CSE 330 – Midterm 2 – Devyash Lodha

Table of Contents

[Chapter 5 – CPU Scheduling 4](#_Toc3973652)

[CPU Burst 4](#_Toc3973653)

[CPU Scheduler (short term scheduler / dispatcher) 4](#_Toc3973654)

[o Dispatcher: 4](#_Toc3973655)

[Preemptive vs Non-preemptive Scheduling 5](#_Toc3973656)

[o Non-preemptive Scheduling: 5](#_Toc3973657)

[o Preemptive Scheduling: 5](#_Toc3973658)

[Dispatch Latency 5](#_Toc3973659)

[Scheduling Criteria – User vs. System: 5](#_Toc3973660)

[ CPU Utilization: 5](#_Toc3973661)

[ Throughput: 5](#_Toc3973662)

[ Turnaround Time: 5](#_Toc3973663)

[ Waiting Time: 5](#_Toc3973664)

[ Response Time: 5](#_Toc3973665)

[Scheduling Algorithms 5](#_Toc3973666)

[o Gantt chart 5](#_Toc3973667)

[o FCFS – First Come First Serve 6](#_Toc3973668)

[o SJF – Shortest Job First 6](#_Toc3973669)

[o SRTF – Shortest Remaining Time First 6](#_Toc3973670)

[o RR, given time quantum 6](#_Toc3973671)

[o Priority 6](#_Toc3973672)

[o Multi-level Queue 6](#_Toc3973673)

[o Multi-level Feedback Queue 7](#_Toc3973674)

[Thread scheduling: process vs system-contention scope 7](#_Toc3973675)

[o Contention Scope 7](#_Toc3973676)

[o Pthread Scheduling 7](#_Toc3973677)

[Multi-processor scheduling: asymmetric vs symmetric multi-processor scheduling 7](#_Toc3973678)

[o Asymmetric Scheduling 7](#_Toc3973679)

[o Symmetric Scheduling 8](#_Toc3973680)

[Multi-core processors: memory stalls, load balancing (push vs. pull strategies), processor affinity 8](#_Toc3973681)

[o Multi-core Processor Architectures 8](#_Toc3973682)

[o Memory Stalls 8](#_Toc3973683)

[o Load Balancing 8](#_Toc3973684)

[o Processor Affinity 8](#_Toc3973685)

[Heterogeneous Multi-processing 8](#_Toc3973686)

[Real-time CPU scheduling: soft vs. hard real-time systems 8](#_Toc3973687)

[Soft real-time systems 8](#_Toc3973688)

[Hard real-time systems 9](#_Toc3973689)

[Earliest-deadline-first (EDG) scheduling 9](#_Toc3973690)

[Chapter 7 – Synchronization 10](#_Toc3973691)

[Bounded-buffer problem (Producer-Consumer Problem) 10](#_Toc3973692)

[Readers-writers problem 11](#_Toc3973693)

[Dining philosophers problem 12](#_Toc3973694)

[Synchronization patterns (from class) 12](#_Toc3973695)

[Chapter 8 – Deadlocks 13](#_Toc3973696)

[Deadlock 13](#_Toc3973697)

[Starvation vs. Deadlock 13](#_Toc3973698)

[System Model 13](#_Toc3973699)

[Deadlock vs. Livelock 13](#_Toc3973700)

[Deadlock characterization 13](#_Toc3973701)

[o Necessary Conditions: 13](#_Toc3973702)

[o Resource-Allocation Graphs (RAGs) 14](#_Toc3973703)

[Methods for handling deadlocks: 14](#_Toc3973704)

[o Deadlock Prevention 14](#_Toc3973705)

[o Deadlock Avoidance 14](#_Toc3973706)

[o Deadlock Detection 14](#_Toc3973707)

[Deadlock prevention: prevent the 4 conditions from arising 14](#_Toc3973708)

[o Mutual Exclusion 14](#_Toc3973709)

[o Hold and Wait 14](#_Toc3973710)

[o No Preemption 15](#_Toc3973711)

[o Circular Wait 15](#_Toc3973712)

[Deadlock avoidance: 15](#_Toc3973713)

[o Safe vs. unsafe states 15](#_Toc3973714)

[o Banker’s algorithm / equivalent reductions in RAGs 15](#_Toc3973715)

[o Safety algorithm 16](#_Toc3973716)

[Deadlock detection: 16](#_Toc3973717)

[o Variation of Banker’s Algorithm 16](#_Toc3973718)

[o Properties of RAGs 16](#_Toc3973719)

[Recovery from deadlock 16](#_Toc3973720)

[Other Things to Know 17](#_Toc3973721)

[Basic pthreads calls 17](#_Toc3973722)

[Section 4.4 – Thread libraries / 4.4.1 – pthreads 17](#_Toc3973723)

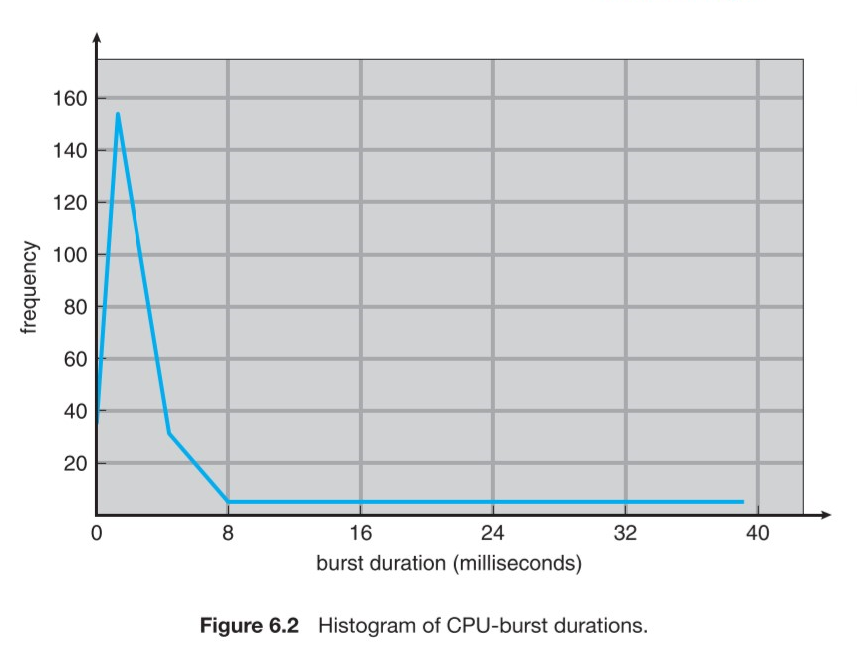
[Section 7.3 – POSIX synchronization / 7.3.2 – POSIX semaphores / 7.3.2 – POSIX condition variables 17](#_Toc3973724)

# Chapter 5 – CPU Scheduling

## CPU Burst

The execution of a process by the CPU

Most processes have a CPU burst where they perform computations, followed by an I/O burst, where they access I/O devices.



## CPU Scheduler (short term scheduler / dispatcher)

* When CPU becomes idle, the OS must select a process in the ready queue to execute
  + This is performed by the **short-term scheduler**
* Scheduling decisions may take place under four circumstances:
  1. Process switches from running state to waiting state
  2. Process switches from running to ready state
  3. Process switches from waiting to ready state
  4. Process terminates

### Dispatcher:

* + The component in CPU scheduling that gives control of the CPU to the process selected by the short-term scheduler. Has the following functions:
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart it
* The dispatcher MUST be designed to be as fast as possible because it is invoked during every process switch.

## Preemptive vs Non-preemptive Scheduling

### Non-preemptive Scheduling:

* + scheduling decisions only take place under 1 and 4 above (in CPU Scheduler).
  + Once time has been allocated to a process, it has the CPU time to use, regardless of what it chooses to do with it.

### Preemptive Scheduling:

* + the opposite of non-preemptive scheduling; scheduling decisions can take place under all 4 conditions above.

## Dispatch Latency

The amount of time it takes for the dispatcher to stop one process and start another.

## Scheduling Criteria – User vs. System:

### CPU Utilization:

* + attempt to keep the CPU as busy as possible

### Throughput:

* + Number of processes completed per time unit

### Turnaround Time:

* + how long it takes to execute a process; the sum of periods waiting to get into memory, executing on the CPU, waiting in the ready queue, and I/O.

### Waiting Time:

* + the CPU scheduling algorithm does not affect the amount of time that a process spends waiting in the ready queue; is the sum of the periods spent waiting in the ready queue.

### Response Time:

* + the time from the submission of a request until the first response is produced. It is the time It takes to start responding, not the time it takes to output a response.

## Scheduling Algorithms

### Gantt chart

* + A bar-type chart which illustrates time usage
  + X-axis is time
  + Y-axis is task

### FCFS – First Come First Serve

* + The process that requests the CPU first is allocated the CPU first
  + Easily managed by a FIFO queue
  + The average waiting time under the FCFS policy tends to be quite long
  + **Convoy Effect:** all other processes wait for one long-running process to get off the CPU.

### SJF – Shortest Job First

* + Determines the next burst length for each process
  + Run processes which have the shortest burst length first
  + If there is a tie in the burst length of two processes, FCFS is used
  + Next CPU burst is often predicted as an exponential average of the measured lengths of previous CPU bursts

### SRTF – Shortest Remaining Time First

* + Determines which process requires the least compute time, and runs that first
  + If there is a tie, use FCFS

### RR, given time quantum

* + Designed for time-sharing systems
  + Like FSFS, but preemption is added to enable the system to switch between processes
  + A unit of time – **time quantum** – or **time slice** is defined – usually 10-100 ms in length
  + Ready queue is treated as a circular queue
  + New processes are added to the tail of the queue

### Priority

* + SJF is a type of priority-scheduling algorithm
  + A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
  + Equal-priority processes are scheduled FCFS
  + Some systems use higher numbers to consider a higher priority; others use a lower number to consider a higher priority
  + Problems:
    - **Indefinite Blocking / Starvation:** a process is ready to run but is waiting for the CPU can be considered blocked
    - A heavy stream of higher-priority processes can prevent the consideration of other processes

### Multi-level Queue

* + Used when processes can be classified into different groups
  + Often, a division is made between **foreground** (interactive) processes, and **background** (batch) processes
    - Foreground / background applications have different response-time requirements – they have different scheduling needs
  + The ready queue is partitioned into multiple separate queues
  + Processes are “permanently” assigned to a queue, usually because of a process property such as memory size, process priority or process type
  + Each queue has its own scheduling algorithm
  + Scheduling over the queues often uses the FCFS algorithm

### Multi-level Feedback Queue

* + In multi-level queues, processes are permanently assigned to a queue when they enter the system
  + Now, we can move processes around, between the different queues
  + It a process uses too much CPU, it can be moved to a lower priority state
  + If a process is waiting too long, it may be moved up in its priority

## Thread scheduling: process vs system-contention scope

### Contention Scope

* + One distinction between user and kernel threads is in scheduling
  + **Process Contention Scope:** user-level threads; threads fight for CPU time for the same process
  + **System Contention Scope:** kernel-level threads; the kernel can decide how to share CPU time between the threads

### Pthread Scheduling

* + Pthreads allows process contention scope as well as system contention scope
  + PTHREAD\_SCOPE\_PROCESS schedules a thread using PCS
  + PTHREAD\_SCOPE\_SYSTEM schedules a thread using SCS
  + If a system allows the many-many model, PTHREAD\_SCOPE\_PROCESS will perform the many to many model
  + pthread\_attr\_getscope(pthread\_attr\_t\* attr, int scope) <- Gets the current scope
  + pthread\_attr\_setscope(pthread\_attr\_t\* attr, int\* scope) <- Sets the scope of the thread

## Multi-processor scheduling: asymmetric vs symmetric multi-processor scheduling

### Asymmetric Scheduling

* + One master CPU/core (the server) should handle the following:
    - Scheduling decisions
    - I/O processing
    - Other system activities
  + Simple, because only one CPU accesses the system’s data structures, reducing the need for data sharing and synchronization

### Symmetric Scheduling

* + Each processor / core is self-scheduling
  + Processes may be in a common ready queue / each processor may have its own private queue of ready processes
  + Scheduler must be programmed carefully to prevent data sharing and corruption issues

## Multi-core processors: memory stalls, load balancing (push vs. pull strategies), processor affinity

### Multi-core Processor Architectures

* + One core / Multiple CPU
  + Multiple cores / Single CPU
  + Multiple threads / Multiple cores

### Memory Stalls

* + Waiting for data from RAM that is not in cache memory
  + Caused by a cache miss or other reasons

### Load Balancing

* + Attempting to distribute the workload evenly between all processors in an SMP system

#### Push Strategy

* + - A specific task periodically checks the load on each processor and rebalances if it there is an imbalance
    - Processes are “pushed” from an overloaded to an idle / less busy CPU

#### Pull Strategy

* + - Idle processor “pulls” a waiting task from a busy processor
  + Often, the two strategies are used in conjunction

### Processor Affinity

* + A process has an affinity for the processor on which it is currently running -> the process can only run on said processor.

## Heterogeneous Multi-processing

Example: ARM processors often have the big.LITTLE architecture, which consists of a set of powerful processors for general runtime use, plus some processors which use less power for other tasks.

## Real-time CPU scheduling: soft vs. hard real-time systems

### Soft real-time systems

* Provide no guarantee as to when a critical real-time process will be scheduled

### Hard real-time systems

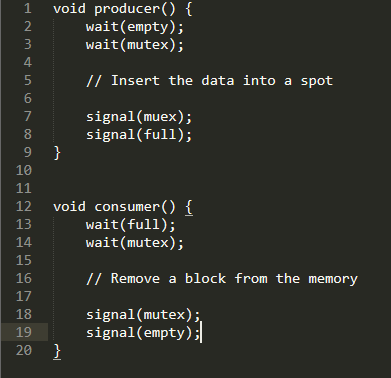
* Have stricter requirements; a task MUST be serviced before its deadline

### Earliest-deadline-first (EDG) scheduling

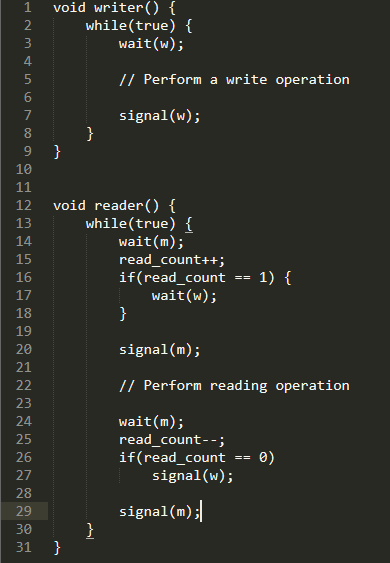
* Whichever process has the earliest deadline should be serviced first

# Chapter 7 – Synchronization

## Bounded-buffer problem (Producer-Consumer Problem)

* There is a buffer of slots, each of which can store one unit of data
* Two processes are running – the **producer** and the **consumer**
* A producer tries to insert data into an empty slot of the buffer
* A consumer tries to remove data from a filled slot in the buffer
* This creates a race condition when the producer and consumer are executing concurrently
* Solution to the problem:
  + 
  + Producer
    1. Waits until there is at least one empty slot
    2. Decrements the empty semaphore because one block will be inserted
    3. Acquires mutual exclusion to the buffer
    4. Inserts the block
    5. Releases from ME and increments full
  + Consumer
    1. Waits until one full slot is in the buffer
    2. Decrements the full semaphore because a slot is emptied
    3. Consumer acquires ME for the buffer
    4. Consumer removes a block from the buffer
    5. Consumer releases ME for the buffer
    6. Empty semaphore is incremented by one because an empty slot has been created

## Readers-writers problem

* One process can write at a time
* Multiple processes can read at a time
* Mutex m and semaphore w are used; an integer variable reader\_count is used to track the number of readers accessing the resource
* read\_count is initialized to 0; m and w are initialized to 1
* Solution:
  + 
* The write worker waits until w is available
* is acquired whenever read\_count is updated by a process
* When a reader wants to access the resource, it increments read\_count, then accesses the resources, then decrements the read\_count value
* The semaphore is used by the first reader which enters the critical section and the last reader which exits the critical section
  + Allows the writer to unblock
  + When the last reader exits, it allows the writer to continue

## Dining philosophers problem

* There are 5 philosophers sitting around a circular dining table – each has five chopsticks and a bowl of rice in the middle
* At any instant, a philosopher is either thinking or eating
* When a philosopher wants to eat, he uses two chopsticks, one from his left, and one from his right
* When a philosopher wants to think, he keeps down both chopsticks in their original place
* A philosopher can think for an indefinite amount of time
* A philosopher can only eat for so long
* Solution:
  + 5 semaphores are needed, for each chopstick
  + Code for each philosopher: 
* Issues we Encounter
  + If all five philosophers are simultaneously hungry, and each of them pick up one chopstick, then a deadlock situation occurs; they will be waiting for another chopstick forever
  + Solutions:
    1. A philosopher must be allowed to pick up the chopsticks if only both left and right are available. (ME?)
    2. Allow only four philosophers to sit at a table so there is always a chopstick left on the table

## Synchronization patterns (from class)

# Chapter 8 – Deadlocks

## Deadlock

When a waiting process is never able to change state because the requested resources are not available at a time; often caused because resources requested are held by other waiting processes

## Starvation vs. Deadlock

**Starvation:** when a low priority process does not get access to resources it needs because there is a high priority process accessing the resources

The entire system of processes hasn’t come to halt in this case

**Deadlock:** when all the processes do not get access to resources because every process is waiting for some other processes

Cycle in the RAG

## System Model

* A system consists of a finite number of resources distributed among several competing processes
* If a process requests an instance of a resource type, the allocation of *any* instance of the type should satisfy the request
* Under normal operation, a process may utilize a resource in the following sequence:
  1. **Request**: get access to a resource instance
  2. **Use:** use the resource
  3. **Release:** release the resource

## Deadlock vs. Livelock

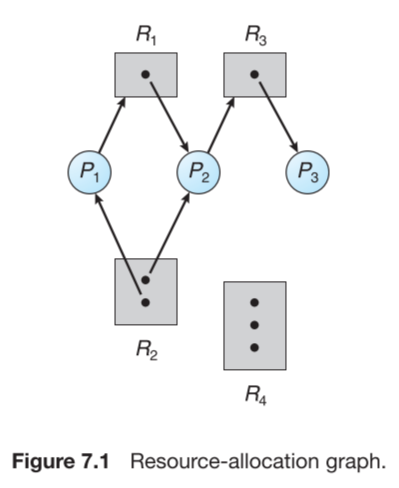
* **Deadlock:** all processes are sleeping while waiting for available resources
* **Livelock:** quite similar, but the processes try to actively access the same resource
  + Similar to trying to avoid someone, but they go the same direction as you indefinitely

## Deadlock characterization

## Necessary Conditions:

* 1. **Mutual Exclusion**: at least one resource must be held in a non-shareable mode – only one process can use it at one time; other processes must wait for that resource
  2. **Hold and Wait**: a process must be holding at least one resource and waiting to acquire additional resources
  3. **No preemption**: resources cannot be preempted – a resource can only be released voluntarily by the process holding it after that process completed its task
  4. **Circular Wait:** a cycle of processes waiting for resources before they can continue

### Resource-Allocation Graphs (RAGs)

* + Resources are represented as rectangles
  + Processes are represented as circles
  + Dots in the rectangles show different instances of the resources
  + Arrow from process to resource means write
  + Arrow from resource to process means read
  + 

## Methods for handling deadlocks:

### Deadlock Prevention

* + A set of methods to ensure at least one of the necessary conditions for deadlock CANNOT hold

### Deadlock Avoidance

* + The operating system must be given extra information in advance, concerning which resources a process will request and whether it should wait

### Deadlock Detection

* + The operating system can determine a deadlock if all processes are waiting on a resource to be readied up, but the processes holding on the resources are waiting too.

## Deadlock prevention: prevent the 4 conditions from arising

### Mutual Exclusion

* + We could try to make mutual exclusion not possible
  + This makes the issue that not all data is inherently shareable, so we will corrupt our data via simultaneous access

### Hold and Wait

* + We could try to make sure that if a process holds one resource, it cannot hold another
  + We could only process resource requests when no resources are held
  + Starvation is possible because a process that requires multiple resources may have to wait indefinitely

### No Preemption

* + Allow resources to be snatched from processes when other processes need them

### Circular Wait

## Deadlock avoidance:

## Safe vs. unsafe states

* + A state is **safe** if the system can allocate resources to each processes in some order and still avoid deadlock
    1. A system is in a safe state only if there exists a **safe sequence**
  + A safe state is a state in which all resources of the system are **well managed**. All resources are not assigned to one request
  + There should be no **circular demand** of resources in the system
  + Basically, there should be no possible deadlocks

## Banker’s algorithm / equivalent reductions in RAGs

* + When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need
  + This number should not exceed the capabilities of the system
  + When a process requests a set of resources, the system must determine whether the allocation of these will leave the system in a safe state; if it will, processes will be allocated, otherwise, it must wait for enough resources
  + Data structures required for Banker’s algorithm:
    1. **Available:** vector of length indicates the number of available resources of each type. If , then instances of resource type are available
    2. **Max:** An matrix defines the maximum demand of each process
       - If , then process may request at most instances of resource type
    3. **Allocation:** an matrix indicates the umber of resources of each type currently allocated to each process
       - If , then process is currently allocated instances of resource type
    4. **Need:** an matrix indicates the remaining resource needs of each process
       - If , then process may need more instances of resource type

## Safety algorithm

* + Determines if the system is in a safe state
  + Process:
    1. Let work and finish be vectors of length and respectively
       - Initialize
       - Initialize for
    2. Find an index such that both:
       - If no such exists, go to step 4
    3. Following
       - Go to step 2
    4. If for all , then the system is in a safe state
  + This algorithm may require operations to determine whether a state is safe!!!

## Deadlock detection:

* If a system does not employ deadlock-prevention or deadlock-avoidance, a deadlock may occur
* The system may provide:
  + An algorithm to examine the state of the system to determine whether a deadlock occurred
  + An algorithm to recover from the deadlock

### Variation of Banker’s Algorithm

### Properties of RAGs

## Recovery from deadlock

# Other Things to Know

## Basic pthreads calls

## Section 4.4 – Thread libraries / 4.4.1 – pthreads

## Section 7.3 – POSIX synchronization / 7.3.2 – POSIX semaphores / 7.3.2 – POSIX condition variables