OPERATING SYSTEMS

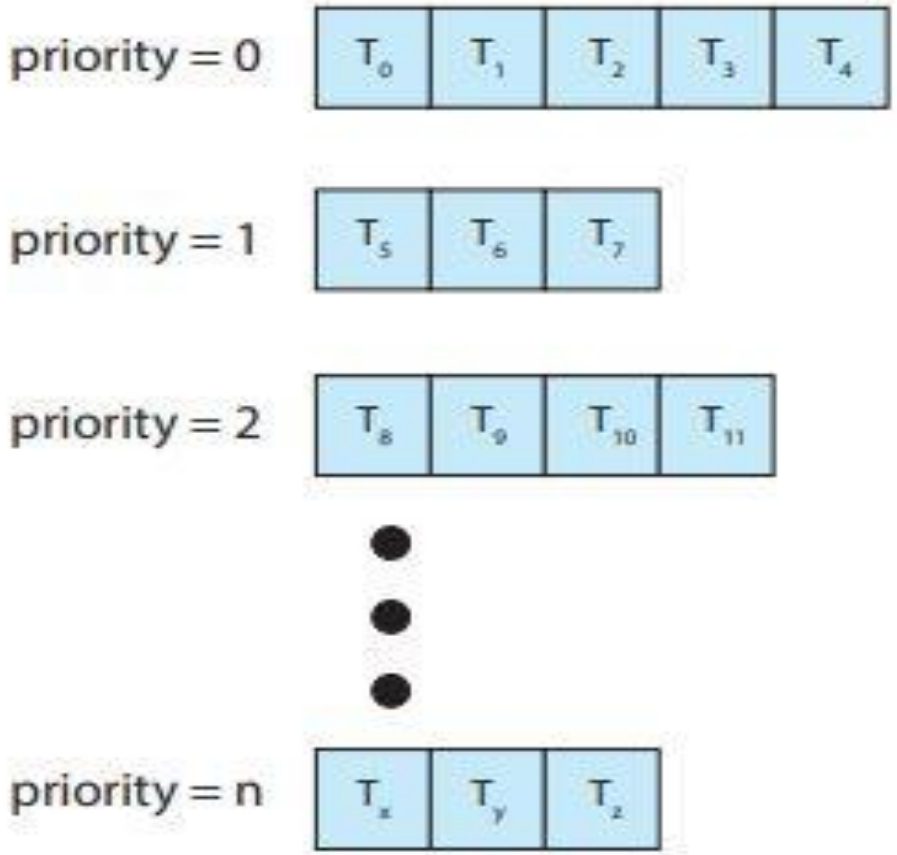
Razi Uddin

Lecture # 14

MULTILEVEL QUEUE SCHEDULING

- With both priority and round-robin scheduling, all processes may be placed in a single queue, and the scheduler then selects the process with the highest priority to run.
- Depending on how the queues are managed, an O(n) search may be necessary to determine the highest-priority process.
- In practice, it is often easier to have separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue.
- This approach—known as multilevel queue.

MULTILEVEL QUEUE SCHEDULING



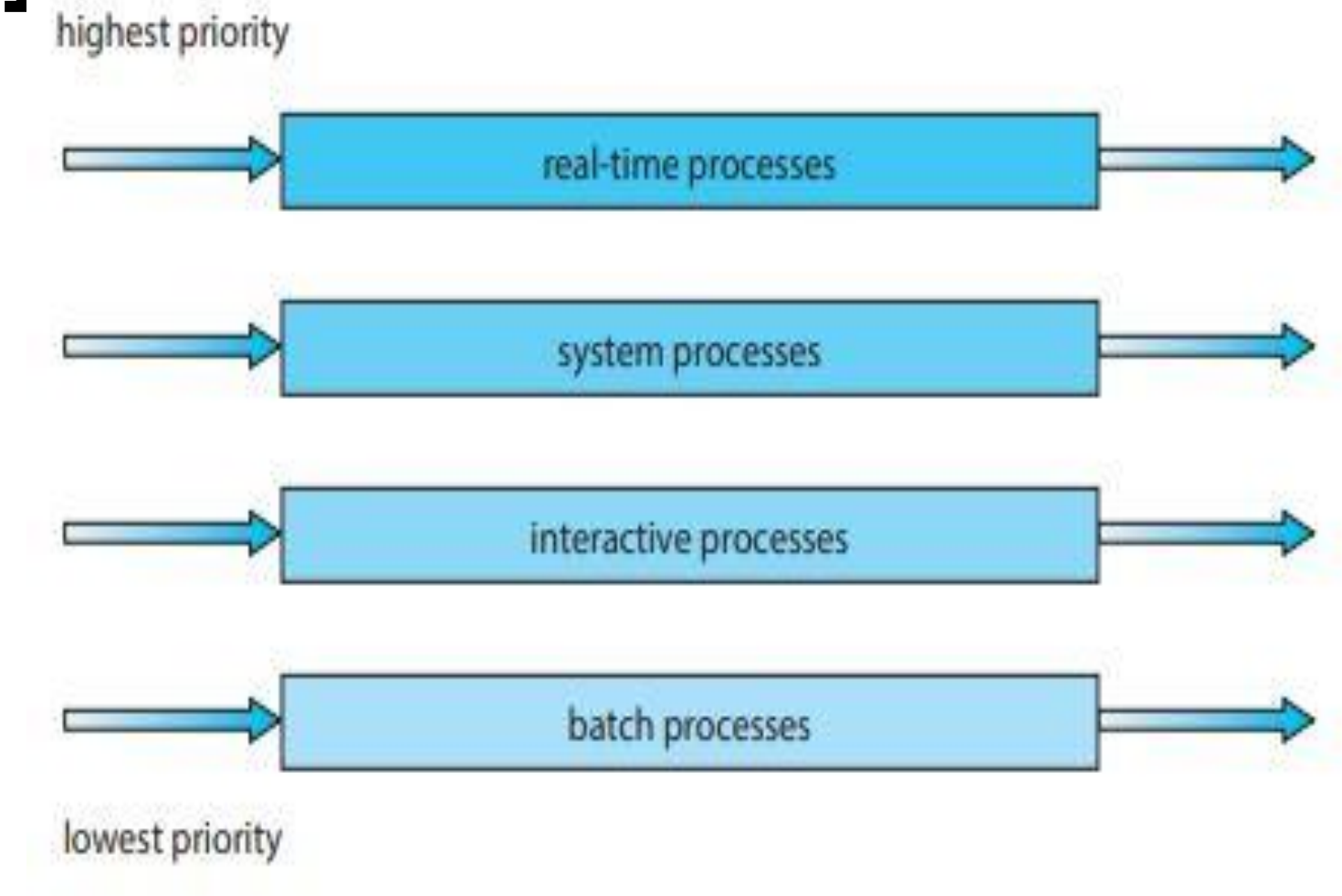
MULTILEVEL QUEUE SCHEDULING

- A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues.
- Each queue has its own priority and scheduling algorithm.
- Processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority or process type.
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling i.e., serve all from the foreground then from the background.
- Another possibility is to time-slice between queues.
- Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue, e.g., 80% to foreground in RR and 20% to background in FCFS.
- Scheduling across queues prevents starvation of processes in lower-priority queues.

MULTILEVEL QUEUE SCHEDULING

- A common division is made between foreground (interactive) processes and background (batch) processes.
- These two types of processes have different response-time requirements and so may have different scheduling needs.
- In addition, foreground processes may have priority (externally defined) over background processes.
- Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm.
- The foreground queue might be scheduled by an RR algorithm, for example, while the background queue is scheduled by an FCFS algorithm.

MULTILEVEL QUEUE SCHEDULING



MULTILEVEL FEEDBACK QUEUE SCHEDULING

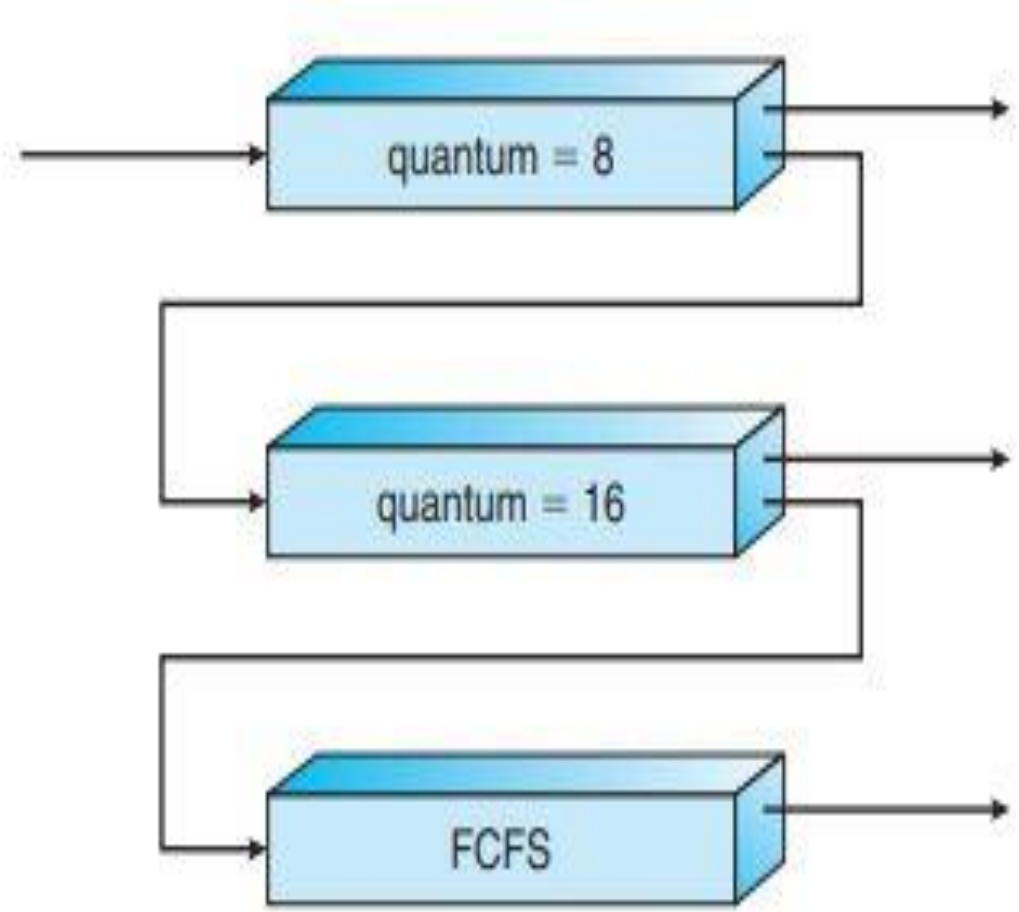
- It allows a process to move between queues.
- The idea is to separate processes with different CPU burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- This scheme leaves I/O bound and interactive processes in the higher-priority queues.
- Similarly, a process that waits too long in a lower-priority queue may be moved o a higher priority queue.
- This form of aging prevents starvation.

MULTILEVEL FEEDBACK QUEUE SCHEDULING

In general, a multi-level feedback queue scheduler is defined by the following parameters:

- Number of queues
- Scheduling algorithm for each queue
- Method used to determine when to upgrade a process to the higher priority queue
- Method used to determine when to demote a process
- Method used to determine which queue a process enters when it needs service

MULTILEVEL FEEDBACK QUEUE SCHEDULING



SHARED MEMORY

SHARED MEMORY

- Typically, a shared-memory region resides in the address space of the process creating the shared memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Recall that, normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.
- They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

PROCESS SYNCHRONIZATION

- Concurrent processes or threads often need access to shared data and shared resources.
- If there is no controlled access to shared data, it is often possible to obtain an inconsistent state of this data.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes, and hence various process synchronization methods are used.

PRODUCER-CONSUMER PROBLEM

- A producer process produces information that is consumed by a consumer process. For Example compiler and assembler.
- The producer-consumer problem also provides a useful metaphor for the client-server paradigm.
- We generally think of a server as a producer and a client as a consumer.
- For example, a web server produces (that is, provides) web content such as HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.
- Producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.

PRODUCER-CONSUMER PROBLEM

Two types of buffers can be used:

- Bounded- Buffer—assumes a fixed buffer size. (consumer must wait if the buffer is empty, and the producer must wait if the buffer is full)
- Un-Bounded Buffer—places no practical limit on the size of the buffer.(consumer may have to wait for new items, but the producer can always produce new items.)

PRODUCER-CONSUMER PROBLEM

```
#define BUFFER_SIZE 10
typedef struct
{
   ...
} item;
item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

PRODUCER-CONSUMER PROBLEM

The code for the producer process is:

```
while(1)
{
   /*Produce an item in nextProduced*/
   while(counter == BUFFER_SIZE); /*do nothing*/
   buffer[in]=nextProduced;
   in=(in+1)%BUFFER_SIZE;
   counter++;
}
```

PRODUCER-CONSUMER PROBLEM

The code for the consumer process is:

```
while(1)
{
   while(counter==0); //do nothing
   nextConsumed=buffer[out];
   out=(out+1)%BUFFER_SIZE;
   counter--;
   /*Consume the item in nextConsumed*/
}
```

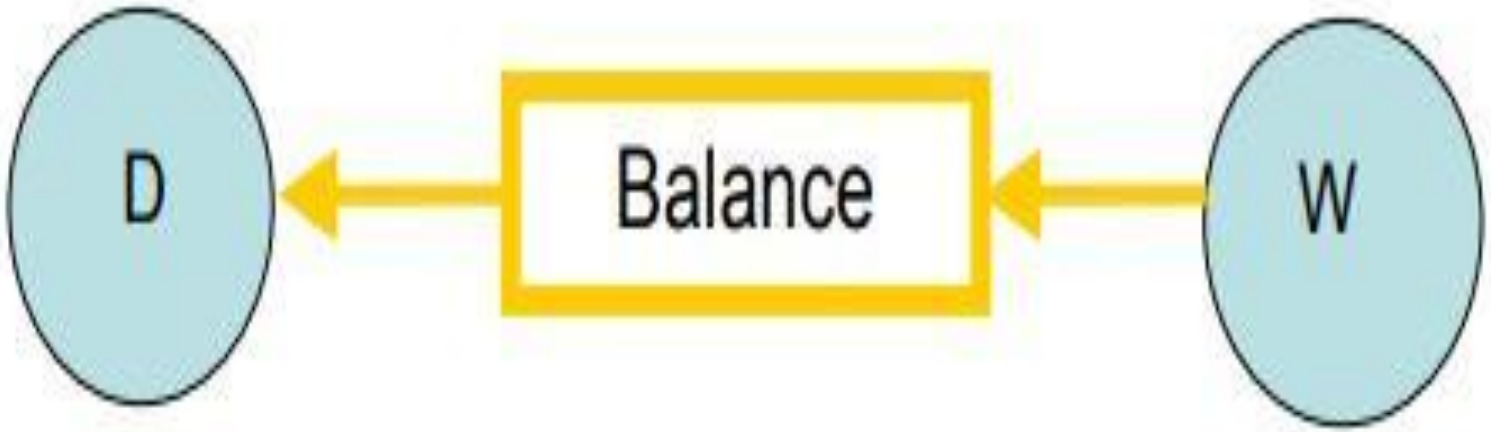
PRODUCER-CONSUMER PROBLEM

- Problem—Suppose that the value of the counter is 5, and that both the producer and the consumer execute the statement counter++ and counter- - concurrently.
- Following the execution of these statements the value of the counter maybe 4,5, or 6.
- The only correct result of these statements should be counter= =5, which is generated if the consumer and the producer execute separately.

PRODUCER-CONSUMER PROBLEM

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the manipulation depends on the particular order in which the access takes place, is called a race condition. To guard against such race conditions, we require synchronization of processes.

BANK TRANSACTION EXAMPLE

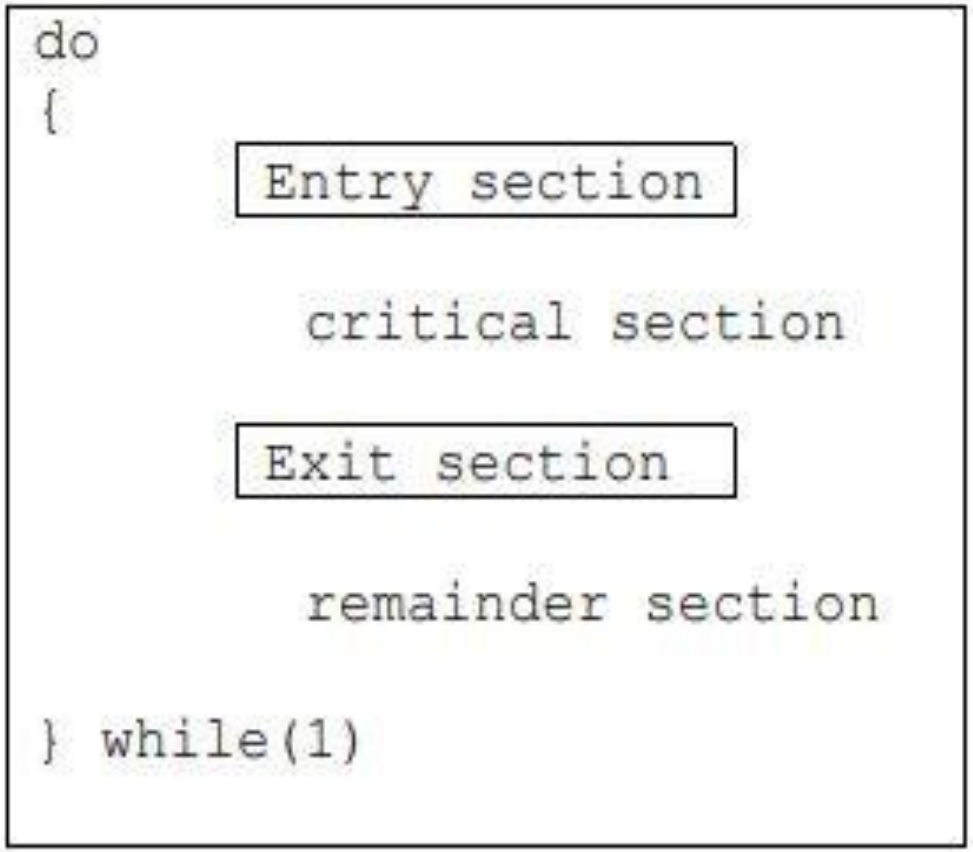


Critical Section—A piece of code in a cooperating process in which the process may updates shared data (variable, file, database, etc.).

CRITICAL SECTION PROBLEM

- Serialize executions of critical sections in cooperating processes.
- When a process executes code that manipulates shared data (or resource), we say that the process is in its critical section (for that shared data).
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors).
- So each process must first request permission to enter its critical section.
- The section of code implementing this request is called the entry section.
- The remaining code is the remainder section.

CRITICAL SECTION PROBLEM



CRITICAL SECTION PROBLEM

There can be three kinds of solutions to the critical section problem:

- ✔ Software based solutions
- ✔ Hardware based solutions
- ✔ Operating system based solution

SOLUTION TO CRITICAL SECTION PROBLEM

A solution to the critical section problem must satisfy the following three requirements:

1. **Mutual Exclusion—**If process Pi is executing in its critical section, then no other process can be executing in its critical section.
2. **Progress—**If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting—**There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.