_this_block > remaining_size) {
            write_this_block = remaining_size;
        }

        memcpy(simulated_disk + (block_num * BLOCK_SIZE) + offset_in_block,
               buffer + bytes_written, write_this_block);

        bytes_written += write_this_block;
        remaining_size -= write_this_block;
        inode_table[fd].file_size += write_this_block;

        // Move to next block if current one is full
        if (offset_in_block + write_this_block >= BLOCK_SIZE) {
            current_block_idx++;
            offset_in_block = 0;
        } else {
            offset_in_block += write_this_block;
        }
    }
    printf("Wrote %d bytes to file '%s'. Current size: %d\n", bytes_written,
inode_table[fd].filename, inode_table[fd].file_size);
    return bytes_written;
}

int my_read(file_descriptor fd, char *buffer, int size) {
    if (fd == -1 || !inode_table[fd].in_use) {
        printf("Error: Invalid file descriptor.\n");
        return -1;
    }

    int bytes_read = 0;
    int remaining_to_read = size;
    if (remaining_to_read > inode_table[fd].file_size) {
        remaining_to_read = inode_table[fd].file_size; // Cannot read more than
file size
    }

    int current_block_idx = 0;
    int offset_in_block = 0;

    while (bytes_read < remaining_to_read && current_block_idx <
MAX_FILE_BLOCKS) {
        int block_num = inode_table[fd].block_pointers[current_block_idx];
        if (block_num == -1) {
```

```
            break; // No more blocks for this file
        }

        int read_this_block = BLOCK_SIZE - offset_in_block;
        if (read_this_block > (remaining_to_read - bytes_read)) {
            read_this_block = (remaining_to_read - bytes_read);
        }

        memcpy(buffer + bytes_read,
               simulated_disk + (block_num * BLOCK_SIZE) + offset_in_block,
               read_this_block);

        bytes_read += read_this_block;
        offset_in_block += read_this_block;

        if (offset_in_block >= BLOCK_SIZE) {
            current_block_idx++;
            offset_in_block = 0;
        }
    }
    buffer[bytes_read] = '\0'; // Null-terminate the read buffer
    printf("Read %d bytes from file '%s'.\n", bytes_read,
inode_table[fd].filename);
    return bytes_read;
}

void my_close(file_descriptor fd) {
    if (fd != -1 && inode_table[fd].in_use) {
        printf("File '%s' closed.\n", inode_table[fd].filename);
    } else {
        printf("Error: Invalid file descriptor for close.\n");
    }
}

void my_ls() {
    printf("\n--- Files in File System ---\n");
    printf("Filename        | Size (bytes)\n");
    printf("----------------|--------------\n");
    bool found_files = false;
    for (int i = 0; i < NUM_INODES; i++) {
        if (inode_table[i].in_use) {
            printf("%-15s | %d\n", inode_table[i].filename,
inode_table[i].file_size);
            found_files = true;
        }
    }
    if (!found_files) {
        printf("No files found.\n");
    }
    printf("----------------------------\n");
    printf("Free blocks: %d / %d\n", sb.free_blocks_count, sb.num_blocks);
    printf("Free inodes: %d / %d\n", sb.free_inode_count, sb.num_inodes);
}

int main() {
    my_mkfs();
    my_ls();

    file_descriptor fd1 = my_open("my_file.txt", "w");
    if (fd1 != -1) {
        my_write(fd1, "Hello, this is a test file content for our simple file
system.", 60);
        my_write(fd1, "Adding more data to the file to exceed one block size.",
55);
        my_close(fd1);
    }
```

```c
    file_descriptor fd2 = my_open("another.log", "w");
    if (fd2 != -1) {
        my_write(fd2, "Log entry 1.\nLog entry 2.", 25);
        my_close(fd2);
    }

    my_ls();

    file_descriptor fd_read = my_open("my_file.txt", "r");
    if (fd_read != -1) {
        char read_buffer[200];
        int bytes_read = my_read(fd_read, read_buffer, sizeof(read_buffer) - 1);
        if (bytes_read > 0) {
            printf("Content of 'my_file.txt':\n%s\n", read_buffer);
        }
        my_close(fd_read);
    }

    file_descriptor fd_non_existent = my_open("non_existent.txt", "r");

    return 0;
}
```

Input

N/A (Operations are hardcoded in `main`)

Expected Output

```
Simulated disk initialized.
Superblock initialized.
Inode table initialized.
Block bitmap initialized (all blocks free).
File system formatted successfully.

--- Files in File System ---
Filename        | Size (bytes)
----------------|--------------
No files found.
----------------------------
Free blocks: 80 / 80
Free inodes: 16 / 16
File 'my_file.txt' created (inode 0).
Wrote 60 bytes to file 'my_file.txt'. Current size: 60
Wrote 55 bytes to file 'my_file.txt'. Current size: 115
File 'my_file.txt' closed.
File 'another.log' created (inode 1).
Wrote 25 bytes to file 'another.log'. Current size: 25
File 'another.log' closed.

--- Files in File System ---
Filename        | Size (bytes)
----------------|--------------
my_file.txt     | 115
another.log     | 25
----------------------------
Free blocks: 78 / 80
Free inodes: 14 / 16
File 'my_file.txt' opened for reading.
Read 115 bytes from file 'my_file.txt'.
Content of 'my_file.txt':
Hello, this is a test file content for our simple file system.Adding more data
to the file to exceed one block size.
File 'my_file.txt' closed.
```

```
Error: File 'non_existent.txt' not found for reading.
```

(Exact memory addresses or block numbers will vary based on execution.)

Lab 12: Understand the Basic Methods of Free Space

Title

Understanding and Simulating Free Space Management Methods

Aim

To understand different techniques for managing free space within a file system, such as bitmaps, linked lists, and grouping, and to simulate a basic free space allocation/deallocation process.

Procedure

1. **Conceptual Understanding:** Discuss the concepts of free space management, including:
   - **Bitmap:** Using an array of bits where each bit represents a block's status (free/used).
   - **Linked List:** Linking all free blocks together.
   - **Grouping:** Storing addresses of free blocks in the first free block.
2. **Simulation Program (Bitmap):**
   - Write a C program to simulate free space management using a bitmap.
   - Initialize a disk with a certain number of blocks, all marked as free.
   - Implement functions to:
     - `allocate_block()`: Find and allocate a free block, updating the bitmap.
     - `deallocate_block(block_num)`: Mark a block as free, updating the bitmap.
     - `print_bitmap()`: Display the current state of the free space bitmap.
   - Demonstrate allocation and deallocation of several blocks.

Source Code

```
// free_space_bitmap.c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define TOTAL_BLOCKS 32 // Total number of blocks in the simulated disk

// Bitmap: true means free, false means used
bool free_space_bitmap[TOTAL_BLOCKS];

void initialize_free_space() {
    for (int i = 0; i < TOTAL_BLOCKS; i++) {
        free_space_bitmap[i] = true; // All blocks are initially free
    }
    printf("Free space bitmap initialized. All %d blocks are free.\n",
TOTAL_BLOCKS);
}

// Function to allocate a free block
int allocate_block() {
    for (int i = 0; i < TOTAL_BLOCKS; i++) {
        if (free_space_bitmap[i]) {
            free_space_bitmap[i] = false; // Mark as used
            printf("Allocated block: %d\n", i);
            return i; // Return the allocated block number
        }
    }
    printf("Error: No free blocks available.\n");
```

```
    return -1; // No free block
}

// Function to deallocate a block
void deallocate_block(int block_num) {
    if (block_num >= 0 && block_num < TOTAL_BLOCKS) {
        if (!free_space_bitmap[block_num]) {
            free_space_bitmap[block_num] = true; // Mark as free
            printf("Deallocated block: %d\n", block_num);
        } else {
            printf("Warning: Block %d was already free.\n", block_num);
        }
    } else {
        printf("Error: Invalid block number %d for deallocation.\n", block_num);
    }
}

// Function to print the current state of the bitmap
void print_bitmap() {
    printf("\n--- Free Space Bitmap State ---\n");
    printf("Block | Status\n");
    printf("------|--------\n");
    for (int i = 0; i < TOTAL_BLOCKS; i++) {
        printf("%5d | %s\n", i, free_space_bitmap[i] ? "FREE" : "USED");
    }
    printf("------------------------------\n");
}

int main() {
    initialize_free_space();
    print_bitmap();

    int block1 = allocate_block();
    int block2 = allocate_block();
    int block3 = allocate_block();
    print_bitmap();

    deallocate_block(block2);
    print_bitmap();

    int block4 = allocate_block(); // Should reuse block2
    print_bitmap();

    deallocate_block(99); // Invalid block
    deallocate_block(block1);
    deallocate_block(block3);
    deallocate_block(block4);
    print_bitmap();

    return 0;
}
```

Input

N/A (Operations are hardcoded in `main`)

Expected Output

```
Free space bitmap initialized. All 32 blocks are free.

--- Free Space Bitmap State ---
Block | Status
------|--------
    0 | FREE
```

```
    1 | FREE
    ... (all 32 blocks)
   31 | FREE
-------------------------------
Allocated block: 0
Allocated block: 1
Allocated block: 2

--- Free Space Bitmap State ---
Block | Status
------|--------
    0 | USED
    1 | USED
    2 | USED
    3 | FREE
    ...
   31 | FREE
-------------------------------
Deallocated block: 1

--- Free Space Bitmap State ---
Block | Status
------|--------
    0 | USED
    1 | FREE
    2 | USED
    3 | FREE
    ...
   31 | FREE
-------------------------------
Allocated block: 1

--- Free Space Bitmap State ---
Block | Status
------|--------
    0 | USED
    1 | USED
    2 | USED
    3 | FREE
    ...
   31 | FREE
-------------------------------
Error: Invalid block number 99 for deallocation.
Deallocated block: 0
Deallocated block: 2
Deallocated block: 1

--- Free Space Bitmap State ---
Block | Status
------|--------
    0 | FREE
    1 | FREE
    2 | FREE
    3 | FREE
    ...
   31 | FREE
-------------------------------
```

Lab 13: Programs to Implement the Various CPU Scheduling Algorithms

Title

Implementation of CPU Scheduling Algorithms

Aim

To understand and implement various CPU scheduling algorithms, analyze their performance metrics (turnaround time, waiting time), and compare their effectiveness.

Procedure

1. **Conceptual Understanding:** Review the principles of CPU scheduling, including:
   - **FCFS (First-Come, First-Served):** Non-preemptive, simple.
   - **SJF (Shortest Job First):** Optimal (non-preemptive), minimizes average waiting time.
   - **Priority Scheduling:** Processes with higher priority execute first.
   - **Round Robin:** Preemptive, time-sliced, fair.
2. **Implementation for Each Algorithm:**
   - For each algorithm, write a C program that:
     - Defines a set of processes with `process_id`, `arrival_time`, `burst_time`, (and `priority` for Priority Scheduling).
     - Simulates the execution of processes according to the algorithm's rules.
     - Calculates and displays:
       - Gantt Chart (textual representation).
       - Completion Time (CT) for each process.
       - Turnaround Time (TAT = CT - Arrival Time) for each process.
       - Waiting Time (WT = TAT - Burst Time) for each process.
       - Average Turnaround Time.
       - Average Waiting Time.

Source Code

```
// cpu_scheduling_fcfs.c
#include <stdio.h>
#include <stdlib.h>

// Structure to represent a process
typedef struct {
    int pid;          // Process ID
    int arrival_time; // Arrival time
    int burst_time;   // CPU burst time
    int completion_time;
    int turnaround_time;
    int waiting_time;
} Process;

// Function to sort processes by arrival time (for FCFS)
void sort_by_arrival_time(Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].arrival_time > p[j + 1].arrival_time) {
```

```c
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

void calculate_times(Process p[], int n) {
    int current_time = 0;
    float total_turnaround_time = 0;
    float total_waiting_time = 0;

    printf("\n--- FCFS Scheduling ---\n");
    printf("Gantt Chart:\n");

    for (int i = 0; i < n; i++) {
        // If process arrives after current_time, CPU is idle
        if (current_time < p[i].arrival_time) {
            printf("| Idle (%d-%d) ", current_time, p[i].arrival_time);
            current_time = p[i].arrival_time;
        }

        printf("| P%d (%d-%d) ", p[i].pid, current_time, current_time +
p[i].burst_time);
        p[i].completion_time = current_time + p[i].burst_time;
        p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
        p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;

        total_turnaround_time += p[i].turnaround_time;
        total_waiting_time += p[i].waiting_time;

        current_time = p[i].completion_time;
    }
    printf("|\n");

    printf("\nProcess Details:\n");
    printf("PID | Arrival | Burst | CT  | TAT | WT\n");
    printf("----|---------|-------|-----|-----|----\n");
    for (int i = 0; i < n; i++) {
        printf("%3d | %7d | %5d | %3d | %3d | %3d\n",
               p[i].pid, p[i].arrival_time, p[i].burst_time,
               p[i].completion_time, p[i].turnaround_time, p[i].waiting_time);
    }

    printf("\nAverage Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

int main() {
    // Example processes
    Process processes[] = {
        {1, 0, 5},
        {2, 1, 3},
        {3, 2, 8},
        {4, 3, 6}
    };
    int n = sizeof(processes) / sizeof(processes[0]);

    // Sort processes by arrival time for FCFS
    sort_by_arrival_time(processes, n);

    calculate_times(processes, n);

    return 0;
}
```

(Note: Implementations for SJF, Priority, and Round Robin would follow a similar structure, but with different sorting/selection logic. Due to space, only FCFS is provided as an example.)

Input

N/A (Process details are hardcoded in `main`)

Expected Output

```
--- FCFS Scheduling ---
Gantt Chart:
| P1 (0-5) | P2 (5-8) | P3 (8-16) | P4 (16-22) |

Process Details:
PID | Arrival | Burst | CT  | TAT | WT
----|---------|-------|-----|-----|----
  1 |       0 |     5 |   5 |   5 |   0
  2 |       1 |     3 |   8 |   7 |   4
  3 |       2 |     8 |  16 |  14 |   6
  4 |       3 |     6 |  22 |  19 |  13

Average Turnaround Time: 11.25
Average Waiting Time: 5.75
```

(Output will vary depending on the processes defined and the specific algorithm implemented.)