# All Assignments-ClassTests-LabTests Together

**Shakil Hossan**
**Roll: 2011176131**
**11 November 2024**

# Assignment-12.02.2024

## 1 What is PID? Why is it necessary?

A PID(Process Identifier) is a unique number given to each process running in an operating system. Here's why it is important:

**1.Process Management:** The OS uses PIDs to keep track of each running process. Without PIDs, it would be hard to organize and control processes.

**2.Resource Allocation:** The OS assigns resources like memory and processing power based on PIDs, so each process gets what it needs.

**3.Process Communication:** Processes can talk to each other using PIDs. For example, one process can send a message to another by using its PID.

**4.Process Termination:** PIDs let the OS stop specific processes when needed, freeing up resources for other tasks.

**5.Error Handling:** If something goes wrong, the OS can use the PID to find and fix the problematic process.

## 2 Is PID necessary only in Ubuntu? In Windows or other operating system, is there any similar concept?

No, PIDs are not exclusive to Ubuntu.PIDs are a common concept across many operating systems. The concept of Process Identifiers is used in many operating systems, including Windows, macOS, Linux, and others. Here is how it works in different systems:

**1.Windows:** Like Linux, Windows assigns a PID to each process. PIDs are used for managing processes, allocating resources, handling communication between processes, and debugging. You can see the list of processes and their PIDs using Task Manager or PowerShell.

**2.macOS:** Being Unix-based, macOS uses PIDs in a similar way to Linux for process management.

**3.Other OS:** Other systems like Unix (FreeBSD, OpenBSD) and even embedded or real-time OSs also use PIDs to manage processes, though the details may vary.

## 3 Is there any process having PID = 0? If yes, write down the tasks or responsibilities of that process.

Yes, there is a process with PID = 0 in most operating systems, and its responsibilities vary slightly depending on the system:
**In Unix-like systems (Linux, macOS):**
PID 0 is usually the **"idle" or "swapper" process**. It manages the scheduling of other processes and allocates CPU time based on priorities.It runs when no other processes need CPU time, keeping the system ready for tasks when they arise.

**In Windows:**
PID 0 is the **"System Idle Process"**. Its main role is to run when no other processes are active, representing unused CPU time. It keeps the CPU from entering a low-power state unnecessarily, but it doesn't consume significant resources.

## 4 Write down the task the tasks or responsibilities of that process having PID 1.

The process with PID 1 called **"init"** process, it has several key tasks:
**System Startup:** It starts the system and sets up services when the computer boots up.
**Manage Other Processes:** It starts and supervises other important processes, making sure they are running.
**Shutdown and Restart:** It handles shutting down or restarting the system properly.
**Handle Orphaned Processes:** If a process loses its parent, init takes care of it and prevents it from causing problems.

# Assignment-13.02.2024

## 1 Based on Silberschatz's and Tanenbaum's Operating System Books, point out the similarities of scheduling algorithms with the approaches discussed at 13.02.2024 classroom.

**Active Process First:** Similar to **preemptive scheduling**, where active processes are prioritized over inactive ones.

**First Come First Serve (FCFS):** Matches **non-preemptive scheduling**, where processes run in the order they arrive without interruption.

**Priority and Urgency:** This is like **priority scheduling**, where processes are scheduled based on their importance.

**Time Allocation for Processes:** Similar to **round-robin scheduling**, where processes get a set amount of CPU time before waiting their turn.

**Service Based on Resource Needs:** Similar to **Shortest Job First (SJF)**, where processes with fewer resource needs are prioritized.

**Group Processes by Requirements:** Similar to **priority-based scheduling**, where processes with similar needs are grouped and prioritized.

## 2 Write down positive and negative points of each scheduling algorithm.

**First Come First Serve (FCFS)**
**Positive:** Simple and fair, processes run in the order they arrive.
**Negative:** Can cause delays for short tasks if long tasks arrive first.

**Shortest Job First (SJF)**
**Positive:** Minimizes waiting time for short tasks.
**Negative:** Can starve long tasks; requires knowing process durations in advance.

**Priority Scheduling**
**Positive:** Prioritizes important tasks.
**Negative:** Low-priority tasks may be starved and delayed.

**Round Robin (RR)**
**Positive:** Fairly allocates CPU time to all processes.
**Negative:** High overhead from frequent switching; inefficient for CPU-heavy tasks.

## 3 Is their any scheduling algorithm which was not discussed at today's class? If yes, write down short notes on that algorithm.

One scheduling algorithm not discussed in class is **Multilevel Feedback Queue Scheduling**.

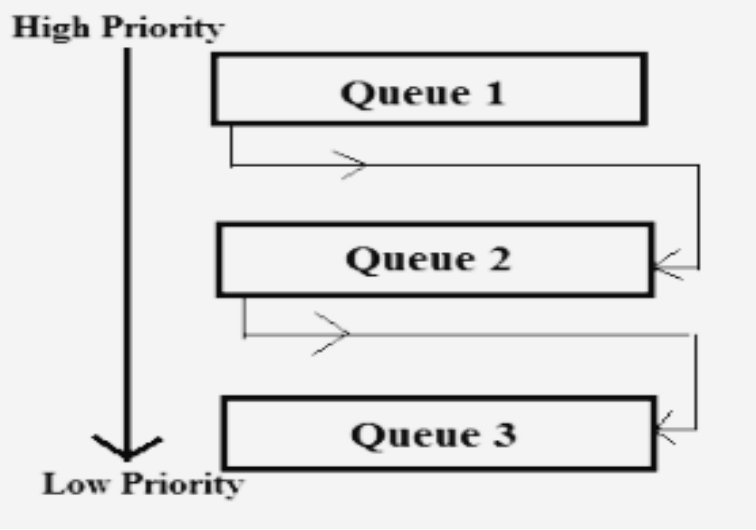



Figure 1: Multilevel-Feedback Queue Scheduling

It uses multiple queues with different priorities. Processes move between queues based on their CPU usage and needs.

**Advantages:**
- Works well for different types of processes.
- Prevents starvation by allowing processes to move to higher queues if needed.

**Disadvantages:**
- Can be complex to manage and set up.
- Requires careful tuning of parameters.

# Final Assignment (Individual Submission)

## 1 Process

### a What do you know about:

**i. Process:** A process is a program in execution. It includes the program code and resources like memory and CPU time.

**ii.Process Control Block (PCB):** The PCB stores important information about a process, such as its PID, state, CPU registers, and memory details.

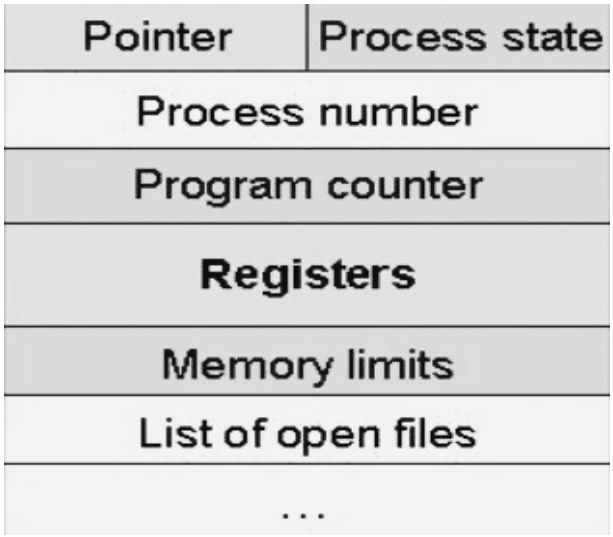



Figure 2: Process Control Block(PCB)

**iii. PCB Table:** A table that holds a PCB for every process in the system, allowing the operating system to manage them.

**iv. Process ID (PID):** A unique number given to each process to identify it.

**v. Process Tree:** A structure showing the relationship between processes. The init process (PID 1) is the root, and other processes branch off from it.

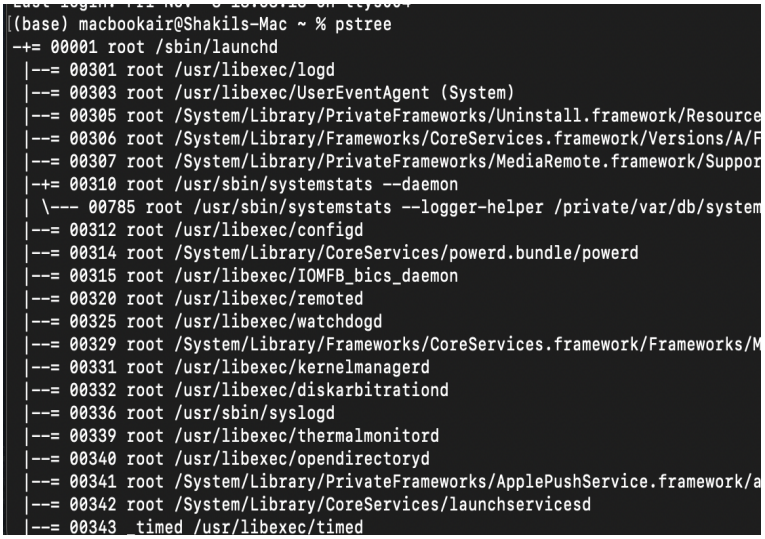



Figure 3: Process Tree in Macbook Air M2

**vi. PID 0 and PID 1:**
**PID 0:** Represents the idle process, running when no other processes are using the CPU.
**PID 1:** Represents the init process, which starts other system processes during boot.

## b. Write down the differences between the following terms:

### i. Program and Process

| Program | Process |
|---|---|
| A program is a set of instructions stored on disk. | A process is a program that is currently running in memory. |
| Static (not running). | Dynamic (actively running and using resources). |
| No memory is used when the program is not running. | Uses memory for execution. |
| No execution, just stored code. | Actively executing the program code. |
| Doesn't use CPU or resources until executed. | Uses CPU, memory, and other system resources. |
| Created by the developer (stored on disk). | Created by the OS when the program is executed. |

Table 1: Difference between Program and Process

**ii. Process and Light-Weight Process**

| Process | Light-Weight Process |
|---|---|
| A process is a full execution of a program. | An LWP is a smaller unit (thread) within a process. |
| Has its own memory space. | Shares memory space with the parent process. |
| Uses system resources like CPU, memory, etc. | Uses fewer resources than a full process. |
| Runs independently with its own execution flow. | Executes as part of a larger process, sharing resources. |
| Created by the OS when a program is run. | Created by a process to manage tasks or threads. |
| May have more overhead in context switching. | Has less overhead due to shared resources. |

Table 2: Difference between Process and Light-Weight Process (LWP)

### c. Can you kill processes having PID 0 and PID 1? If yes, write down the steps of killing processes having PID 0 and PID 1.

**PID 0**: This is the **Idle process** in the kernel. It's crucial for the system and cannot be killed.
**PID 1**: This is the **init process**, which starts and manages other system processes. You *can* technically kill it, but it's dangerous and can cause system issues. **How to Kill PID 1: Using kill:**

```
sudo kill 1
```

This may not work because PID 1 is essential. textbfUsing pkill (forcefully):

```
sudo pkill -9 1
```

This will forcefully stop the process, but it's risky and may crash the system.

## 2. Write down the differences between the following terms:

### a. Preemptive and Non-Preemptive Scheduling Algorithms

| Preemptive Scheduling | Non-Preemptive Scheduling |
|---|---|
| The OS can interrupt and switch processes anytime. | Process runs until it finishes or yields. |
| More responsive to high-priority processes. | Less responsive, as tasks wait their turn. |
| Used in real-time and multi-user systems. | Used in simpler, single-user systems. |

Table 3: Difference between Preemptive and Non-Preemptive Scheduling Algorithms

### b. User mode and kernel mode

| User Mode | Kernel Mode |
|---|---|
| Restricted access to system resources. | Full access to system resources. |
| Runs application code. | Handles core OS functions. |
| Safer, less risk to OS. | Can modify OS components directly. |
| Example: Running apps. | Example: Accessing device drivers, system calls. |

Table 4: Difference between User Mode and Kernel Mode

### c. Named Pipe and Unnamed Pipe

| Named Pipe | Unnamed Pipe |
|---|---|
| Has a specific name, can be accessed by name. | Exists only while connected processes run. |
| Allows communication between unrelated processes. | Limited to related processes. |
| Often used for inter-process communication. | Temporary data transfer between processes. |

Table 5: Difference between Named Pipe and Unnamed Pipe

### d. Conventional File and Named Pipe

| Conventional File | Named Pipe |
|---|---|
| Stores data persistently on disk. | Temporary, doesn't store data persistently. |
| Random or sequential access. | Sequential access only. |
| Holds data and information. | Used for communication between processes. |

Table 6: Difference between Conventional File and Named Pipe

### e. Logical Core and Physical Core

| Logical Core | Physical Core |
|---|---|
| Virtual core created by hyperthreading. | Actual, physical CPU core. |
| Shares resources with other logical cores. | Owns dedicated processing resources. |
| Improves multitasking. | Higher performance for individual tasks. |

Table 7: Difference between Logical Core and Physical Core

### f. Default Signal Handler and User-Defined Signal Handler

| Default Signal Handler | User-Defined Signal Handler |
|---|---|
| Automatically handles signals in a set way. | Allows custom behavior in response to signals. |
| Limited control over signal response. | Customizable based on app needs. |
| Example: Terminates process on interrupt. | Example: Saves data before exiting on interrupt. |

Table 8: Difference between Default Signal Handler and User-Defined Signal Handler

## 3. Write down the positive and negative sides of the following scheduling algorithms

### a. First Come, First-Served

| Positive Sides | Negative Sides |
|---|---|
| Easy to use and fair in order. | Long tasks can slow down short ones. |
| Works well in simple systems. | Can lead to high wait times. |

Table 9: First Come, First-Served (FCFS)

### b. Shortest Job First

| Positive Sides | Negative Sides |
|---|---|
| Reduces average wait time effectively. | Needs to know task duration, which isn't always known. |
| Great for quick, short tasks. | Can cause long tasks to be delayed. |

Table 10: Shortest Job First (SJF)

### c. Round Robin

| Positive Sides | Negative Sides |
|---|---|
| Fair time for each task. | Frequent switching can slow performance. |
| Good for sharing CPU time fairly. | Short tasks may wait longer than needed. |

Table 11: Round Robin (RR)

## Which scheduling algorithms could result in starvation? How can the OS solve it?

- **Priority Scheduling:**
  - **Problem:** Low-priority processes can be blocked forever if high-priority processes keep arriving.
- **Shortest Job First (SJF)** and **Shortest Remaining Time Next (SRTN):**

– **Problem:** Long processes may never get executed if shorter ones keep arriving.

**Solution:**

- **Aging:** A technique where the priority of long-waiting processes is gradually increased over time. For example, a low-priority process slowly gains priority until it can eventually execute.This ensures that no process is stuck waiting forever.

## 5. Write a C program having initialized and uninitialized local and global variables and describe the memory layout of its process.

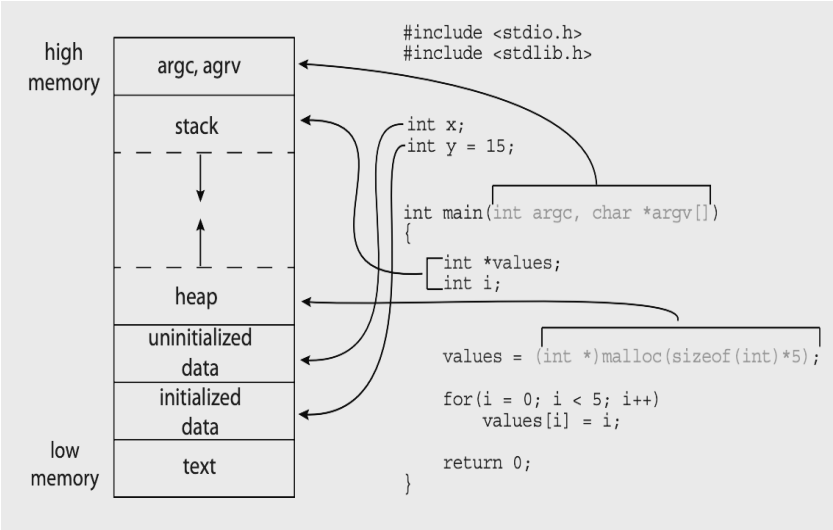



Figure 4: Memory Layout of a Process

## 6. When instructions after execlp() are executed? Why?

When `execlp()` is called, it replaces the current process with a new one. The instructions after `execlp()` are **not executed** if `execlp()` is successful, because the process is replaced.
They will only be executed if `execlp()` fails, like when the program can't be found or run.

## 7. Process Creation

### a. Consider the following code segment:

```
pid_t pid;
pid = fork();
if (pid == 0) {
    fork();
    pthread_create(&tid, NULL, (void*)thread_handler, NULL);
}
fork();
```

**i. How many unique processes are created?**
**ii. How many unique threads are created?**

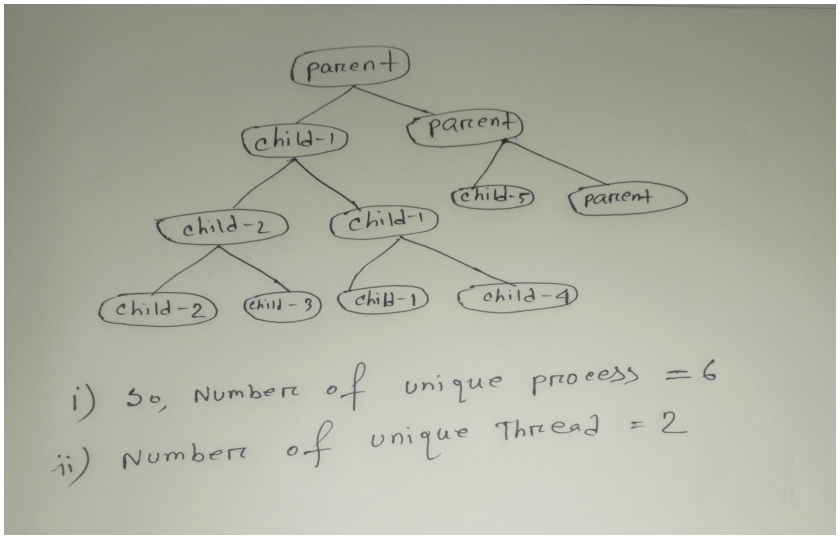



Figure 5: Unique Process and Thread

## b. What will be the output for the following code segment?

```
fork();
printf('Bangladesh');
fork();
printf('Bangladesh');
fork();
```

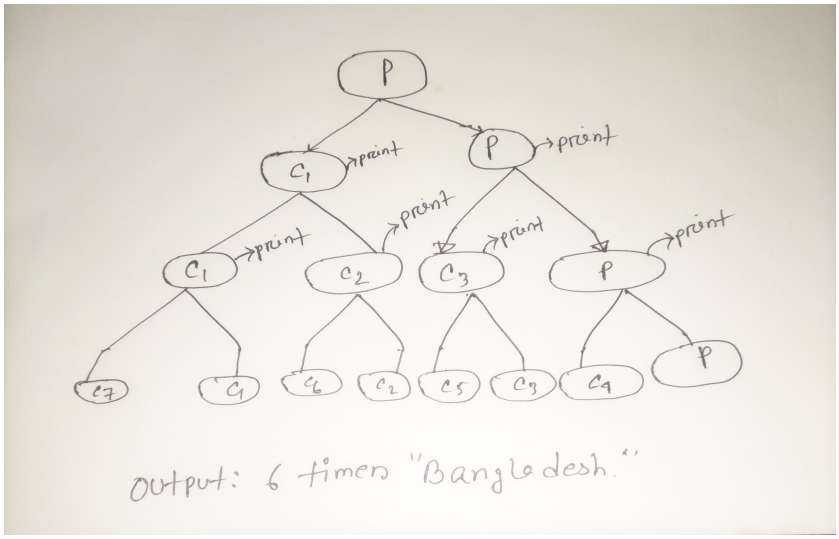



Figure 6: Output of the given code

**c. How many child processes will be created if the following loop is in a program code? Give a proper explanation for your answer.**

```
for (i = 0; i < n; i++)
    fork();
Draw the process tree when n = 4.
```

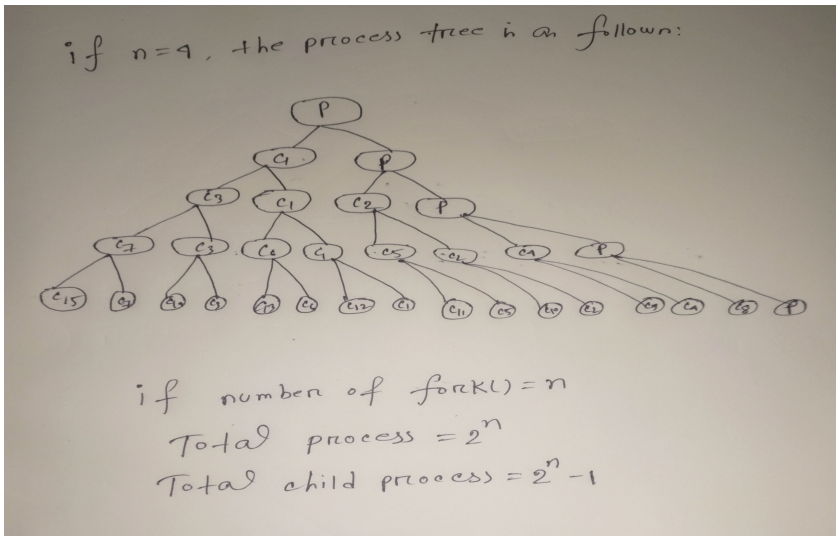



Figure 7: Answer of the given question

# 8. What do you know about

## Orphan and Zombie Processes in Linux

### Definitions

| | |
|---|---|
| **Orphan Process** | A child process whose parent has ended, leaving it to be adopted by the `init` process. |
| **Zombie Process** | A process that has finished executing but still appears in the process table because its parent hasn't read its exit status. |

Table 12: Definitions of Orphan and Zombie Processes

## a. When can you call a process an orphan process or a zombie process?

- **Orphan Process:** A process is called an orphan process when its parent process has terminated. The `init` process (PID 1) then adopts the orphan process.
- **Zombie Process:**A zombie process is a process that has completed its execution but still remains in the process table because its parent has not read its exit status.

## b. Creating Orphan and Zombie Processes

### Orphan Process Code:

```
#include <stdlib.h>
#include <unistd.h>

int main() {
    if (fork() > 0) {
        exit(0); // Parent exits, child becomes orphan
    } else {
        sleep(10); // Child continues running
    }
    return 0;
}
```

**Zombie Process Code:**

```
#include <stdlib.h>
#include <unistd.h>

int main() {
    if (fork() == 0) {
        exit(0); // Child exits, becomes zombie
    } else {
        sleep(10); // Parent doesn't wait(), so child remains zombie
    }
    return 0;
}
```

## c. Tracing Orphan and Zombie Processes

- **Orphan Process:** Use `ps -e -o pid,ppid,stat,cmd` to find processes with `PPID` of 1.
- **Zombie Process:** Use `ps aux | grep Z` to list processes with status `Z`.

## d. OS Treatment of Orphan and Zombie Processes

| | |
|---|---|
| **Orphan Process** | Adopted and managed by the `init` process. |
| **Zombie Process** | Removed when the parent process reads its exit status. |

Table 13: OS Treatment of Orphan and Zombie Processes

## e. Advantages and Disadvantages

| Process Type | Advantages | Disadvantages |
|---|---|---|
| Orphan | Child can continue independently | Can accumulate if mismanaged, using resources |
| Zombie | Minimal resource usage | Excessive zombies can clutter process table, affecting performance |

Table 14: Advantages and Disadvantages of Orphan and Zombie Processes

# ClassTest-23-05-2024

## 1

### a. Can you kill processes having PID 0 and PID 1? If yes, write down the steps of killing processes having PID 0 and PID 1.

**PID 0**: This is the **Idle process** in the kernel. It's crucial for the system and cannot be killed.
**PID 1**: This is the **init process**, which starts and manages other system processes. You *can* technically kill it, but it's dangerous and can cause system issues. **How to Kill PID 1: Using kill:**

```
sudo kill 1
```

This may not work because PID 1 is essential. textbfUsing pkill (forcefully):

```
sudo pkill -9 1
```

This will forcefully stop the process, but it's risky and may crash the system.

### Consider the following code segment:

```
pid_t pid;
pid = fork();
if (pid == 0) {
    fork();
    pthread_create(&tid, NULL, (void*)thread_handler, NULL);
}
fork();
```

**i. How many unique processes are created?**
**ii. How many unique threads are created?**

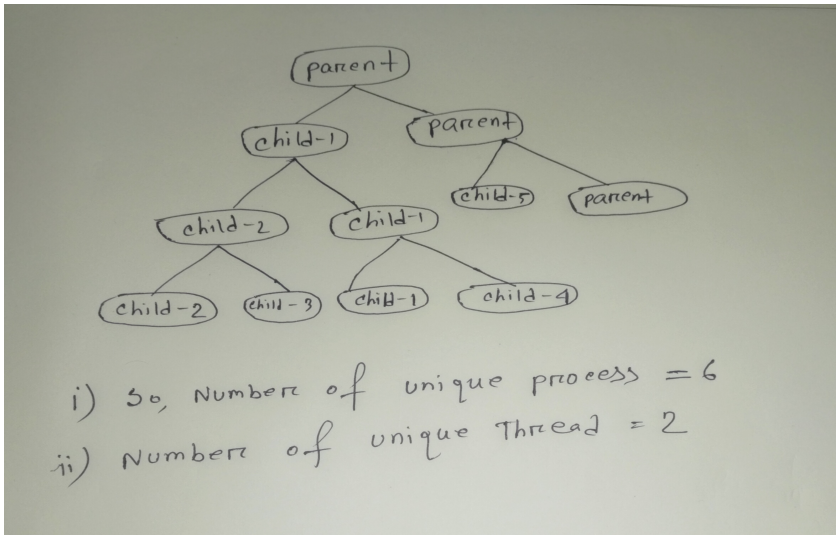



Figure 8: Unique Process and Thread

**b. What will be the output for the following code segment?**

```
fork();
printf('Bangladesh');
fork();
printf('Bangladesh');
fork();
```

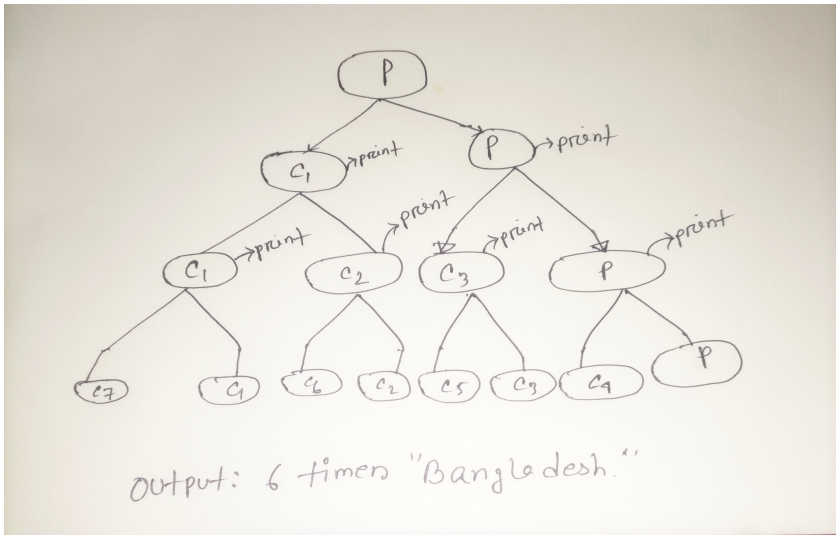



Figure 9: Output of the given code

**c. How many child processes will be created if the following loop is in a program code? Give a proper explanation for your answer.**

```
for (i = 0; i < n; i++)
    fork();
Draw the process tree when n = 4.
```

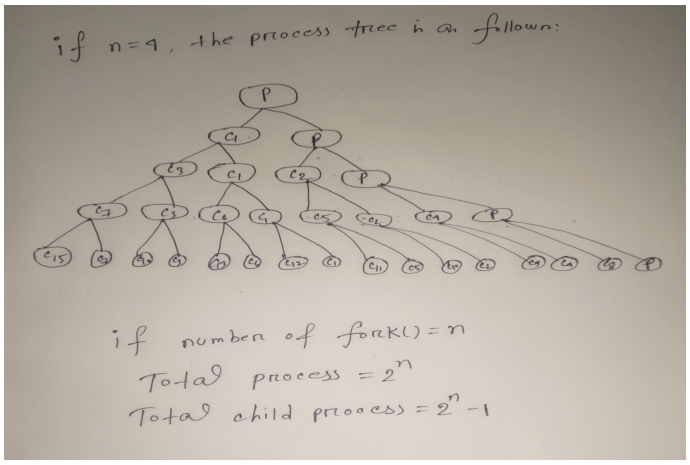



Figure 10: Answer of the given question

# 2

# a. What do you know about

## Orphan and Zombie Processes in Linux

### Definitions

| | |
|---|---|
| **Orphan Process** | A child process whose parent has ended, leaving it to be adopted by the `init` process. |
| **Zombie Process** | A process that has finished executing but still appears in the process table because its parent hasn't read its exit status. |

Table 15: Definitions of Orphan and Zombie Processes

## b. When can you call a process an orphan process or a zombie process?

- **Orphan Process:** A process is called an orphan process when its parent process has terminated. The `init` process (PID 1) then adopts the orphan process.
- **Zombie Process:**A zombie process is a process that has completed its execution but still remains in the process table because its parent has not read its exit status.

## c. Creating Orphan and Zombie Processes

**Orphan Process Code:**

```
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid;
    pid = fork();
    if (fork() > 0) {
        exit(0); // Parent exits, child becomes orphan
    } else {
        sleep(10); // Child continues running
    }
    return 0;
}
```

**Zombie Process Code:**

```
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid;
    pid = fork();
    if (fork() == 0) {
        exit(0); // Child exits, becomes zombie
    } else {
        sleep(10); // Parent doesn't wait(), so child remains zombie
    }
```

```
    return 0;
}
```

### d. Tracing Orphan and Zombie Processes

- **Orphan Process:** Use `ps -e -o pid,ppid,stat,cmd` to find processes with `PPID` of 1.
- **Zombie Process:** Use `ps aux | grep Z` to list processes with status `Z`.

### e. OS Treatment of Orphan and Zombie Processes

| | |
|---|---|
| **Orphan Process** | Adopted and managed by the `init` process. |
| **Zombie Process** | Removed when the parent process reads its exit status. |

Table 16: OS Treatment of Orphan and Zombie Processes

### f. Advantages and Disadvantages

| **Process Type** | **Advantages** | **Disadvantages** |
|---|---|---|
| Orphan | Child can continue independently | Can accumulate if mismanaged, using resources |
| Zombie | Minimal resource usage | Excessive zombies can clutter process table, affecting performance |

Table 17: Advantages and Disadvantages of Orphan and Zombie Processes

## 3. Do we need an operating system for all cases? Justify your answer.

- **A. Single User, Single Task, Limited Hardware:**
  An OS isn't strictly needed, but it can help manage resources and make things easier.
- **B. Single User, Multiple Tasks Simultaneously, Limited Hardware:**
  An OS is essential to handle multitasking, as it manages resources and lets tasks share CPU time.
- **C. Single User, Multiple Tasks Consecutively, Limited Hardware:**
  An OS isn't required but helps run tasks smoothly, making task management easier.
- **D. Single User, Single Task, Single Hardware Device:**
  No OS needed if the device is built for just that one task.
- **E. Multiple Users, Multiple Tasks Simultaneously, Limited Hardware:**
  An OS is a must to share resources fairly and avoid conflicts between users.

## 4

### a. How to increase the degree of multiprogramming?

Adding more memory and improving CPU scheduling allows the operating system to handle more programs at once, increasing multiprogramming.

**b. Explain the journey of a source code (say Addition.c) to turn into a process by the help of a figure.**

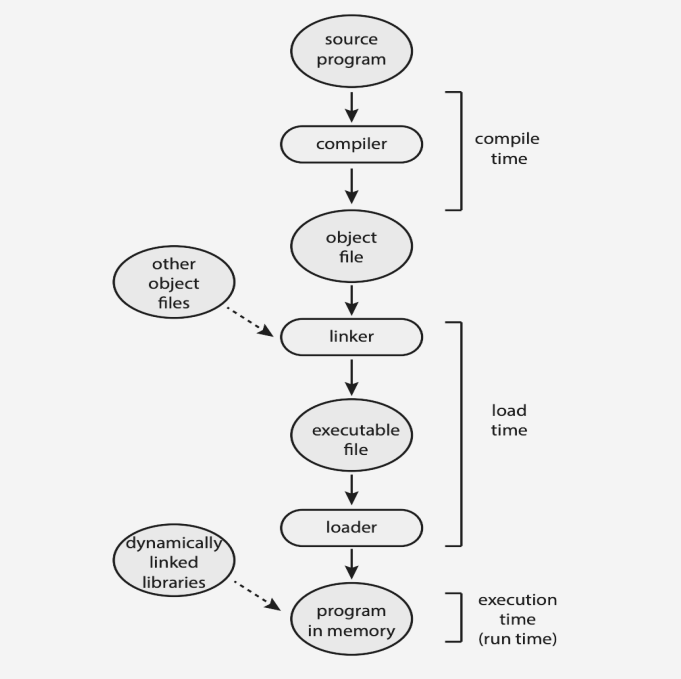



Figure 11: Multistep processing of a user program.

**d. Draw the memory layout for the following C program.**

```
#include <stdio.h>
#include <stdlib.h>
int x, y = 10;
void f2(){
int x, y = 10;
int *ptr;
ptr = (int ) malloc(sizeof(int) * 5);
}
void f1(){
int x, y = 10;
int *ptr;
ptr = (int ) malloc(sizeof(int) * 5);
f2();
}
int main(int argc, char
*
argv[]){

int x, y = 10;
int *ptr;
ptr = (int ) malloc(sizeof(int) * 5);
return 0;
}
```

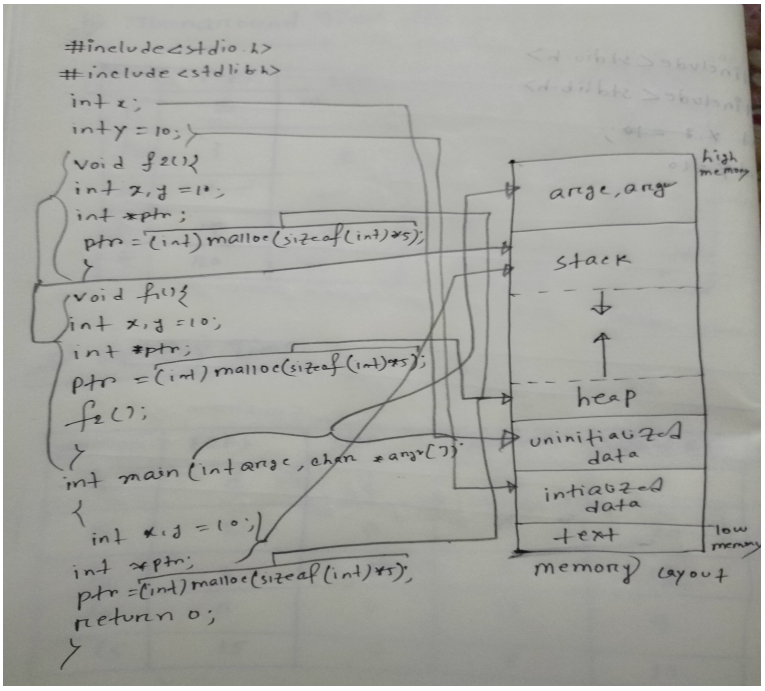



Figure 12: Memory layout of the given program

# 5

## a. How many processes can be in a running, waiting, and ready state at a time?

- **Running State:** Only one process per CPU core can be in the running state at a time.
- **Waiting State:** Multiple processes can be in the waiting state, as they are paused and waiting for resources.
- **Ready State:** Many processes can be in the ready state, waiting their turn to use the CPU.

## b. Who can access the Process Control Block (PCB) of a process? Justify your answer.

Only the **Operating System (OS)** has access to a process's PCB because it contains essential data for managing and tracking the process safely and securely.

## c. Where are PCBs stored? Justify your answer.

PCBs are stored in **OS-managed memory**, often in a table or structure within system memory, so the OS can access process information efficiently.

## d. Is it possible to handle multiple processes without PCBs? Justify your answer.

No, PCBs are crucial for handling multiple processes. They store necessary data about each process's state and resources, which is vital for organized process management and switching.

# 6

## a. Write down the difference between:

### i. unnamed pipe and named pipe

| Unnamed Pipe | Named Pipe |
|---|---|
| Used for communication between parent-child processes | Used for communication between unrelated processes |
| Temporary, not saved in the file system | Persistent, exists as a file in the file system |
| No name, anonymous | Has a name, can be referred to by any process |

Table 18: Difference between Unnamed Pipe and Named Pipe

### ii. named pipe and regular file

| Named Pipe | Regular File |
|---|---|
| Used for process-to-process communication | Used for data storage for permanent use |
| Works as a stream, doesn't store data permanently | Stores data persistently |
| Only one process can write, and another can read | Data can be read or written randomly |

Table 19: Difference between Named Pipe and Regular File

### iii. program and process

| Program | Process |
|---|---|
| A set of instructions stored on disk | A running instance of a program |
| Does not use system resources until executed | Uses CPU, memory, and other resources while running |
| Static, not executing | Dynamic, actively running |

Table 20: Difference between Program and Process

### iv. user mode and kernel mode

| User Mode | Kernel Mode |
|---|---|
| Programs run with limited access to resources | OS runs with full access to system resources |
| Restricted, no direct access to hardware or core functions | Full access to system and hardware resources |
| Limited, for user-level applications | High-level security, for OS and system tasks |

Table 21: Difference between User Mode and Kernel Mode

## b. What happens when a child process updates its local and global variables?

- **Local Variables:**
  - The child process has its own copy of local variables.
  - Changes to local variables in the child do not affect the parent process.
- **Global Variables:**
  - Both the parent and child processes start with the same global variables.
  - When the child updates a global variable, it gets its own copy (due to copy-on-write), so the parent's global variables remain unaffected.

### c. Explain how many ways two processes can work on a shared data when two processes:

**i. Child-Parent Relationship:**

- The child process inherits resources from the parent.
- Shared data can be exchanged through:
  - **Shared Memory:** Both processes can access the same memory.
  - **Pipes:** Data flows from parent to child.
  - **Message Queues:** Parent and child exchange messages.

**ii. No Child-Parent Relationship:**

- No direct link between processes.
- Shared data can be exchanged through:
  - **Shared Memory:** Common memory for both processes.
  - **Pipes/Named Pipes:** One process writes, the other reads.
  - **Message Queues:** Sending and receiving messages.
  - **Files:** Both processes can read from and write to the same file.
  - **Sockets:** Communication over a network or locally.

# Class Test, 26-05-2024

## 1

### a. How can a process send a signal to

1. **Another Process**
   - **How:** Use `kill(pid, signal);`
   - **What It Does:** Sends a signal to another process by specifying its `pid` (process ID). This lets one process communicate with or control another.
2. **A Specific Thread of the Same Process**
   - **How:** Use `pthread_kill(thread_id, signal);`
   - **What It Does:** Sends a signal to a specific thread within the same process using its `thread_id`.
3. **A Specific Thread in a Different Process**
   - **How:** Generally, this isn't directly possible.
   - **Alternative:** Use communication methods like sockets or message queues to pass information between threads in different processes.

# 2

## a. Write down the differences between:

### A. User Thread vs. Kernel Thread

| User Thread | Kernel Thread |
|---|---|
| Controlled by libraries, not the OS. | Controlled by the OS. |
| Faster to create and switch. | Slower to create and switch. |
| Can't use multiple CPUs at the same time. | Can use multiple CPUs for true multitasking. |
| If one thread is blocked, others are too. | Other threads can keep running if one is blocked. |

Table 22: User Thread vs. Kernel Thread

### B. Child Process vs. Thread

| Child Process | Thread |
|---|---|
| Runs separately with its own memory. | Runs within the same memory as others. |
| Created using `fork()` call. | Created with thread functions like `pthread`. |
| Needs special methods to communicate. | Can easily share data with other threads. |
| Slower due to separate memory. | Faster because of shared memory. |

Table 23: Child Process vs. Thread

# 4

## a. Why Separate Registers, Stack, and Program Counter for Each Thread?

- **Registers**: Each thread requires its own set of registers to maintain its execution state independently of other threads.
- **Stack**: Each thread has its own stack for handling function calls and local variables, avoiding interference between threads.
- **Program Counter**: Each thread has its own program counter to keep track of its position in execution separately.

## b. Do multiple child processes face any problems if they try to update:

- **Local Variables**: No issues arise, as each child process has its own copy, allowing changes to be independent.
- **Global Variables**: Issues can occur if shared, as one process might overwrite another's changes, leading to inconsistencies.

### c. Do multiple threads face any problems if they try to update:

- **Local Variables**: No issues arise since each thread has its own stack for local variables.
- **Global Variables**: Problems like race conditions can occur if threads try to change shared data at the same time. Synchronization methods, such as locks, help manage this.

## 5

### a. What happens when the main thread terminates before the termination of:

**Child Processes:** The child processes continue running even after the main thread finishes.

**(Subordinate) Threads:** Threads may continue to run after the main thread ends, depending on the program setup.

### b. Five Examples When Multithreading is Beneficial

**1. Faster Performance:**
Multithreading allows multiple tasks to run at the same time, speeding up processing.

**2. Efficient Resource Use:**
Threads share resources, leading to better memory and resource management.

**3. Better User Experience:**
One thread handles user input while others work on background tasks, making the app more responsive.

**4. Faster Computation:**
Large tasks can be split into smaller threads, which process faster.

**5. Handling I/O Efficiently:**
While one thread handles input/output, other threads can continue processing, preventing delays.

# Class Test, 30-05-2024

## 1

### a. Figure out how often context switches occur in your system and describe it with a screen shot.

**Context switch** is defined as the process of saving the state of the currently running process so that it can be restored and resumed later, and loading the saved state of the next process to be run.
context switching occurs frequently on general-purpose systems when interrupts cause the operating system to switch a CPU core from its current task to run a kernel routine. The system must save the current context (including CPU registers, process state, and memory-management information) in the Process Control Block (PCB)

of the current process before switching. This task involves performing a state save of the current process and a state restore of a different process, as illustrated in Figure 3.6. Context-switch time is considered overhead since no useful work is done during the switch, and the speed of context switching varies based on factors like memory speed, the number of registers, and hardware support. Advanced memory-management techniques can increase the complexity and time required for context switching.
The context switching from process to process is illustrated in the Figure 13.

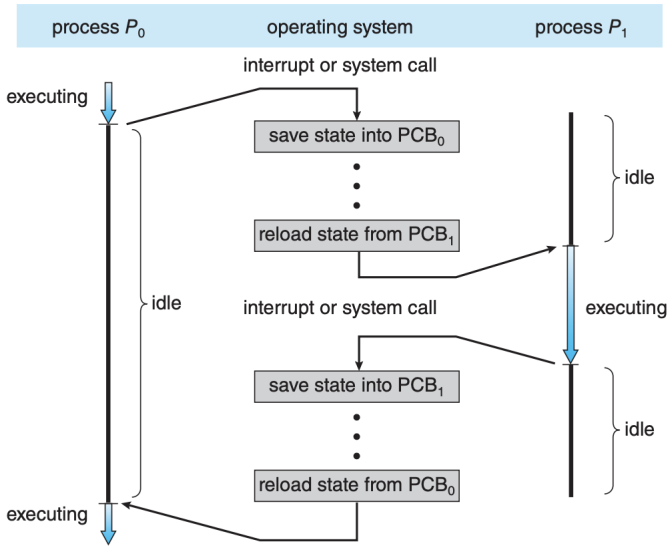



Figure 13: Diagram showing context switch from process to process.

## b. Figure out the number of context switches occurred for a specific process in your system. Describe investigation steps in detail.

After running the command **pidstat 739** i saw the number of context switch occured on my system.

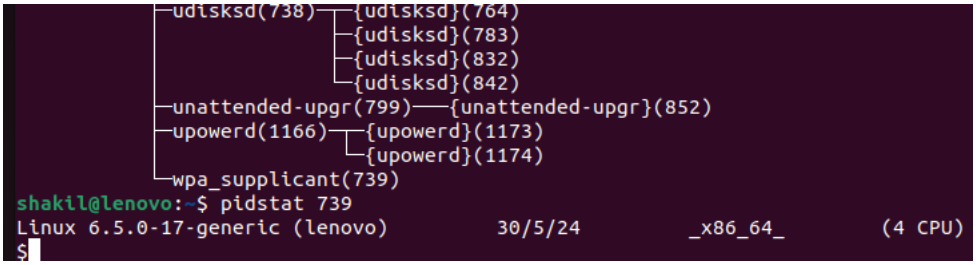



Figure 14: Number of Context switching occured on my system.

## c. What are the catchy things about:
## A. processor affinity
## B. dispatcher
## C. starvation of a process

### A. processor affinity

Processor affinity, also known as CPU pinning, refers to binding a process or thread to a specific CPU core or a set of cores to optimize performance.

### B. dispatcher

The dispatcher is a component of the operating system responsible for the actual process of switching the CPU from one process to another.

### C. starvation of a process

Starvation occurs when a process is continuously denied necessary resources, leading to an indefinite delay in its execution.

# 2

## a. Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| $P1$ | 2 | 2 |
| $P2$ | 1 | 1 |
| $P3$ | 8 | 4 |
| $P4$ | 4 | 2 |
| $P5$ | 5 | 3 |

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

**A. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms:**

- FCFS (First-Come, First-Served)
- SJF (Shortest Job First)
- Non-preemptive Priority (A larger priority number implies a higher priority)
- RR (Round Robin, quantum = 2)

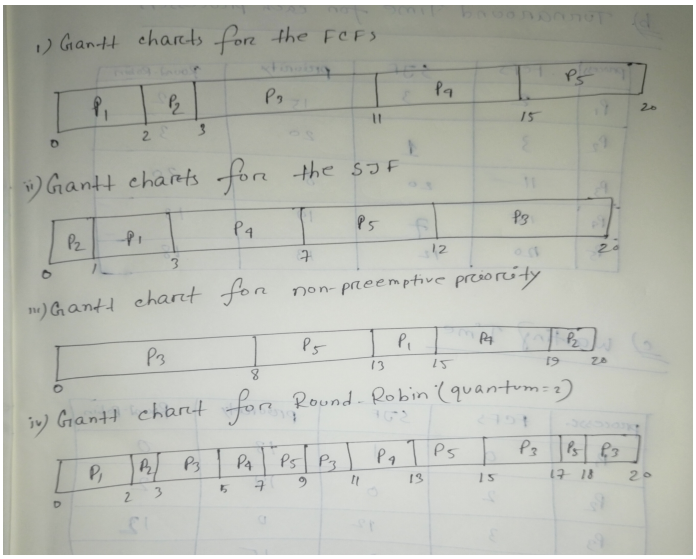
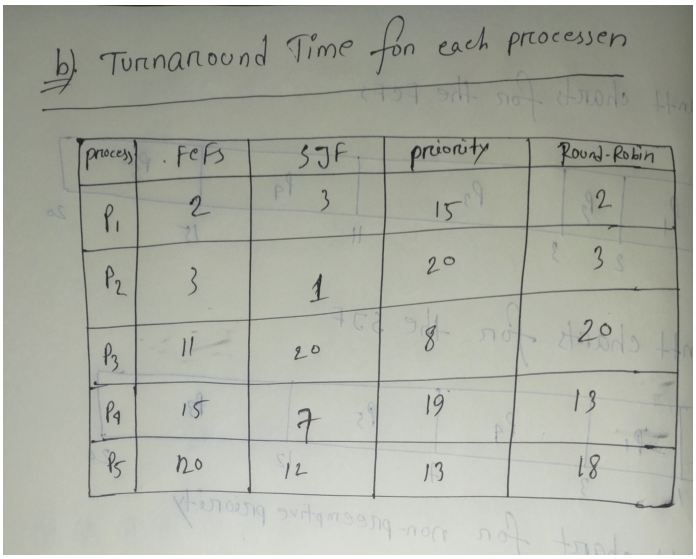



Figure 15: Gantt chart for all scheduling algorithms

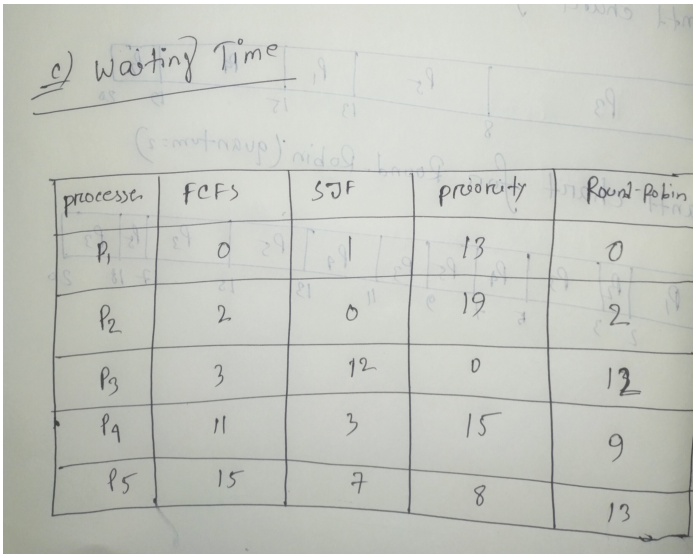
**B. What is the turnaround time of each process for each of the scheduling algorithms in part A?**




| process | FCFS | SJF | priority | Round-Robin |
|---|---|---|---|---|
| P1 | 2 | 3 | 15 | 2 |
| P2 | 3 | 1 | 20 | 3 |
| P3 | 11 | 20 | 8 | 20 |
| P4 | 15 | 7 | 19 | 13 |
| P5 | 20 | 12 | 13 | 18 |

Figure 16: Turnaround time

**C. What is the waiting time of each process for each of these scheduling algorithms?**



c) Waiting Time

| processes | FCFS | SJF | priority | Round-Robin |
|---|---|---|---|---|
| $P_1$ | 0 | 1 | 13 | 0 |
| $P_2$ | 2 | 0 | 19 | 2 |
| $P_3$ | 3 | 12 | 0 | 12 |
| $P_4$ | 11 | 3 | 15 | 9 |
| $P_5$ | 15 | 7 | 8 | 13 |

Figure 17: Waiting time (turnaround time - burst time)

**D. Which of the algorithms results in the minimum average waiting time (over all processes)?**

For first come first serve:

$$\frac{0+2+3+11+15}{4} = 6.2$$

for shortest job first:

$$\frac{1+0+12+3+7}{4} = 4.6$$

For priority scheduling:

$$\frac{13+19+0+15+8}{4} = 11$$

For Round Robin scheduling:

$$\frac{0+2+12+9+13}{4} = 7.2$$

Hence, shortest job first has the shortest wait time.

# 3

The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function: Priority = (recent CPU usage / 2) + base where base = 60 and recent CPU usage refers to a value indicating how often a process has used the CPU since priorities were last recalculated. Assume that recent CPU usage for process P 1 is 40, for process P 2 is 18, and for process P 3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler

To calculate the new priorities for the three processes (P1, P2, and P3) using the traditional UNIX scheduler's priority formula, we'll use the given recent CPU usage

values and the base value.
**Given:**
- Base value (base) = 60
- Recent CPU usage for P1 = 40
- Recent CPU usage for P2 = 18
- Recent CPU usage for P3 = 10

**We'll use the formula:**

$$\text{Priority} = \left(\frac{\text{recent CPU usage}}{2}\right) + \text{base}$$

**Calculations:**
**1.For Process P1**

$$\text{Priority for P1} = \left(\frac{40}{2}\right) + 60 = 20 + 60 = 80$$

**2.For Process P2:**

$$\text{Priority for P2} = \left(\frac{18}{2}\right) + 60 = 9 + 60 = 69$$

**3.For Process P3:**

$$\text{Priority for P3} = \left(\frac{10}{2}\right) + 60 = 5 + 60 = 65$$

**New Priorities:**
- Priority for Process P1: 80
- Priority for Process P2: 69
- Priority for Process P3: 65

# Class Test, 04-06-2024

## 2. Illustrate three cases of race conditions.

**1. Incrementing a Shared Variable:** Two threads try to increase the same counter. If both read the same value before updating, one update might be lost.
Example: If both threads read `counter = 5` and add 1, the final value might be `6` instead of `7`.

**2. Reading and Writing Shared Data:**
One thread reads data while another writes to it at the same time. This can cause the read thread to get incorrect data.
Example: Thread 1 reads from a file, and Thread 2 writes to it at the same time.

**3. Bank Account Withdrawal:**
Two threads try to withdraw money from the same account. If both check the balance at the same time, they may both withdraw money based on the same balance, causing an incorrect final balance.
Example: Both threads read the balance as $500 and withdraw $100, resulting in a balance of $400 instead of $300.

## 5

A multithreaded web server wishes to keep track of the number of requests it services (known as hits). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex lock hit_lock;
hit_lock.acquire();
hits++;
hit_lock.release();
```

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

**Answer:**
The **atomic integer approach** is generally more efficient for this task.

- **Using Mutex Lock:**
  - A mutex lock ensures only one thread can update `hits` at a time, preventing conflicts.
  - However, using locks requires additional steps to acquire and release, slowing down the process, especially in high-traffic systems.
- **Using Atomic Integer:**
  - Atomic integers update `hits` in a single, thread-safe step, without needing locks.
  - These operations are handled by hardware and are therefore much faster than locks, making them ideal in cases of frequent updates.

# Class Test, 09-06-2024

## 1

**a. Assume val is an atomic integer in a Linux system. What is the value of val after the following operations have been completed?**

```
atomic
atomic
atomic
atomic
atomic
atomic
set(&val, 10);
sub(8, &val);
inc(&val);
inc(&val);
add(6, &val);
sub(3, &val);
```

### Solution

Starting with an initial value of `val = 10`, here's the result after each operation:

- `atomic_set(&val, 10);`
  Sets `val` to 10.
- `atomic_sub(8, &val);`
  $10 - 8 = 2$
- `atomic_inc(&val);`
  $2 + 1 = 3$
- `atomic_inc(&val);`
  $3 + 1 = 4$
- `atomic_add(6, &val);`
  $4 + 6 = 10$
- `atomic_sub(3, &val);`
  $10 - 3 = 7$

**Final value of `val` = 7**

## 2

Given: 1. **Allocation Matrix**

| | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $T0$ | 0 | 0 | 1 | 2 |
| $T1$ | 1 | 0 | 0 | 0 |
| $T2$ | 1 | 3 | 5 | 4 |
| $T3$ | 0 | 6 | 3 | 2 |
| $T4$ | 0 | 0 | 1 | 4 |

2. **Max Matrix**

| | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $T0$ | 0 | 0 | 1 | 2 |
| $T1$ | 1 | 7 | 5 | 0 |
| $T2$ | 2 | 3 | 5 | 6 |
| $T3$ | 0 | 6 | 5 | 2 |
| $T4$ | 0 | 6 | 5 | 6 |

3. **Available Resources**

| $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|
| 1 | 5 | 2 | 0 |

## Question

Using the Banker's algorithm, answer the following:

(a) What is the content of the matrix *Need*?

(b) Is the system in a safe state?

(c) If a request from thread $T1$ arrives for $(0, 4, 2, 0)$, can the request be granted immediately?

## Solution

(a) The **Need** matrix is calculated as follows:

$$\text{Need} = \text{Max} - \text{Allocation}$$

Calculating for each thread:

| | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $T0$ | 0 | 0 | 0 | 0 |
| $T1$ | 0 | 7 | 5 | 0 |
| $T2$ | 1 | 0 | 0 | 2 |
| $T3$ | 0 | 0 | 2 | 0 |
| $T4$ | 0 | 6 | 4 | 2 |

(b) **Safe State Check:**
To determine if the system is in a safe state, we would check if we can find an order in which each process can complete. Starting with the available resources, we would assess each process to confirm that each can finish and free up resources. Using Banker's algorithm, we verify if a safe sequence exists.

(c) **Request from Thread $T1$:**

If thread $T1$ requests resources $(0, 4, 2, 0)$, we need to check the following conditions to see if the request can be granted immediately:

- **Request $\leq$ Need:** Since $T1$'s *Need* is $(0, 7, 5, 0)$, the request $(0, 4, 2, 0)$ is valid as it does not exceed $T1$'s need.
- **Request $\leq$ Available:** The available resources are $(1, 5, 2, 0)$, which meets $T1$'s request $(0, 4, 2, 0)$.

Since both conditions are satisfied, the request can be **granted immediately**.

# 3

## a. Can a system detect that some of its threads are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

Yes, a system can detect starvation by checking if threads are waiting too long for resources. If a thread's wait time exceeds a certain limit, it can be flagged as starving. The system can fix this by using **aging**, where the priority of waiting threads increases over time, ensuring they eventually get resources.

### b. As an OS developer what do you prefer: deadlock avoidance, deadlock prevention or recovery from deadlock? Justify your answer.

**Deadlock prevention** is preferred because it stops deadlocks before they happen by avoiding risky situations. It is simpler and more efficient than recovery, which can disrupt the system. Deadlock avoidance checks if deadlocks might happen, which uses more resources, while recovery involves fixing problems after they occur, which is complex and slow.

## 5

Consider the following snapshot of a system:

| | Allocation | | | | Max | | | |
|---|---|---|---|---|---|---|---|---|
| | $A$ | $B$ | $C$ | $D$ | $A$ | $B$ | $C$ | $D$ |
| $T_0$ | 3 | 0 | 1 | 4 | 5 | 1 | 1 | 7 |
| $T_1$ | 2 | 2 | 1 | 0 | 3 | 2 | 1 | 1 |
| $T_2$ | 3 | 1 | 2 | 1 | 3 | 3 | 2 | 1 |
| $T_3$ | 0 | 5 | 1 | 0 | 4 | 6 | 1 | 2 |
| $T_4$ | 4 | 2 | 1 | 2 | 6 | 3 | 2 | 5 |

Using the Banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the threads may complete. Otherwise, illustrate why the state is unsafe.

a. Available = (0, 3, 0, 1)

b. Available = (1, 0, 0, 2)

### Answer

**a. Available = (0, 3, 0, 1)**

**Need Matrix:**

| | Need | | | | Max | | | |
|---|---|---|---|---|---|---|---|---|
| | $A$ | $B$ | $C$ | $D$ | $A$ | $B$ | $C$ | $D$ |
| $T_0$ | 2 | 1 | 0 | 3 | 5 | 1 | 1 | 7 |
| $T_1$ | 1 | 0 | 0 | 1 | 3 | 2 | 1 | 1 |
| $T_2$ | 0 | 2 | 0 | 0 | 3 | 3 | 2 | 1 |
| $T_3$ | 4 | 1 | 0 | 2 | 4 | 6 | 1 | 2 |
| $T_4$ | 2 | 1 | 1 | 3 | 6 | 3 | 2 | 5 |

- Process $T_2$ can finish as its Need = (0, 2, 0, 0) and Available = (0, 3, 0, 1).
- After $T_2$ finishes, Available = (3, 4, 2, 2).
- Process $T_0$ can finish as its Need = (2, 1, 0, 3) and Available = (3, 4, 2, 2).
- After $T_0$ finishes, Available = (6, 4, 3, 6).
- Process $T_1$ can finish as its Need = (1, 0, 0, 1) and Available = (6, 4, 3, 6).
- After $T_1$ finishes, Available = (8, 6, 4, 6).
- Process $T_3$ can finish as its Need = (4, 1, 0, 2) and Available = (8, 6, 4, 6).
- After $T_3$ finishes, Available = (8, 11, 5, 8).
- Process $T_4$ can finish as its Need = (2, 1, 1, 3) and Available = (8, 11, 5, 8).
- After $T_4$ finishes, Available = (12, 13, 6, 10).

**Safe Order**: $T_2, T_0, T_1, T_3, T_4$ **Result**: Safe State

**b. Available = (1, 0, 0, 2)**

- Process $T_1$ can finish as its Need = (1, 0, 0, 1) and Available = (1, 0, 0, 2).
- After $T_1$ finishes, Available = (3, 2, 1, 2).
- Process $T_0$ can finish as its Need = (2, 1, 0, 3) and Available = (3, 2, 1, 2).
- After $T_0$ finishes, Available = (6, 2, 2, 6).
- Process $T_2$ can finish as its Need = (0, 2, 0, 0) and Available = (6, 2, 2, 6).
- After $T_2$ finishes, Available = (9, 3, 4, 7).
- Process $T_3$ can finish as its Need = (4, 1, 0, 2) and Available = (9, 3, 4, 7).
- After $T_3$ finishes, Available = (9, 8, 5, 9).
- Process $T_4$ can finish as its Need = (2, 1, 1, 3) and Available = (9, 8, 5, 9).
- After $T_4$ finishes, Available = (13, 10, 6, 11).

**Safe Order**: $T_1, T_0, T_2, T_3, T_4$ **Result**: Safe State

# Class Test, 25-06-2024

## 1

### b. Why are page sizes always powers of 2?

- **Hardware Simplicity**: Computers use binary (0s and 1s), so using powers of 2 makes it easier to divide memory. This allows for faster and simpler calculations when the computer needs to find the correct memory location.
- **Efficient Paging**: In a paging system, powers of 2 make it quicker to calculate where data is stored. The system can use simple bitwise operations to find the right page and the exact location inside that page.

## 2

Assuming a 1- KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
**a. 3085**
**b. 42095**
**c. 215201**
**d. 650000**
**e. 2000001**

$$\text{Page number} = \left\lfloor \frac{\text{Address}}{\text{Page Size}} \right\rfloor$$

$$\text{Offset} = \text{Address} \mod \text{Page Size}$$

**a. Address = 3085:**

$$\text{Page number} = \left\lfloor \frac{3085}{1024} \right\rfloor = 3$$

$$\text{Offset} = 3085 \mod 1024 = 13$$

**b. Address = 42095:**

$$\text{Page number} = \left\lfloor \frac{42095}{1024} \right\rfloor = 41$$

$$\text{Offset} = 42095 \mod 1024 = 111$$

**c. Address = 215201:**

$$\text{Page number} = \left\lfloor \frac{215201}{1024} \right\rfloor = 210$$

$$\text{Offset} = 215201 \mod 1024 = 161$$

**d. Address = 650000:**

$$\text{Page number} = \left\lfloor \frac{650000}{1024} \right\rfloor = 634$$

$$\text{Offset} = 650000 \mod 1024 = 784$$

**e. Address = 2000001:**

$$\text{Page number} = \left\lfloor \frac{2000001}{1024} \right\rfloor = 1953$$

$$\text{Offset} = 2000001 \mod 1024 = 129$$

# 3

## a

The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2- KB page size. How many entries are there in each of the following?
**a. A conventional, single-level page table**
**b. An inverted page table**
What is the maximum amount of physical memory in the BTV operating system?

A. **Single-Level Page Table:**

- Page size = 2 KB = $2^{11}$ bytes, so each page has an 11-bit offset.
- Number of pages (or entries in the page table) = $\frac{2^{21}}{2^{11}} = 2^{10} = 1024$.

**Maximum Physical Memory:** The maximum amount of physical memory in the BTV operating system is 64 KB.

## b. Avoiding Internal and External Fragmentation

- **Internal Fragmentation:** Can be minimized by using smaller page sizes, as smaller pages fit memory more precisely.
- **External Fragmentation:** Can be reduced by using paging or segmentation. These techniques ensure memory is allocated in fixed sizes (paging) or only as needed (segmentation).

# 4

## a. Given six memory partitions of 300 KB , 600 KB , 350 KB , 200 KB , 750 KB , and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB , 500 KB , 358 KB , 200 KB , and 375 KB (in order)?

### i. First-Fit Allocation

The *first-fit* algorithm allocates the first partition that is large enough for each process.

| Process Size | Partition Chosen | Remaining Partitions After Allocation |
|---|---|---|
| 115 KB | 300 KB | 185 KB, 600 KB, 350 KB, 200 KB, 750 KB, 125 KB |
| 500 KB | 600 KB | 185 KB, 100 KB, 350 KB, 200 KB, 750 KB, 125 KB |
| 358 KB | 750 KB | 185 KB, 100 KB, 350 KB, 200 KB, 392 KB, 125 KB |
| 200 KB | 200 KB | 185 KB, 100 KB, 350 KB, 0 KB, 392 KB, 125 KB |
| 375 KB | 392 KB | 185 KB, 100 KB, 350 KB, 0 KB, 17 KB, 125 KB |

### ii. Best-Fit Allocation

The *best-fit* algorithm allocates the smallest partition that is large enough for each process.

| Process Size | Partition Chosen | Remaining Partitions After Allocation |
|---|---|---|
| 115 KB | 125 KB | 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, 10 KB |
| 500 KB | 600 KB | 300 KB, 100 KB, 350 KB, 200 KB, 750 KB, 10 KB |
| 358 KB | 750 KB | 300 KB, 100 KB, 350 KB, 200 KB, 392 KB, 10 KB |
| 200 KB | 200 KB | 300 KB, 100 KB, 350 KB, 0 KB, 392 KB, 10 KB |
| 375 KB | 392 KB | 300 KB, 100 KB, 350 KB, 0 KB, 17 KB, 10 KB |

### iii. Worst-Fit Allocation

The *worst-fit* algorithm allocates the largest available partition for each process.

| Process Size | Partition Chosen | Remaining Partitions After Allocation |
|---|---|---|
| 115 KB | 750 KB | 300 KB, 600 KB, 350 KB, 200 KB, 635 KB, 125 KB |
| 500 KB | 635 KB | 300 KB, 600 KB, 350 KB, 200 KB, 135 KB, 125 KB |
| 358 KB | 600 KB | 300 KB, 242 KB, 350 KB, 200 KB, 135 KB, 125 KB |
| 200 KB | 350 KB | 300 KB, 242 KB, 150 KB, 200 KB, 135 KB, 125 KB |
| 375 KB | None | No partition large enough |

For the *worst-fit* algorithm, the last process (375 KB) cannot be allocated, as no remaining partition is large enough.

## b. Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

### Solution

a. **Logical Address Bits Calculation**

- Total pages in logical address space = 64
- Each page has 1,024 words

To calculate the number of bits in the logical address:

Number of bits for page number = $\log_2(64) = 6$ bits
Number of bits for offset within a page = $\log_2(1024) = 10$ bits
Total bits in logical address = $6 + 10 = 16$ bits

b. **Physical Address Bits Calculation**

- Physical memory has 32 frames
- Each frame holds 1,024 words

To calculate the number of bits in the physical address:

Number of bits for frame number = $\log_2(32) = 5$ bits
Number of bits for offset within a frame = $\log_2(1024) = 10$ bits
Total bits in physical address = $5 + 10 = 15$ bits

# Class Test, 26-06-2024

## 5. a. Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rate. Separate those algorithms that suffer from Belady's anomaly from those that do not.

1. LRU replacement
2. FIFO replacement
3. Optimal replacement
4. Second-chance replacement

### Solution:

| Rank | Algorithm | Suffer from Belady's anomaly |
|---|---|---|
| 1 | Optimal | no |
| 2 | LRU | no |
| 3 | Second-chance | yes |
| 4 | FIFO | yes |

- **Belady's Anomaly**: This occurs when increasing the number of page frames leads to more page faults. FIFO and Second-chance suffer from this anomaly, while Optimal and LRU do not.
- **Optimal**: The optimal page replacement algorithm replaces the page that will be used furthest in the future. It minimizes page faults but is impractical because it requires future knowledge of page requests.
- **LRU (Least Recently Used)**: This algorithm replaces the least recently used page. It does not suffer from Belady's anomaly, but it is more complex to implement due to the need to track page usage history.

- **Second-chance**: A modified version of FIFO that gives pages a second chance before replacing them. Despite being more efficient than FIFO, it still suffers from Belady's anomaly.
- **FIFO (First-In, First-Out)**: This algorithm replaces the oldest page in memory. It is simple but suffers from Belady's anomaly and is less efficient than other algorithms.

# 5. b. Consider the following page reference string:

$$7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.$$

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

## Solution is as follows:

| **Algorithm** | **Page Faults** |
|---|---|
| LRU | 18 |
| FIFO | 17 |
| Optimal | 13 |

# LabTest 25-05-2024

## 1. Write a C program to create a main process named 'parent_process' having 3 child processes without any grandchildren processes.

### Trace parent and child processes in the process tree.

### Show that child processes are doing addition, subtraction and multiplication on two variables initialized in the parent_process.

The required code is as follows:

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

void main()
{
    int x = 4, y = 6;
    pid_t pid1, pid2, pid3;

    pid1 = fork();
    if (pid1 < 0)
    {
        printf("Child Process Creation Failed\n");
        exit(1);
    }
    else if (pid1 == 0)
    { // First child
        printf("Child1 : Addition       : %d + %d = %d\n", x, y, x + y);
    }
    else
    {
        pid2 = fork();
        if (pid2 < 0)
        {
            printf("Child Process Creation Failed\n");
        }
        else if (pid2 == 0)
        { // Second child
            printf("Child2 : Subtraction    : %d - %d = %d\n", x, y, x - y);
        }
        else
        {
            pid3 = fork();
            if (pid3 < 0)
            {
                printf("Child Process Creation Failed\n");
            }
            else if (pid3 == 0)
            { // Third child
                printf("Child3 : Multiplication : %d * %d = %d\n", x, y, x * y);
```

```
            }
        }
    }
    sleep(30);
    wait(NULL);
    wait(NULL);
    wait(NULL);
}
```

The process tree for the above code is as follows:

```
| | \-+= 11638 macbookair /bin/zsh -il
| |   \-+= 11675 macbookair ./parent_child
| |     |--- 11676 macbookair ./parent_child
| |     |--- 11677 macbookair ./parent_child
| |     \--- 11678 macbookair ./parent_child
```

Figure 18: Process Tree

The output for the above code is as follows:

```
(base) macbookair@Shakils-Mac Test_1 % ./parent_child
Child2 : Subtraction    : 4 - 6 = -2
Child1 : Addition       : 4 + 6 = 10
Child3 : Multiplication : 4 * 6 = 24
```

Figure 19: Output of the above code

**Explanation:**
- The 'sleep()' function is included to keep the processes alive temporarily. This delay allows us to observe and trace them in the process tree using tools like 'ps' or 'top'.

- In the process tree, both the parent process and its child processes might be labeled as 'parent process'. This naming is due to the fact that when a process is forked, the child process is a copy of the parent process and shares the same memory space. Therefore, they start with the same name and code.

- When you use tools to view the process tree, you may see multiple processes with the same name because they originated from the same parent process and haven't been given unique identifiers or changed their names in the code.

## 2. Write a program to create an orphan process.

The following code of orphan_process.c is created to replace the memory space. It has an infinite loop to exist even after termination of it's parent process.

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main() {
      for(int i =0; ; i++)
```

```
        {
            printf("This is child process\n");
        }
    return 0;
}
```

Here is the parent_process.c:

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();  // Create a new process

    if (pid < 0) {
        // Fork failed
        printf("fork failed");
        exit(1);
    } else if (pid == 0) {
        // Child process
        execlp("./orphan_process", "2_orphan_process", NULL);
        // If execlp fails
        printf("execlp failed");
        exit(1);
    } else {
        // Parent process
        sleep(30);  // Keep the parent process alive for 20 seconds
    }

    return 0;
}
```

The parent process creates a child process. The child process runs the ‘orphan_process‘ program, replacing its memory space with it. The parent process then terminates itself after 30 seconds without using ‘wait()‘. While the parent process is alive for those 30 seconds, the child process remains under its control.This is illustrated in Figure 20.


Figure 20: Child having a parent

But after 30 seconds, there is no parent for child, making it an orphan and the ’systemd’ adopts it. See Figure 21. Here ’systemd’ can’t be shown because the tree of ’systemd’ is too large to accommodate in a single screenshot.

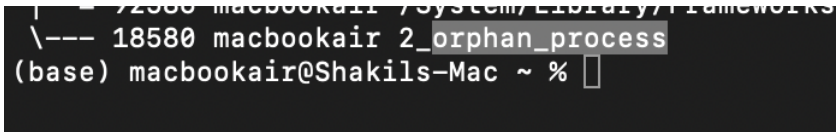



Figure 21: Child as an orphan process

## 3. Write a program to create a zombie process.

The code of the zombie_process.c is as follows:-
which is created to replace the memory space.

```
void main() {
   //Nothing is available to perform.
}
```

As this process has no operation, it immediately terminates.
The code for parent_process.c is as follows:-

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
    pid_t pid = fork();
    if(pid == 0) {
        execlp("./zombie_process", "zombie_process", NULL);
    }
    else {
        for(int i = 0; i < 10000000000 ; i++) {
            // Some operation
        }
    }

    wait(NULL);
}
```

The parent process creates a child process. The child process replaces its memory space with a new program. Both processes run at the same time. The child process finishes quickly, but the parent process keeps running a long loop. Since the parent process doesn't call 'wait()' to collect the child's termination status, the child remains in the process table with the status 'Z' for zombie.

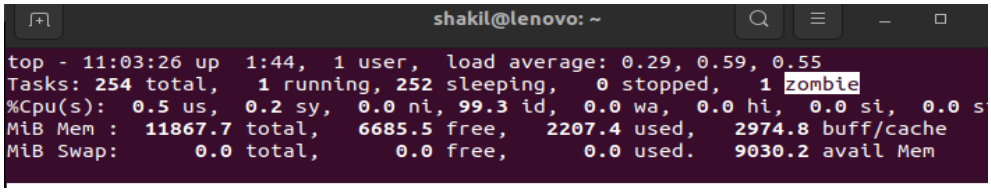



Figure 22: Existence of zombie process using "TOP" command

## 4. Write a C program to create a main process named 'parent_process' having 3 child processes without any grandchildren processes. Child Processes' names are child_1, child_2, child_3. Trace the position in the process tree.

The code for the given question is as follows:-

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
#include<sys/prctl.h>

int main() {
    pid_t pid1, pid2, pid3;

    pid1 = fork();
    if (pid1 < 0) {
        printf("Child Process Creation Failed\n");
        exit(1);
    }
    else if (pid1 == 0)
    {
        // first child
        prctl(PR_SET_NAME, "Child-1", 0, 0 , 0);
        printf("First child created\n");
        sleep(30);
    }
    else
    {
        pid2 = fork();
        if (pid2 < 0)
        {
            printf("Child Process Creation Failed\n");
            exit(1);
        }
        else if (pid2 == 0)
        {    prctl(PR_SET_NAME, "Child-2", 0, 0 , 0);
            printf("Second child created\n");
            sleep(30);
        }
        else
        {
            pid3 = fork();
            if (pid3 < 0)
            {
                printf("Child Process Creation Failed\n");
                exit(1);
            }
            else if (pid3 == 0)
            {    prctl(PR_SET_NAME, "Child-3", 0, 0 , 0);
                printf("Third child created\n");
                sleep(30); // Sleep to simulate some work
                exit(0);
            }
        }
    }

    // Parent process
    sleep(30); // Sleep to keep parent process alive for a while
    wait(NULL); // Wait for first child
    wait(NULL); // Wait for second child
    wait(NULL); // Wait for third child
```

```
    return 0;
}
```

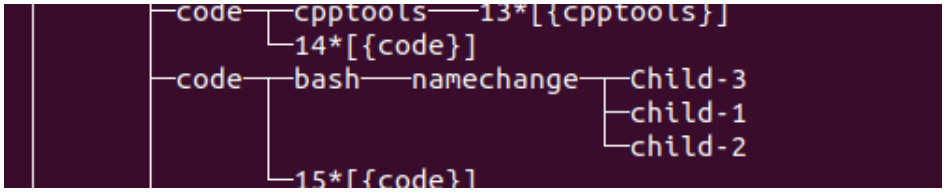



Figure 23: Changed the name of processes

**Explanation:**
This code creates three child processes. Each child prints a message saying it's created and then sleeps for 30 seconds to simulate some work before exiting. The parent process waits for all child processes to finish.

## 5. Write a C program to create a main process named 'parent_process' having 'n' child processes without any grandchildren processes. Child Processes' names are child_1, child_2, child_3,....., child_n. Trace the position in the process tree. Number of child processes (n) and name of child processes will be given in the CLI of Linux based systems.

**Example:**

```
$./parent_process 3 child_1 child_2 child_3
```

Hint: fork, exec, fopen, system
Here is the code.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/prctl.h>

int main(int argc, char *argv[]) {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        pid_t pid = fork();
        if (pid < 0) {
            printf("Child Process Creation Failed\n");
            exit(1);
        } else if (pid == 0) {
            char name[100];
            snprintf(name, sizeof(name), "Child-%d", i);
            prctl(PR_SET_NAME, name, 0, 0, 0);
```

```
            printf("Child Process %d name as %s\n", i, name);
            exit(0);
        }
    }

    sleep(100);
    for (int i = 0; i < n; i++) {
        wait(NULL);
    }
    return 0;
}
```


Figure 24: Multiple Child are created(n = 10)

**Explanation:-**
This code creates multiple child processes based on the number provided in the command line arguments. Each child process is given a name provided as an argument. After creating the processes, the parent process sleeps for 100 seconds and then waits for all child processes to finish before exiting.

# LabTest 26-05-2024

## 1. Write a C program for creating a multi-threaded process and check:

### 1.A. If one thread in the process calls fork(), does the new process duplicate all threads, or is the new process single-threaded?

No, the new process does not duplicate all threads. It is a single-threaded process. Here is the code to illustrate the answer.

```
#include <unistd.h>
```

```
#include <pthread.h>
#include <sys/prctl.h>

void thread_handler1() {
    pid_t pid = fork();
    if(pid == 0) {
        prctl(PR_SET_NAME, "child_process", NULL, NULL, NULL);
        sleep(100);
    }
    else {
        sleep(100);
    }
}

void thread_handler2() {
    sleep(100);
}

void main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, (void*)thread_handler1, NULL);
    pthread_create(&tid2, NULL, (void*)thread_handler2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
}
```

The code does not output anything. But we can see this from the process tree. This is shown in Figure 25.

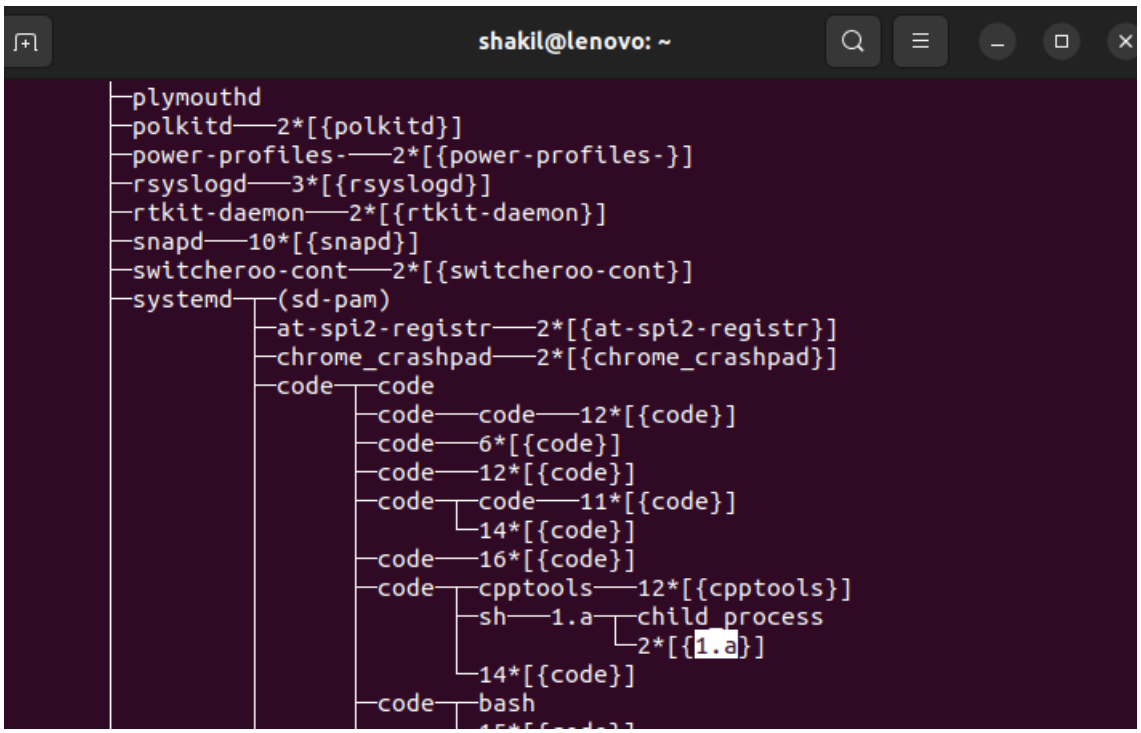



Figure 25: fork() called by a thread

### 1.B. If a thread invokes the exec() system call, does it replace the entire code of the process?

```
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>

void thread_handler1() {
    execlp("ls", "ls", NULL);
}

void thread_handler2() {
    for(int i = 0; i < 10; i++) {
        printf("%d : Thread2\n", i);
        sleep(0.05);
    }
}

void main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, (void*)thread_handler1, NULL);
    pthread_create(&tid2, NULL, (void*)thread_handler2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Parent Process\n");
}
```

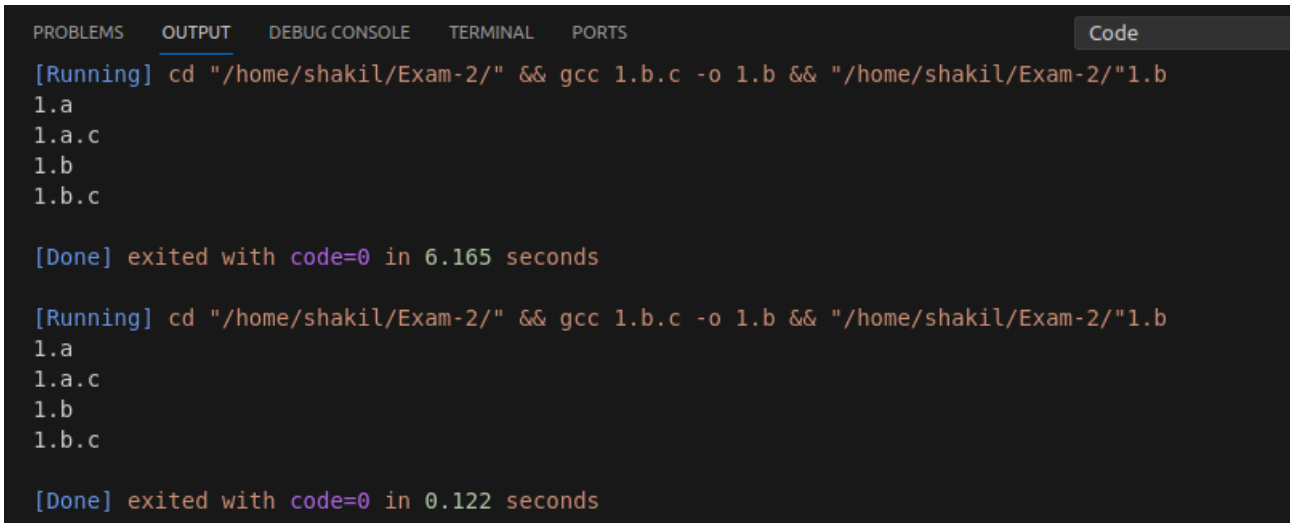



Figure 26: exec() by a thread

### 1.C. If exec() is called immediately after forking, will all threads be duplicated?

```
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>

void thread_handler1() {
    pid_t pid = fork();
    execlp("ls", "ls", NULL);
```

```
}

void thread_handler2() {
    for(int i = 0; i < 10; i++) {
        printf("%d : Thread2\n", i);
        sleep(0.05);
    }
}

void main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, (void*)thread_handler1, NULL);
    pthread_create(&tid2, NULL, (void*)thread_handler2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Parent Process\n");
}
```

```
[Running] cd "/home/shakil/Exam-2/" && gcc 1.c -o 1 && "/home/shakil/Exam-2/"1
1
1.a
1.a.c
1.b
1.b.c
1.c
1
1.a
1.a.c
1.b
1.b.c
1.c

[Done] exited with code=0 in 0.11 seconds
```

Figure 27: Output of execlp() immediately after fork()

## 3. Write a C program to show how data inconsistency arises in a multi-threaded process.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

int x = 3, y = 2;
int output = 0;

void sum() {
    for(int i = 0; i < 10 ; i++) {
        output += x + y;
        printf("Summation : i = %d x + y = %d output = %d\n", i, x + y, output);
    }
}

void sub() {
```

```
    for(int j = 0; j < 10; j++) {
        output += x - y;
        printf("Subtraction : j = %d x - y = %d output = %d\n", j, x - y, output);
    }
}

void main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, (void*)sum, NULL);
    pthread_create(&tid2, NULL, (void*)sub, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
}
```

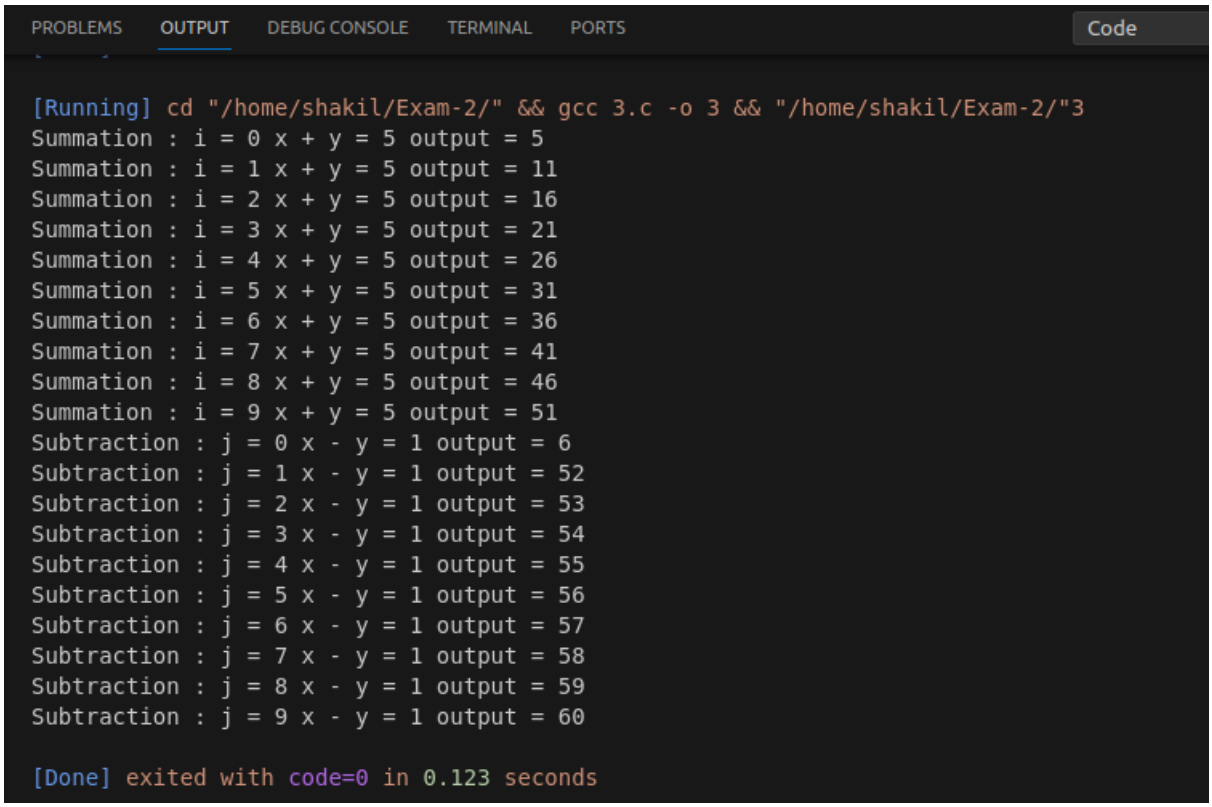



Figure 28: Data inconsistency in multithreading

# LabTest 29-06-2024

## 1

| | | |
|---|---|---|
| 1. | Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers<br>90 81 78 95 79 72 85<br>The program will report<br>A. The average value is 82<br>B. The minimum value is 72<br>C. The maximum value is 95<br>The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. | 15 |

Figure 29: Question of the first problem

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int average = 0;
int minimum = 0;
int maximum = 0;

void avg(int *numbers) {
    for(int i = 1; i <= numbers[0]; i++) {
        avrg += numbers[i];
    }
    avrg /= numbers[0];
}

void min(int *numbers) {
    minimum = numbers[1];
    for(int i = 2; i <= numbers[0]; i++) {
        if(numbers[i] < minimum) {
            minimum = numbers[i];
        }
    }
}

void max(int *numbers) {
    maximum = numbers[1];
    for(int i = 2; i <= numbers[0]; i++) {
        if(numbers[i] > maximum) {
            maximum = numbers[i];
        }
    }
}

void main(int argc, char* argv[]) {
    if(argc < 2) {
```

```
        printf("Too few arguments.\n");
        exit(1);
    }

    int *numbers = malloc(argc * sizeof(int));
    numbers[0] = argc - 1; // numbers[0] contains the value count
    for (int i = 1; i < argc; i++) {
        numbers[i] = atoi(argv[i]);
    }

    pthread_t tid1, tid2, tid3;
    pthread_create(&tid1, NULL, (void*)avg, numbers);
    pthread_create(&tid2, NULL, (void*)min, numbers);
    pthread_create(&tid3, NULL, (void*)max, numbers);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    printf("The average value is %d\n", average);
    printf("The minimum value is %d\n", minimum);
    printf("The maximum value is %d\n", maximum);
}
```

```
(base) macbookair@Shakils-Mac Test-3 % ./1.statistical_values 90 71 58 95
The average value is 78
The minimum value is 58
The maximum value is 95
(base) macbookair@Shakils-Mac Test-3 % ./1.statistical_values 90 81 78 95 79 72 3
The average value is 75
The minimum value is 30
The maximum value is 95
```

Figure 30: Multithreaded program for calculating various statistical values for a list of numbers

**The explanation of the above program is as follows:**
**1. Global Variables:** Store average, minimum, and maximum values.
**2. Functions:**
- ‘avg()‘: Computes the average.
- ‘min()‘: Finds the minimum value.
- ‘max()‘: Finds the maximum value.
**3. Main Function:**
- Checks for sufficient input.
- Allocates memory and converts input to integers.
- Creates threads to calculate average, minimum, and maximum.
- Waits for threads to complete.
- Prints the results.

## 2

| | | |
|---|---|---|
| 2. | The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ....<br>Formally, it can be expressed as:<br>$fib_0 = 0$<br>$fib_1 = 1$<br>$fib_n = fib_{n-1} + fib_{n-2}$<br>Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish. | 15 |

Figure 31: Question of the Second problem

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/wait.h>

int *array;

void fibonacci(int *n) {
    array = (int*)malloc(*n * sizeof(int));
    array[0] = 0;
    array[1] = 1;

    for(int i = 2; i < *n; i++) {
        array[i] = array[i-1] + array[i-2];
    }
}

void main(int argc, char* argv[]) {
    if(argc != 2) {
        printf("Invalid arguments list.\n");
        exit(1);
    }

    int n = atoi(argv[1]);

    pthread_t tid1;
    pthread_create(&tid1, NULL, (void*)fibonacci, &n);
    pthread_join(tid1, NULL);

    printf("Fibonacci sequence (n = %d): ", n);
    for(int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
```

```
}
```

The sample output is in Figure 32 **The explanation:-**

```
(base) macbookair@Shakils-Mac Test-3 % ./2.fibonacci_sequence
Invalid arguments list.
(base) macbookair@Shakils-Mac Test-3 % ./2.fibonacci_sequence 15
Fibonacci sequence (n = 15): 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
(base) macbookair@Shakils-Mac Test-3 % ./2.fibonacci_sequence 7
Fibonacci sequence (n = 7): 0 1 1 2 3 5 8
(base) macbookair@Shakils-Mac Test-3 %
```

Figure 32: Program for generating the Fibonacci sequence

**1.Global Array:** ‘array‘ stores the Fibonacci sequence.
**2.Fibonacci Function:**
- Allocates memory for the array.
- Computes the Fibonacci sequence up to ‘n‘.
**3.Main Function:**
- Checks for exactly one argument.
- Converts argument to integer ‘n‘.
- Creates a thread to run the ‘fibonacci‘ function.
- Waits for the thread to finish.
- Prints the Fibonacci sequence.
- Frees the allocated memory.