

Operating Systems

Prepared by Dr. Alaa Mohamed

Module 4

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Scheduling and Deadlocks



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

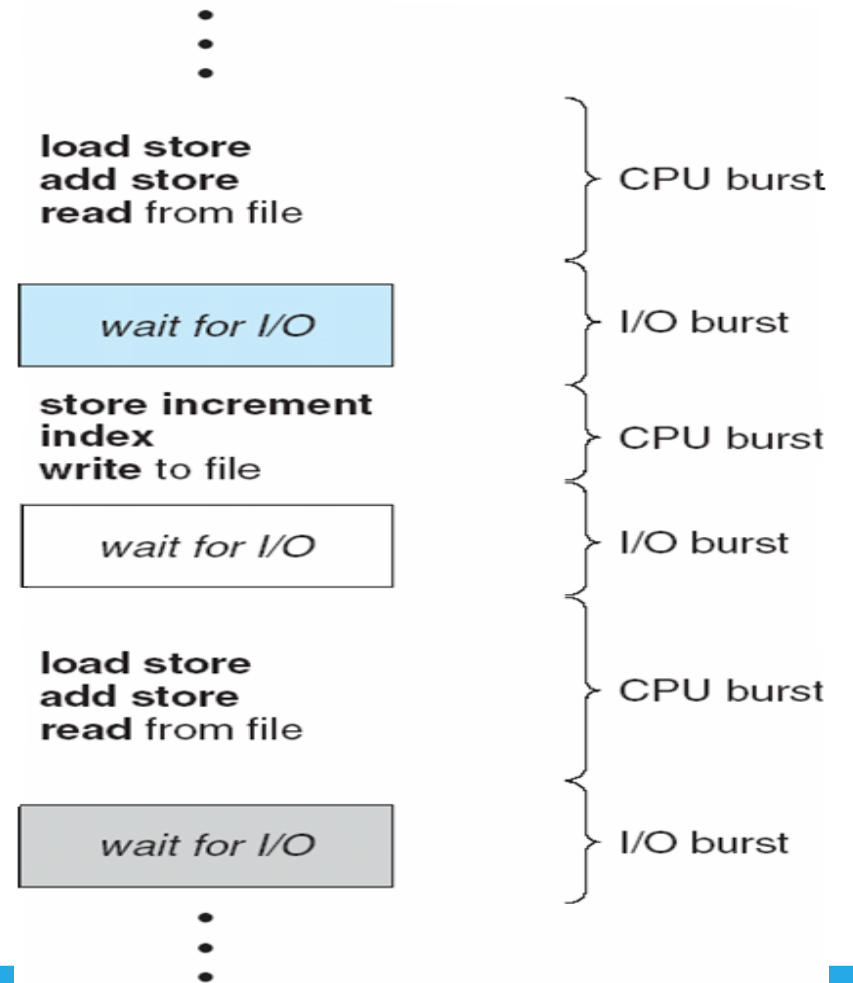
- Objectives:
 - Basic Concepts
 - Scheduling Criteria
 - Scheduling Algorithms
 - Deadlock Characterization
 - Methods for Handling Deadlocks
 - Deadlock Prevention
 - Deadlock Avoidance
 - Deadlock Detection
 - Recovery from Deadlock

- Topics To Be Covered:
 - To introduce CPU scheduling, which is the basis for multiprogram on operating systems
 - To describe various CPU-scheduling algorithms
 - To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

- The Objective is to Maximum CPU utilization obtained with multiprogramming
- A process is executed until it must wait, typically for the completion of some I/O request.
- In a simple computer system, the CPU then sits idle. All this is wasted: no useful work is accomplished.
- Several processes are kept in memory at one time.
- When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process and this pattern continues.

- CPU–I/O Burst Cycle ,
Process execution consists of
a cycle of **CPU execution** and
I/O wait.
- Processes alternate between.
those two states,

CPU Burst is the time when
the process is being executed in CPU.
I/O burst is the time when a process
wait for I/O to complete.



- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
- Scheduling only under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

- **Nonpreemptive**

- Once a process is allocated the CPU, it does **not** leave unless:
 1. it has to wait, e.g., for I/O request or for a child to terminate
 2. it terminates

- **Preemptive**

- OS can force (preempt) a process from CPU at **anytime**
 - Say, to allocate CPU to another higher-priority process

- Which is harder to implement? and why?
 - Preemptive is harder: Need to maintain **consistency** of data shared between processes, and more importantly, kernel data structures (e.g., I/O queues)
 - Think of a preemption while kernel is executing a sys call on behalf of a process (many OSs, wait for sys call to finish)

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

- **CPU utilization** – keep the CPU as busy as possible
 - Maximize
- **Throughput** – # of processes that complete their execution per time unit
 - Maximize
- **Turnaround time** – amount of time to execute a particular process (time from submission to termination)
 - Minimize
- **Waiting time** – amount of time a process has been waiting in **ready queue**
 - Minimize
- **Response time** – amount of time it takes from when a request is submitted until the first response is produced, not output (for time-sharing environment)
 - Minimize

- First Come, First Served
- Shortest Job First
- Priority
- Round Robin



First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3
- The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

First-Come, First-Served (FCFS) Scheduling

Suppose that the processes arrive in the order

P_2, P_3, P_1

3, 3, 24

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Convoy effect: short process behind long process

First-Come, First-Served (FCFS) Scheduling

- FCFS is fair in the formal sense or human sense of fairness
- but it is unfair in the sense that long jobs take priority over short jobs and unimportant jobs make important jobs wait
- One of the major drawbacks of this scheme is that the waiting time and the average turnaround time is often quite long
- FCFS Scheme is not useful in scheduling interactive users because it cannot guarantee good response time.
- The code for FCFS scheduling is simple to write and understand.

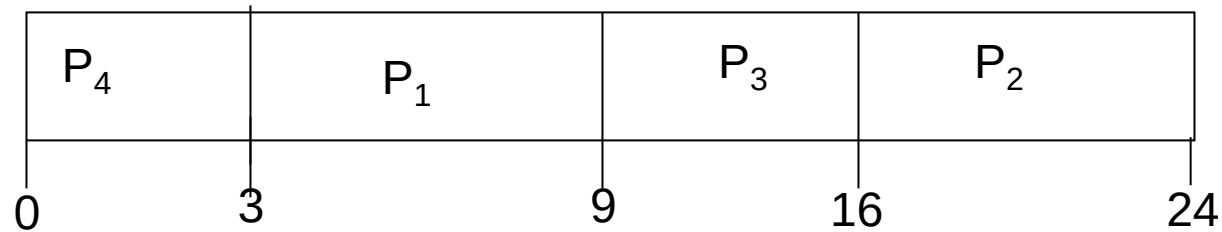
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
 - Non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
 - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the **Shortest-Remaining-Time-First (SRTF)**
- SJF is optimal – gives minimum average waiting time for a given set of processes
- The difficulty is knowing the length of the next CPU request

Shortest-Job-First (SJF) Scheduling

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

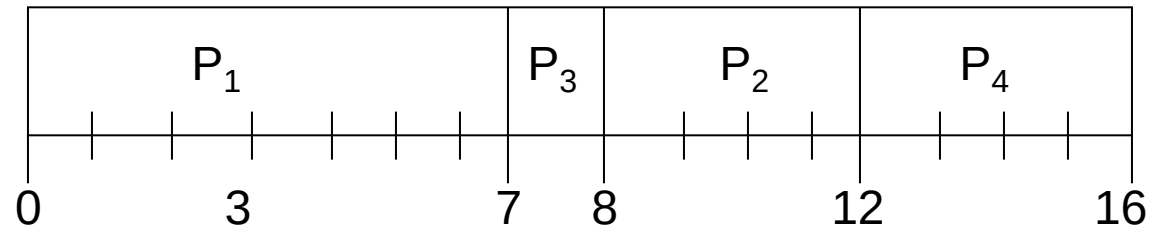


- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- The Gantt chart for non-preemptive SJF is:



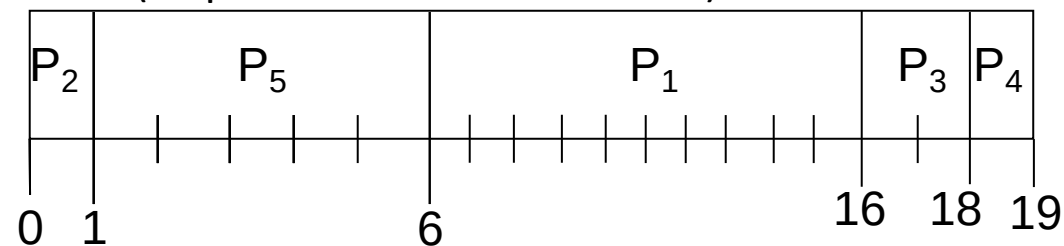
- Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)
 - Preemptive
 - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem $\equiv$ **Starvation** – low priority processes may never execute
- Solution $\equiv$ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling (nonpreemptive)

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- The Gantt chart is (all processes arrive at time 0):



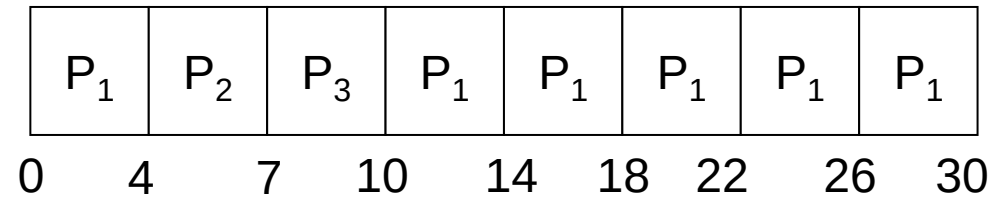
- Average waiting time = $(6 + 0 + 16 + 18 + 1)/5 = 8.2$

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
 - q large $\Rightarrow$ FIFO
 - q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*

- System consists of resources
 - Resource types $R_1, R_2, \dots, R_m$
CPU cycles, memory space, I/O devices
 - Each resource type R_i has W_i instances.
 - Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**
- Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

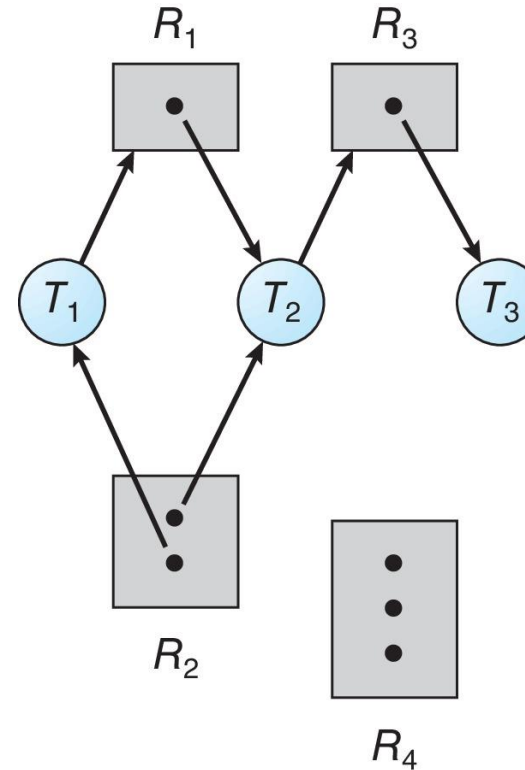
Deadlock can arise if four conditions hold simultaneously.

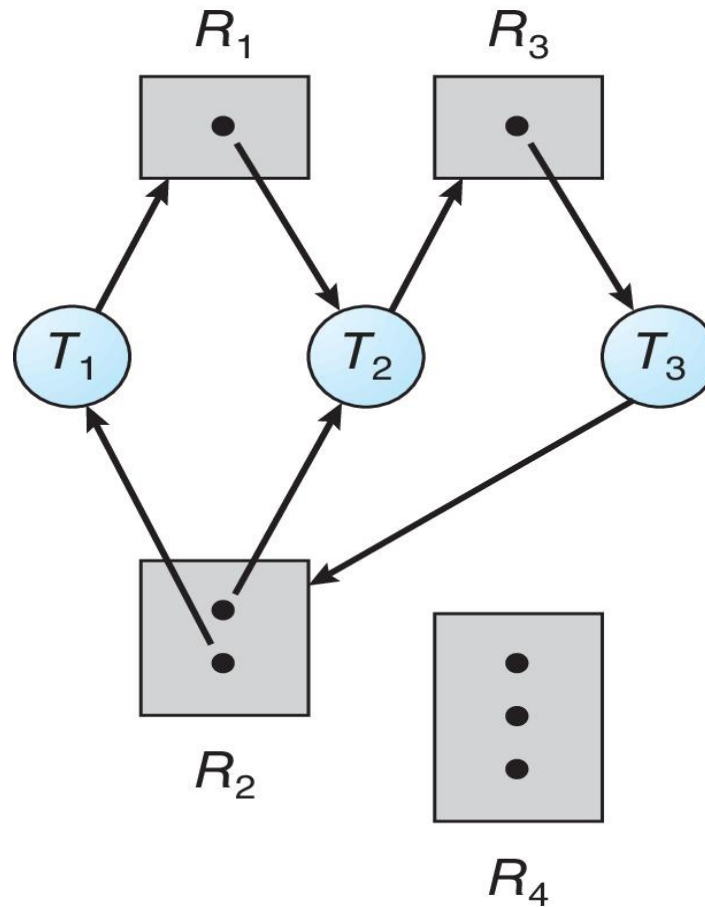
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

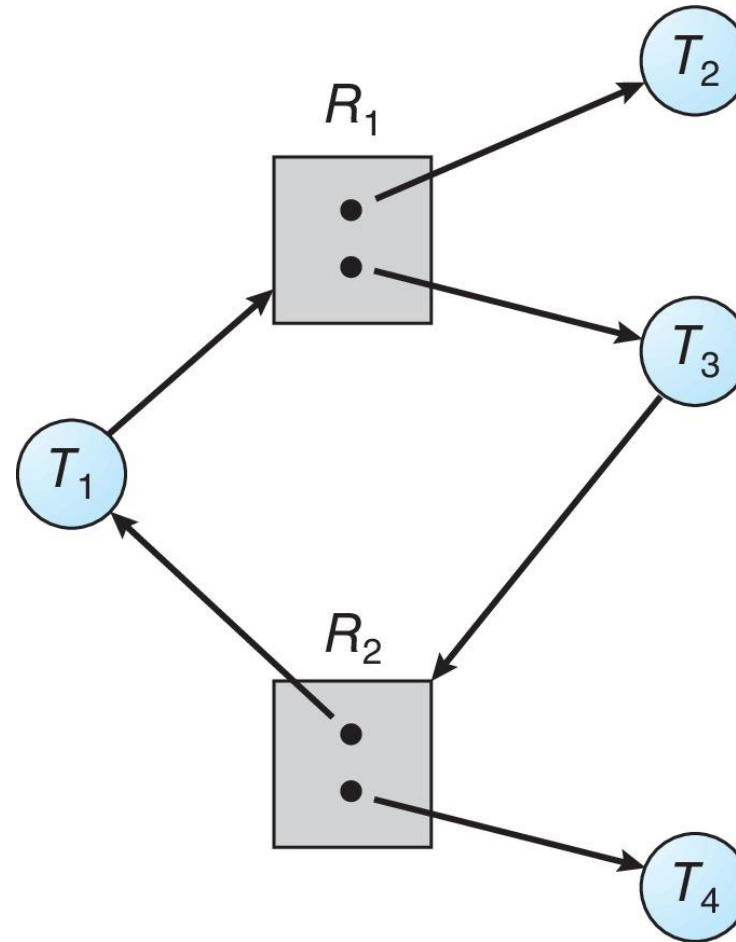
A set of vertices V and a set of edges E .

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3







- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, possibility of deadlock

- Ensure that the system will *never* enter a deadlock state:
 - Deadlock prevention
 - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to request and be allocated all its resources before it begins execution or allow process to request resources only when the process has none allocated to it.
 - Low resource utilization; starvation possible

- **No Preemption** –
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
 - Preempted resources are added to the list of resources for which the process is waiting
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Requires that the system has some additional *a priori* information available

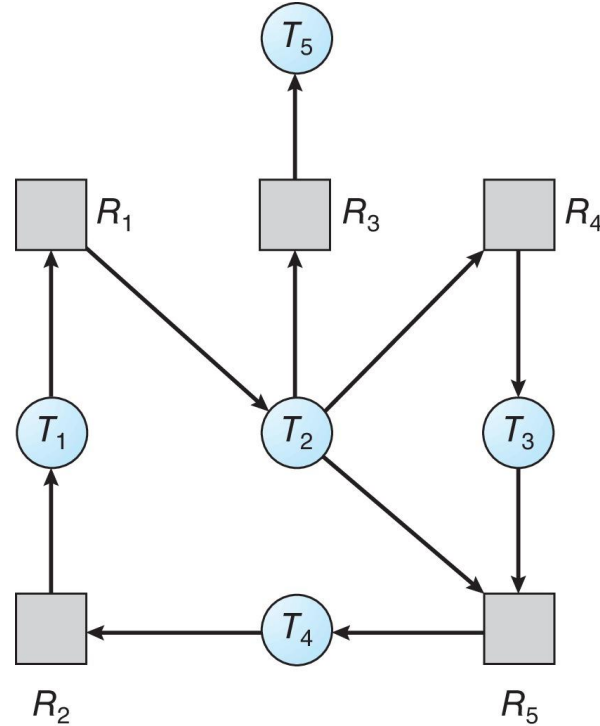
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

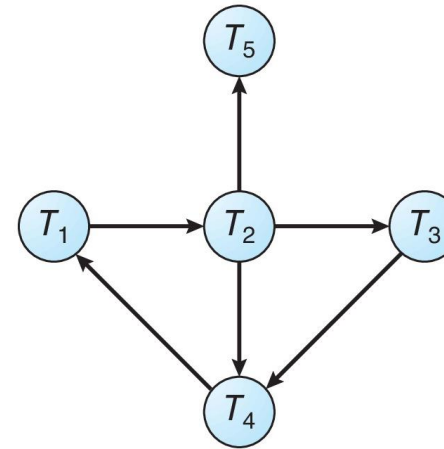
- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

- Maintain **wait-for** graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock



(a)



(b)

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
 1. Priority of the process
 2. How long process has computed, and how much longer to completion
 3. Resources the process has used
 4. Resources process needs to complete
 5. How many processes will need to be terminated
 6. Is process interactive or batch?

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

End of Module 4

