6530262_MinMyintMohSoe_Assigment_1

1. What are the three main purposes of an operating system?
An operating system serves **three main purposes**: it manages the computer hardware, provides a basis for application programs, and acts as an intermediary between users and the hardware. It aims to execute user programs to solve problems efficiently, make the system convenient for users, and ensure the hardware is used effectively.

---

2. What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?
In a real-time environment, the main challenge for programmers is **ensuring that the operating system responds within strict time constraints**. This includes managing processes and hardware resources to meet deadlines, which requires precise scheduling and synchronization to avoid delays.

3.1. What are two such problems in multiprogramming and time-sharing environments?
Two security problems in such environments are **unauthorized access to data** and **resource misuse.** Users sharing the system may accidentally or maliciously interfere with each other's data or monopolize hardware resources, leading to data breaches or system inefficiencies.

3.2. Can we ensure the same degree of security in a time-shared machine as in a dedicated machine? Explain your answer.
No, achieving the same level of security is difficult because shared systems involve multiple users and processes running simultaneously, increasing the risk of interference or data leaks. Dedicated systems minimize these risks by isolating user activities.

---

4. What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?
Interrupts signal the operating system to handle events by diverting the CPU to execute a service routine(ISR). A trap is a type of software-generated interrupt caused by errors or system calls. Traps can be intentionally generated by user programs to request OS services, such as file handling.

---

5. Define the following: multiprogramming, multitasking, and multiprocessing.
Multiprogramming involves keeping multiple jobs in memory to utilize the CPU more effectively. Multitasking is an extension where the CPU rapidly switches between tasks to give users the impression of concurrent execution. Multiprocessing uses multiple processors to execute tasks simultaneously, improving performance and reliability.

---

6. Distinguish the terms interrupt, trap (system call), and mode bit.
Interrupts are hardware or software signals that alter the CPU's flow to handle specific events. Traps, a type of interrupt, are software-generated and often result from system calls or errors. Mode bits indicate the CPU's operating mode: **user mode (1)** or **kernel mode (0)**, ensuring privilege separation.

---

7. Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device it is caching, why not make it that large and eliminate the device?
Caches improve system speed by providing faster data access and reducing memory latency. They solve slow access issues but may introduce consistency problems. Large caches are impractical due to cost and complexity; devices like disks offer larger, cheaper storage than feasible caches.

---

8. Describe the differences between symmetric and asymmetric multiprocessing. What are the three advantages and disadvantages of multiprocessor systems?
In symmetric multiprocessing **(SMP)**, **all processors are peers** and share tasks equally, while asymmetric multiprocessing **(AMP)** assigns specific tasks to certain processors under **a master-slave model.** Advantages include increased throughput, cost efficiency, and reliability. Disadvantages involve complexity, potential resource contention, and higher costs than single-processor systems.

---

9. Why are caching systems designed this way (local and shared caches)?
Local caches allow each core to access data quickly without waiting for shared resources, reducing latency. Shared caches enable communication between cores and reduce duplicate data storage. This design balances speed, efficiency, and resource sharing among processing cores.

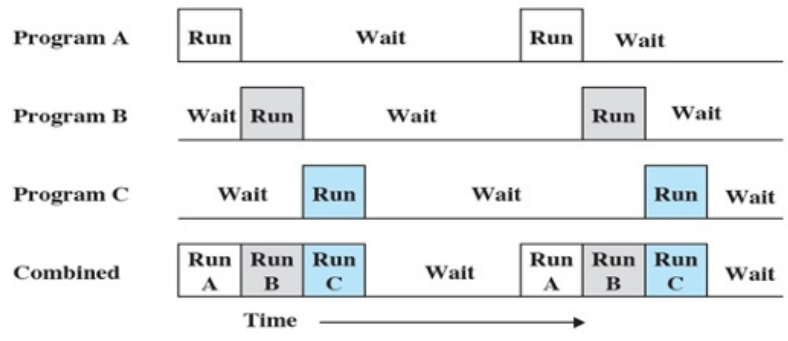


Figure 1.

10.1. Is this multiprocessing or multiprogramming? Why?
This is **multiprogramming** because **a single CPU is shared among multiple processes** (Programs A, B,

and C) by switching between them. The **"Run"** and **"Wait"** states in the diagram show that the CPU executes one program while others wait, which **aligns with multiprogramming's focus** on maximizing CPU utilization by keeping it busy with one process at a time.

10.2. Can the processes execute without utilizing the 'wait' state? Briefly describe your reasons.
No, the processes cannot execute without utilizing the "wait" state. As shown in the diagram, each process **alternates between "Run" and "Wait" states** because **certain tasks**, such as I/O operations, **require the process to pause and wait for resources or operations to complete**. This behavior is inherent in multiprogramming systems, where processes depend on resource availability.

10.3. Is there any system call involved in this process's execution? Why?
Yes, there is system call involvement in this process's execution. **System calls** are required to **transition between the "Run" and "Wait" states**. Specifically, when a process needs I/O or other OS services, it uses system calls to request these operations. The transition to "Wait" indicates that the operating system is managing resources or awaiting I/O completion before allowing the process to resume execution.

Min_Myint_Moh_Soe-6530262_Assignment(2)

1. What is the purpose of system calls?

System calls provide an interface between a process and the operating system, allowing user-level processes to request specific services from the OS kernel. They are used for:

- Program execution
- File manipulation
- Communication
- Device and resource management

2.**Five major activities** of an operating system for **process managemen**t:

1. Process creation and termination
2. Process scheduling
3. Synchronization
4. Deadlock handling
5. Inter-process communication (IPC)

3.**Three major activities** of an operating system for **memory management**:

1. Memory allocation and deallocation
2. Memory protection

3. Virtual memory management

---

4. Purpose of the **command interpreter** and why it is separate from the kernel:

- The command interpreter interprets and executes user commands, translating them into system calls.
- It is separate from the kernel to ensure flexibility; the command interpreter can be modified or replaced without altering the kernel

---

5. A command interpreter or shell must execute these system calls to start a new process.

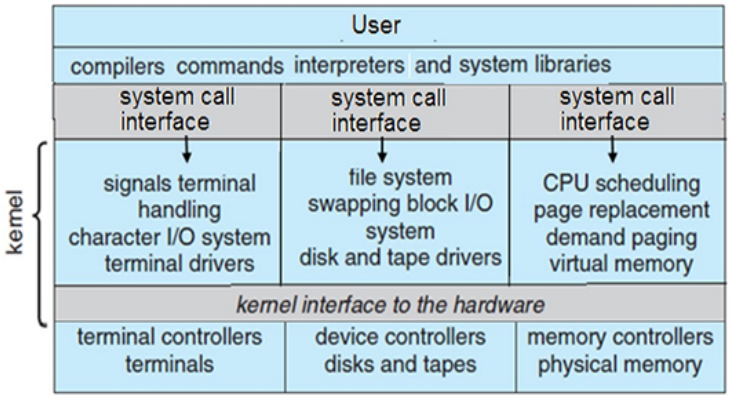
1. **fork()**: Creates a new process.
2. **exec()**: Loads a program into the process's memory space.
3. **wait()**: Waits for the process to complete.
4. **exit()**: Terminates the process

---

6. Purpose of system programs:

System programs provide convenient functionalities for users and developers. They support:

- File management
- Process monitoring
- Communication
- Programming tools like compilers and debuggers



7. OS architecture: traditional UNIX system architecture, which uses a monolithic kernel design.

7.2 Advantages and disadvantages of this architecture:

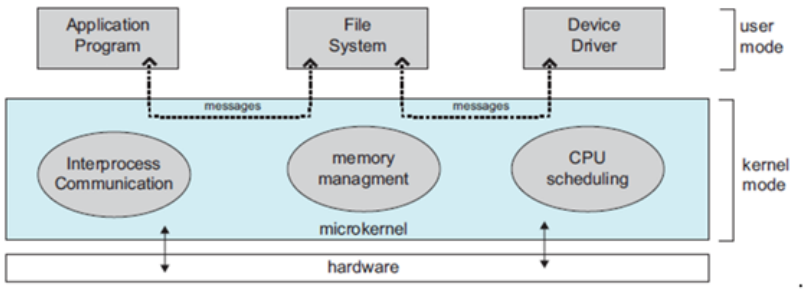
Advantages:

1. **Performance:** Communication between OS components is efficient due to their integration within the kernel space.
2. **Simplicity in Design:** The design allows direct access to hardware resources, leading to high-speed operations.
3. **Compatibility:** Traditional UNIX systems are well-tested and widely used, ensuring a robust user experience.

Disadvantages:

1. **Maintenance Challenges:** Adding or updating features requires modifying the entire kernel, which can introduce bugs.
2. **Stability Risks**: A bug in any kernel component can potentially crash the whole system.
3. **Lack of Modularity**: The tightly integrated design makes debugging and isolating issues more complex

---



8. OS architecture: a microkernel architecture. In this design, the kernel contains only essential functions, while other functionalities run in user space.

8.2 Advantages and disadvantages of this method:

Advantages:

1. **Extensibility**: New services can be added without modifying the kernel, allowing for easier system updates.
2. **Security**: Fewer components run in kernel mode, reducing the likelihood of catastrophic system crashes.
3. **Reliability**: Bugs in user-space services do not directly impact the kernel.

Disadvantages:

1. **Performance Overhead**: Increased communication between user space and kernel space leads to slower execution.
2. **Complexity in Communication**: Message passing between processes can make system interactions more complicated and inefficient.
3. **Initial Development Effort:** More effort is needed to design and implement a microkernel system compared to a monolithic kernel

---

9. Main advantage of the layered approach to system design is simplicity and modularity as each layer interacts only with adjacent layers, making debugging and system enhancements easier. The disadvantage is the added overhead of transitions between layers.

---

10. **Five OS services** and their user convenience:
    1. **Program Execution**
        - Loads and runs programs.
        - Makes it easy to execute tasks without dealing with hardware-level details.
        - User programs can't do this because they don't have access to hardware resources.
    2. **File System Management**
        - Creates, reads, writes, and deletes files.
        - Lets users store/retrieve data without worrying about disk operations.
        - User programs can't do this because they can't access storage devices directly.
    3. **I/O Operations**
        - Handles communication with devices like printers or monitors.
        - Allows users to interact with devices without managing drivers.
        - User programs can't do this because direct device access requires kernel permissions.
    4. **Interprocess Communication (IPC)**
        - Allows processes to share data or messages.
        - Makes multitasking smooth for users.
        - User programs can't do this because they can't ensure synchronization or data protection.
    5. **Security and Protection**
        - Controls access to system resources.
        - Protects user data from unauthorized access.
        - User programs can't do this because they can't enforce access controls.

---

11. **Two models of inter-process communication (IPC):**

    1. **Message passing:**
        - Strengths: Easier to implement, good for small data exchange.
        - Weaknesses: Slower for large data transfers.
    2. **Shared memory:**

    - Strengths: High speed and efficiency.
    - Weaknesses: Requires synchronization and careful design

---

12.Advantages of loadable kernel modules:

- **Flexibility** to add or remove functionalities at runtime.
- **Reduces kernel complexity** and allows system updates without rebooting.
- **Efficient use of resource**s since modules are loaded on demand

---

13. Designing a system to allow a choice of OS at boot:

- A bootloader program must support multiple OS options.
- It should:
    - Display a menu to choose an OS.
    - Load the selected OS kernel into memory.
    - Transfer control to the OS for initialization

---

14. **Yes,** it's possible for the user to develop a new command interpreter using the system-call interface provided by the operating system. The new command interpreter can use system calls to:

- Manage processes (fork, exec).
- Handle files (open, read, write).
- Communicate with other processes (IPC)

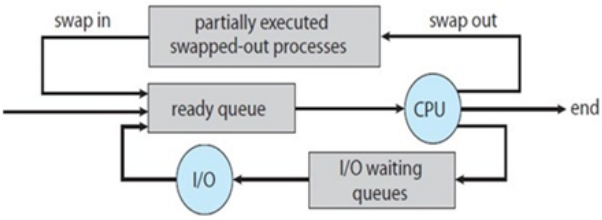
6530262_MinMyintMohSoe_Assignment_3

1. **Differences between short-term, medium-term, and long term scheduling**

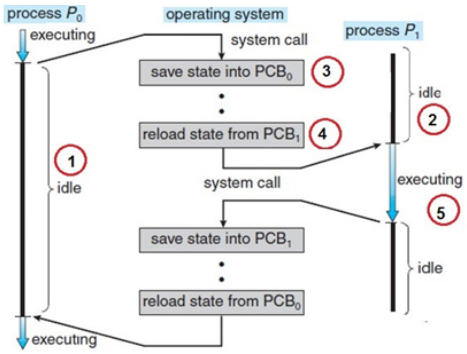
- **Short-term scheduling** selects which **process from the ready queue** gets CPU time, happening **frequently** to maximize CPU usage.
- **Medium-term scheduling** temporarily **removes processes from memory (swapping)** to reduce load and improve performance, occurring **less often**.
- **Long-term scheduling** decides which **processes to load into memory from a pool**, focusing on balancing CPU-bound and I/O-bound processes, and it happens **rarely.**

---

2. During a context switch, the kernel **saves the current process's state**, including CPU registers, program counter, and memory details, **into its PCB**. It then **restores the state of the next process from its PCB**, ensuring a smooth transition as the CPU starts executing the new process.



3. The CPU scheduler in the diagram uses medium-term scheduling, which swaps out processes to free memory or resources. This happens when a process waits for I/O or when the system reduces active processes in memory. Swapped-out processes are stored and brought back later to continue. This method is common in preemptive scheduling, like Round-Robin, to manage resources and keep the system efficient.



4.1 Importance of "Save State into PCB_0" and "Reload State from PCB_1"

"Save state into PCB_0" stores the current process's context, like CPU registers and program counter, into its PCB. This ensures the process can continue later without errors. "Reload state from PCB_1" restores the context of another process from its PCB, allowing it to run. These steps are crucial for switching processes smoothly and keeping them consistent.

4.2. Source of the System Call Causing Context Switching

The system call for context switching happens due to events like a process waiting for I/O, using up its time slice, or an interrupt like a timer signal. These events let the operating system decide to switch the CPU to another process.

4.3. CPU Mode During Steps 1 and 2

At points 1 and 2, the process lines show **"idle."** This means the CPU is running the **operating system's idle routine**, which is part of the kernel. Since the idle routine is in the kernel, the CPU is **in kernel mode** during both points.

4.4. CPU Mode During Steps 1 and 5

At point 1, the CPU is running the operating system's idle routine, so it is in kernel mode. At point 5, the process transitions to "executing," meaning the CPU is running the user process. When running a user process, the CPU is in user mode, so it operates in user mode at point 5.

6530262_MinMyintMohSoe_Assignment_4

1. Yes, concurrency is possible without parallelism. **Concurrency** means tasks progress by being interleaved, even on a single core, like **switching quickly between them**. **Parallelism**, on the other hand, means **tasks run at the same time, usually on multiple cores**. So, concurrency can appear simultaneous without actual simultaneous execution.

---

2. For S = 0.6 (60% serial,40% parallel),

   - (a) Two processing cores:
     Speedup = 1 / (S + (1 - S) / 2) = 1 / (0.6 + (1 - 0.6) / 2) = 1.25.
   - (b) Four processing cores:
     Speedup = 1 / (S + (1 - S) / 4) = 1 / (0.6 + (1 - 0.6) / 4) ≈ 1.43.

---

3. **The two differences between user-level threads and kernel-level threads**
   - User-level threads are managed by user-level libraries, while kernel-level threads are managed directly by the operating system.
   - User-level threads are faster to create and switch because they don’t involve the kernel, while kernel-level threads are slower due to system calls. User-level threads are better when blocking is rare, but kernel-level threads are better when blocking or true parallelism is needed.

---

4. When context-switching between kernel-level threads, the kernel first saves the state of the current thread (like registers and program counter) into its thread control block. It then updates the thread's status, selects the next thread to execute, restores its saved state, and transfers control to it. This ensures the new thread can resume execution seamlessly.

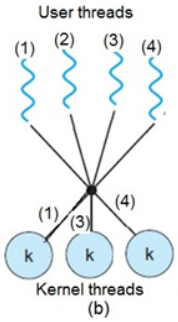
---

5. Yes, user-level threads can perform better on a multiprocessor system than on a single processor. This is because multiple threads can run simultaneously on different processors, enabling true parallelism. However, for user-level threads to achieve this, kernel-level thread support is required; otherwise, they are restricted to a single processor.

6.1 **Why is thread mapping important?**

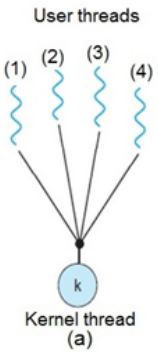
Thread mapping determines how user-level threads are linked to kernel-level threads, impacting performance and system efficiency. It controls how threads are scheduled, how blocking operations are handled, and how well parallelism is utilized.

6.2 **Thread mapping scheme for multiprogramming and multiprocessing**



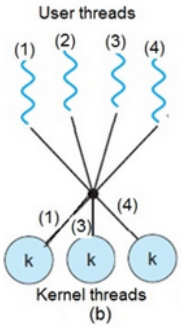
The **many-to-many mapping** (shown in Figure (b)) is more suitable for both multiprogramming and multiprocessing. It allows multiple user threads to be mapped to multiple kernel threads, enabling true parallelism and better performance on multiprocessor systems.

6.3 **Impact of user thread3 in (a) causing a system call**



In the **many-to-one mapping** , when a user thread makes a system call, the entire process is blocked. This happens because all user threads rely on the same kernel thread, and the kernel cannot schedule other user threads until the system call is complete.

6.4 **Impact of user thread1 in (b) causing a system call**



In the **many-to-many mapping** , when user thread1 makes a system call, only the kernel thread associated with it is blocked. Other user threads can continue executing on their respective kernel threads, ensuring better concurrency.

6530262_MinMyintMohSoe_Assignment_5

1.A semaphore is a variable used to manage access to shared resources. It prevents multiple processes from using the same resource at the same time. Two operations control it:

- **wait (P)**: Decreases the semaphore value. If the value is 0 or less, the process waits.
- **signal (V)**: Increases the semaphore value. If any process is waiting, it resumes.

A bounded buffer with n locations;
Semaphore mutex initialized to the value 1;

Semaphore full initialized to the value 0;
Semaphore empty initialized to the value n;

Producer process

```
do {
        produce an Item;

        wait(empty); // 1.1
        wait(mutex); // 1.2

        add item to the buffer; // Critical Section

        signal(mutex); // 1.3
        signal(full);  // Signal to consumer that the buffer is full

} while (TRUE);
```

Consumer process

```
do {
        wait(full);  // 1.4
        wait(mutex); // 1.5

        remove an item from the buffer; // Critical Section

        signal(mutex); // Signal to producer that the mutex is free
        signal(empty); // 1.6
} while (TRUE);
```

2.

i. Dataset
ii. Semaphore rw_mutex initialized to 1
iii. Semaphore mutex initialized to 1
iv. Integer read_count initialized to 0

Readers process

```
do {
        // first finding readers using mutex
        2.1 wait(mutex);
        read_count++;             // // find readers
        if (read_count == 1)      // If this is the first reader
        wait(rw_mutex);           // then a writer should wait
        2.2 signal(mutex);
```

```
        /* reading is performed */

        2.3 wait(mutex);
        read_count--;              // reading by readers
        if (read_count == 0)       // // if no more readers
        2.4 signal(rw_mutex);
        signal(mutex);             // signal 'mutex' to synchronized writers
} while (TRUE);
```

---

3. A Critical Section (CS) is a part of a program where shared resources like data or hardware are accessed. If multiple processes execute their CS simultaneously, it can cause data inconsistency or race conditions. Semaphores solve this problem by using **wait()** and **signal()** operations to allow only one process at a time in the CS. This ensures mutual exclusion and prevents conflicts, maintaining synchronization between processes.

---

4. The OS does not directly recognize user-level threads; instead, they are managed by a user-level thread library. The OS interacts with Light Weight Processes (LWPs), which act as intermediaries between user threads and kernel threads. LWPs are important because they improve scheduling, enable concurrency on multiple processors, and reduce overhead compared to directly managing user-level threads in the kernel.

---

5. The two IPC models are shared memory and message passing. Shared memory allows processes to communicate by directly accessing a shared memory space, making it fast and efficient for large data but requires complex synchronization to avoid conflicts. Message passing involves processes sending and receiving messages, which is simpler to implement and avoids race conditions, but it is slower due to system call overhead and limited by message size

---

6.

```
while (true)
    {
        if (counter == BUFFER_SIZE)
           /* do nothing */
        Buffer[i] = next_item;
        in = (in + 1) % BUFFER_SIZE;
        counter++;  }
```

This process structure represents a Producer process from the Producer-Consumer problem, using the Shared Memory IPC model. Shared resources like counter, Buffer, and BUFFER_SIZE are accessed by both the Producer and Consumer. The Producer writes to the buffer until it is full, tracked by the counter, while the Consumer retrieves items.

Time	Process	Register-counter Status	Value
T_0	producer	$register_1 = counter$	$register_1 = 5$
T_1	producer	$register_1 += 1$	$register_1 = 6$
T_2	producer	$counter = register_1$	$counter = 6$
T_3	consumer	$register_2 = counter$	$register_2 = 6$
T_4	consumer	$register_2 -= 1$	$register_2 = 5$
T_5	consumer	$counter = register_2$	$counter = 5$
T_6	producer	$register_1 = counter$	$counter = 5$

7.1 A race condition occurs in this interleaved execution because the Producer and Consumer processes access the shared counter variable simultaneously without proper synchronization. At T0, the Producer reads counter as 5, increments it, and writes 6 at T2. However, at T3, the Consumer reads the same counter value 6, decrements it, and writes 5 at T5. This overwrites the Producer's update, leading to inconsistent results. The problem arises because the operations on counter are not atomic, and there is no mutual exclusion to prevent both processes from modifying the variable at the same time.

7.2 This is a process synchronization problem because the Producer and Consumer processes are modifying a shared resource (counter) without coordination. Proper synchronization mechanisms, like semaphores or mutex locks, are required to ensure that only one process can access the shared variable at a time. Without synchronization, a race condition occurs, leading to incorrect and inconsistent results when the processes access and modify the shared resource concurrently.

---

8. Busy waiting happens when a process continuously checks for a condition, like resource availability, while consuming CPU cycles unnecessarily. Other types of waiting include blocking, where a process is put to sleep until the resource becomes available, and non-blocking waiting, where a process can perform other tasks while waiting. Busy waiting can be avoided by using blocking mechanisms or event-driven synchronization, which suspend the process and place it in a queue until the resource is free, reducing CPU wastage.

---

9. If the **wait()** and **signal()** operations are not atomic, multiple processes can modify the semaphore value at the same time, violating mutual exclusion. For example, two processes could execute **wait()**

simultaneously on a semaphore initialized to 1, both decrement it, and think they have access to the critical section. This would allow both processes to enter the critical section, causing race conditions and inconsistent behavior.

---

10. To avoid busy waiting, mutex locks can be modified to use a blocking mechanism. When a process attempts to acquire a locked mutex, it is placed in a waiting queue managed by the OS and suspended. Once the mutex is released, one process from the queue is awakened and allowed to acquire the lock. This ensures that CPU cycles are not wasted on spinning and improves system efficiency.

MinMyintMohSoe_6530262_Assignment_2_Chapter(6)

1.

Process	Arrival Time (ms)	Burst (ms)	Priority
P_1	0	8	3
P_2	1	9	1
P_3	2	7	2
P_4	3	3	5
P_5	4	12	4

FCFS

- **P1**: Completion Time = **8**
- **P2**: Completion Time = 8 + 9 = **17**
- **P3**: Completion Time = 17 + 7 = **24**
- **P4**: Completion Time = 24 + 3 = **27**
- **P5**: Completion Time = 27+ 12 = **39**

- **P1**: Waiting Time = 8 - 8 - 0 = **0**
- **P2**: Waiting Time = 17- 9 - 1 = **7**
- **P3**: Waiting Time = 24 - 7- 2 = **15**
- **P4**: Waiting Time = 27 - 3 - 3 = **21**
- **P5**: Waiting Time = 39 - 12 - 4 = **23**
- **Average Waiting Time:** (0 + 7 + 15 + 21 + 23) / 5 = **13.2 ms**

STRF

Gantt Chart

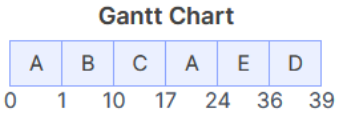
Time	Process Executing
0-3	P1
3-6	P4
6-11	P1
11-18	P3
18-27	P2
27-39	P5

- **P1**: Waiting Time = 11 - 8 - 0 = **3**
- **P2**: Waiting Time = 27 - 9 - 1 = **17**
- **P3**: Waiting Time = 18 - 7- 2 = **9**
- **P4**: Waiting Time = 6 - 3 - 3 = **0**
- **P5**: Waiting Time = 39 - 12 - 4 = **23**
- **Average Waiting Time:** (3 + 17 + 9 + 0 + 23) / 5 = **10.4 ms**

Priority Scheduling

Output

PP



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	8	24	24	16
B	1	9	10	9	0
C	2	7	17	15	8
D	3	3	39	36	33
E	4	12	36	32	20
			Average	116 / 5 = 23.2	77 / 5 = 15.4

- **P1**: Waiting Time = 24 - 8 - 0 = **16**
- **P2**: Waiting Time = 10 - 9 - 1 = **0**
- **P3**: Waiting Time = 17 - 7- 2 = **8**
- **P4**: Waiting Time = 39 - 3 - 3 = **33**
- **P5**: Waiting Time = 36 - 12 - 4 = **20**
- **Average Waiting Time: 15.4ms**

RR scheduling

Time	Process Executing
0-4	P1
4-8	P2
8-12	P3
12-15	P4
15-19	P5
19-23	P1

23-27	P2
27-30	P3
30-34	P5
34-35	P2
35-39	P5

- **P1**: Waiting Time = 23 - 8 - 0 = **15**
- **P2**: Waiting Time = 35 - 9 - 1 = **25**
- **P3**: Waiting Time = 30 - 7- 2 = **21**
- **P4**: Waiting Time = 15 - 3 - 3 = **9**
- **P5**: Waiting Time = 39 - 12 - 4 = **23**
- **Average Waiting Time:** (15 + 25 + 21 + 9 + 23) / 5 = **18.6ms**

2.

Process	Arrival Time (ms)	CPU Burst (ms)	Priority
P_1	0	6	4
P_2	1	3	1
P_3	2	10	2
P_4	3	8	3

FCFS

- **P1**: Completion Time = **6**
- **P2:** Completion Time = 6 + 3 = **9**
- **P3:** Completion Time = 9 + 10 = **19**
- **P4**: Completion Time = 19 + 8 = **27**

- **P1:** Waiting Time = 6 - 6 - 0 = **0**
- **P2**: Waiting Time = 9 - 3 - 1 = **5**

- **P3:** Waiting Time = 19 - 10 - 2 = **7**
- **P4:** Waiting Time = 27 - 8 - 3 = **16**
- **Average Waiting Time:** (0 + 5 + 7 + 16) / 4 = 7 ms

SRTF

Time	Process Executing
0-1	P1
1-4	P2
4-9	P1
9-17	P4
17-27	P3

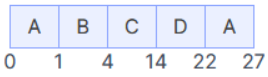
- **P1:** Waiting Time = **9 - 6 - 0 = 3**
- **P2:** Waiting Time = **4 - 3 - 1 = 0**
- **P3:** Waiting Time = **27 - 10 - 2 = 15**
- **P4:** Waiting Time = **17 - 8 - 3 = 6**
- **Average Waiting Time**: (3 + 0 + 15 + 6) / 4 = **6 ms**

Priority Scheduling

Output

PP

Gantt Chart



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	6	27	27	21
B	1	3	4	3	0
C	2	10	14	12	2
D	3	8	22	19	11
			Average	61 / 4 = 15.25	34 / 4 = 8.5

- **P1:** Waiting Time = 27 - 6 - 0 = **21**
- **P2:** Waiting Time = 4 - 3 - 1 = **0**
- **P3:** Waiting Time = 14 - 10 - 2 = **2**
- **P4:** Waiting Time = 22 - 8 - 3 = **11**
- **Average Waiting Time**: **8.5 ms**

RR scheduling

Gantt Chart

Time	Process Executing
0-4	P1
4-7	P2
7-11	P3
11-15	P4
15-17	P1
17-21	P3
21-25	P4

25-27	P3

- **P1:** Waiting Time = 17 - 6 - 0 = **11**
- **P2:** Waiting Time = 7 - 3 - 1 = **3**
- **P3:** Waiting Time = 27 - 10 - 2 = **15**
- **P4:** Waiting Time = 25 - 8 - 3 = **11**
- **Average Waiting Time**: (11 + 3 + 15 + 14) / 4 = **10.75 ms**

3.

Process	Arrival Time (ms)	CPU Burst (ms)	Priority
P_1	0	11	1
P_2	1	4	3
P_3	2	8	4
P_4	3	24	2

FCFS

- **P1**: Completion Time = **11**
- **P2**: Completion Time =11 + 4 = **15**
- **P3**: Completion Time = 15 + 8 = **23**
- **P4**: Completion Time = 23 + 24 = **47**

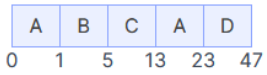
- **P1**: Waiting Time = 11 - 11 - 0 = **0**
- **P2**: Waiting Time = 15 - 4 - 1 = **10**
- **P3**: Waiting Time = 23 - 8 - 2 = **13**
- **P4**: Waiting Time = 47 - 24 - 3 = **20**
- **Average Waiting Time:** (0 + 10 + 13 + 20) / 4 = **10.75 ms**

STRF

Output

SRTF

Gantt Chart



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	11	23	23	12
B	1	4	5	4	0
C	2	8	13	11	3
D	3	24	47	44	20
			Average	82 / 4 = 20.5	35 / 4 = 8.75

- **P1:** Waiting Time = 23 - 11 - 0 = **12**
- **P2:** Waiting Time = 5 - 4 - 1 = **0**
- **P3:** Waiting Time = 13 - 8 - 2 = **3**
- **P4:** Waiting Time = 47 - 24 - 3 = **20**
- **Average Waiting Time**: **8.75 ms**

Priority Scheduling

Gantt Chart

Time	Process Executing
0-11	P1
11-35	P4
35-39	P2
39-47	P3

- **P1:** Waiting Time = 11 - 11 - 0 = **0**
- **P2:** Waiting Time = 39 - 4 - 1 = **34**
- **P3:** Waiting Time = 47 - 8 - 2 = **37**
- **P4:** Waiting Time = 35 - 24 - 3 = **8**
- **Average Waiting Time**: (0 + 8 + 34 + 37) / 4 **= 19.75 ms**

RR scheduling

Gantt Chart

Time	Process Executing
0-4	P1
4-8	P2
8-12	P3
12-16	P4
16-20	P1
20-24	P3
24-28	P4
28-32	P1
32-36	P4
36-40	P4
40-44	P4
44-48	P4

- **P1:** Waiting Time = 32 - 11 - 0 = **21**
- **P2:** Waiting Time = 8 - 4 - 1 = **3**
- **P3:** Waiting Time = 24 - 8 - 2 = **14**
- **P4:** Waiting Time = 48 - 24 - 3 = **21**
- **Average Waiting Time**: (21 + 3 + 14 + 21) / 4 = **14.75 ms**

---

4. The number of different schedules possible for n processes to be scheduled on one processor is n! (n factorial), which is calculated as n × (n - 1) × (n - 2) × ... × 1.

5.

Preemptive Scheduling:

- The CPU can be taken away from a running process and allocated to another process.
- It allows higher-priority processes to interrupt a lower-priority process.
- Useful for real-time and time-sharing systems.

Non-Preemptive Scheduling:

- Once a process starts execution, it cannot be interrupted until it completes or voluntarily gives up the CPU.
- Simpler to implement as no context switches occur during execution.
- Suitable for batch systems where processes do not require frequent switching.

---

6. Yes, it is necessary to bind a real-time thread to an LWP. This ensures that the real-time thread has direct access to the kernel and can be scheduled immediately without waiting for the thread library to map it to an available LWP. Binding eliminates potential delays caused by the many-to-many mapping, allowing the real-time thread to meet strict timing requirements.

---

7. It is important for the scheduler to distinguish I/O-bound programs from CPU-bound programs because:

- **I/O-bound programs** spend more time waiting for I/O operations and less time using the CPU. Scheduling them promptly can improve system responsiveness and prevent the CPU from being idle.
- **CPU-bound programs** spend most of their time performing computations and use the CPU intensively. Scheduling them effectively ensures that they do not monopolize the CPU, allowing I/O-bound programs to make progress.

By distinguishing between the two, the scheduler can balance resource utilization, improve throughput, and enhance overall system performance.

Quiz 1

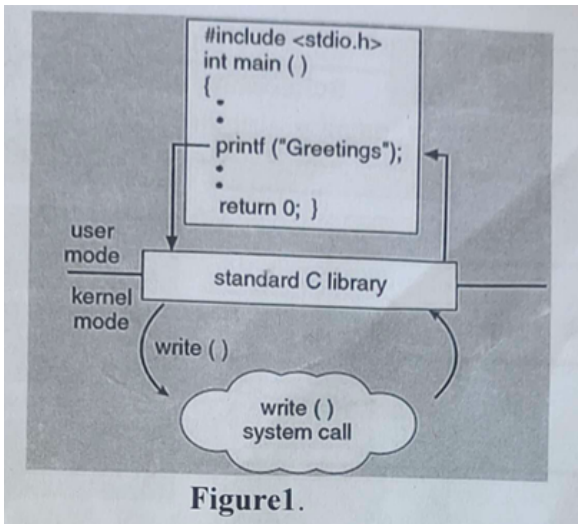


Figure1.

1.1 What is a system call?

A **system call** allows a user program to request services from the operating system, like accessing hardware or performing I/O. It switches execution from **user mode** to **kernel mode** to safely perform the task.

1.2 Why does the user code need a system call?

User code needs system calls because it **cannot access hardware or OS services directly** while in **user mode**. The system call requests the OS to perform the task in **kernel mode**, ensuring safety and control.

1.3 How does the CPU behave during a system call?

1. The user program calls a **system call** (e.g., write() in Figure 1).
2. The CPU **switches to kernel mode**.
3. The OS executes the requested service (like writing).
4. The CPU **returns to user mode** and resumes the program.

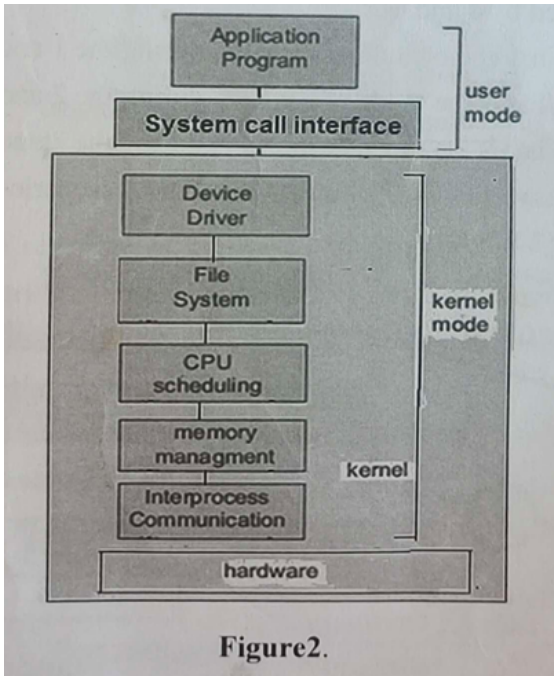


Figure2.

2.1 Identify the OS architecture.

The architecture shown is a **Monolithic Kernel**.

2.2 What is the main advantage of this architecture?

The main advantage is **efficiency**. All core OS services (like file system, CPU scheduling, and memory management) run in kernel mode, making execution faster.

2.3 What is the disadvantage of this architecture?

The main disadvantage is that it **lacks modularity**. If one part fails, the whole system can crash because all components share the same kernel space.

---

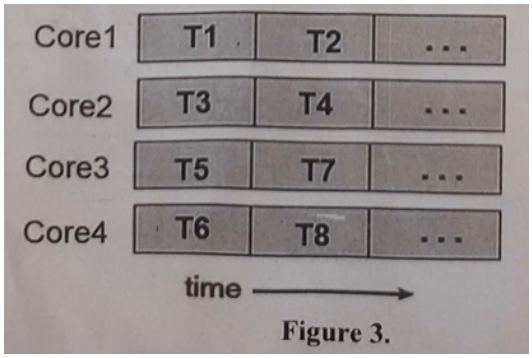
3. 

Figure 3.

- Formula: Speedup = 1 / (S + (1 - S) / N)
- S = 0.5 (parallelizable portion is 0.5 as seen in the handwritten notes).
- N = 4 (number of cores)
- Speedup = 1 / (0.5 + 0.125)
- Speedup = 1 / 0.625 = 1.6

---

4. What is a process? Briefly describe its various states.

- A **process** is a program in execution. It has a program counter, stack, and data section.
- **States of a Process**:
  - **New**: Being created.
  - **Ready**: Waiting for CPU time.
  - **Running**: Currently executing.
  - **Waiting**: Paused, waiting for an event like I/O.
  - **Terminated**: Finished execution.

---

5. Why is the process concept important in an OS? Briefly describe how the OS creates a process for a user application.

- **Importance**: It lets the OS run multiple tasks, manage resources, and execute programs efficiently.
- **How the OS creates a process**:
  1. Load the program into memory.
  2. Create a **PCB** (Process Control Block) with details like state and resources.
  3. Set up the stack, heap, and data sections.
  4. Move the process to the **ready queue** to wait for CPU execution.

6.

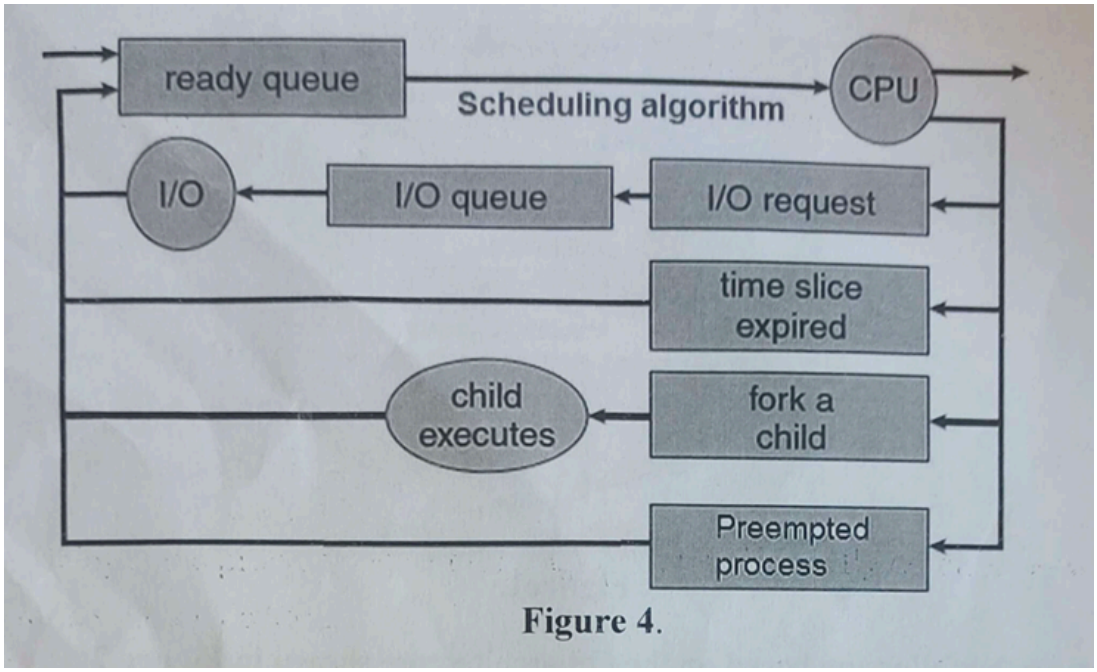


Figure 4.

Ready Queue

The ready queue holds processes waiting for CPU time. Scheduling algorithms like FCFS, SJF, Priority, or Round Robin decide which process runs next.

Time Slice Expired

In Round Robin, processes get a fixed time to execute. If they don't finish, they return to the ready queue, ensuring fairness.

I/O Queue

Processes needing I/O are moved to the I/O queue. Once I/O finishes, they go back to the ready queue. This keeps the CPU busy.

Preempted Process

A preempted process happens when the CPU is taken from a running process. This allows higher-priority or waiting processes to run.

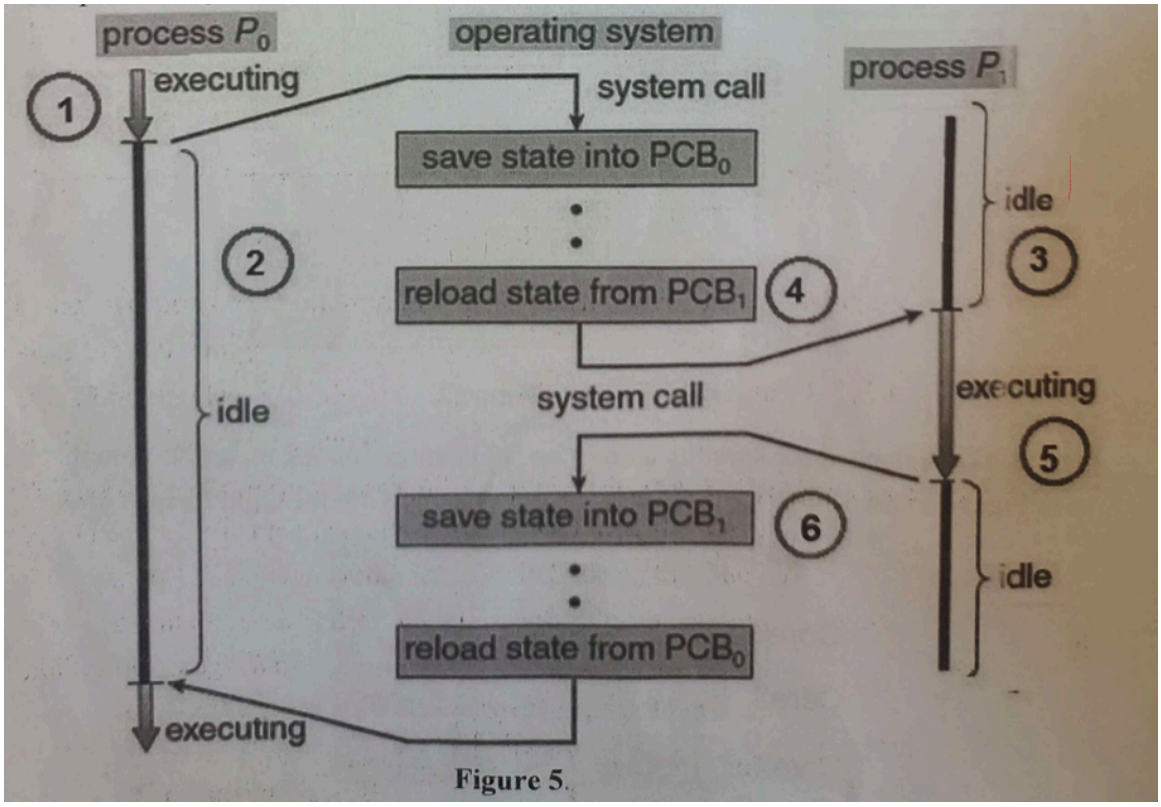


Figure 5.

9.1: Saving and reloading process states using PCBs is critical because it preserves a process's execution context (CPU registers, program counter) when it is switched out. This lets the OS later restore the process exactly where it left off, ensuring smooth execution after context switches.

9.2 (During 1 and 3):

- At step 1, the CPU is executing a user process, so it is in user mode.
- At step 3, the CPU is idle, which is handled by the OS, so it is in kernel mode.

9.3 (During 2 and 5):

- At step 2, the OS is performing system tasks (saving/reloading states), so the CPU is in kernel mode.
- At step 5, the CPU runs another user process, so it is in user mode.

9.4 (During 2 and 3):

- At step 2, the CPU is handling OS-level operations, so kernel mode.
- At step 3, the CPU is idle (an OS-controlled state), so kernel mode.

8.How the Mutex lock solves the critical section issue and its disadvantage

- **Solution**: A Mutex lock ensures that only **one process** can enter the critical section at a time. It locks the resource before use and unlocks it after finishing, avoiding data conflicts.
- **Disadvantage**: It causes **busy waiting**, where processes repeatedly check the lock, wasting CPU time.

---

9. What is a semaphore and how it solves readers' priority in a reader-writer problem

- **Semaphore**: A semaphore is a tool that controls access to shared resources using two operations: wait and signal.
- **Readers' Priority Solution**:
    1. Multiple readers can enter the critical section at the same time.
    2. Writers wait until all readers are done.
    3. Readers are allowed as long as no writer is waiting.

This gives readers priority while ensuring fairness.

10.

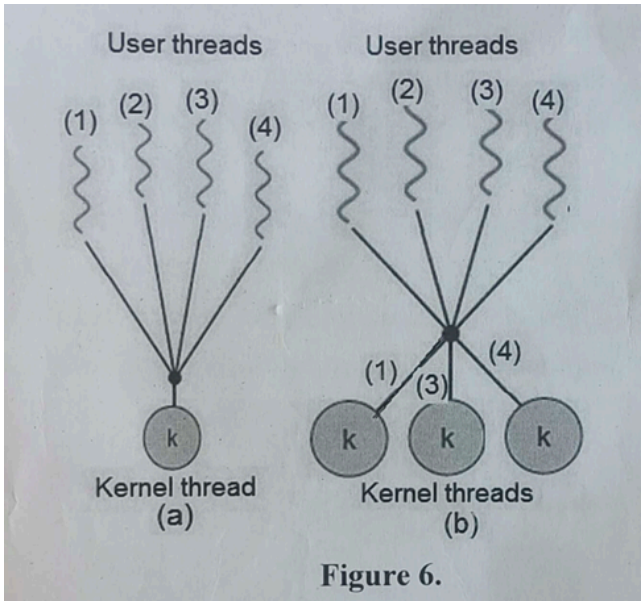


Figure 6.

10.1 Why is thread mapping important?

Thread mapping is important because it determines how **user-level threads** are mapped to **kernel-level threads**, which affects performance, concurrency, and system resource utilization.

10.2 Identify the thread mapping scheme suitable for multiprogramming and multiprocessing.

- **Scheme (a): Many-to-One**: Not suitable for multiprogramming or multiprocessing because only one kernel thread handles all user threads, blocking other threads if one makes a system call.
- **Scheme (b): One-to-One**: Suitable for **multiprogramming and multiprocessing** as each user thread maps to its own kernel thread, allowing true parallelism on multiple cores.

10.3 Impact when user thread3 in (a) causes a system call.

In **(a) Many-to-One mapping**, if user thread3 makes a system call, **all other user threads are blocked** because the single kernel thread is busy handling the call.

10.4 Impact when user thread1 in (b) causes a system call.

In **(b) One-to-One mapping**, if user thread1 makes a system call, **other user threads can continue executing** because they have separate kernel threads.

11.

Time	Process	Register-counter Status	Value
T_0	producer	$register_1 = counter$	$register_1 = 5$
T_1	producer	$register_1 += 1$	$register_1 = 6$
T_2	producer	$counter = register_1$	$counter = 6$
T_3	consumer	$register_2 = counter$	$register_2 = 6$
T_4	consumer	$register_2 -= 1$	$register_2 = 5$
T_5	consumer	$counter = register_2$	$counter = 5$
T_6	producer	$register_1 = counter$	$counter = 5$

Figure7.

11.1 What is a race condition?

A **race condition** happens when two or more processes access and modify a shared variable at the same time, leading to unpredictable or incorrect results.

11.2 Check whether any race condition occurs and describe its reasons.

Yes, a **race condition** occurs in the interleaved execution.

- **Reason**:
  The producer and consumer processes both read and write to the shared counter. At time **T2**, the producer writes **6** to the counter, but at **T5**, the consumer overwrites it to **5**. This happens because both processes use their own registers and update the counter without proper synchronization.

11.3 Is this situation a process synchronization problem? Why?

Yes, this is a **process synchronization problem**.

- **Why**:
  The shared counter variable is updated by multiple processes without coordination. Proper synchronization mechanisms, like **mutex locks** or **semaphores**, are needed to avoid conflicting updates and ensure consistency.

Process	Arrival Time (ms)	CPU Burst (ms)	Priority
P_1	0	8	3
P_2	1	9	1
P_3	2	7	2
P_4	3	3	5
P_5	4	12	4

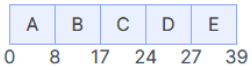
Table1.

12.1 FCFS

Output

FCFS

Gantt Chart



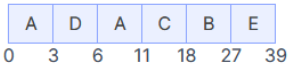
Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	8	8	8	0
B	1	9	17	16	7
C	2	7	24	22	15
D	3	3	27	24	21
E	4	12	39	35	23
			Average	105 / 5 = 21	66 / 5 = 13.2

12.2 SRTF

Output

SRTF

Gantt Chart



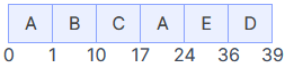
Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	8	11	11	3
B	1	9	27	26	17
C	2	7	18	16	9
D	3	3	6	3	0
E	4	12	39	35	23
			Average	91 / 5 = 18.2	52 / 5 = 10.4

12.3 Priority

Output

PP

Gantt Chart



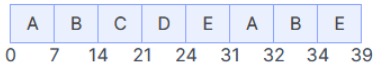
Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	8	24	24	16
B	1	9	10	9	0
C	2	7	17	15	8
D	3	3	39	36	33
E	4	12	36	32	20
			Average	116 / 5 = 23.2	77 / 5 = 15.4

12.4

Output

RR

Gantt Chart



Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	0	8	32	32	24
B	1	9	34	33	24
C	2	7	21	19	12
D	3	3	24	21	18
E	4	12	39	35	23
			Average	140 / 5 = 28	101 / 5 = 20.2

Definition of Deadlock: "Deadlock is a permanent blocking situation where processes wait indefinitely for resources held by each other. The four necessary conditions are: (1) Mutual Exclusion, (2) Hold and Wait, (3) No Preemption, and (4) Circular Wait."

Address Binding: "Address binding is the process of mapping logical addresses to physical memory addresses. It occurs at compile-time, load-time, or execution-time. This allows programs to run regardless of their location in memory and enables memory protection between processes."

Demand Paging: "Demand paging loads pages into memory only when they are accessed, triggering page faults when needed. It uses valid/invalid bits to track page status and allows programs larger than physical memory to execute by swapping pages between disk and RAM."

Direct I/O (Polling) is where the CPU passes commands through the I/O subsystem to the device driver and repeatedly checks the device status register until the device is ready. During this time, the requesting process is blocked.

The main drawback is CPU time waste - the processor spends valuable cycles repeatedly checking device status instead of performing other useful tasks.

This is an **Interrupt-driven I/O operation**. After issuing an I/O command, the CPU continues other tasks without waiting for the device. Once the device status is available(either ready or error), it send interrupt(IRQ) to the CPU. The CPU then decide whether to handle this interrupt based on the device priority

DMA : The OS handles DMA by first setting up the DMA controller through a system call. After that, the DMA controller takes control of the system bus (cycle stealing), transfers data directly between memory and the I/O device, and finally interrupts the CPU when finished. DMA is preferable for transferring large blocks of data, as it reduces CPU involvement and increases overall efficiency

6530262_MinMyintMohSoe_Assignment(6)

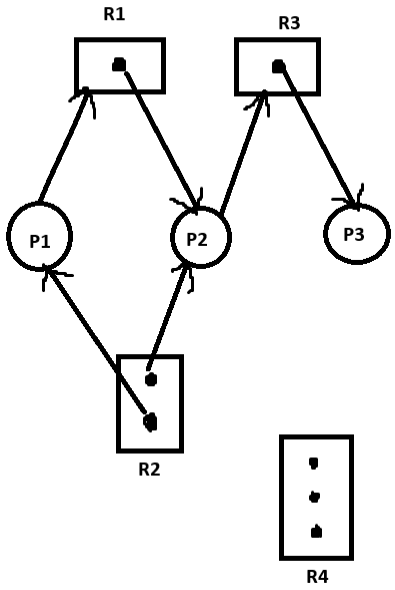
1. Define the term deadlock. Briefly describe the four conditions that can cause a deadlock in a system.

A deadlock happens when a set of processes cannot move forward because they are waiting for resources held by each other. In this situation, none of the processes can complete their tasks.

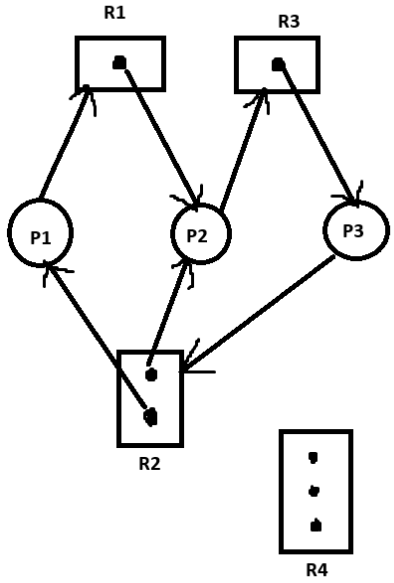
Four Conditions for Deadlock:

1. Mutual Exclusion: Only one process can use a resource at a time.
2. Hold and Wait: A process holding resources waits for more resources held by others.
3. No Preemption: Resources cannot be forcibly taken from processes.
4. Circular Wait: Processes form a cycle, each waiting for a resource held by the next.

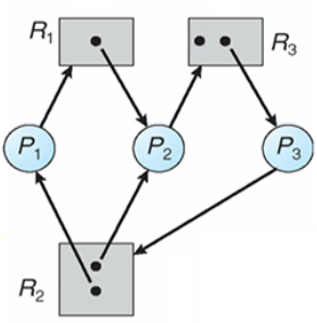
2. “Deadlocks can be described more precisely in terms of a directed graph called a resource allocation graph”. Based on this description, show resource allocation graphs with and without deadlocks.
Resource allocation graph without deadlocks:



Resource allocation graph with deadlock:



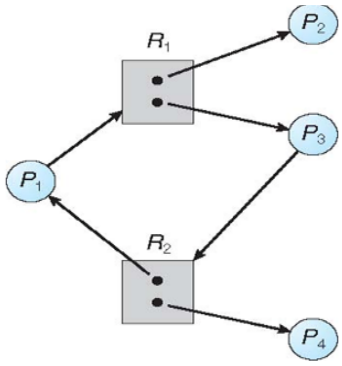
3. Consider the following resource allocation graph (Figure 1), attempt the graph reduction method, and determine whether any deadlock exists (where R1, R2, and R3 are resources and P1, P2, P3, and P4 are processes).



Yes deadlock exists since :

- P1: Waiting for R1, held by P2.
- P2: Waiting for R3, held by P3.
- P3: Waiting for R2, held by P1.

4.Consider the following resource allocation graph (shown in Figure 1), attempt the graph reduction method, and determine whether any deadlock exists (where R1, R2, and R3 are resources and P1, P2, P3, and P4 are processes).



No deadlock because P4 can release R2, breaking the cycle.

5. Consider the following snapshot of a system (where P0, P1, P2, P3, P4, A, B, C, and D are processes and resources, respectively):

	Allocation	*Max*	*Available*	*Need*
	A B C D	*A B C D*	*A B C D*	
P_0	0 0 1 2	0 0 1 2	1 5 2 0	
P_1	1 0 0 0	1 7 5 0		
P_2	1 3 5 4	2 3 5 6		
P_3	0 6 3 2	0 6 5 2		
P_4	0 0 1 4	0 6 5 6		

5.1. What is the content of the matrix Need?

Need

P0 : 0 0 0 0

P1 : 0 7 5 0

P2 : 1 0 0 2

P3 : 0 0 2 0

P4 : 0 6 4 2

5.2. Is the system in a safe state? If yes, then show the safe state sequence.

Sequence : (P0, P2, P1,P3, P4)

5.3. If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?

Yes a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately since it is less than Need for P1 : 0 7 5 0 and also less than Available value (1 5 2 0).

6. Describe various options for breaking a deadlock.

Options for Breaking Deadlock:

1. Process Termination:
    - Abort all deadlocked processes.
    - Abort one process at a time until the deadlock is resolved.
2. Resource Preemption:
    - Take resources from processes and reassign them.
    - Requires careful selection of processes and rollback mechanisms to avoid starvation.

6530262_MinMyintMohSoe_OS_Assignment(7)

1. Consider the following segment table (shown in Table 1)

Segment	Base	Limit
0	216	600
1	2300	20
2	90	100
3	1327	510

Table 1.

Calculate the physical addresses (in decimal) of the following logical addresses:

1.1 (0,430)

- Limit = 600
- Since 430<600, it is valid.
- Physical Address = 216+430=646

1.2 (1,20)

- Limit = 20
- Since 20≮20, it is invalid.
- Result: Trap (out of range).

1.3 (2,111)

- Limit = 100
- Since 111≮100,, it is invalid.
- Result: Trap (out of range).

1.4 (3,400)

- Limit = 510
- Since 400<510, it is valid.
- Physical Address = 1327+400=1727.

2. What is the purpose of paging the page tables? Describe the advantage(s) of an inverted page table over a direct page table.

Paging the page tables reduces memory usage by breaking large page tables into smaller pages. Instead of storing an entire page table in memory, only needed parts are loaded when required. This makes memory

management more efficient, especially for large address spaces. The advantages of an inverted page table over a direct page table include less memory usage, scales better , supports multiple processes, and slower lookups.

3. Consider a virtual memory system with a page size of 128-bit and a physical memory of 256 bytes. Each logical address has a 4-bit offset value. The page table is shown in Table 2.

Page no.	Frame address
0	11
1	6
2	10
⋮	⋮
14	7
15	2

Table 2.

Based on the page table, calculate the physical addresses (in decimal) of the following logical addresses: 1

3.1. 0116 = 0000 0001 = 11 x 8 + 1 = 89

3.2. 2B16 = 0010 1011 = 10× 8 + 11 = 91

3.3. 1A16 = 0001 1010 = 6×8 + 10 = 48 + 10 = **58**

3.4. FC16 = 1111 1100 = 2 x 8 + 12 = 28

4. (4 Points) “Any attempt by a program executing in user mode to access operating system memory or other users’ memory generates a trap to OS”. Briefly discuss how the OS would protect memory access during program execution.

The OS protects memory using base and limit registers, which define the valid memory range for a program. If a program in user mode tries to access OS memory or another program’s memory, the CPU generates a trap and stops execution. Privileged instructions, such as modifying base and limit registers, can only be executed in kernel mode, ensuring memory isolation and security.

5. What is address binding? Why is dynamic address binding preferable in a computer system?

Address binding assigns memory addresses to a program. A program starts as a file on disk but must be placed in RAM to execute. The system maps logical (virtual) addresses to physical addresses during this process.It can happen at compile time, load time, or execution time (dynamic binding).

Dynamic binding is preferable because it allows programs to move in memory during execution, improves memory utilization, supports multitasking, and enhances security by keeping physical addresses hidden.

6. Consider a computer with a 32-bit virtual address (logical address) and a 4K-bit page size. How many entries are required in a page table? If each page table entry requires 4 bytes, what is the total size of the page table?

A 32-bit address has 32 bits in total.

Page size is 4 KB = 4096 bytes, which is 2^{12} bytes, so the offset size is 12 bits.

Total number of pages = 2^{32} total bytes ÷ 2^{12} bytes per page = 2^{20} pages.

Each page table entry is 4 bytes, so the total page table size: $2^{20} \times 4 = 2^{22}$ bytes = 4 MB.

7. Given five memory holes of sizes 300 KB, 550 KB, 250 KB, 600 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit dynamic processes allocating algorithms allocate four processes of size 115 KB, 500 KB, 358 KB, and 200 KB (in order) into the given holes? Rank the algorithms by comparing the total hole space generated after their hole allocation.

First_Fit Allocation

Process	Process Size	Hold Index	Hole Allocated
0	115 KB	0	300 KB
1	500 KB	1	550 KB
2	358 KB	3	600 KB
3	200 KB	2	250 KB

Total Hole Space Left = 185 + 50 + 50 + 242 + 125 = 652 KB

Best_Fit Allocation

Process	Process Size	Hold Index	Hole Allocated
0	115 KB	0	125 KB
1	500 KB	1	550 KB
2	358 KB	3	600 KB
3	200 KB	2	600 KB

Total Hole Space Left = 300 + 50 + 250 + 42 + 10 = 652 KB

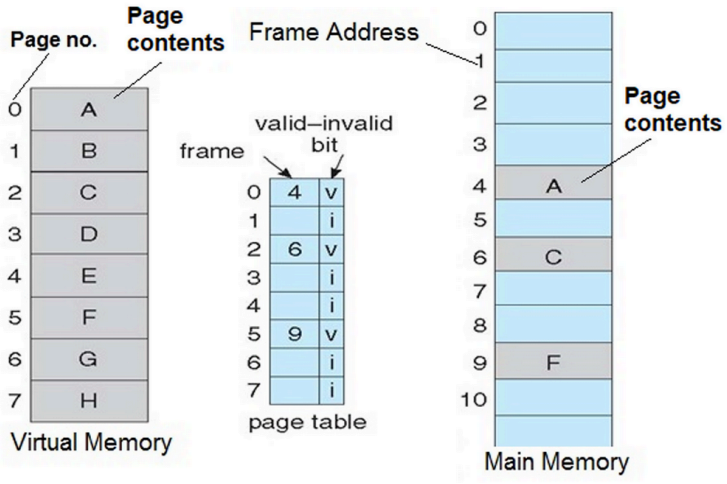
First_Fit Allocation

Process	Process Size	Hold Index	Hole Allocated
0	115 KB	0	600 KB
1	500 KB	1	550 KB
2	358 KB	3	600 KB
3	200 KB	2	300 KB

Total Hole Space Left = 100 + 50 + 250 + 127 + 125 = 652 KB

So rank-in-order = First-fit, Best-fit, Worst-fit

8.A virtual memory (VM) system is shown in Figure below. Based on the VM:



(a). Why does the page table need the valid-invalid bit?

The page table needs the valid-invalid bit because it tells whether a page is in main memory (valid) or not (invalid). If invalid, a page fault occurs, and the OS loads the page from disk.

(b). Assume page no. 4 is selected for the subsequent execution. Briefly describe how the OS handles the situation.

Page 4 is invalid, so a page fault happens. The OS:

1. Find a free frame (if available).
2. Uses a page replacement algorithm (if no free frame).
3. Loads page 4 from disk into memory.
4. Updates the page table to mark it as valid.
5. Restarts the instruction that caused the fault.

(c). Is any page replacement algorithm needed to solve the issue in (b)? Why?

Yes. A page‑replacement algorithm is needed here, because all frames allocated to the process are full. The OS must decide which resident page (0, 2, or 5) to remove in order to bring page 4 into memory.

(d) Suppose there are 1M pages in the VM. Show the row size (no. of rows) of the page table.

If there are 1M pages in the VM, there will be 1,048,576(2^{20}) rows in the page table.

6530262_MinMyintMohSoe_OS_Assignment(8)

1. Why do computer systems need address binding during a program execution?

Address binding maps a program's logical addresses to actual physical memory locations. This lets programs be loaded anywhere in memory and supports protection and sharing.

2. Briefly describe the importance of dynamic address binding in a computer system.
Dynamic address binding allows this mapping to happen at runtime. It makes moving programs in memory easier and improves memory use by adjusting to the program's needs as they run.

3. What is dynamic loading? Briefly describe how the dynamic loading is related to demand paging.
Dynamic loading means only loading parts of a program into memory when they are needed rather than loading everything at once. It is similar to demand paging, where pages are loaded on demand, reducing memory use and speeding up program startup.

4. Describe the term page fault. What happens if there is no free memory frame during a page fault (along with your answer, you should cover the details of the page table, including its valid-invalid control bit)?

A page fault occurs when a program accesses a page that is not in main memory. When this happens, the OS checks the page table; the page's valid-invalid bit will be set to "invalid" if it isn't in memory. If there's no free memory frame, the OS uses a page replacement algorithm to choose a victim frame, writes its contents back to disk if needed, and loads the required page into that frame before updating the page table.

5. (3 Points) Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

A page fault happens when a process accesses a page not in memory. When this occurs, the OS:

- Checks if the access is valid.
- Finds a free frame or chooses one to replace.
- Loads the required page from disk into memory.
- Sets the page table's valid-invalid bit to "valid."
- Restarts the interrupted instruction.

6. Briefly describe at which situation the page fault handler routine of an OS needs a page replacement algorithm. Assume that a page reference string of a user process is "12 3 2 4 2 5 3 4 5 3 2" and three fixed memory-frames are allocated for this process. Show the performance of optimal, FIFO, and LRU page replacement algorithms regarding their page fault.

The page fault handler needs a page replacement algorithm when no free frame is available to load the demanded page. For a reference string "12 3 2 4 2 5 3 4 5 3 2" of a user process and three fixed frames:

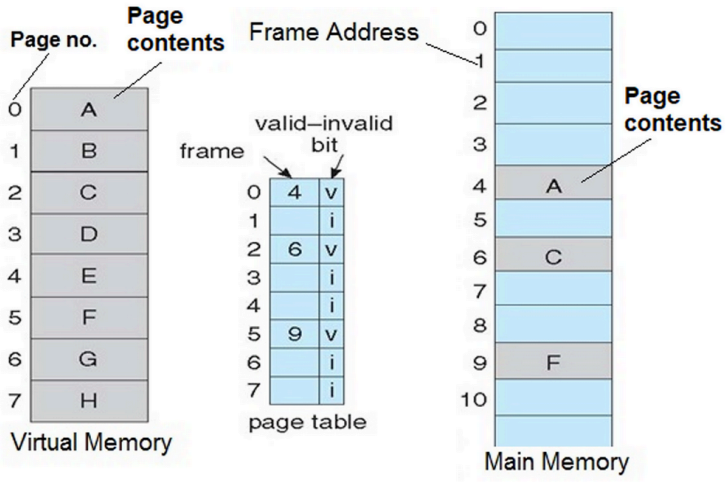
- Optimal: 6 faults
- FIFO: 7 faults
- LRU: 8 faults

7. Assume that a page reference string of a user process is “2 3 2 1 4 2 5 3 4 5 3 2” and three fixed memory frames are allocated for this process. Show the performance of optimal, FIFO, and LRU page replacement algorithms regarding their page fault.

With the reference string “2 3 2 1 4 2 5 3 4 5 3 2” of a user process and three fixed frames, the performance is:

- Optimal: 6 faults
- FIFO: 9 faults
- LRU: 8 faults

8.A virtual memory (VM) system is shown in Figure below. Based on the VM:



(a). Why does the page table need the valid-invalid bit?

The page table needs the valid-invalid bit because it tells whether a page is in main memory (valid) or not (invalid). If invalid, a page fault occurs, and the OS loads the page from disk.

(b). Assume page no. 4 is selected for the subsequent execution. Briefly describe how the OS handles the situation.

Page 4 is invalid, so a page fault happens. The OS:

1. Find a free frame (if available).
2. Uses a page replacement algorithm (if no free frame).
3. Loads page 4 from disk into memory.
4. Updates the page table to mark it as valid.
5. Restarts the instruction that caused the fault.

(c). Is any page replacement algorithm needed to solve the issue in (b)? Why?

Yes. A page‑replacement algorithm is needed here, because all frames allocated to the process are full. The OS must decide which resident page (0, 2, or 5) to remove in order to bring page 4 into memory.

(d) Suppose there are 1M pages in the VM. Show the row size (no. of rows) of the page table.

If there are 1M pages in the VM, there will be 1,048,576(2^{20}) rows in the page table.

6530262_MinMyintMohSoe_OS_Assigment(9)

1. What are the advantages and disadvantages of supporting memory mapped I/O to device control registers?
Memory-mapped I/O allows memory and device registers to share the same instructions, simplifying programming and speeding up transfers. However, it consumes memory address space and requires careful handling to avoid caching and security issues.

2. Describe I/O Polling. Why is I/O polling inefficient? Suggests solution(s) to overcome the inefficiency of the I/O polling operation.
I/O polling involves repeatedly checking a device's status, wasting CPU cycles that could be used for other tasks. Interrupt-driven I/O or DMA can improve efficiency by letting the CPU focus elsewhere until the device is ready.

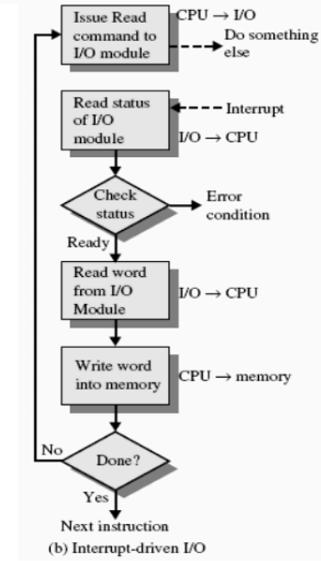
3. Briefly describe the Direct Memory Access (DMA) I/O operation. Point out the advantages of the DMA operation over memory-mapped and direct I/O operations.
Direct Memory Access (DMA) transfers data between devices and memory without CPU intervention. This reduces overhead, speeds up large transfers, and frees the CPU for other tasks.

4. Describe various circumstances under which nonblocking I/O should be used. Why not just implement non-blocking I/O and keep processes busy until their devices are ready?
Nonblocking I/O helps applications stay responsive by letting processes check for data without pausing execution. Busy-waiting, however, wastes CPU resources and reduces overall performance.

5. What are the advantages and disadvantages of supporting memory-mapped I/O to device control registers?
Memory-mapped I/O allows memory and device registers to share the same instructions, simplifying programming and speeding up transfers. However, it consumes memory address space and requires careful handling to avoid caching and security issues.

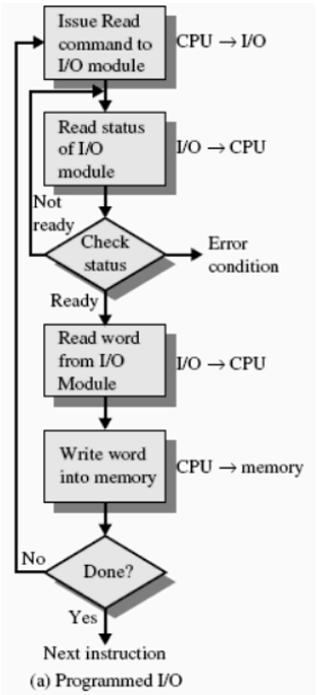
6. What are the advantages and disadvantages of supporting memory-mapped I/O to device control registers?
Memory-mapped I/O allows memory and device registers to share the same instructions, simplifying programming and speeding up transfers. However, it consumes memory address space and requires careful handling to avoid caching and security issues.

7. Identify the nature of I/O operation from the given flow chart:



The flowchart illustrates an **interrupt-driven I/O** process where the CPU issues a command, performs other work, and then is interrupted when the I/O module is ready.

8. Identify the nature of the I/O operation from the given flow chart:



The flowchart represents programmed I/O (polling). The CPU continuously checks the device's status rather than being interrupted.