CHAPTER – TWO

Process Management

Process concepts

Process vs. Program

- Program
  - It is sequence of instructions defined to perform some task
  - It is a passive entity
  - Not in execution; just a file.
- Process
  - It is a program in execution
  - It is an instance of a program running on a computer
  - It is an active entity, which Actively running on the CPU.

The process model

- A process is just an executing program.
- The CPU switches back and forth from process to process, this rapid switching back and forth is called multiprogramming.
- However, at any instant of time, the CPU runs only one program.
- Thus giving the users illusion of parallelism.

System calls

- The interface between the **operating system** and **running program** is defined by the set system calls that the operating system provides.
  - Generally available as assembly-language instructions.
  - Languages defined to replace assembly language for systems
  - Programming allow system calls to be made directly (e.g., C, C++)

System calls…

- Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in registers.
  - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  - Push (store) the parameters onto the stack by the program, and pop off the stack by operating system.

Process Management

Overview

- The most fundamental function of modern operating systems is process management.
- It includes:
    - Creation of processes
    - Allocation of resources to processes
    - Protecting resources of each processes from other processes
    - Enabling processes to share and exchange information
    - Enabling synchronization among processes for proper sequencing and coordination when dependencies exist
    - Termination of processes

Types of Processes

- There are two types of processes:
  - **Sequential Processes**
    - Execution progresses in a sequential fashion, i.e. one after the other
    - At any point in time, at most one process is being executed

Types of Processes

- **Concurrent Processes**
  - There are two types of concurrent **processes**
    - True Concurrency (Multiprocessing)
      - Two or more processes are executed simultaneously in a multiprocessor environment

        Supports real parallelism
    - Apparent Concurrency (Multiprogramming)
      - Two or more processes are executed in parallel in a **uniprocessor** environment by switching from one process to another
      - Supports pseudo parallelism, i.e. fast switching among processes gives illusion of parallelism

Process creation

- Operating system needs some way to make sure all the necessary processes are **created** and **terminated**.

- There are four principal events that cause a processes to be created:

  1. System initialization

  2. Execution of a process creation system call by a running process.

  3. A user request to create a new process.

  4. Initialization of a batch of job.

1. System initialization - When an operating system is booted, typically several processes are created.

- Some of these are foreground processes, (i.e. processes that interact with users and perform work for them.)
- Others are background processes, which are not associated with particular users.

  Example: one background process may be designed to accept incoming-

  mail sleeping most of the time, incoming request for web pages
- Processes that stay in the background to handle some activities are called daemons.

2- Creation of processes by running process

- Often a running process will issue system calls to create one or more new processes to help it to do its job.

Example: If a large amount of data is being fetched over a network for subsequent processing,

- one process to fetch the data and put them in a shared buffer,
- While the second process removes the data item and process them.

3 - A user request to create processes

- Users can start a program by typing a command or (double) clicking an icon.
- Taking either of these actions starts a new process and runs the selected program in it.

4 - Initiation of a batch of job

- Here user can submit batch of jobs to the system (possibly remotely) in main frame computer.
- When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.
- In all these cases, a new process is created by having an existing process execute a process creation system call.

Process Termination

- After a process has been created, it starts running and does whatever its job is.
- Sooner or later the new process will terminate, due to one of the following conditions:

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

1- Normal exit

- Most process terminates because they have done their work.

Example:

- When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished.
- Screen oriented programs also support voluntary termination. Word processors, Internet browser and similar programs always have an icon or menu item that the user can click to tell the process to terminate.

File -> Exit

2. **Error exit :** When the process discovers an error.

Example: If the user typed a command

`javac Function.java`

- To compile the program and no such file exists, the compiler simply exits.

3. **Fata error** : is an error caused by the process, often due to a program bug.

Example:

- Executing an illegal instruction
- Referencing non existent memory
- Dividing by zero

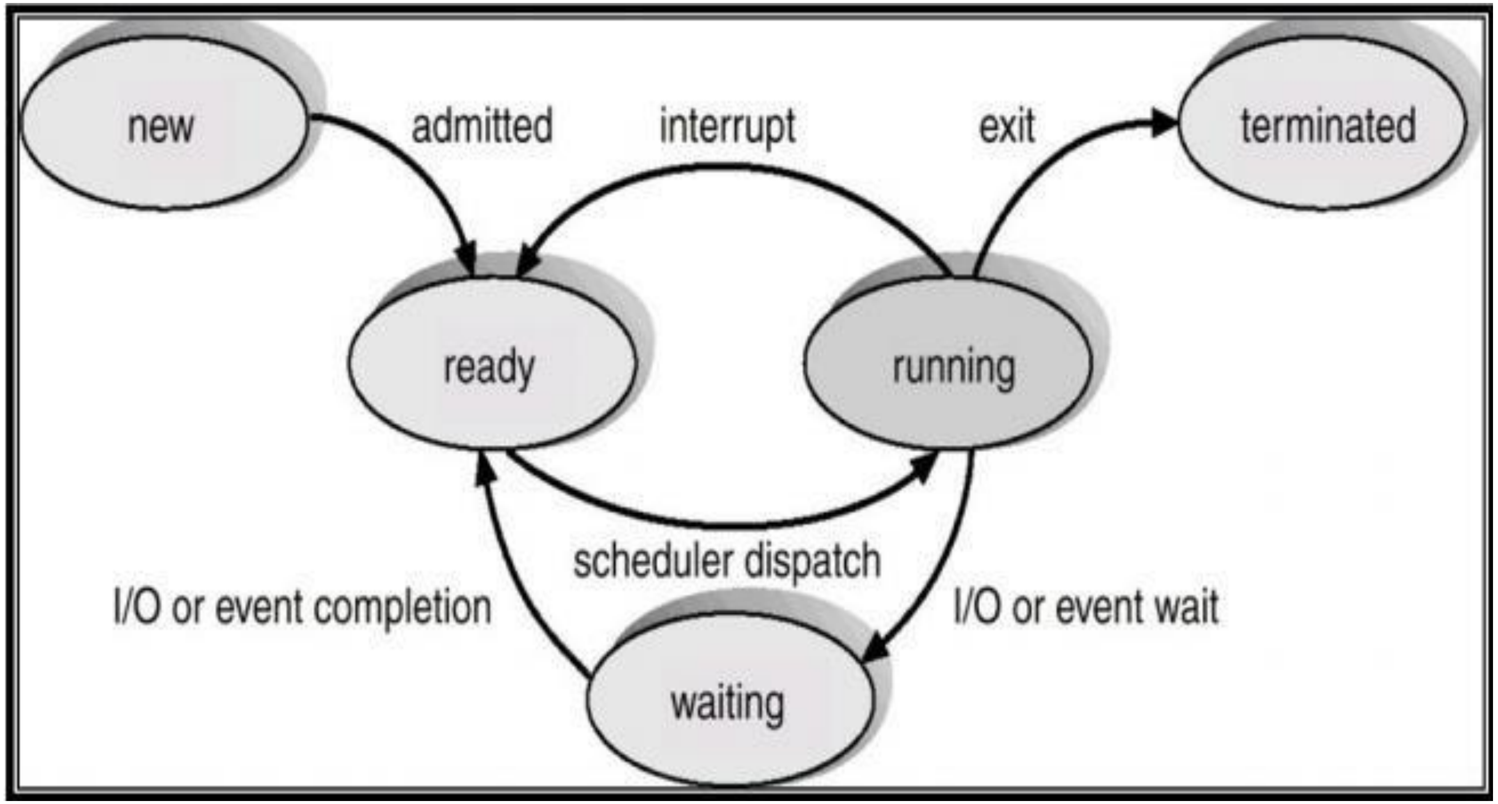
4. **Killed by another process** – a process executes a system call telling the operating system to kill some other process.

Process States

- During its lifetime, a process passes through a number of states.

- new: The process is being created.
- ready: The process is waiting to be assigned to a processor.
- running: Instructions are being executed.
- Waiting/blocked: The process is waiting for some event to occur such as I/O operation.
- terminated: The process has finished execution.

Diagram of Process State



State Transitions in Five-State Process Model

- **New -> ready**
  - ✓ Admitted to ready queue; can now be considered by CPU scheduler.
- **ready -> running**
  - ✓ CPU scheduler chooses that process to execute next, according to some scheduling algorithm
- **running -> ready**
  - ✓ Process has used up its current time slice
- **running -> blocked**
  - ✓ Process is waiting for some event to occur (for I/O operation to complete, etc.)
- **blocked -> ready**
  - ✓ Whatever event the process was waiting on has occurred

Process Control Block (PCB)

- It is a data structure used by the operating system to manage information about a process. It plays a crucial role in process management and is essential for the operating system to track the status and control of processes.

- **Example**:

When a process is created, the operating system allocates a PCB for it. As the process runs and interacts with system resources, the PCB is updated with the process's state and resource usage. If the process is interrupted or blocked, the PCB provides all the necessary information to resume it later without loss of context.

Thread

- Thread vs. Process
  - A thread is a dispatchable unit of work (lightweight process) that has independent context, state and stack
  - A process is a collection of one or more threads and associated system resources
- The thread has:
  - A program counter that keeps track of which instruction to execute next.
  - Has registers which hold its current working variables
  - Has a stack which contains the execution history

Thread

- What threads add to the process model is to allow multiple executions to take place in the same process environment.
- Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer.

– In the former case the threads share an address space, open files, and other resources.

– In the later case processes share physical memory, disks, printers and other resources.

Thread

- Threads are sometimes called lightweight process.
- Multithreading - is to describe the situation of allowing multiple threads in the same process.
- A thread is a single sequence of execution within a program
  - Multithreading refers to multiple threads of control within a single program each program can run multiple threads of control within it, e.g., Web Browser, MS Words,…

Thread Usage

- Improved Responsiveness:
- Concurrency:
- Resource Sharing:
- Simplified Design:

CPU Scheduling

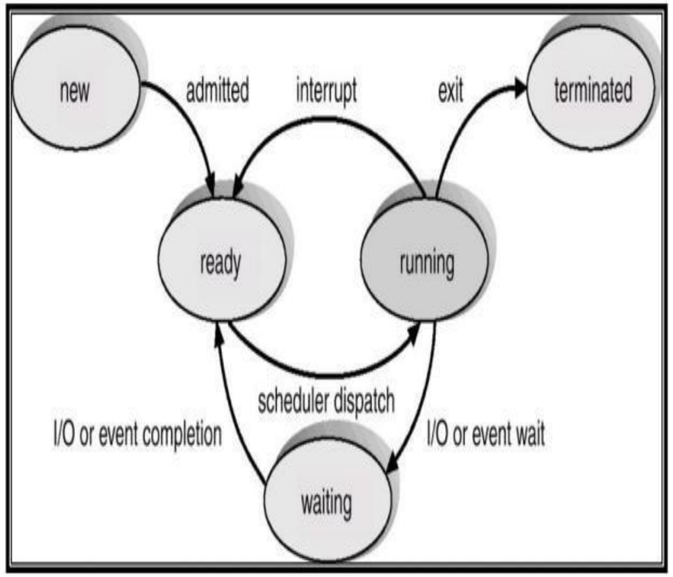
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms

CPU Scheduler: OS

- Selects among processes in memory that are ready to be executed, and allocates 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 under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive*.

CPU Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per unit time
- Turnaround time – total time required to execute a particular process
- Waiting time – amount of time a process waits in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced.

CPU Optimization Criteria

- Maximize throughput - run as many jobs as possible in a given amount of time.
  - This could be accomplished easily by running only short jobs or by running jobs without interruptions.
- Minimize response time - quickly turn around interactive requests.
  - This could be done by running only interactive jobs and letting the batch jobs wait until the interactive load ceases.
- Minimize turnaround time - move entire jobs in and out of the system quickly.
  - This could be done by running all batch jobs first (because batch jobs can be grouped to run more efficiently than interactive jobs).

CPU Optimization Criteria…

- Minimize waiting time - move jobs out of the READY queue as quickly as possible.
  - This could only be done by reducing the number of users allowed on the system so the CPU would be available immediately whenever a job entered the READY queue.
- Maximize CPU efficiency - keep the CPU busy 100 percent of the time.
  - This could be done by running only CPU-bound jobs (and not I/O- bound jobs).
- Ensure fairness for all jobs - give everyone an equal amount of CPU and I/O time.
  - This could be done by not giving special treatment to any job, regardless of its processing characteristics or priority.

Process Scheduling Algorithms

- Part of the operating system that makes scheduling decision is called scheduler and the algorithm it uses is called scheduling algorithm
- The Process Scheduler relies on a process scheduling algorithm,
  - based on a specific policy, to allocate the CPU and move jobs through the system.
- There are six process scheduling algorithms that have been used extensively.

First-Come, First-Served (FCFS) Scheduling

Basic Concept

- is a non preemptive scheduling algorithm that handles jobs according to their arrival time:
- the earlier they arrive, the sooner they're served.
- It's a very simple algorithm to implement because it uses a FIFO queue.
- In a strictly FCFS system there are no WAIT queues (each job is run to completion).

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time (msec)
P_1	24
P_2	3
P_3	3

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

- P_1 P_2 P_3
- 0 24 27 30

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Turnaround time for P_1=24, P_2=27, P_3=30
- Avg turnaround time: (24+27+30)/3= 27

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2, P_3, P_1 .

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Turnaround time for P_1=30, P_2=3, P_3=6
- Avg turnaround time: (30+3+6)/3= 13
- Much better than previous case.

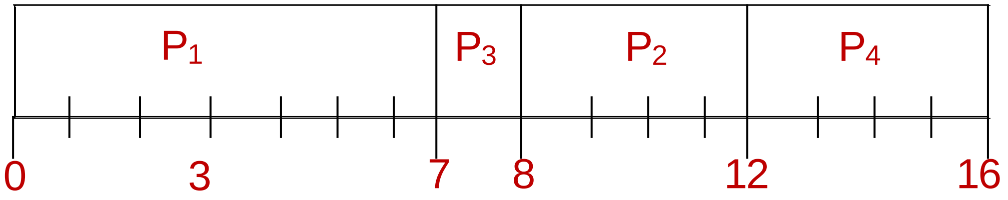
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - nonpreemptive – 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 know as the Shortest-Remaining- Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

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

- SJF (non-preemptive)

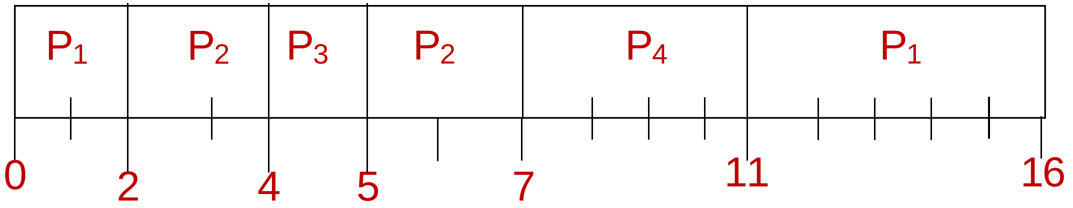


- Waiting time P_1=0-0=0, P_2=8-2=6, P_3=7-4=3, P_4=12-5=7
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4
- Turnaround time: P_1= 7-0=7, P_2=12-2=10, P_3=8-4=4, P_4=16-5=11
- Avg. turnaround time: (7+10+4+11)/4= 8

Example of 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

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3
- Turnaround time:P_1=16-0=16, P_2=7-2=5, P_3=5-4=1, P_4=11-5=6
- Avg. turnaround time= (16+5+1+6)/4=7

Priority Scheduling

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

- **Example of** Priority Scheduling

Let's assume we have **4 processes** with the following details:

Process	Burst Time	Priority
P1	10	2
P2	1	1
P3	2	4
P4	1	3

Scheduling Order:

Based on the table above, the order is:

1. **P2** (priority 1)
2. **P1** (priority 2)
3. **P4** (priority 3)
4. **P3** (priority 4)

Calculation of Waiting Time and Turnaround Time:

Step-by-step Calculations:

1. **P2** runs first (highest priority).
    1. **Waiting Time (WT)** for P2 = 0 (it starts immediately)
    2. **Turnaround Time (TT)** for P2 = 1 (WT + Burst Time)
2. **P1** runs next.
    1. **Waiting Time** for P1 = 1 (P2's burst time)
    2. **Turnaround Time** for P1 = 1 + 10 = 11
3. **P4** runs third.
    1. **Waiting Time** for P4 = 1 + 10 = 11
    2. **Turnaround Time** for P4 = 11 + 1 = 12
4. **P3** runs last.
    1. **Waiting Time** for P3 = 1 + 10 + 1 = 12
    2. **Turnaround Time** for P3 = 12 + 2 = 14

Average Times:

- **Average Waiting Time** = (0 + 1 + 11 + 12) / 4 = 6
- **Average Turnaround Time** = (1 + 11 + 12 + 14) / 4 = 9.5

Round Robin (RR)

- 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 ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- **Avg turnaround time** = (134+37+162+121)/4 = 113.5
- **Average Waiting Time** = (81 + 20 + 94 + 97) / 4 = 73
- Typically, higher average turnaround than SJF, but better response.

Deadlock Management

Outline

- Deadlock: definition
- Conditions for Deadlock
- Deadlock examples
- Starvation
- Methods for handling deadlock
  - Ostrich algorithm
  - Deadlock detection and recovery
  - Deadlock prevention
  - Deadlock avoidance

Deadlock

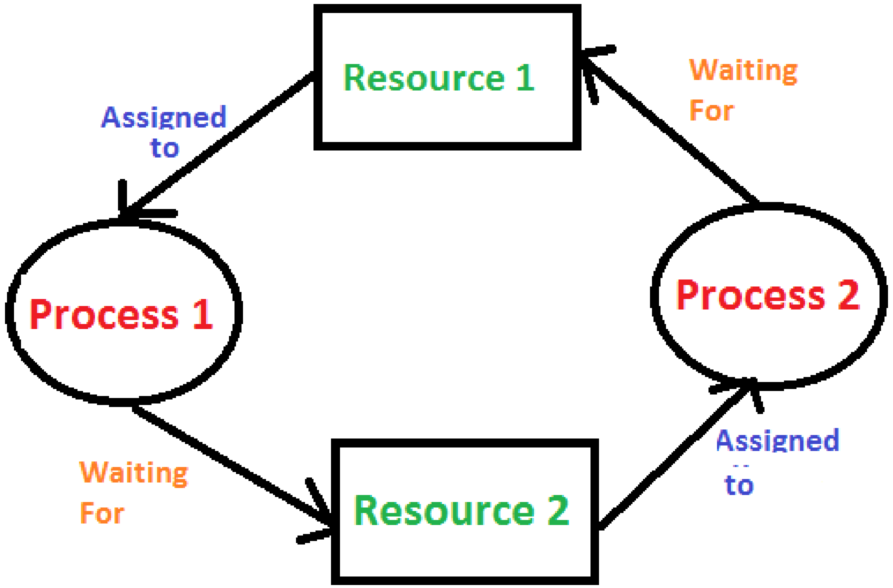
- For many applications, a process needs **exclusive** access to not one resource, but several.
- Suppose, for example, two processes each want to record a scanned document on a CD.
  - Process A requests permission to use the scanner and is granted it.
  - Process B is programmed differently and requests the CD recorder first and is also granted it.
- Now A asks for the CD recorder, but the request is **denied** until B releases it.
- Unfortunately, instead of releasing the CD recorder B asks for the scanner.
- At this point both processes are blocked and will remain so forever.
- This situation is called a **deadlock.**

Deadlock

- Deadlock can be defined formally as follows:
  - A *deadlock* is a situation where a set of processes is blocked because each process is holding a resource and waiting for another resource acquired by some other process. If a processes are in deadlock state:
    - None of the processes can run,
    - None of them can release any resources, and
    - all the processes continue to wait forever.

Deadlock …

For example, in the below diagram, **Process 1** is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and **process 2** is waiting for resource 1.



Conditions for Deadlock

- Four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Two or more resources are non-shareable (Only one process can use at a time).
2. Hold and wait condition. A process is holding at least one resource and waiting for another resources.
    - In this condition, a process holding one or more resources can still request additional resources without releasing its current ones. If these additional resources are unavailable, the process waits, potentially causing a deadlock if other processes are also holding resources and waiting indefinitely for new ones.

Conditions for Deadlock

Four conditions must hold for there to be a deadlock:

3.No preemption condition. A resource cannot be taken from a process unless the process releases the resource.

- Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them. It means that resources already assigned to a process cannot be forcibly removed; they must be voluntarily released by the process. This condition contributes to deadlocks, as a process may hold onto resources indefinitely while waiting for additional resources, preventing others from accessing them.

4. Circular wait condition. A set of processes waiting for each other in circular form.

- It means there's a closed loop where each process holds a resource that the next process in the chain needs. As each process waits for a resource held by another in the chain, none can proceed, creating a cycle that leads to a deadlock.

Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

Deadlock Examples

- examples
  - studying students
  - traffic intersection
- evaluation
  - four conditions: mutual exclusion, hold and wait, no preemption, circular wait

Studying Students

- studying students: both students need the textbook and the course notes to study, but there is only one copy of each
- consider the following situation:

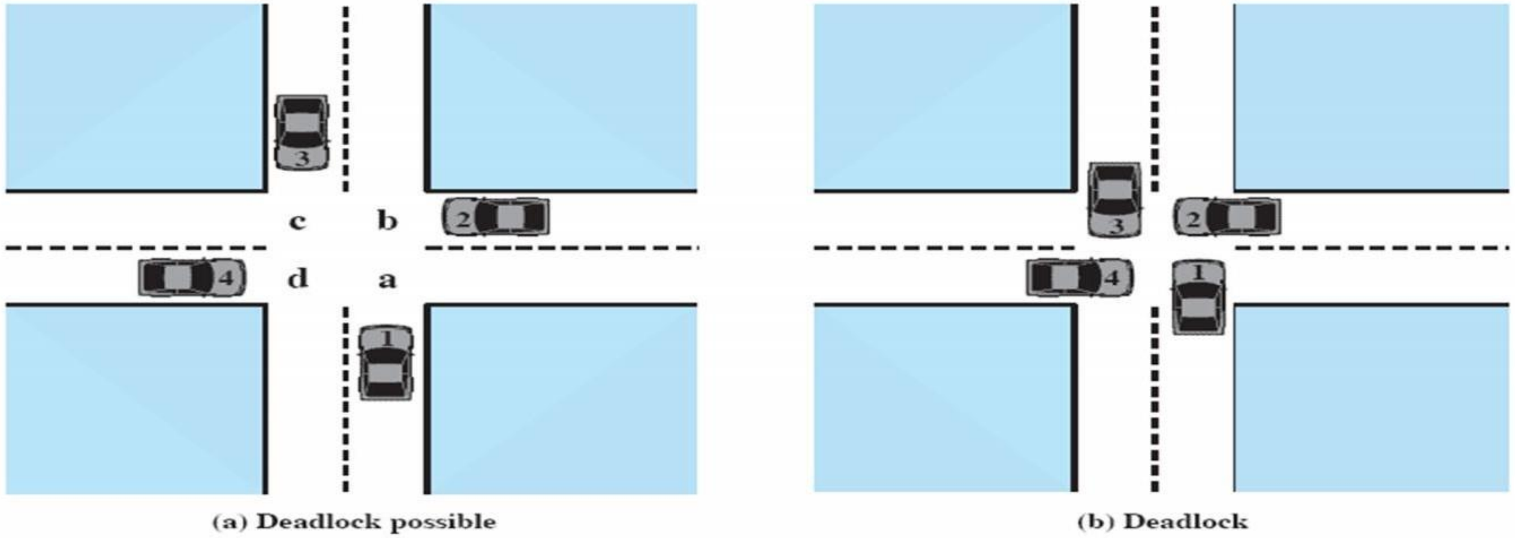
Student A	Student B
get coursenotes	**get** textbook
get textbook	**get** coursenotes
study	**study**
release textbook	**release** coursenotes
release coursenotes	**release** textbook

Students Evaluation

- mutual exclusion
  - books and course notes can be used only by one student
- hold and wait
  - a student who has the book waits for the course notes, or vice versa
- no preemption
  - there is no authority to take away book or course notes from a student
- circular wait
  - student A waits for resources held by student B, who waits for resources held by A

Traffic Intersection

- at a four-way intersection, four cars arrive simultaneously
- if all proceed, they will be stuck in the middle



(a) Deadlock possible (b) Deadlock

Traffic Evaluation

- mutual exclusion
  - cars can’t pass each other in the intersection
- hold and wait
  - vehicles proceed to the center, and wait for their path to be clear
- no preemption
  - there is no authority to remove some vehicles
- circular wait
  - vehicle 1 waits for vehicle 2 to move, which waits for 3, which waits for 4, which waits for 1

Starvation

- **Starvation** is a situation where a process can’t proceed because other processes always have the resources it needs.
    - the request of the process is never satisfied.
    - where higher-priority tasks or other processes continuously receive resources, leaving the affected process "starving" without access to the CPU, memory, or other necessary resources.
- in principle, it is possible to get the resource, but doesn’t because of
    - low priority of the process
    - timing of resource requests
    - ill-designed resource allocation or scheduling algorithm
- So, starvation is different from deadlock

Examples Starvation

Imagine there are three processes in a system:

- **Process A**: High-priority, short-running task
- **Process B**: Medium-priority, medium-running task
- **Process C**: Low-priority, long-running task

If **Process A** and **Process B** keep entering the queue repeatedly, **Process C** (the low-priority task) might never get a turn to use the CPU because the scheduler always favors higher-priority tasks. As a result, **Process C** "starves" and might never complete.

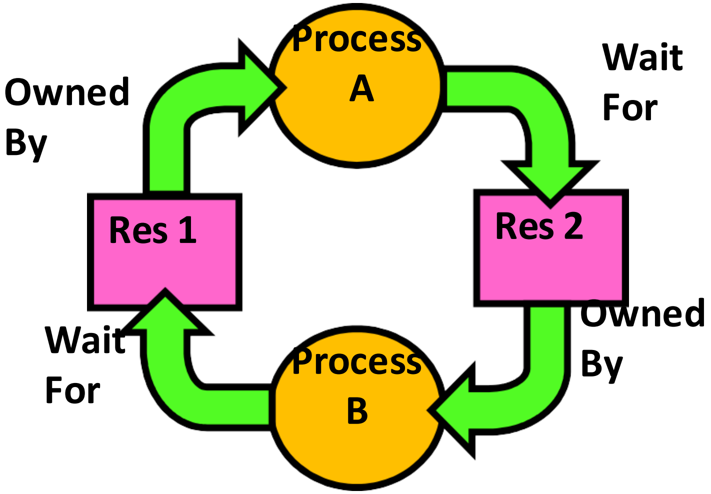
Solution Starvation

- fairness: each process gets its fair share of all resources it requests
- aging
  - the priority of a request is increased the longer the process waits for it.

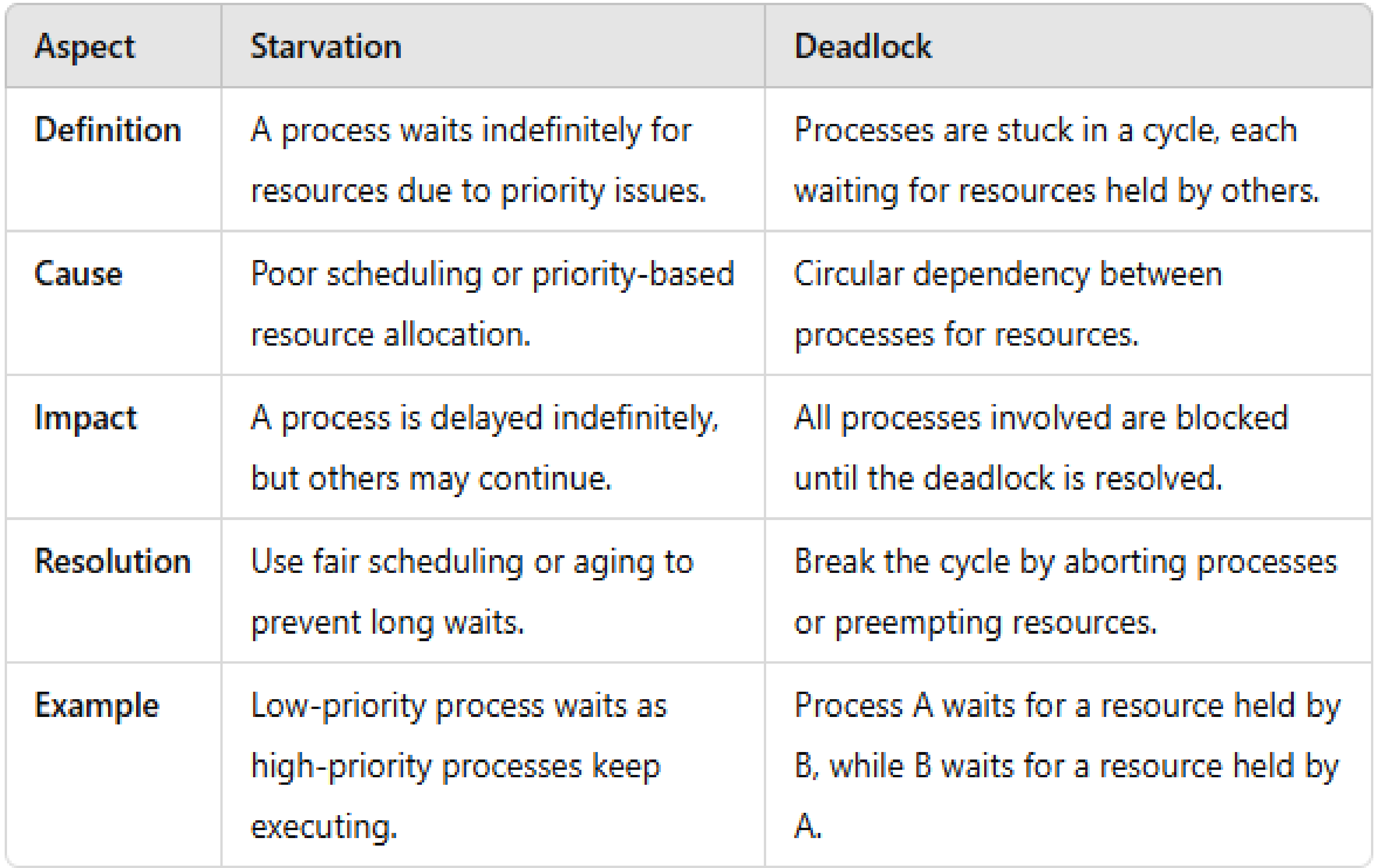
✓ The previous example process **C “starves” ,** so To avoid this an aging technique could be applied, which gradually increases **Process C's** priority over time, eventually making it high enough for the OS to allocate CPU time to it.

Starvation vs Deadlock

- Starvation: process waits indefinitely
  - Example, low-priority process waiting for resources constantly in use by high-priority processes
- Deadlock: circular waiting for resources
  - Process A owns Res 1 and is waiting for Res 2
  - Process B owns Res 2 and is waiting for Res 1



- Deadlock -> Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention



Aspect	Starvation	Deadlock
Definition	A process waits indefinitely for resources due to priority issues.	Processes are stuck in a cycle, each waiting for resources held by others.
Cause	Poor scheduling or priority-based resource allocation.	Circular dependency between processes for resources.
Impact	A process is delayed indefinitely, but others may continue.	All processes involved are blocked until the deadlock is resolved.
Resolution	Use fair scheduling or aging to prevent long waits.	Break the cycle by aborting processes or preempting resources.
Example	Low-priority process waits as high-priority processes keep executing.	Process A waits for a resource held by B, while B waits for a resource held by A.

Starvation is caused by **unfair scheduling**, while deadlock arises from **circular dependencies**.

Methods for handling deadlock

1 **Deadlock Ignorance** - the system "ignores" the possibility of deadlock entirely by restarting the system or terminating processes.

2 **Deadlock Detection and recovery** - manage deadlocks by **identifying** when they occur and **taking** corrective actions to resolve them.

3 **Deadlock avoidance:** seeks to prevent them from happening in the first place by carefully controlling how resources are allocated.

4 **Deadlock Prevention** - aims to structurally eliminate the possibility of deadlocks altogether by preventing one or more of the four necessary conditions for deadlock.

Deadlock Ignorance (THE OSTRICH ALGORITHM)

- When **storm** approaches, an ostrich puts his head in the sand(ground) and pretend (imagine) that there is no problem at all.
- Ignore the deadlock and pretend that deadlock never occur.
- Reasonable if
    - Deadlocks occur very rarely
    - Difficult to detect
    - Cost of prevention is high

Deadlock Detection and Recovery

Detection Algorithms

- Deadlock Detection with One Resource of Each Type
- Deadlock Detection with Multiple Resources of Each Type

Recovery from Deadlock

- Recovery through Preemption
- Recovery through Rollback
- Recovery through Killing Processes

Deadlock Detection with single Resource

- Assume that only one resource of each type exists.
    - Such a system might have one scanner, one CD recorder, one plotter, and one tape drive.
- We can construct a resource graph.
- If this graph contains one or more cycles, a deadlock exists.
- Any process that is part of a cycle is deadlocked.
- If no cycles exist, the system is not deadlocked.

Example

- Consider a system with seven processes, *A* though *G*, and six resources, *R* through *W.*

- The state are as follows:

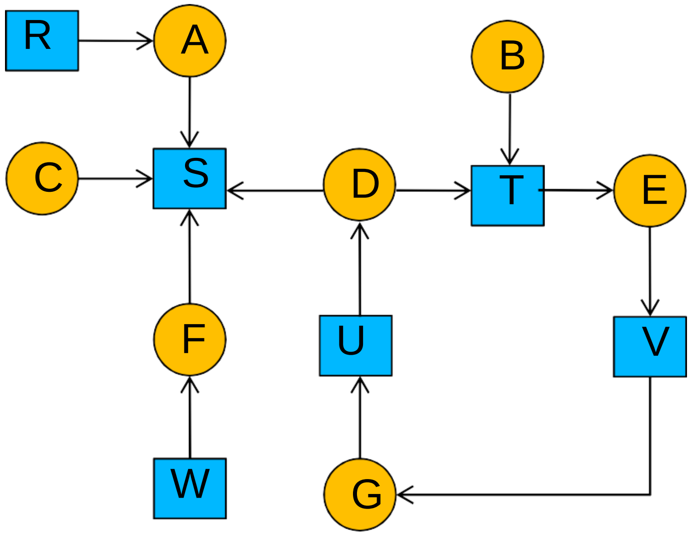
1. Process A holds R and wants S.
2. Process B holds nothing but wants T.
3. Process C holds nothing but wants S.
4. Process D holds U and wants S and T.
5. Process E holds T and wants V.
6. Process F holds W and wants S.
7. Process G holds V and wants U.

Question

- "Is this system deadlocked, and if so, which processes are involved?"

Answer

Construct wait for graph



Process *D*,*E*, and *G* are deadlocked

Deadlock Detection with Multiple Resources

- Matrix-based algorithm for detecting deadlock among n processes, P_1 through P_n.
- E is the existing resource vector. It gives the total number of instances of each resource in existence.
- For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives.
- Let A be the available resource vector, with A_i giving the number of instances of Resource i that are currently available (i.e unassigned).

Deadlock Detection with Multiple Resources…

- Two arrays: C - the current allocation matrix, and R - the request matrix.
- **C_{ij}** is the number of instances of resource j that are held by process i.
- **R_{ij}** is the number of instances of resource j that P_i wants.

Resource in existence
(E_1, E_2, E_3,… E_m)

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \dots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \dots & C_{2m} \\ \dots & \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & C_{n3} & \dots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

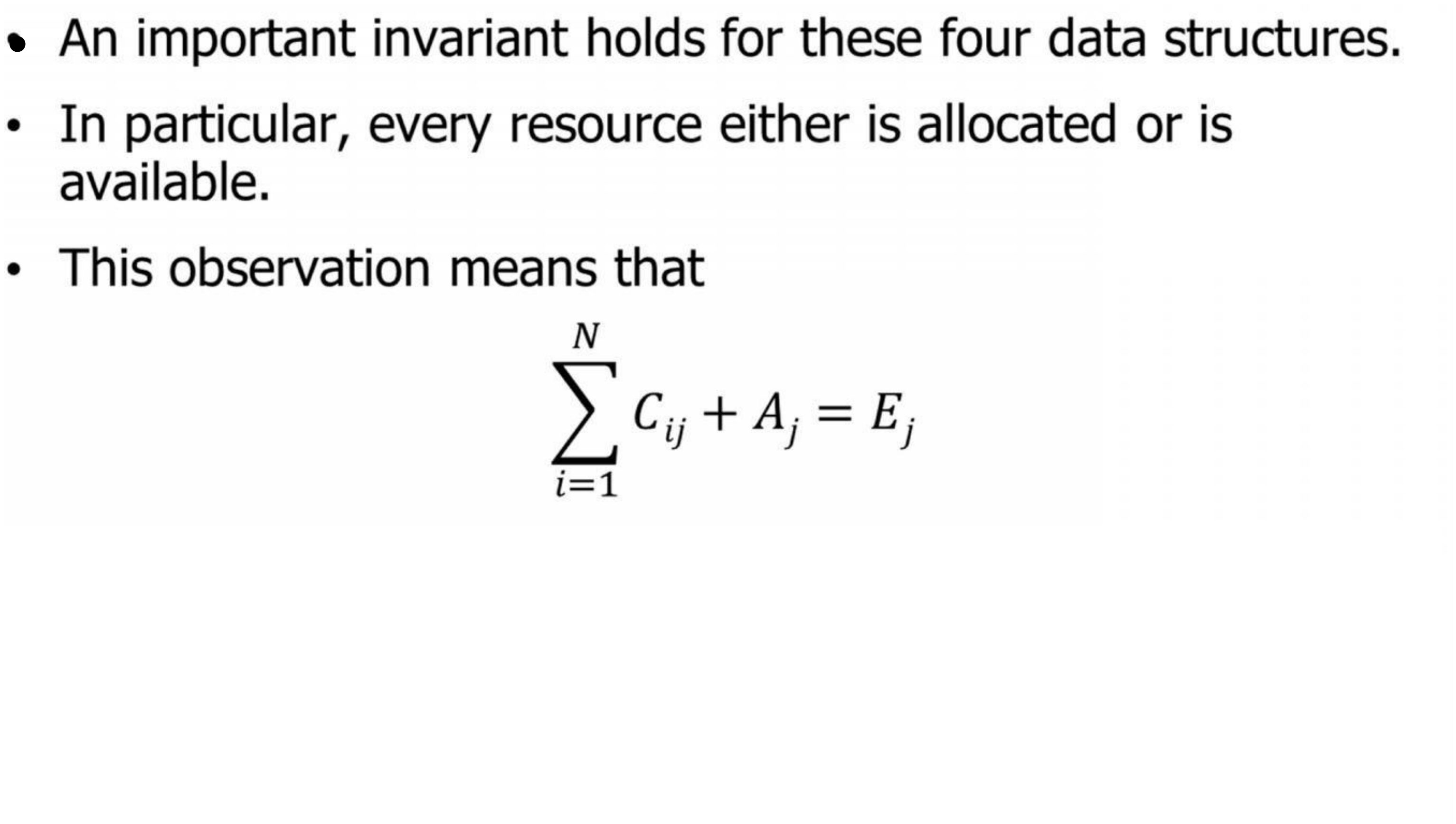
Resource available
(A_1, A_2, A_3,… A_m)

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \dots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \dots & R_{2m} \\ \dots & \dots & \dots & \dots & \dots \\ R_{n1} & R_{n2} & R_{n3} & \dots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Deadlock Detection with Multiple



- An important invariant holds for these four data structures.
- In particular, every resource either is allocated or is available.
- This observation means that

$$\sum_{i=1}^{N} C_{ij} + A_j = E_j$$

Deadlock Detection with Multiple Resources…

Algorithm

1. Look for an unmarked process, **P_i**, for which the i-th row of **R** is less than or equal to **A**.
2. If such a process is found, add the i-th row of **C** to **A**, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.
4. When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

Example

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (2 & 1 & 0 & 0) \end{array}$$

Current allocation matrix

$$\begin{array}{c|cccc|} P_1 & 0 & 0 & 1 & 0 \\ P_2 & 2 & 0 & 0 & 1 \\ P_3 & 0 & 1 & 2 & 0 \end{array}$$

Request matrix

$$\begin{array}{c|cccc|} P_1 & 2 & 0 & 0 & 1 \\ P_2 & 1 & 0 & 1 & 0 \\ P_3 & 2 & 1 & 0 & 0 \end{array}$$

Process 3 run first and return all its resources: A = (2 2 2 0)

Process 2 can run next and return its resources: A = (4 2 2 1)

Now process 1 can run. There is no deadlock in the system.

Recovery from Deadlock

- Suppose that our deadlock detection algorithm has succeeded and detected a deadlock.
- In this section, we will discuss various ways of recovering from deadlock.

Recovery from Deadlock…

- Roll back each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur
- Successively kill deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- 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.

Deadlock Prevention

- set of rules ensures that at least one of the four necessary conditions for deadlock doesn’t hold
  - mutual exclusion
  - hold and wait
  - no preemption
  - circular wait
- may result in low resource utilization, reduced system throughput

Deadlock Prevention…

1. Prevent the *circular-wait condition* by defining a linear ordering of resource types
   - A process can be assigned resources only according to the linear ordering (e.g., sequence number)
   - Disadvantages
     - Resources cannot be requested in the order that are needed
     - Resources will be longer than necessary
2. Prevent the *hold-and-wait condition* by requiring the process to acquire all needed resources before starting execution
   - Disadvantages
     - Inefficient use of resources
     - Reduced concurrency
     - Process can become deadlocked during the initial resource acquisition
     - Future needs of a process cannot be always predicted

Deadlock Prevention…

3. Denying No Preemption

- means that processes may be preempted by the OS
  - should only done when necessary
    - resources of a process trying to acquire another unavailable resource may be preempted
    - preempt resources of processes waiting for additional resources, and give some to the requesting process
- possible only for some types of resources
  - state must be easily restorable
  - e.g. CPU, memory

Deadlock Prevention …

1.e Use of time-stamps

- Example: Use time-stamps for transactions to a database – each transaction has the time-stamp of its creation
- The circular wait condition is avoided by comparing time-stamps: strict ordering of transactions is obtained, the transaction with an earlier time-stamp always wins
  - "**Wait-die**" method

```
if  [ e (T2)  <  e (T1) ]
     halt_T2 ('wait');
else
     kill_T2 ('die');
```

  - "**Wound-wait**" method

```
if  [ e (T2)  <  e (T1) ]
   kill_T1 ('wound');
else
   halt_T2 ('wait');
```

Timestamped Deadlock-Prevention Scheme

- Each process P_i is assigned a unique timestamp

- Timestamps are used to decide whether a process P_i should wait for a process P_j; otherwise P_i is rolled back.

- The scheme prevents deadlocks.
- For every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority *(lower timestamp)* than P_j.
- Thus a cycle cannot exist.

Wait-Die Scheme

- Based on a nonpreemptive technique.

- If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a smaller timestamp than does P_j (P_i is older than P_j).
- Otherwise, P_i is rolled back (dies).

- Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15 respectively.
  - if P_1 request a resource held by P_2, then P_1 will wait.
  - If P_3 requests a resource held by P_2, then P_3 will be rolled back (dies).

Wound-Wait Scheme

- Based on a preemptive technique; counterpart to the wait-die system.

- If P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a larger timestamp than does P_j (P_i is younger than P_j).
- Otherwise P_j is rolled back (P_j is wounded by P_i).

- Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15 respectively.
  - If P_1 requests a resource held by P_2, then the resource will be preempted from P_2 and P_2 will be rolled back.
  - If P_3 requests a resource held by P_2, then P_3 will wait.

Deadlock Avoidance

- **Basic Principle:** 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 to hold simultaneously. (maximum demand)
- 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

Deadlock Avoidance…

- The system must be able to decide whether granting a resource is **safe or not** and only make the allocation when it is safe.
- Thus, the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time?
- The answer is a qualified yes-we can avoid deadlocks.
- Algorithms
  - Safe and Unsafe States
  - The Bankers algorithm

Safe state

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- A state is **safe** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.

- We are considering a worst-case situation here.
- Even in the worst case (process requests up their maximum at the moment), we don’t have deadlock in a safe state.

Safe state…

- More formally: A system **state** is **safe** if there exists a *safe sequence* of all processes ($<P_1, P_2, \ldots, P_n>$)
- 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.

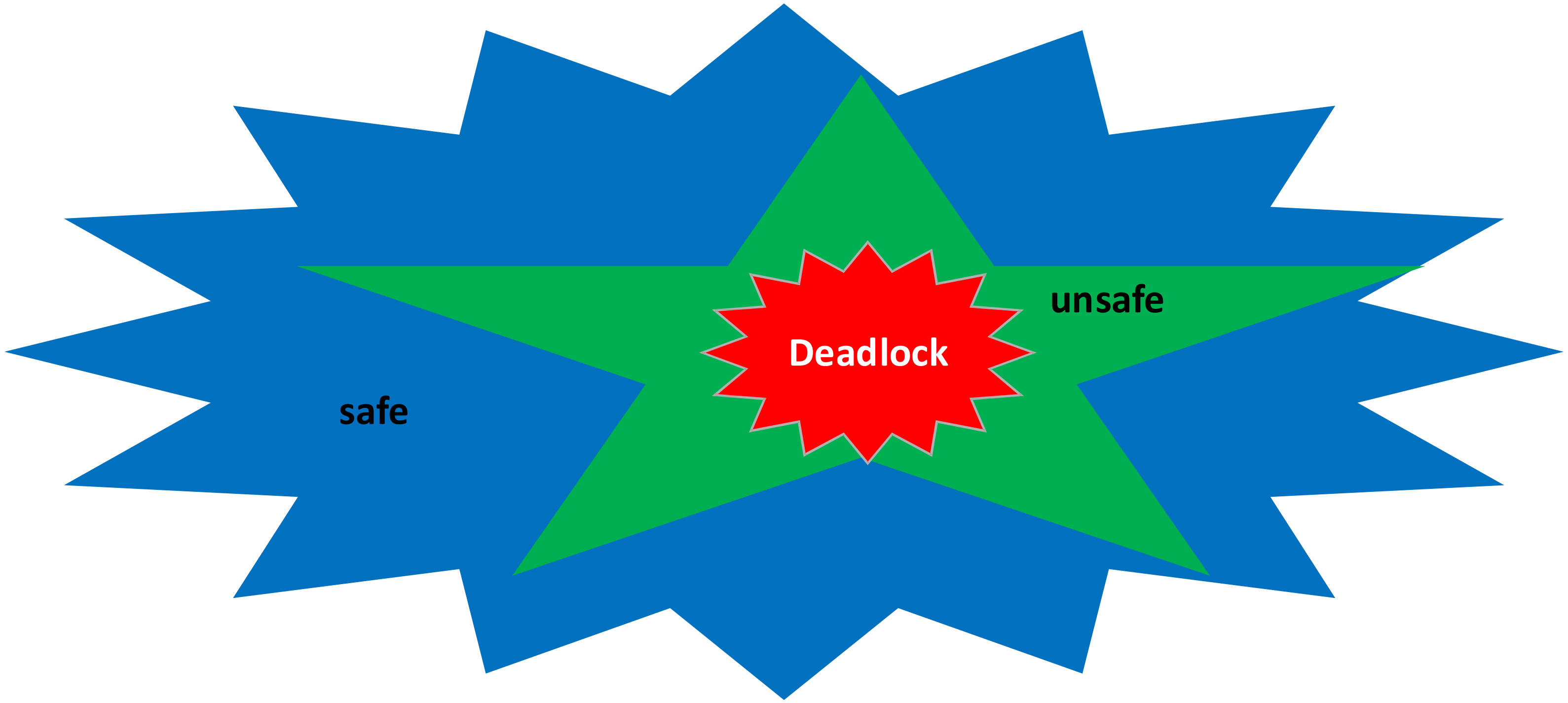
Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock

- Avoidance ⇒ **ensure that a system will never enter an unsafe state.**
  - When a request is done by a process for some resource(s):
  - check before allocating resource(s);
  - if it will leave the system in an unsafe state, then do not allocate the resource(s);
  - process is waited and resources are not allocated to that process.

Safe State Space

- if a system is in a safe state there are no deadlocks
- in an unsafe state, there is a possibility of deadlocks



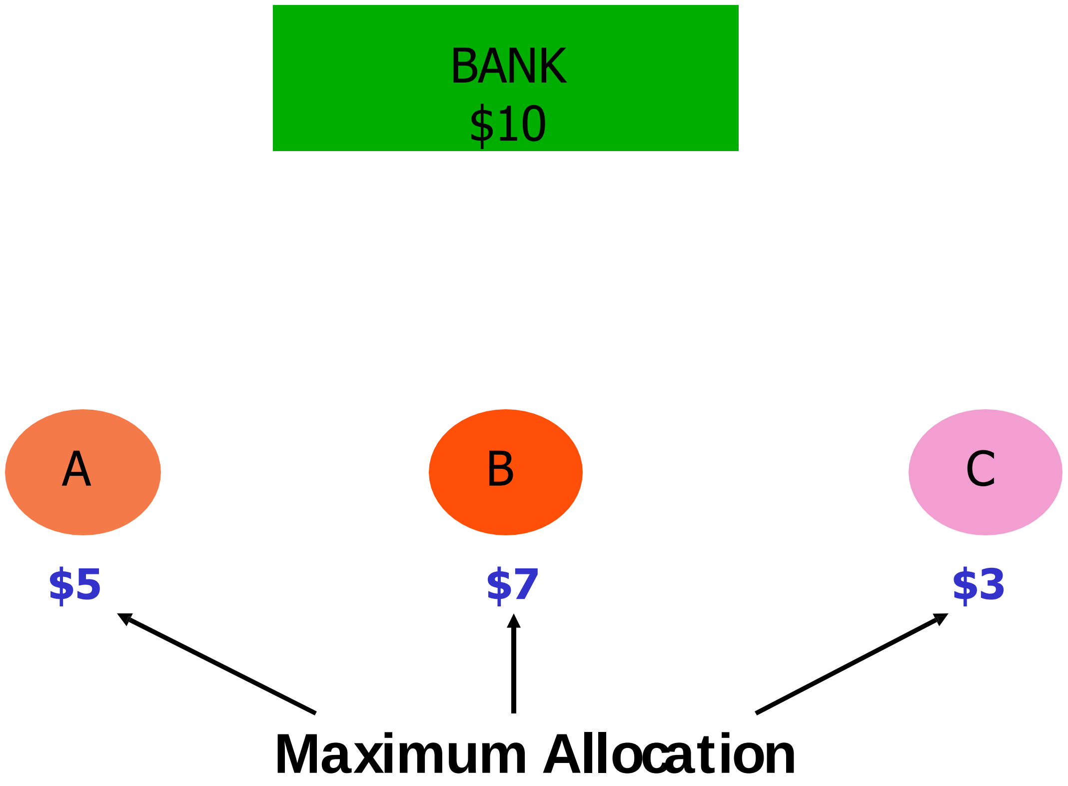
Deadlock Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker's algorithm

Example

- Bank gives loans to customers
  - maximum allocation = credit limit



BANK
$10

A

$5

B

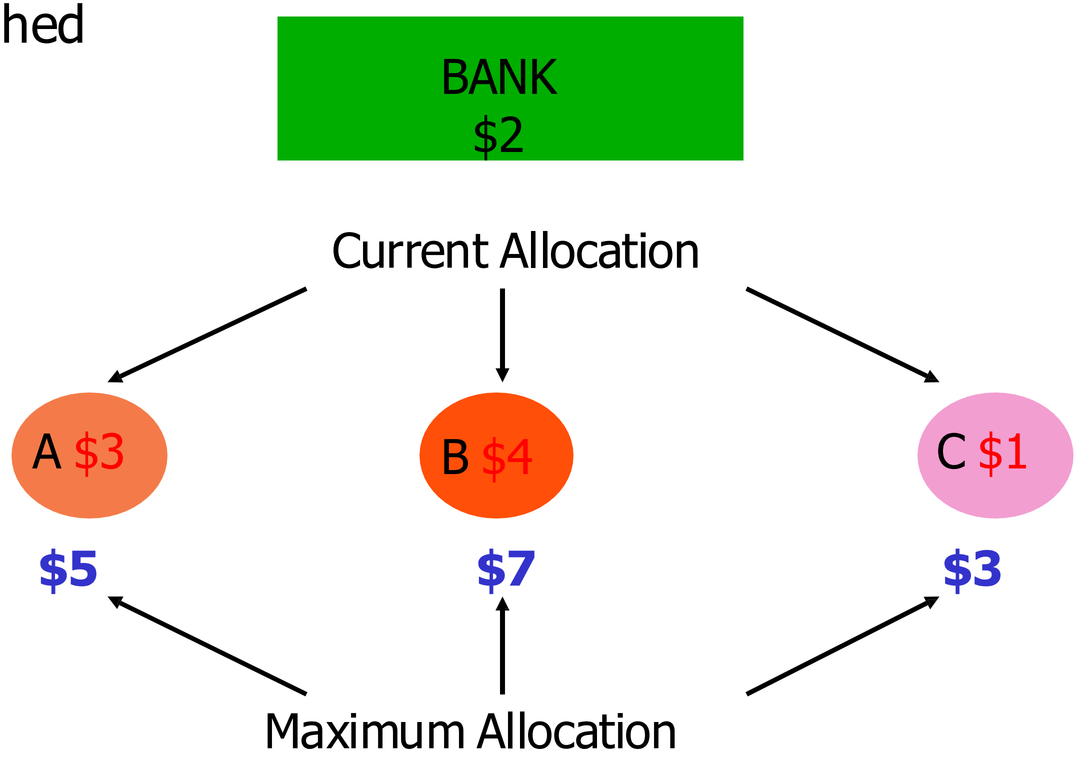
$7

C

$3

Maximum Allocation

- **Safe State?**
  - Will the bank be able to give each customer a loan up to the full credit limit?
    - not necessarily all customers simultaneously
    - order is not important
    - customers will pay back their loan once their credit limit is reached



BANK
$2

Current Allocation

A $3

B $4

C $1

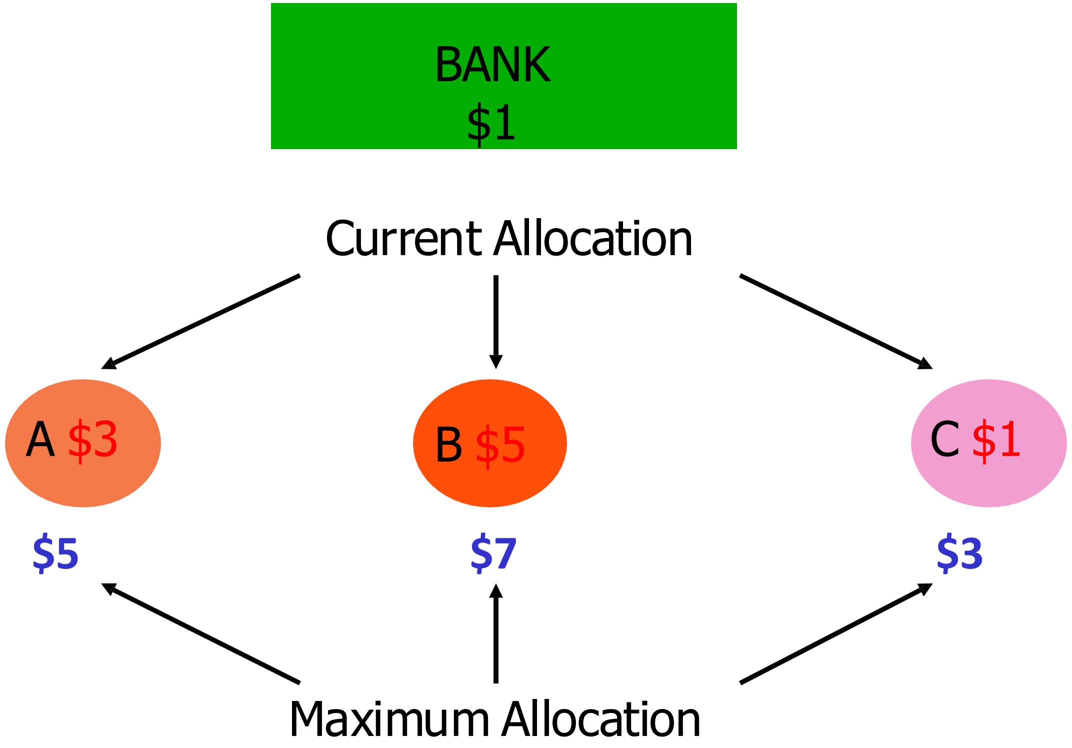
$5

$7

$3

Maximum Allocation

- **Still Safe?**
  - after customer B requests and is granted $1, is the bank still safe? NO



BANK
$1

Current Allocation

A $3

B $5

C $1

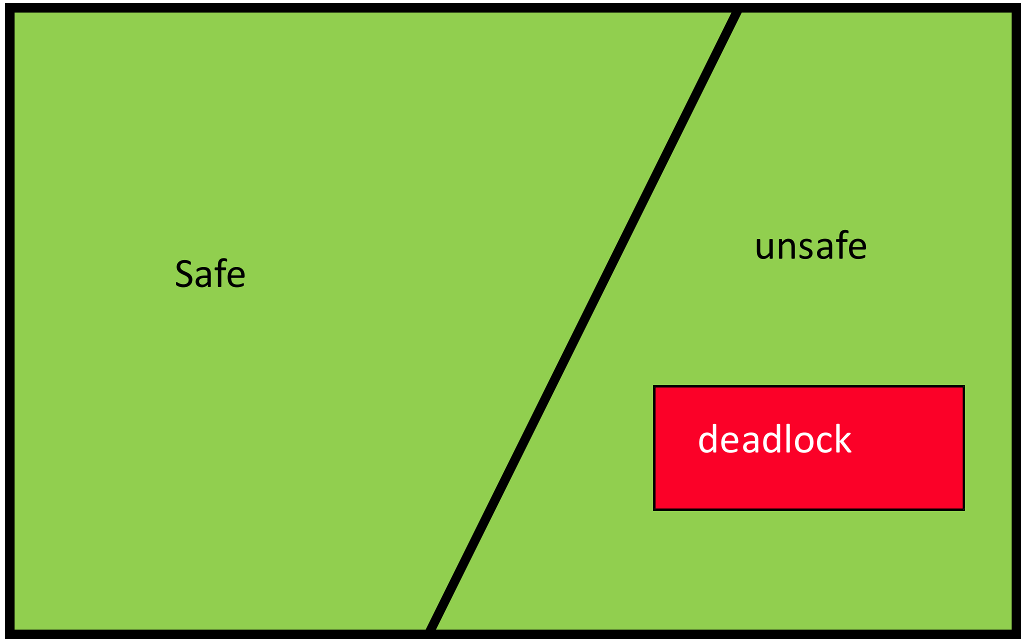
$5

$7

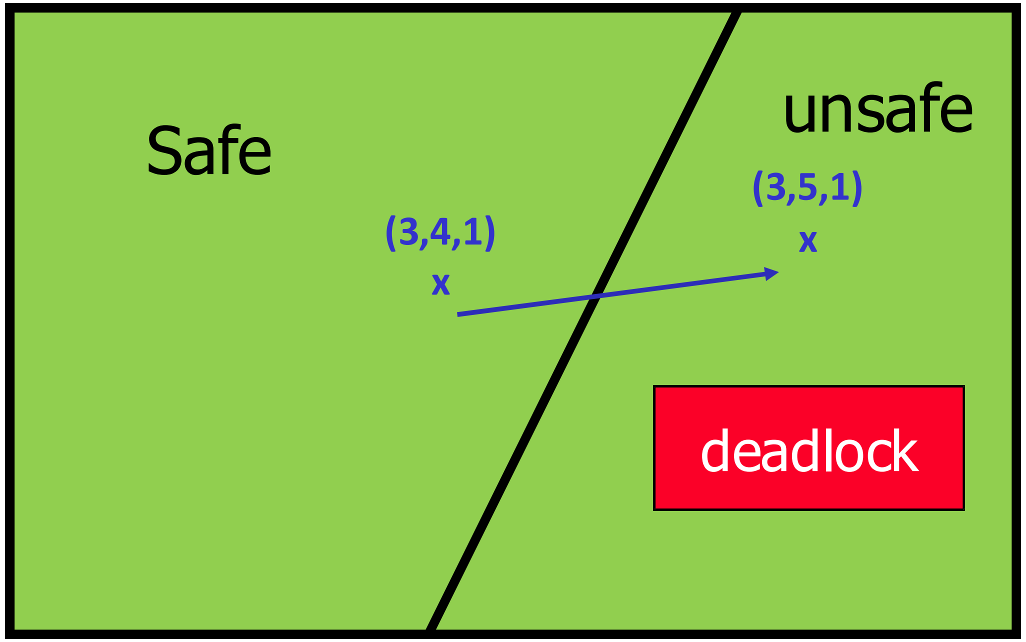
$3

Maximum Allocation

Safe State Space



Bank Safe State Space



Safe and Unsafe States: Example

- A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free. Is the state safe or not?

	Has	Max
A	3	9
B	2	4
C	2	7
Free = 3		

	Has	Max
A	3	9
B	4	4
C	2	7
Free = 1		

	Has	Max
A	3	9
B	0	--
C	2	7
Free = 5		

	Has	Max
A	3	9
B	0	--
C	7	7
Free = 0		

	Has	Max
A	3	9
B	0	--
C	0	--
Free = 7		

- Scheduler can run B first, then C and finally A.
- Thus, the state is **safe** because the system, by careful scheduling can avoid deadlock.

	Has	Max
A	3	9
B	2	4
C	2	7
Free = 3		

	Has	Max
A	4	9
B	2	4
C	2	7
Free = 2		

	Has	Max
A	4	9
B	4	4
C	2	7
Free = 0		

	Has	Max
A	4	9
B	--	--
C	2	7
Free = 4		

unsafe

The Banker's Algorithm

- before a request is granted, check the system's state
  - assume the request is granted
  - if it is still safe, the request can be honored
  - otherwise the process has to wait
  - overly careful
    - there are cases when the system is unsafe, but not in a deadlock

The Banker’s Algorithm: Single resource

- What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied.
- If granting the request leads to a safe state, it is carried out.
- The banker reserved 10 instead of 22.

SAFE

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7
Free = 10		

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7
Free = 2		

	Has	Max
A	1	6
B	1	5
C	4	4
D	4	7
Free = 0		

	Has	Max
A	1	6
B	1	5
C	--	--
D	4	7
Free = 4		

	Has	Max
A	1	6
B	5	5
C	--	--
D	4	7
Free = 0		

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7
Free = 10		

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7
Free = 2		

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7
Free = 1		

UNSAFE

Banker’s Algorithm: Multiple resource

- Each process must **a priori claim maximum use**
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and
m = number of resources types.

- **Available**: Vector of length m. If Available[j] == k, there are k instances of resource type R_j at the time deadlock avoidance algorithms is run.
- **Max**: n x m matrix. If *Max* [i,j] == k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If *Allocation* [i,j] == k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If *Need* [i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] – Allocation[i,j]$$

An example system state

All resources in the system

Existing A B C
10 5 7

Available A B C
10 5 7

Initially Available == Existing

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

system state at some t (may change)

Need = Max - Allocation

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
3 3 2

Notation

	X A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

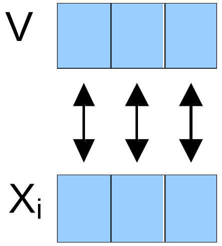
X is a matrix.

X_i is the ith row of the matrix: it is a vector.
For example, X_3 = [2 1 1]

V A B C
3 3 2

V is a vector; V = [3 3 2]

Compare two vectors:
Ex: compare V with X_i



$V == X_i$?
$V <= X_i$?
$X_i <= V$?
….

Ex: Compare [3 3 2] with [2 2 1]

[2 2 1] <= [3 3 2]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available* (initialize Work temporary vector)

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*-1

   (Work is a <u>temporary vector</u> initialized to the Available (i.e., free) resources at that time when the safety check is performed)

2. Find an *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available
[3 3 2]

4. If *Finish* [*i*] == true for all *i*, then the system state is safe; o.w. unsafe.

Resource-Request Algorithm for Process P_i

Request : request vector for process P_i.
If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j

Algorithm

1.If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \le Available$, go to step 3. Otherwise P_i must wait, since resources are not available

Resource-Request Algorithm for Process P_i

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request_i$; $Allocation_i = Allocation_i + Request_i$; $Need_i = Need_i - Request_i$;

Run the Safety Check Algorithm:

- *If safe* ⇒ *the requested resources are allocated to* P_i
- *If unsafe* ⇒ *The requested resources are not allocated to* P_i.

  P_i *must wait.*

  The old resource-allocation state is restored.

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types: A, B, and C

Existing Resources: *A* (10 instances), *B* (5 instances), and *C* (7 instances)

Existing = [10, 5, 7]

initially, Available = Existing.

Assume, processes indicated their maximum demand as follows:

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

Initially, Allocation matrix will be all zeros.
Need matrix will be equal to the Max matrix.

Example of Banker’s Algorithm

- Assume later, at an arbitrary time t, we have the following system state:

Existing = [10 5 7]

Need = Max - Allocation

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
3 3 2

Is it a safe state?

Example of Banker's Algorithm

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
3 3 2

Try to find a row in $Need_i$ that is <= Available.

P1. run completion. Available becomes = [3 3 2] + [2 0 0] = [5 3 2]
P3. run completion. Available becomes = [5 3 2] + [2 1 1] = [7 4 3]
P4. run completion. Available becomes = [7 4 3] + [0 0 2] = [7 4 5]
P2. run completion. Available becomes = [7 4 5] + [3 0 2] = [10 4 7]
P0. run completion. Available becomes = [10 4 7] + [0 1 0] = [10 5 7]

We found a sequence of execution: P1, P3, P4, P2, P0. State is safe

Example: P_1 requests (1,0,2)

- At that time Available is [3 3 2]
- First check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.
- Then check the new state for safety:

	Max A B C
P0	7 5 3
P1	3 2 2
P2	9 0 2
P3	2 2 2
P4	4 3 3

	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 3 0

new state (we did not go to that state yet; we are just checking)

Example: P_1 requests (1,0,2)

	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

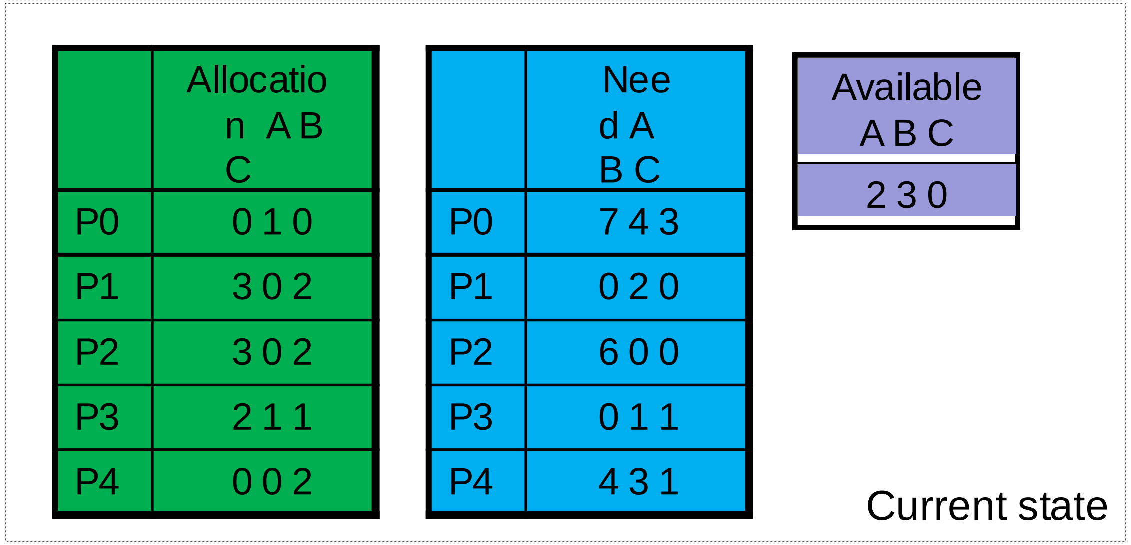
Available A B C
2 3 0

new state

Can we find a sequence?
Run P1. Available becomes = [5 3 2]
Run P3. Available becomes = [7 4 3]
Run P4. Available becomes = [7 4 5]
Run P0. Available becomes = [7 5 5]
Run P2. Available becomes = [10 5 7]

Sequence is:
P1, P3, P4, P0, P2
Yes, New State is safe.
We can grant the request.
Allocate desired resources to process P1.

P_4 requests (3,3,0)?



	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

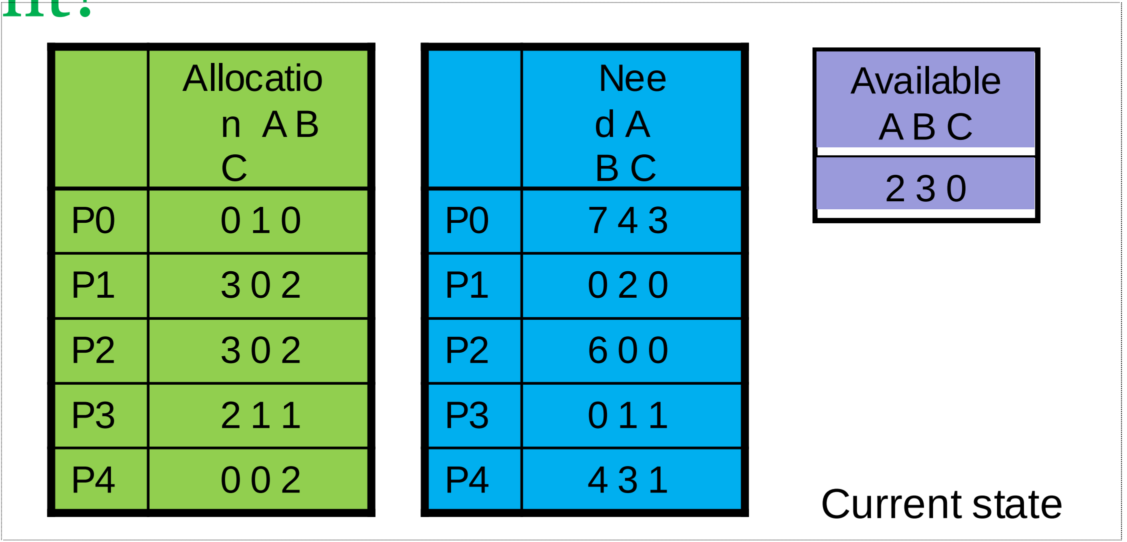
Available A B C
2 3 0

Current state

If this is current state, what happens if P4 requests (3 3 0)?

There is no available resource to satisfy the request. P4 will be waited.

P_0 requests (0,2,0)? Should we grant?



	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 4 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 3 0

Current state

System is in this state.
P0 makes a request: [0, 2, 0].   Should we grant.

P_0 requests (0,2,0)? Should we grant?

Assume we allocate 0,2,0 to P0. The new state will be as follows.

	Allocation A B C
P0	0 3 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

	Need A B C
P0	7 2 3
P1	0 2 0
P2	6 0 0
P3	0 1 1
P4	4 3 1

Available A B C
2 1 0

New state

Is it safe?

No process has a row in Need matrix that is less than or equal to Available.
Therefore, the new state would be **UNSAFE**.
Hence we should not go to the new state.
The request is not granted. **P0 is waited.**

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

  allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.