**Assignment No. 02**

A logo of a university of engineering and technology

Description automatically generated

**Spring 2024**

**CSE-204**

**Operating System**

Submitted by: **NAVEED AHMAD**

Registration No.: **22PWCSE2165**

Class Section: **B**

“On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work.”

Student Signature: A blue line on a white surface

Description automatically generated



Submitted to:

Dr. Madiha Sher

09/05/2024

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

**Question No. 1**: Name different types of operating systems structures. Discuss thepros and cons of each one of them.

**Answer:**

* **Monolithic kernel:**

Pros: Efficiency, performance, simplicity.

Cons: Lack of modularity, stability issues, security concerns.

* **Microkernel:**

Pros: Modularity, flexibility, security.

Cons: Performance overhead, complexity, dependency issues.

* **Hybrid Kernel:**

Pros: Balance of monolithic and microkernel advantages, flexibility, security.

Cons: Complexity, maintenance overhead, less modularity.

* **Exokernel:**

Pros: Fine-grained resource management, performance, flexibility.

Cons: Complexity, security challenges, compatibility issues.

**Question No. 2**: How are privileged instructions executed by a user process? Show the transition of modes using the figure.

**Answer:**

Privileged instructions in a user process are executed through system calls. Here's the mode transition:

* **User Mode:** User processes run here, with restricted access to system resources.
* **System Call:** When a user process needs privileged access, it triggers a system call.
* **Kernel Mode:** CPU switches to kernel mode, granting full access to system resources.
* **Return from System Call**: After executing the requested operation, control returns to the user process.
* **User Mode**: Process continues execution with limited privileges.

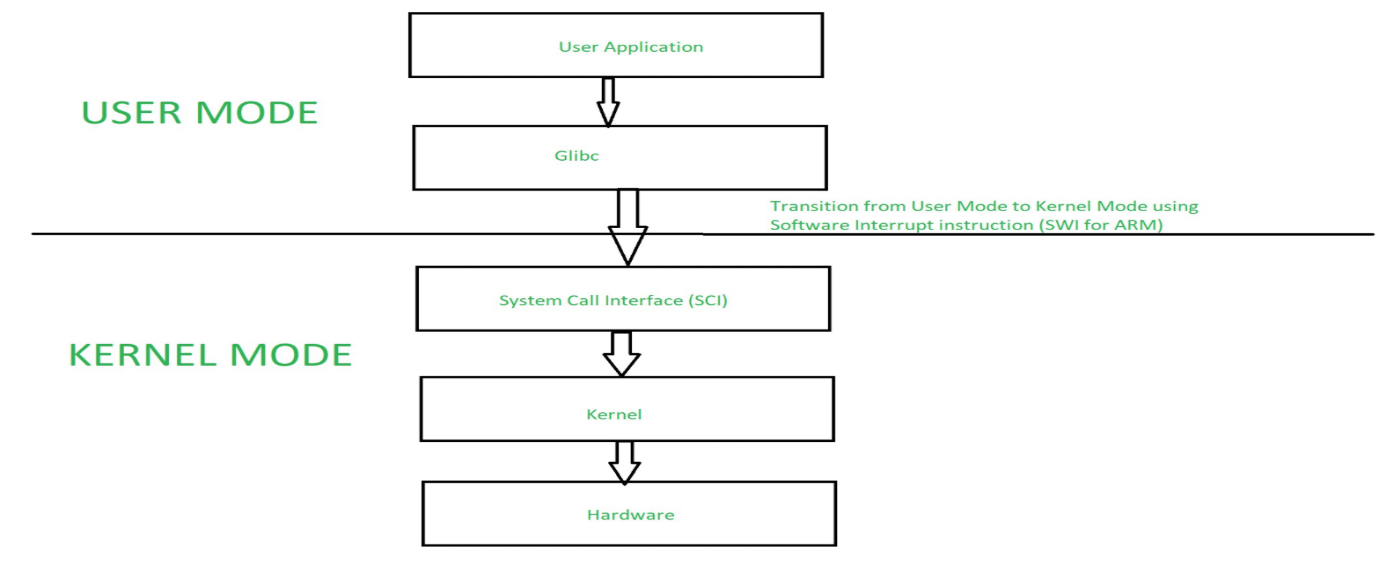


Fig 1. privileged instructions executed by a user process.

**Question No. 3:** What information is stored in the PCB of a process? Draw figure. Explain the purpose of each entry.

**Answer:**

T he Process Control Block (PCB) of a process typically stores various pieces of information necessary for managing the process. Here are some common entries in a PCB and their purposes:

* Process ID (PID): Unique identifier for the process.
* Program Counter (PC): Address of the next instruction to be executed.
* CPU Registers: Storage for general-purpose registers, program status word, etc.
* Process State: Current state of the process (e.g., running, waiting, ready).
* Priority: Priority level of the process for scheduling purposes.
* Memory Pointers: Pointers to the process's memory allocation (e.g., stack, heap).
* File Descriptors: Information about open files and I/O devices.
* Parent Process ID: PID of the parent process.
* Accounting Information: CPU usage, execution time, etc.
* Scheduling Information: Details related to process scheduling, such as time quantum, scheduling algorithm used.

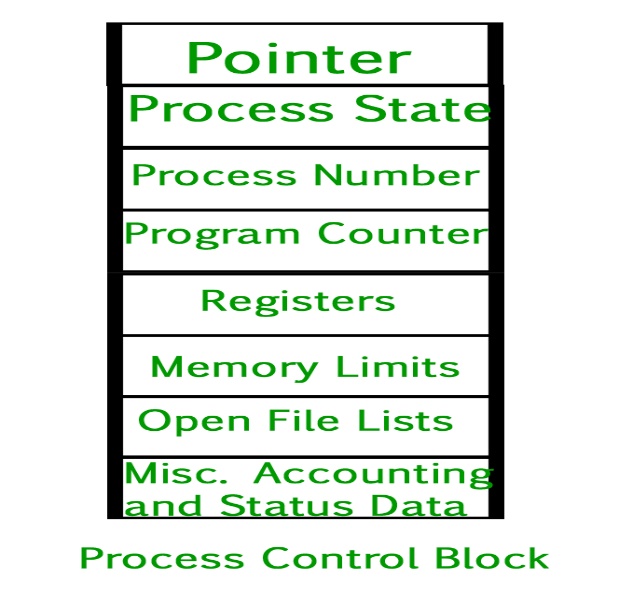
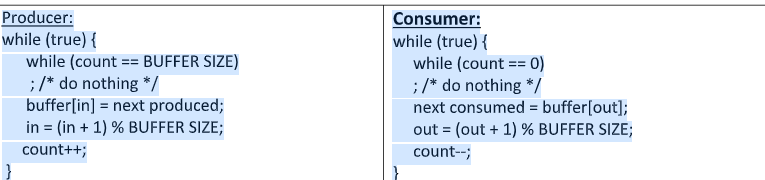


Fig 2. information is stored in the PCB of a process.

**Question No 4: What can go wrong if we execute the following code in producer**

**and consumer processes? Give Solution.**

**Answer:**

**ISSUES:**

1. Race conditions due to concurrent access to shared variables.

2. Risk of deadlock when the buffer is full or empty.

3. Potential buffer overflow/underflow.

**SOLUTION:**

* Utilize synchronization mechanisms such as semaphores.
* Ensure mutual exclusion and proper coordination between producer and consumer processes to prevent race conditions, deadlock, and buffer overflow**/underflow.**

***#define BUFFER\_SIZE 10***

***int buffer[BUFFER\_SIZE];***

***int in = 0;***

***int out = 0;***

***int count = 0;***

***semaphore mutex = 1; // Controls access to critical sections***

***semaphore full = 0; // Counts number of full slots in the buffer***

***semaphore empty = BUFFER\_SIZE; // Counts number of empty slots in the buffer***

***// Producer***

***while (true) {***

***produce\_item(); // Produce the next item***

***wait(empty); // Wait if buffer is full***

***wait(mutex); // Wait for exclusive access to buffer***

***buffer[in] = next\_produced;***

***in = (in + 1) % BUFFER\_SIZE;***

***count++;***

***signal(mutex); // Release buffer access***

***signal(full); // Increment count of full slots***

***}***

***// Consumer***

***while (true) {***

***wait(full); // Wait if buffer is empty***

***wait(mutex); // Wait for exclusive access to buffer***

***next\_consumed = buffer[out];***

***out = (out + 1) % BUFFER\_SIZE;***

***count--;***

***signal(mutex); // Release buffer access***

***signal(empty); // Increment count of empty slots***

***consume\_item(next\_consumed); // Consume the next item***

***}***

**Question No 5:** Describe the various stages a process undergoes throughout its existence, including the factors that prompt transitions between these stages.

**Answer: The lifecycle of a process typically consists of several stages. Here are the common stages:**

* **New:**

The process is being created but has not yet been admitted to the system.

Factors prompting transition: Invocation of a process creation system call by the operating system or by a running process.

* **Ready:**

The process is prepared to execute but is waiting for the CPU to be assigned to it.

Factors prompting transition: After the process is created, or after it has been waiting for I/O or an event to occur.

* **Running:**

The process is being executed on the CPU.

Factors prompting transition: The operating system scheduler allocates CPU time to the process.

* **Blocked (Waiting):**

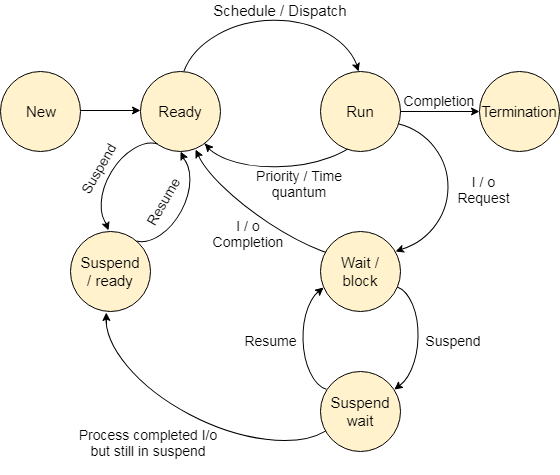
The process is unable to execute because it's waiting for an event or resource (such as I/O completion or a lock).

Factors prompting transition: The process initiates an I/O operation, requests a resource held by another process, or awaits a specific signal or event.

* Terminated (Exit):

The process has finished execution.

Factors prompting transition: The process completes its execution, explicitly terminates itself, or is terminated by the operating system due to an error.



**Fig 3. The lifecycle of a process**

**Question No. 6**: Write a program that creates a chain of 3 processes. Make sure there is no orphan or zombie process. Execute the “ls” command in child 1, the “ps” command in child 2, and the “who” command in child 3.

**Answer:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**int main() {**

**pid\_t pid1, pid2, pid3;**

**// Create child 1**

**if ((pid1 = fork()) == 0) {**

**// Child 1 executes "ls" command**

**execlp("ls", "ls", NULL);**

**perror("execlp(ls) failed");**

**exit(EXIT\_FAILURE);**

**} else if (pid1 < 0) {**

**perror("fork() failed");**

**exit(EXIT\_FAILURE);**

**}**

**// Wait for child 1 to complete before creating child 2**

**waitpid(pid1, NULL, 0);**

**// Create child 2**

**if ((pid2 = fork()) == 0) {**

**// Child 2 executes "ps" command**

**execlp("ps", "ps", NULL);**

**perror("execlp(ps) failed");**

**exit(EXIT\_FAILURE);**

**} else if (pid2 < 0) {**

**perror("fork() failed");**

**exit(EXIT\_FAILURE);**

**}**

**// Wait for child 2 to complete before creating child 3**

**waitpid(pid2, NULL, 0);**

**// Create child 3**

**if ((pid3 = fork()) == 0) {**

**// Child 3 executes "who" command**

**execlp("who", "who", NULL);**

**perror("execlp(who) failed");**

**exit(EXIT\_FAILURE);**

**} else if (pid3 < 0) {**

**perror("fork() failed");**

**exit(EXIT\_FAILURE);**

**}**

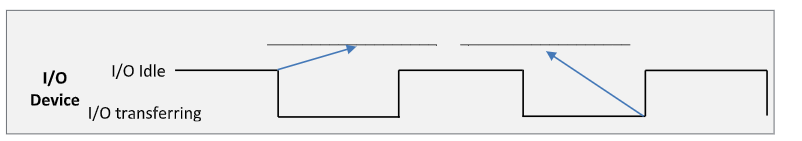
**// Wait for child 3 to complete before exiting**

**waitpid(pid3, NULL, 0);**

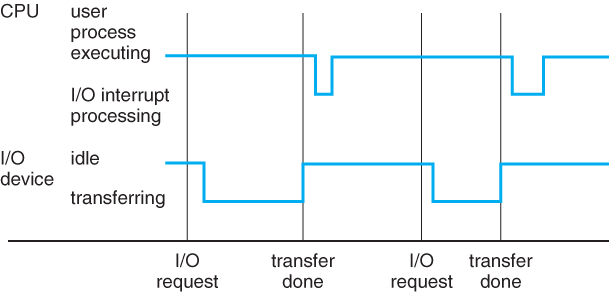
**return 0;**

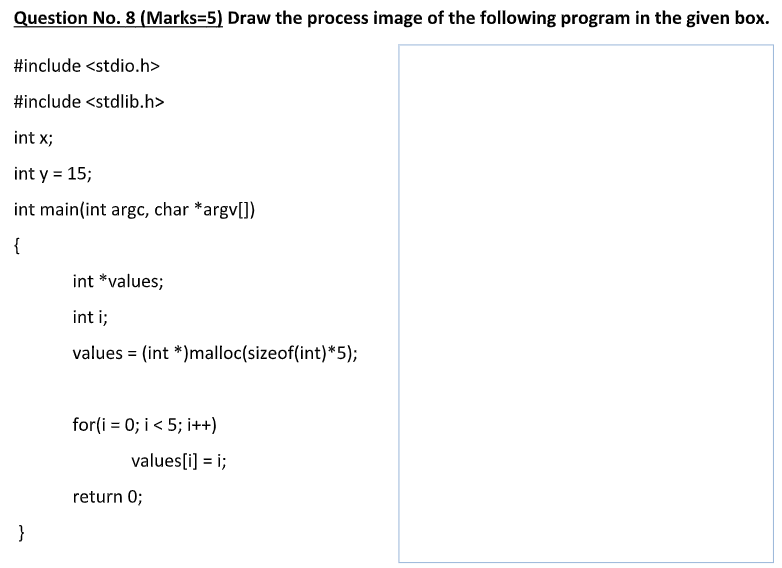
**}**

**Question No 7**: Consider the following timeline of an I/O device and draw the timeline of the CPU where the user program is currently executing. Also, mention the operation taking place that changes the state of the I/O timeline in the given blanks.

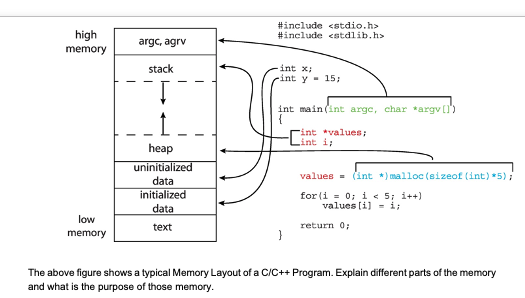
****

**Answer:**



****

**Answer:**



**Fig 4. Process image.**

**Question No. 9**: What are the two models of inter process communication? What are the strengths and weaknesses of the two approaches?

**Answer:**

1. **shared Memory Model:**

**strengths:** High performance, flexibility, direct memory access.

**Weaknesses:** Requires explicit synchronization, complexity, security risks.

1. **Message Passing Model:**

**Strengths:** Simplicity, isolation, security.

**Weaknesses:** Performance overhead, limited scalability, synchronization **challenges.**

**Question No. 10:** Why is the separation of mechanism and policy desirable?

**Answer:**

Separation of mechanism and policy is desirable for several reasons:

1. **Flexibility:** It allows for changes in policy (decision-making rules) without altering the underlying mechanisms (implementation details). This flexibility enables systems to adapt to evolving requirements without significant redesign.
2. **Modularity:** By separating mechanism from policy, systems become more modular and easier to maintain. Changes to one aspect of the system can be made without affecting other parts, leading to better code organization and readability.
3. **Reusability:** Mechanisms can be reused across different policies, promoting code reuse and reducing redundancy. This reusability leads to more efficient development processes and better resource utilization.
4. **Simplicity:** Separation of mechanism and policy simplifies system design by breaking down complex problems into smaller, more manageable components. This simplification makes it easier to understand, debug, and maintain the system.
5. **Interchangeability:** Different policies can be plugged into the same mechanism, allowing users to choose the most suitable policy for their specific requirements. This interchangeability enhances system adaptability and versatility.