Chapter 3: Processes
Processes are a core concept in OS, representing the active state of programs.

1. **Process Concept**:
   - A process is an instance of a running program, containing its code, data, and state (program counter, stack, heap).
   - Processes cycle through states—**New, Ready, Running, Waiting, Terminated**—depending on the OS's resource allocation.

2. **Process Control Block (PCB)**:
   - The PCB stores essential process information, such as process ID, state, CPU registers, memory pointers, and accounting data, to track and manage processes.

3. **Process Scheduling**:
   - The **scheduler** decides which process will execute next based on priority and state. **Queues** (ready, waiting) are used to manage the processes based on their execution stage.

4. **Inter-Process Communication (IPC)**:
   - **Message Passing** and **Shared Memory** are two primary IPC methods. Message passing uses direct or indirect channels (ports/mailboxes), while shared memory lets processes communicate via a common space in memory, requiring synchronization.

5. **Client-Server Communication**:
   - Processes on distributed systems communicate through **sockets, RPCs (Remote Procedure Calls)**, or **pipes** to share data or request services across different machines.

Chapter 4: Threads
Threads, or lightweight processes, are subdivisions of a process that allow parallel execution of tasks within the same application.

1. **Thread Overview**:
   - Threads share the same data and resources of the parent process, making context switching between threads more efficient than between separate processes.

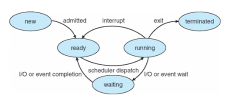
2. **Advantages of Multithreading**:
   - **Responsiveness**: Improves application interactivity, as different tasks can execute in parallel.
   - **Resource Sharing**: Threads within a process share resources, leading to lower memory use.
   - **Economy**: Creating and managing threads require fewer resources than processes.
   - **Scalability**: Threads enhance the performance of multiprocessor systems by allowing multiple tasks to run simultaneously.

3. **Multithreading Models**:
   - **Many-to-One**: Multiple user threads map to a single kernel thread, which is simple but can create bottlenecks.
   - **One-to-One**: Each user thread maps to a kernel thread, allowing better concurrency but with higher overhead.
   - **Many-to-Many**: Multiple user threads map to multiple kernel threads, balancing flexibility with efficient resource use.

4. **Thread Libraries**:
   - Common libraries include **POSIX Pthreads**, **Windows threads**, and **Java threads**. These provide thread management and synchronization functions.

5. **Thread Synchronization and Management**:
   - **Synchronization** tools like **mutexes** and **semaphores** are used to prevent race conditions when threads access shared resources.
   - **Thread Pools** improve performance by reusing threads, and **signal handling** manages interruptions across threads.
   - **Thread Cancellation** methods help safely terminate threads, essential for resource cleanup.
---
1. Process Concepts
- **Process Definition**: A process is an executing instance of a program, containing the program code and its current activity. Processes are fundamental to multitasking.
- **Components of a Process**:
  - **Code**: Also called the text section, contains the executable code.
  - **Data**: Global variables and static variables.
  - **Heap**: For dynamically allocated memory (grows upwards).
  - **Stack**: Stores function parameters, return addresses, and local variables (grows downwards).

2. Process States
- **New**: Process is being created.
- **Running**: Instructions are being executed on the CPU.
- **Waiting**: Process is waiting for an event (e.g., I/O completion).
- **Ready**: Process is prepared to run when CPU is available.
- **Terminated**: Process has completed execution.



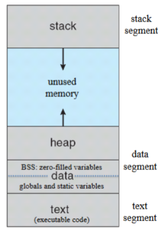
New Process ==> Ready
❍ Allocate resources necessary to run
❍ Place process on process queue (usually at end)
Ready ==> Running
❍ Process is at the head of process queue
❍ Process is scheduled onto an available processor
Running ==> Ready
❍ Process is interrupted
◆ usually by a timer interrupt
❍ Process could still run, in that it is not waiting something
❍ Placed back on the process queue

State Transitions: Page Fault Handling
Running ==> Waiting
❍ Either something exceptional happened that caused an interrupt to occur (e.g., page fault exception) ...
❍ ... or the process needs to wait on some action (e.g., it made a system call or requested I/O)
❍ Process must wait for whatever event happened to be serviced
Waiting ==> Ready
❍ Event has been satisfied so that
the process can return to run
❍ Put it on the process queue
Ready ==> Running
❍ As before...

3. Process Control Block (PCB)
- The PCB is a data structure in the OS kernel that contains essential information for managing processes, including:
  - **Process State**: Current state (New, Ready, Running, Waiting, Terminated).
  - **Program Counter**: The address of the next instruction to execute.
  - **CPU Registers**: Contents saved during context switches.
  - **Memory Management Info**: Base and limit registers, page tables.
  - **I/O Status**: List of I/O devices allocated to the process, open files.
  - **Scheduling Info**: Process priority, scheduling queue pointers.
  - **Accounting Info**: CPU usage, clock time, etc.

4. Process Creation and Termination
- **fork()**: Creates a new process by duplicating the parent's address space. Child process has its own unique Process ID (PID).
- **exec()**: Replaces the process's memory space with a new program. Often used after `fork()` for child process to run a different program.
- **wait()**: Makes a parent process wait until its child process completes.
- **exit()**: Terminates a process, releasing its resources.
- **Process Hierarchy**: OSs maintain a hierarchical tree of parent-child relationships (e.g., Unix init process is at the root).

5. Process Memory Layout
- **Address Space Segmentation** :the process address space is a memory region allocated by the OS where a process stores its code, data, and runtime stack.
  - **Text Segment**: Contains the code (read-only).
  - **Data Segment**: Holds initialized and uninitialized global and static variables.
  - **Stack Segment**: Used for function calls, parameters, return addresses.
  - **Heap Segment**: Memory for dynamic allocations, managed with `malloc()` and `free()` in C/C++.



If there are Multiple Threads:
Text, Data, and Heap segments are shared among threads in a process.
Stack: Each thread has its own stack to keep track of its function calls and local variables.
Purpose of Interrupts and Difference from Traps
Interrupts: Signals from hardware or software indicating an event that needs immediate attention. They pause current execution, allowing the OS to handle the interrupt and then resume.
Traps: Special types of software-generated interrupts often triggered intentionally (e.g., through system calls) by a user program to request OS services.
Can Traps be Generated by User Programs?
Yes, traps can be intentionally generated by a user program for actions like requesting file operations or accessing system resources via system calls.

6. Interprocess Communication (IPC)
- **Definition**: IPC mechanisms allow processes to communicate and synchronize actions without sharing a common address space.
- **Models of IPC**:
  - **Message Passing**: Processes communicate by sending messages.
    - **Direct**: Processes send messages directly to each other.
    - **Indirect**: Messages are sent via mailboxes or message queues.
  - **Shared Memory**: Processes share a memory region for data exchange, requiring synchronization
    - Allows multiple processes to access a common memory space.
    - Requires synchronization mechanisms (e.g., semaphores) to avoid conflicts.

7. IPC Mechanisms (Types)
- **Pipes**: Unidirectional data channels managed by the OS. Typically used for parent-child communication.
- **Sockets**: Network communication endpoints used for inter-process communication over a network (TCP/UDP).
  - **Stream Sockets (TCP)**: Reliable, connection-oriented.
  - **Datagram Sockets (UDP)**: Unreliable, connectionless.
  - **Domain Sockets**: Used for communication between processes on the same host.
- **Remote Procedure Call (RPC)**: Allows programs to invoke procedures on remote systems as if they were local.
  - **Process**:
    - **Marshalling**: Encoding parameters and sending them to the remote system.
    - **Stub**: The server "unmarshalls" data and performs the call, then sends the result back.
  - **Remote Method Invocation (RMI)**: Java-based RPC, where methods are invoked on remote objects.
- **Shared Memory via mmap()**:
  - **mmap()**: Maps files or devices into memory, making them accessible across processes.
  - **shmget() and shmat()**: System calls for managing shared memory segments in Unix/Linux.
- **Pipes (OS-managed Shared Memory)**:
  - Commands in Unix: `prog1 | prog2` links the output of `prog1` to the input of `prog2`.
  - System calls: `pipe()`, `dup()`, `popen()`.

8. IPC Synchronization
- **Blocking (Synchronous)**:
  - **Blocking Send**: Sender waits until message is received.
  - **Blocking Receive**: Receiver waits until a message is available.
- **Non-blocking (Asynchronous)**:
  - **Non-blocking Send**: Sender sends message and continues execution.
  - **Non-blocking Receive**: Receiver receives either a message or a null if none is available.
- **Message Queue Options**:
  - **Synchronous Messaging**: Requires both sender and receiver to be ready at the same time.
  - **Asynchronous Messaging**: Sender and receiver do not need to coordinate timing.

9. Scheduling and Process Management
- **Schedulers**:
  - **Short-Term Scheduler**: Chooses next process to run from the ready queue (fast, frequent).
  - **Medium-Term Scheduler**: Swaps processes in and out of main memory (manages level of multiprogramming).
  - **Long-Term Scheduler**: Decides which processes to load into the system from job pool (controls degree of concurrency).
- **Queues**:
  - **Job Queue**: All processes in the system.
  - **Ready Queue**: Processes ready and waiting to run.
  - **Device Queue**: Processes waiting for I/O devices.

10. Context Switching
- **Context Switch**: The act of saving a process's state and loading another's.
  - **Steps**:
    - Save current process state into its PCB.
    - Load the state of the new process from its PCB.
   -Update memory maps if needed.
   -Switch execution to the new thread

  - **Performance Considerations**:
    - Context switching is costly as it requires saving and loading state, which does not directly contribute to the execution of the user's task.
    - **Hardware Optimizations**: Multiple register sets or fast switching mechanisms can reduce the cost.

11. Concurrency and Process Communication in Client-Server Model
- **Client-Server Model**: Servers often fork new processes for each client to provide concurrent service.
  - **Forking**: Server process forks a child to handle each client request, isolating clients from each other.
  - **Server Workflow**:
    - Parent process waits for a connection request.
    - Upon connection, the server forks a child.
    - The child handles the client connection, while the parent returns to listen for new clients.
- **Concurrency**:
  - Isolated, concurrent child processes allow the server to serve multiple clients simultaneously.

This lecture on **CPU Scheduling** discusses how operating systems manage CPU time allocation for processes to optimize performance, responsiveness, and resource utilization.
Key Topics:
1. **Types of Scheduling**:

  - **Long-Term Scheduling**: Decides which jobs enter the system.
  - **Medium-Term Scheduling**: Manages the degree of multiprogramming.
  - **Short-Term Scheduling**: Determines which process runs next on the CPU.
  - **I/O Scheduling**: Manages I/O requests for devices.

2. **Scheduling Criteria**:
  - **CPU Utilization**: Maximize CPU use.
  - **Throughput**: Maximize jobs completed per unit of time.
  - **Turnaround Time**: Minimize the time from job submission to completion.
  - **Waiting Time**: Reduce time spent in the ready queue.
  - **Response Time**: Improve responsiveness for interactive tasks.
  - **Fairness**: Ensure equal CPU access for all processes.

3. **Scheduling Algorithms**:
  - **First-Come, First-Served (FCFS)**: Simple and fair but can lead to long wait times.
  - **Shortest Job First (SJF)**: Minimizes average waiting time but requires knowledge of future burst times.
    - **Preemptive SJF** (Shortest Remaining Time First, SRTF) preempts if a shorter job arrives.
  - **Priority Scheduling**: Prioritizes processes, but can cause starvation, mitigated by aging.
  - **Round Robin (RR)**: Assigns time slices to each process, balancing responsiveness and overhead.
  - **Multilevel and Feedback Queues**: Use multiple queues for different priority levels or dynamically adjust priorities.

4. **Dispatcher and Preemption**:
  - **Dispatcher**: Switches context between processes.
  - **Preemptive Scheduling**: Allows the OS to switch processes mid-execution to improve responsiveness or efficiency.

5. **Algorithm Evaluation**:
  - **Load Balancing**: Essential in multiprocessor systems to distribute tasks evenly across CPUs.
  - **Real-Time Scheduling**: Ensures time constraints for critical tasks, with hard real-time needing strict timing guarantees.

6. **Thread Scheduling**:
  - **User vs. Kernel Scheduling**: User-level threads are managed by the process, while kernel-level threads compete at the system level.
  - **Processor Affinity**: Keeps threads on the same processor to improve cache efficiency.
Threading Models
1:1 Model: Each user thread maps to a kernel thread, allowing concurrency at the OS level. It's preferred for greater parallelism and better OS-level control.
M:1 Model: Multiple user threads map to a single kernel thread, which is simpler but lacks true concurrency, as only one thread can be in the running state at a time.

7. **Multicore Processors and Real-Time Scheduling**:
  - Modern multicore CPUs support hardware multithreading, allowing parallelism within and across cores.
  - Real-time scheduling emphasizes meeting timing requirements for critical processes.

Summary:
Scheduling is fundamental for efficient OS operation. Techniques vary from simple FCFS to more complex multilevel feedback queues and real-time scheduling, all optimized based on specific system needs. Next topics include **Concurrency and Synchronization**.

Here's a study guide based on **Lecture 5: Threads** from your Operating Systems course. This guide summarizes key concepts, terms, and comparisons for exam preparation.

1. **Processes vs. Threads**
  - **Process**: An independent program with its own memory space and resources. Context switching between processes is costly.
  - **Thread**: A "thread of execution" within a process. Threads share the process's memory and resources but have their own program counters and stack. Less overhead than processes.

2. **Advantages of Threads**
  - **Responsiveness**: Threads can handle different parts of a task simultaneously, improving responsiveness (e.g., GUI remains responsive while background tasks run).
  - **Resource Sharing**: Threads within a process share resources like memory, making data sharing more efficient.
  - **Economy**: Threads are lightweight and less resource-intensive than processes.
  - **Utilization of Multiprocessors**: Threads can run on different CPUs in parallel, leveraging multi-core systems.

3. **Creating and Managing Threads in C**
  - Use `pthread_create()` to start a thread and `pthread_join()` to wait for it to finish.
  - Threads share global variables and heap but maintain individual stacks.

4. **Thread Models**
  - **Many-to-One**: Multiple user-level threads map to one kernel thread (used by libraries, limited by single-thread blocking).
  - **One-to-One**: Each user-level thread has a corresponding kernel thread (used by Linux, better parallelism but more overhead).
  - **Many-to-Many**: Multiple user threads map to multiple kernel threads, balancing parallelism and efficiency (scalable but complex to manage).

5. **POSIX Threads (Pthreads)**
  - Pthreads API is a specification for thread management in UNIX-like systems.
  - Key functions: `pthread_create()`, `pthread_exit()`, `pthread_join()`, `pthread_cancel()`, `pthread_self()`.
  - Pthreads allow thread creation and synchronization, enabling multi-threaded applications.

6. **Thread Pooling**
  - **Thread Pool**: Pre-created threads that handle tasks, reducing thread creation overhead. Once a thread completes a task, it returns to the pool to await new tasks.

7. **Concurrency and Synchronization**
  - **Concurrency**: Multiple threads run "at the same time," essential for tasks like client-server applications where each client request could be handled by a separate thread.
  - **Synchronization**: Mechanisms to prevent data races and ensure threads do not interfere with each other in shared memory.

8. **Signal Handling in Threads**
  - **Synchronous Signals**: Generated by the process (e.g., divide by zero).
  - **Asynchronous Signals**: Generated externally and can disrupt multiple threads.
  - Important to manage carefully to avoid conflicts across threads.

9. **Thread Cancellation**
  - **Synchronous Cancellation**: A thread cancels only at specific points where it is safe to stop.
  - **Asynchronous Cancellation**: Immediate termination of a thread using `pthread_cancel()`, which can be unsafe if the thread is in the middle of a task.

10. **Performance Comparison: Threads vs. Processes**
  - Creating, switching, and terminating threads is generally faster than processes due to less overhead in managing separate memory spaces.

11. **Kernel vs. User-Level Threads**
  - **Kernel-Level Threads**: Managed by the OS and require kernel support; enable better scheduling but with higher overhead.
  - **User-Level Threads**: Managed by a user-space library, allowing faster switching but limited by the OS's handling of only one kernel thread per process in many cases.

12. **Scheduler Activation**
  - **Scheduler Activation**: Mechanism where the kernel informs a process when its threads are blocked, allowing the process to decide if another thread should be scheduled.

Lecture 7: Synchronization

1. **Concurrency and Synchronization**
  - **Concurrency**: The ability to have multiple threads or processes executing logically at the same time, crucial for multiprogramming and operating system design.
  - **Synchronization**: Mechanisms to ensure consistency of shared resources among concurrent processes to prevent issues like race conditions.

2. **Critical-Section Problem**
  - **Critical Section (CS)**: Part of code where processes access shared resources.
  - **Requirements for CS solutions**:
    - **Mutual Exclusion**: Only one process can be in the CS at a time.
    - **Progress**: If no process is in the CS, any process that wishes to enter should be able to.
    - **Bounded Waiting**: There's a limit on how long a process waits to enter the CS.
  - **Solutions**: Include disabling interrupts, busy-waiting with locks, and blocking mechanisms.

3. **Synchronization Techniques**
  - **Peterson's Solution**: For two processes using flags and a turn variable to ensure mutual exclusion, progress, and bounded waiting.
  - **Bakery Algorithm**: Generalized for multiple processes using "tickets" to enter the CS in a specific order.
  - **Hardware Support**: Uses atomic instructions like `test_and_set` and `compare_and_swap` to implement locking mechanisms.

4. **Mutex Locks and Spinlocks**
  - **Mutex**: Basic locking mechanism for mutual exclusion. Uses `acquire()` and `release()` functions, often leading to spinlocks, which involve busy-waiting.
  - **Pthread Mutex**: POSIX library provides functions for thread synchronization with mutex locks.

5. **Semaphores**
  - **Binary Semaphore**: Acts like a mutex with a value of 0 or 1.
  - **Counting Semaphore**: Can take any integer value to manage multiple resources.
  - **Operations**: `wait()` and `signal()` to manage access to resources.
  - **POSIX Semaphores**: Support for managing synchronization in POSIX-compliant systems.

6. **Classical Synchronization Problems**
  - **Bounded-Buffer (Producer-Consumer)**: Uses semaphores to manage items produced and consumed within fixed buffer slots.
  - **Readers-Writers Problem**: Manages access to a shared resource where multiple readers can read concurrently, but only one writer can write.
  - **Dining Philosophers**: Models resource contention and deadlock avoidance by controlling philosophers' access to forks.

Lecture 8: Deadlocks

1. **Deadlock Characterization**
  - **Deadlock**: A situation where processes are blocked, each waiting for resources held by others, preventing any from proceeding.
  - **Necessary Conditions for Deadlock**:
    - **Mutual Exclusion**: Only one process can hold a resource at a time.
    - **Hold and Wait**: Processes hold resources while waiting for others.
    - **No Preemption**: Resources can only be released voluntarily.
    - **Circular Wait**: A closed chain exists where each process holds resources the next process needs.

2. **Resource Allocation Graph (RAG)**
  - Graphical method to visualize resource allocation.
  - **Cycle Detection**: If there is a cycle in a RAG, there may be deadlock (definite if each resource has only one instance).

3. **Deadlock Prevention**
  - **Strategies**:
    - **Mutual Exclusion**: Only for non-sharable resources.
    - **Hold and Wait**: Require processes to request all resources at once.
    - **No Preemption**: Allow preemption of resources if a process can't get all needed.
    - **Circular Wait**: Impose a fixed order on resource requests.

4. **Deadlock Avoidance**
  - **Safe State**: A sequence of processes can complete without deadlock.
  - **Banker's Algorithm**: Allocates resources only if the system remains in a safe state, suitable for multiple instances of resources.

5. **Deadlock Detection and Recovery**
  - **Detection**: Uses algorithms to detect cycles in the wait-for graph.
  - **Recovery**:
    - **Process Termination**: Abort one or more processes to break the deadlock.
    - **Resource Preemption**: Temporarily take resources from processes to resolve the deadlock.

6. **Banker's Algorithm for Resource Allocation**
  - **Data Structures**:
    - `Available`: Tracks free resources.
    - `Max`: Maximum demand of each process.
    - `Allocation`: Current allocation.
    - `Need`: Remaining resources needed by each process (`Max - Allocation`).
  - **Safety and Request Algorithm**: Ensures resources are only allocated if it leaves the system in a safe state.

Questions:
1. **What is the CPU-I/O Burst Cycle and Why is it Relevant to Processor Scheduling?**
  The **CPU-I/O burst cycle** is the alternating pattern of CPU and I/O operations in a program's execution. During a **CPU burst**, the process executes instructions and uses the CPU. This is followed by an **I/O burst**, where the process waits for an I/O operation (like reading from or writing to a disk).
  This cycle is important for **processor scheduling** because understanding the burst pattern helps in assigning the CPU to processes efficiently. For instance, scheduling algorithms can prioritize CPU-bound processes (longer CPU bursts) differently from I/O-bound processes (frequent I/O bursts), ensuring a balanced workload and reduced idle time for both the CPU and I/O devices. This maximizes overall system throughput and minimizes process waiting times.

2. **What is the Difference Between Turnaround Time and Response Time with Respect to CPU Scheduling?**

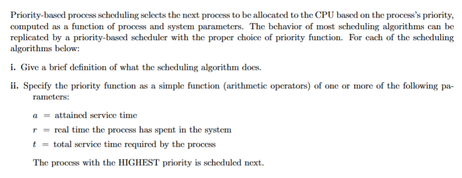
  - **Turnaround Time** is the total time taken from the submission of a process to its completion. It includes all phases: waiting time, CPU execution time, and any I/O time.
  - **Response Time** is the time from the submission of a process until it produces its first response (not completion). It is a measure of how quickly a process begins to execute after submission, which is particularly important in interactive systems.
  In short:
  - **Turnaround Time** = Total time to complete a process.
  - **Response Time** = Time until the process first starts responding.
---
3. **What is the Quantum in Round-Robin Scheduling? How Does It Get Implemented? What Happens in Round-Robin Scheduling as the Quantum Increases? What Happens in Round-Robin Scheduling as the Quantum Decreases?**

  - **Quantum** in **Round-Robin (RR) scheduling** is the fixed time slice each process is allowed to run on the CPU before it is preempted and placed at the end of the ready queue if it hasn't finished execution. This approach is particularly useful in time-sharing systems where fairness and responsiveness are important.

  - **Implementation**: The quantum is typically enforced by a hardware timer that generates an interrupt at the end of each time slice. When the timer triggers, the operating system preempts the running process and schedules the next one in the queue.

  - **If the quantum increases**:
    - Each process runs for a longer period before being preempted. This can reduce the overhead of context switching, which is beneficial for CPU-bound processes.
    - However, as the quantum gets too large, RR starts behaving like First-Come-First-Serve (FCFS), increasing wait times for other processes and lowering system responsiveness.

  - **If the quantum decreases**:
    - Processes are preempted more frequently, which can improve responsiveness, especially for interactive processes.
    - However, frequent context switching can increase overhead, reducing CPU efficiency and slowing down the overall throughput.


Priority-based process scheduling selects the next process to be allocated to the CPU based on the process's priority, computed as a function of process and system parameters. The behavior of most scheduling algorithms can be replicated by a priority-based scheduler with the proper choice of priority function. For each of the scheduling algorithms below:

i. Give a brief definition of what the scheduling algorithm does.

ii. Specify the priority function as a simple function (arithmetic operators) of one or more of the following parameters:

a = attained service time

r = real time the process has spent in the system

t = total service time required by the process

The process with the HIGHEST priority is scheduled next.

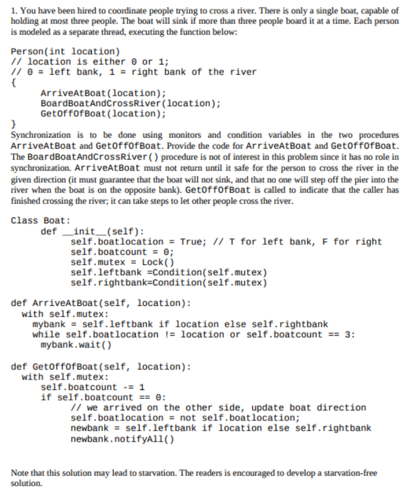
FIFO (First In / First Out)
- **Definition**: Processes are scheduled in the order they arrive. The first process to enter the queue is the first to execute.
- **Priority Function**: Priority = `-r` (or simply `r`).
  - Processes with the earliest arrival time (lowest `r`) have the highest priority.

LIFO (Last In / First Out)
- **Definition**: The most recently arrived process is given the CPU next. This approach is rarely used in practice as it can lead to starvation for older processes.
- **Priority Function**: Priority = `r`.
  - The most recently arrived process (highest `r`) has the highest priority.

SJF (Shortest Job First)
- **Definition**: Processes with the shortest total service time (`t`) are given the CPU first, minimizing the average waiting time. This scheduling method is non-preemptive.
- **Priority Function**: Priority = `-t` (or `1/t`).
  - Processes with the lowest total service time (`t`) have the highest priority.

RR (Round Robin)
- **Definition**: Each process is given a fixed time slice (quantum) in turn. When a process's time slice expires, it's placed at the end of the queue if it still has work to do. Round Robin is often used in time-sharing systems.
- **Priority Function**: Priority = `-a` (or simply `a`).
  - Since all processes are treated equally and are scheduled in a cycle, priority here can be based on the **attained service time (`a`)** or simply follow a cyclic order without using a strict priority function.

SRT (Shortest Remaining Time)
- **Definition**: The process with the shortest remaining service time is executed next. This is a preemptive version of SJF, meaning that if a new process with a shorter remaining time arrives, it can preempt the currently running process.
- **Priority Function**: Priority = `-(t - a)`.
  - Processes with the smallest remaining time (`t - a`) have the highest priority.


1. You have been hired to coordinate people trying to cross a river. There is only a single boat, capable of holding at most three people. The boat will sink if more than three people board it at a time. Each person is modeled as a separate thread, executing the function below:

```
Person(int location)
// location is either 0 or 1;
// 0 = left bank, 1 = right bank of the river
{
      ArriveAtBoat(location);
      BoardBoatAndCrossRiver(location);
      GetOffOfBoat(location);
}
```

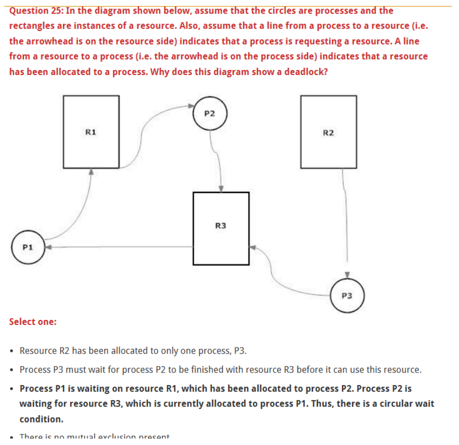
Synchronization is to be done using monitors and condition variables in the two procedures `ArriveAtBoat` and `GetOffOfBoat`. Provide the code for `ArriveAtBoat` and `GetOffOfBoat`. The `BoardBoatAndCrossRiver()` procedure is not of interest in this problem since it has no role in synchronization. `ArriveAtBoat` must not return until it safe for the person to cross the river in the given direction (it must guarantee that the boat will not sink, and that no one will step off the pier into the river when the boat is on the opposite bank). `GetOffOfBoat` is called to indicate that the caller has finished crossing the river; it can take steps to let other people cross the river.

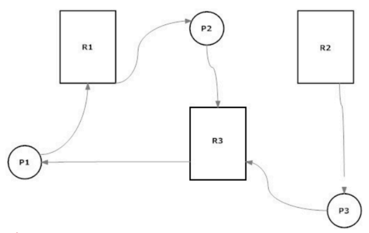
```
Class Boat:
      def __init__(self):
            self.boatlocation = True; // T for left bank, F for right
            self.boatcount = 0;
            self.mutex = Lock()
            self.leftbank =Condition(self.mutex)
            self.rightbank=Condition(self.mutex)

def ArriveAtBoat(self, location):
  with self.mutex:
    mybank = self.leftbank if location else self.rightbank
    while self.boatlocation != location or self.boatcount == 3:
      mybank.wait()

def GetOffOfBoat(self, location):
  with self.mutex:
      self.boatcount -= 1
      if self.boatcount == 0:
            // we arrived on the other side, update boat direction
            self.boatlocation = not self.boatlocation;
            newbank = self.leftbank if location else self.rightbank
            newbank.notifyAll()
```

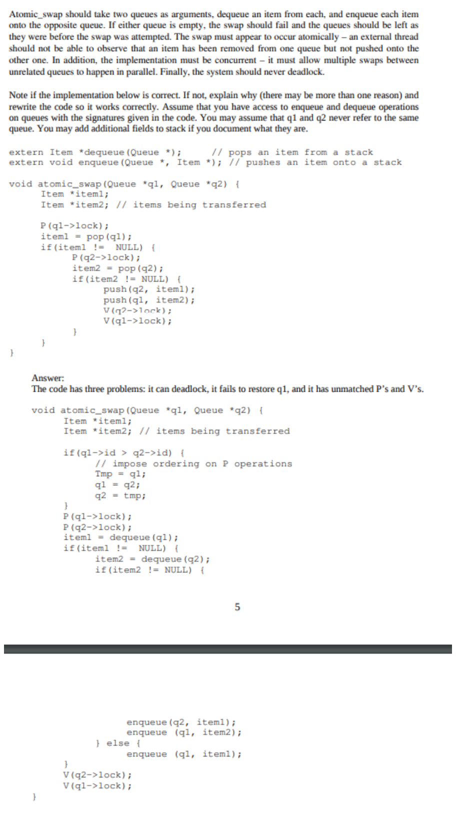
Note that this solution may lead to starvation. The readers is encouraged to develop a starvation-free solution.


Question 25: In the diagram shown below, assume that the circles are processes and the rectangles are instances of a resource. Also, assume that a line from a process to a resource (i.e. the arrowhead is on the resource side) indicates that a process is requesting a resource. A line from a resource to a process (i.e. the arrowhead is on the process side) indicates that a resource has been allocated to a process. Why does this diagram show a deadlock?



Select one:

- Resource R2 has been allocated to only one process, P3.
- Process P3 must wait for process P2 to be finished with resource R3 before it can use this resource.
- **Process P1 is waiting on resource R1, which has been allocated to process P2. Process P2 is waiting for resource R3, which is currently allocated to process P1. Thus, there is a circular wait condition.**
- There is no mutual exclusion present


Atomic_swap should take two queues as arguments, dequeue an item from each, and enqueue each item onto the opposite queue. If either queue is empty, the swap should fail and the queues should be left as they were before the swap was attempted. The swap must appear to occur atomically – an external thread should not be able to observe that an item has been removed from one queue but not pushed onto the other one. In addition, the implementation must be concurrent – it must allow multiple swaps between unrelated queues to happen in parallel. Finally, the system should never deadlock.

Note if the implementation below is correct. If not, explain why (there may be more than one reason) and rewrite the code so it works correctly. Assume that you have access to enqueue and dequeue operations on queues with the signatures given in the code. You may assume that q1 and q2 never refer to the same queue. You may add additional fields to stack if you document what they are.

```
extern Item *dequeue(Queue *);     // pops an item from a stack
extern void enqueue(Queue *, Item *); // pushes an item onto a stack

void atomic_swap(Queue *q1, Queue *q2) {
      Item *item1;
      Item *item2; // items being transferred

      P(q1->lock);
      item1 = pop(q1);
      if(item1 !=  NULL) {
            P(q2->lock);
            item2 = pop(q2);
            if(item2 != NULL) {
                  push(q2, item1);
                  push(q1, item2);
                  V(q2->lock);
                  V(q1->lock);
            }
      }
}
```

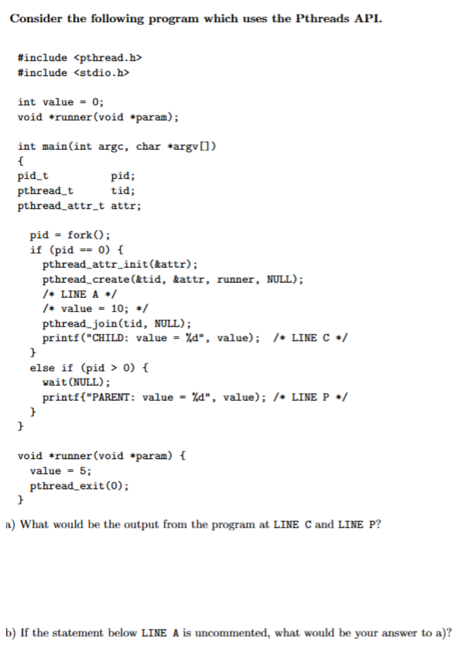
Answer:
The code has three problems: it can deadlock, it fails to restore q1, and it has unmatched P's and V's.

```
void atomic_swap(Queue *q1, Queue *q2) {
      Item *item1;
      Item *item2; // items being transferred

      if(q1->id > q2->id) {
            // impose ordering on P operations
            Tmp = q1;
            q1 = q2;
            q2 = tmp;
      }
      P(q1->lock);
      P(q2->lock);
      item1 = dequeue(q1);
      if(item1 !=  NULL) {
            item2 = dequeue(q2);
            if(item2 != NULL) {
```

5

```
                  enqueue(q2, item1);
                  enqueue (q1, item2);
            } else {
                  enqueue (q1, item1);
      }
      V(q2->lock);
      V(q1->lock);
}
```


Consider the following program which uses the Pthreads API.

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

int value = 0;
void *runner(void *param);

int main(int argc, char *argv[])
{
pid_t          pid;
pthread_t      tid;
pthread_attr_t attr;

  pid = fork();
  if (pid == 0) {
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, NULL);
    /* LINE A */
    /* value = 10; */
    pthread_join(tid, NULL);
    printf("CHILD: value = %d", value);  /* LINE C */
  }
  else if (pid > 0) {
    wait(NULL);
    printf{"PARENT: value = %d", value); /* LINE P */
  }
}

void *runner(void *param) {
  value = 5;
  pthread_exit(0);
}
```

a) What would be the output from the program at `LINE C` and `LINE P`?

b) If the statement below `LINE A` is uncommented, what would be your answer to a)?

1. **Global Variable**: `int value = 0;` is a global variable shared by all processes and threads created by `fork()` and `pthread_create()`.
2. **`fork()` Call**:
  - When `fork()` is called, it creates a new process, duplicating the current process. After `fork()`, there are two processes:
    - **Child process** (where `pid == 0`)
    - **Parent process** (where `pid > 0`)
3. **Child Process**:
  - In the child process (`pid == 0`):
    - A new thread is created with `pthread_create()`, which runs the `runner` function.
    - Inside `runner`, the global variable `value` is set to 5.
    - The child process then waits for the thread to complete with `pthread_join()` before printing `value` at **LINE C**.
4. **Parent Process**:
  - In the parent process (`pid > 0`):
    - The parent waits for the child process to complete with `wait(NULL)`.
    - After the child completes, the parent prints `value` at **LINE P**.

Part (a): Output at LINE C and LINE P

- **LINE C (Child Process)**:
  - The thread in the child process sets `value` to 5. So, when the child prints `value` at **LINE C**, the output will be:
   ```
   CHILD: value = 5
   ```
- **LINE P (Parent Process)**:
  - The parent process has a separate copy of `value` from the initial `fork()` (due to process memory separation). The `runner` function and the assignment in the child process do not affect the parent's copy of `value`.
  - Therefore, when the parent prints `value` at **LINE P**, it remains at its original value of 0:
   ```
   PARENT: value = 0   ```

### Part (b): If the statement `value = 10;` below LINE A is uncommented
- **LINE C (Child Process)**:
  - In this case, after the thread sets `value` to 5, the `value = 10;` statement in the child process overrides it.
  - Therefore, when the child prints `value` at **LINE C**, the output will be:
   ```
   CHILD: value = 10
   ```
- **LINE P (Parent Process)**:
  - The parent process is unaffected by changes to `value` in the child process and still retains its initial value of 0.
  - So, when the parent prints `value` at **LINE P**, it remains:
   ```
   PARENT: value = 0
   ```
   ```