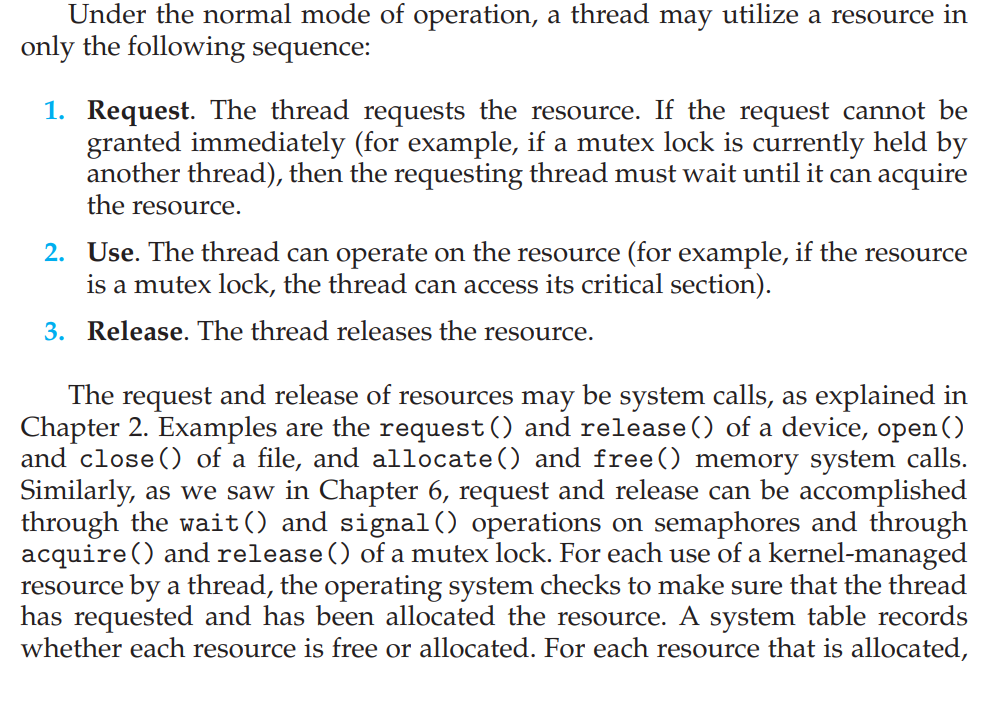
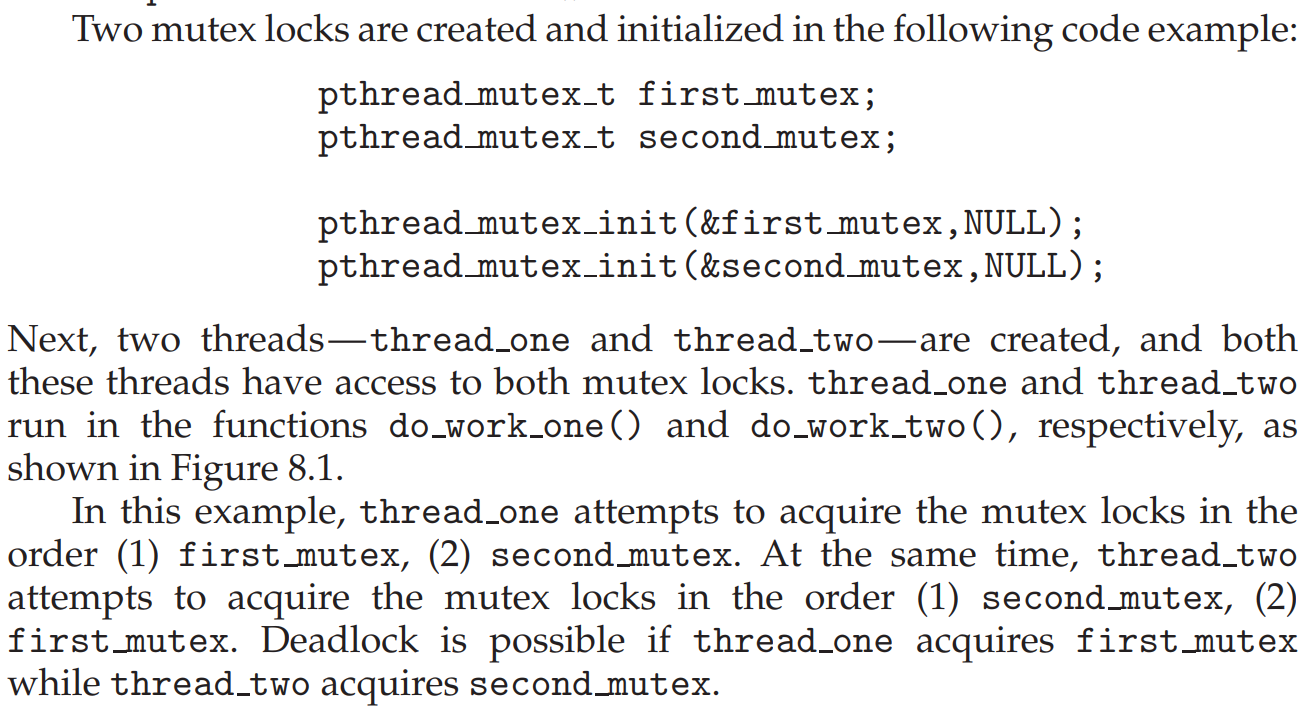
**1. System Model**

* Processes request and release resources (e.g., CPU cycles, memory, I/O devices).
* Resources are categorized into types, each with multiple instances.
* The system must manage resource allocation to prevent deadlocks.



**2. Deadlock With Mutex:**

A deadlock occurs when every member of a set of processes is waiting for an event that can only be caused by a member of the set.

****

A screenshot of a computer code

AI-generated content may be incorrect.

A close up of text

AI-generated content may be incorrect.

**Deadlock With Semaphore:**

A white paper with black text

AI-generated content may be incorrect.

1. **Thread T1​ executes wait(S1):** Since S1​ is initialized to 1, T1​ successfully acquires the semaphore. The value of S1​ becomes 0.
2. **Thread T2​ executes wait(S2):** Similarly, S2​ is initialized to 1, so T2​ acquires it. The value of S2​ becomes 0.

Now, here's where the trouble begins:

1. **Thread T1​ tries to execute wait(S2):** Since S2​ is currently 0 (because T2​ holds it), T1​ will block and wait for S2​ to become available.
2. **Thread T2​ tries to execute wait(S1):** Likewise, S1​ is 0 (because T1​ holds it), so T2​ will block and wait for S1​ to become available.

At this point, both threads are stuck! (**Deadlock Occurred**):

* T1​ is waiting for S2​, which is held by T2​.
* T2​ is waiting for S1​, which is held by T1​.

**4. Deadlock Characterization (Conditions for Deadlock)**

Deadlock occurs when a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. All four conditions must hold for a deadlock to occur **when there are multiple instances of a single resource** are:

1. **Mutual Exclusion**: There should be at least one shared resource or a critical section where only one process is allowed to access it at any given time.
2. **Hold and Wait**: A process is holding at least one resource and waiting to acquire additional resources held by other processes.
3. **No Preemption**: Resources cannot be forcibly removed from the processes holding them; that is, resources cannot be preempted. They must be released voluntarily by theholding process. (Not visibly identifiable)
4. **Circular Wait**: A set of processes are waiting for each other in a circular chain.

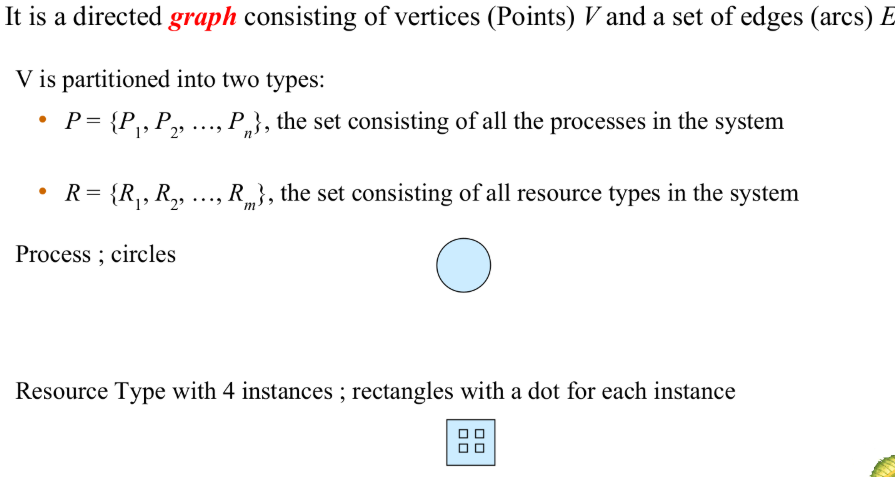
When there is only 1 instance of each resource in the system, if there exists a cycle, then there definitely exists a deadlock.

**Deadlock Problem 1:** The permanent blocking of a set of processes that either compete for a non-shareable resource or communicate with each other. It prevents sets of concurrent processes from completing their tasks.

* Processes require exclusive access to limited, non-sharable resources (e.g., mutex locks, printers, files). If each process in a set holds a resource that another process in the same set needs, a circular dependency can form, leading to deadlock.
* If a set of processes is waiting for messages from each other, and none of them can send a message until they receive one, they can become permanently blocked.

**Deadlock Problem 2:** A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set of processes Deadlock prevents sets of concurrent processes from completing their tasks

**5.Resource Allocation Graph (RAG):**



A diagram of a

AI-generated content may be incorrect.A white paper with black text and red text

AI-generated content may be incorrect.

A diagram of a graph with a cycle

AI-generated content may be incorrect.

**Basic Facts:  
A white background with red text

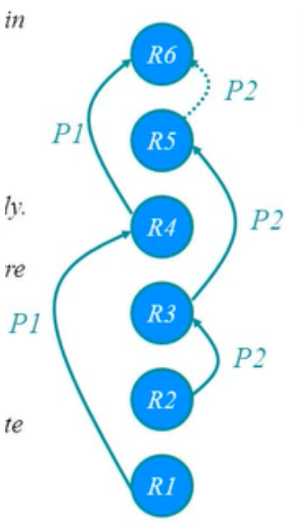
AI-generated content may be incorrect.**

**6. Methods for Handling Deadlocks**

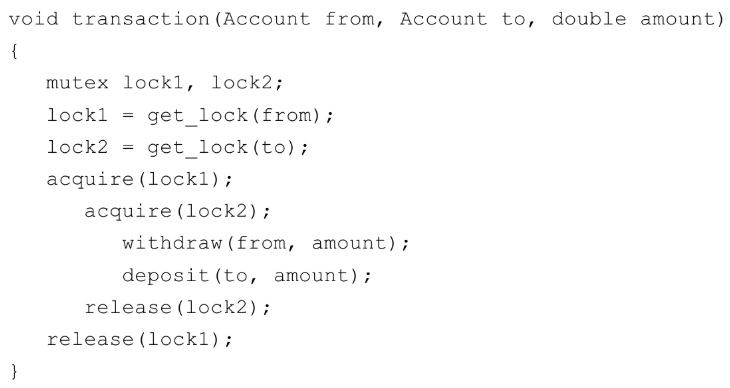
Operating systems employ various strategies to handle deadlocks:

* Method 1:
  + **Deadlock Prevention**: Design the system in a way that the possibility of deadlock is excluded by eliminating at least one of the four necessary conditions.
  + **Deadlock Avoidance**: The system makes decisions about resource allocation in real-time to ensure that a circular wait condition can never exist. OS uses additional knowledge; it keeps track of:
    - Maximum resources of each type a process will request
    - Currently available resources, and currently allocated resources
    - Future release of each process
    - Then decides if a process should wait, or if a request can be satisfied
* Method 2:
  + **Deadlock Detection and Recovery**: Allow the system to enter a deadlock state, detect it, and then recover.
* Method 3:
  + **Ignore the Problem**: In some systems, deadlocks are ignored; this is known as the ostrich algorithm.
* **Deadlock Prevention:**

Deadlock Prevention means making sure deadlocks never occur. By ensuring that at least one of four conditions cannot hold, we can prevent the occurrence of a deadlock.

* **Eliminate Mutual Exclusion:** If resources can be shared by multiple processes simultaneously, then processes needing the resource won't have to wait exclusively for it.
  + Many resources are inherently non-sharable (e.g., a CPU core for a specific critical section, a writable file being updated). For these, we cannot eliminate mutual exclusion.
* **Eliminate Hold and Wait:**
  + **Approach A: Request all necessary resources at the beginning of execution.** A process must request all the resources it will need *before* it starts executing. If all the requested resources are available, they are allocated to the process, and the process can proceed. If any of the required resources are unavailable, the process is blocked and must wait until all of them become free.
  + **Approach B: Release all currently held resources before requesting new ones.** If a process needs a resource and is currently holding others, it must first release all the resources it currently holds. Then, it can request all the resources it needs (including the ones it just released and the new one). The process is blocked until all requested resources become available.
  + **Starvation:** In both approaches, a process might release and then be unable to reacquire all the resources it needs if all of them are rarely available simultaneously.
* **Eliminate No Preemption:** Allow the operating system to forcibly take away resources from a process under certain circumstances.
  + **Approach:** If a process holding some resources requests another resource that cannot be immediately allocated, then all resources currently being held are released (preempted).
  + **Starvation:** If a process's resources are repeatedly preempted, it might never make progress.
* **Eliminate Circular Wait:** Impose a total ordering of all resource types and require that each process requests resources in an increasing order of enumeration.
  + If a process needs a resource with a lower number than one it already holds, it must first release the higher number resource.
  + **Example:** Suppose we have resources R1, R2, and R3 with indices 1, 2, and 3, respectively. A process can request R1, then R2, then R3. It can request R1 and then R3 (skipping R2 if not needed). However, if it holds R3 and later needs R1, it must first release R3.

**Deadlock Example with Lock Ordering**



Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A.

* **Deadlock Avoidance:**

For deadlock avoidance, the operating system typically requires the following **additional information** *in advance* **about each process**:

* **Maximum Resource Needs:** Before the process execution, each process must declare the maximum number of resources of each type that it might need during its entire execution. This provides an upper bound on the process's resource consumption.
* **Currently Allocated Resources:** The OS keeps track of the resources currently held by each process.
* **Currently Available Resources:** The OS knows the number of resources of each type that are currently free and available for allocation.

**A close-up of a white background

AI-generated content may be incorrect.**

**Safe State:** Deadlock avoidance works by ensuring that the system always remains in a **safe state**. A safe state is a state in which there exists at least one sequence of resource allocations to all processes that will allow all processes to complete their execution, even if they request their maximum number of resources. If the system is in a safe state, deadlock cannot occur.

**Unsafe State:** A system is in an unsafe state if there is no such safe sequence. An unsafe state does not necessarily mean that a deadlock *will* occur, but it means that the possibility of deadlock exists if processes make their maximum requests.

Deadlock avoidance requires the operating system to have additional information about how resources are to be requested. The **Banker's Algorithm** is a well-known **deadlock avoidance algorithm** that:

* **Single instance of a resource type**
  + Use a resource-allocation graph
* **Multiple instances of a resource type**
  + Use the banker’s algorithm
* **Deadlock Detection:**

If deadlock prevention and avoidance are not used, the system must provide:

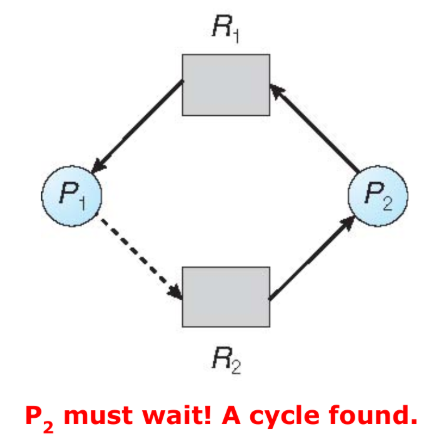
* An algorithm to detect deadlocks: This involves checking for cycles in the resource allocation graph.
* A recovery scheme: Once a deadlock is detected, the system must recover by terminating processes or preempting resources

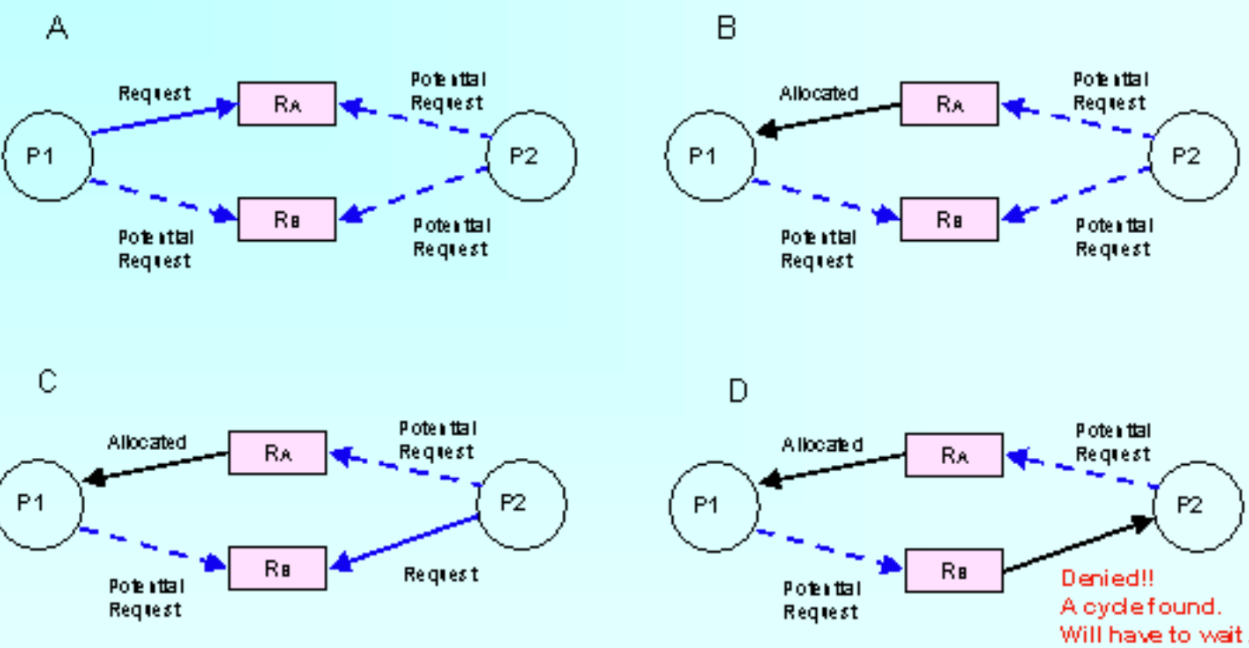
**Resource Allocation Graph (RAG)**

1. **Claim Edge:** Pointed towards the resource R that a process will use/request later. It is represented by a dashed line. When a resource is released by a process, the assignment edge reconverts to a claim edge.
2. **Request Edge:** Pointed towards the resource R that a process wants to use now. Claim edge converts to request edge when a process requests that resource.
3. **Assignment Edge:** Pointed towards the process from the resource R. Request edge is converted to an assignment edge when the resource is allocated to the process.

* A diagram of a diagram of a diagram

  AI-generated content may be incorrect.R2 is **claimed** by P1 and P2.
* R1 is **assigned** to P1.
* P2 is **requesting** R1.
* R2 is free, no process wants to use it right now.

****Now, if P2 requests R2, and that request is granted, then a cycle is formed. The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.

****

**Wait-For Graph (WG)**

**A diagram of a graph

AI-generated content may be incorrect.**

An edge Ti → Tj exists in a **wait-for graph** if and only if the corresponding resource-allocation graph contains two edges Ti → Rq and Rq → Tj for some resource Rq. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.

**Recovery from Deadlock**

**1. Process Termination**

* **Terminate all deadlocked processes:** This is the simplest approach but also the most drastic. It releases all held resources, breaking the deadlock. However, the terminated processes lose all their progress and must be restarted from the beginning.
* **Terminate one process at a time:** This is a more gradual approach. The system terminates one deadlocked process at a time, runs the deadlock detection algorithm after each termination, and continues until the deadlock is broken. This approach minimizes the amount of work lost but involves more overhead.
  + **Process Selection:** When choosing which process to terminate, the following factors can be considered:
    - Priority of the process
    - CPU time consumed by the process
    - Resources the process has used
    - Resources the process needs to complete
    - Number of processes to be terminated
    - Is the process interactive or batch?

**2. Resource Preemption**

* To eliminate deadlocks, the operating system can forcibly take resources away from some processes and give them to other processes.
* **Selecting a victim:**
  + The algorithm must select which resources are to be preempted and from which process.
  + This selection should aim to minimize the cost of preemption. Factors include:
    - Priority of the process holding the resource
    - The amount of time the process has used the resource
    - How much longer the process needs the resource
    - The type of resource (some resources are harder to preempt)
* **Rollback:**
  + If a resource is preempted from a process, the process cannot continue its execution as it lacks the resource.
  + The process must be rolled back to a safe state prior to acquiring the resource.
  + It must be restarted from that state.
* **Starvation:**
  + In resource preemption, it is possible that the same process may repeatedly have its resources taken away from it.
  + This situation, which prevents the process from progressing, is called starvation.
  + To avoid this, the algorithm can limit the number of times a process can be chosen as the victim.

**Banker’s Algorithm**

**It allows:**

* mutual exclusion
* wait and hold
* no preemption

**It prevents:**

* circular wait

**Banker’s algorithm comprises of Data Structures and two algorithms:**

1. Safety algorithm
2. Resource request algorithm

**Requirements**

* Evaluates resource allocation requests by simulating allocation for determining if it leads the system to a safe state.
* Only grants the request if the system remains in a safe state after allocation**.**
* Each process must in advanced 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

**Code:**

* **rows =** Number of Processes
* **cols =** Number of Resource Types
* **initialValue =** 0
* **vector<vector<int>>** mat**(rows, vector<int>(cols, initialValue))**;
* **vector<vector<int>> mat =** { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
* **int mat[rows][cols] =** { {7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3} }; **// 2d array**

1. **Maximum Matrix:** Represents the maximum number of resources each process might need throughout its execution.
2. **Allocation Matrix:** Shows the resources currently allocated to each process.
3. **Need Matrix:** The Need matrix is calculated as: **Need = Maximum - Allocation**. Each entry in the Need matrix represents the number of resources a process still requires to complete.
4. **Available Resources 1D Vector/Array:** Represents the number of resources currently available in the system. It is calculated as: **Available = Total Resources – Total Allocated**. Whereas **Total Resources** = Given, **Total Allocated** = Add all resources of each type separately.

**Safety Algorithm**

* Let **Work** and **Finish** be vectors of length **m** (No. of resource types) and **n** (No. of process), respectively. Initialize:
  + **Work = Available //The number of resources available with the system**
  + **Finish [i] = false for i = 0, 1, …, n- 1 //unfinished processes**
* Find an i such that both:
  + **Finish [i] = false**
  + **Need [i] ≤ Work //resources needed by the process i are less than the available resources**
  + **If no such i exists, go to step 4**
* **Work = Work + Allocation [i]** //assume Pi is finished then return resources back to the system
  + **Finish[i] = true //process is assumed to be finished.**
  + **go to step 2**
* If **Finish [i] == true for all i**, then the system is in a safe state. Otherwise, the processes whose index is **false** are involved in a **deadlock**.

**Resource-Request Algorithm for Process Pi**

**Request [i]** = request vector for process Pi.

If **Request [i] [j] = k** then process **Pi** wants **k** instances of resource type **Rj.**

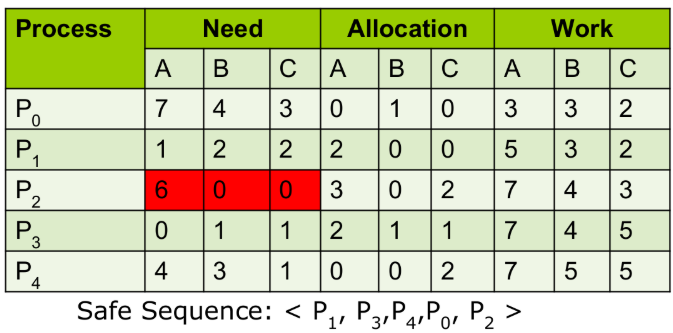
1. If **Request [i] ≤ Need [i]** go to step 2. Otherwise, **raise error condition**, since process has exceeded its maximum claim.

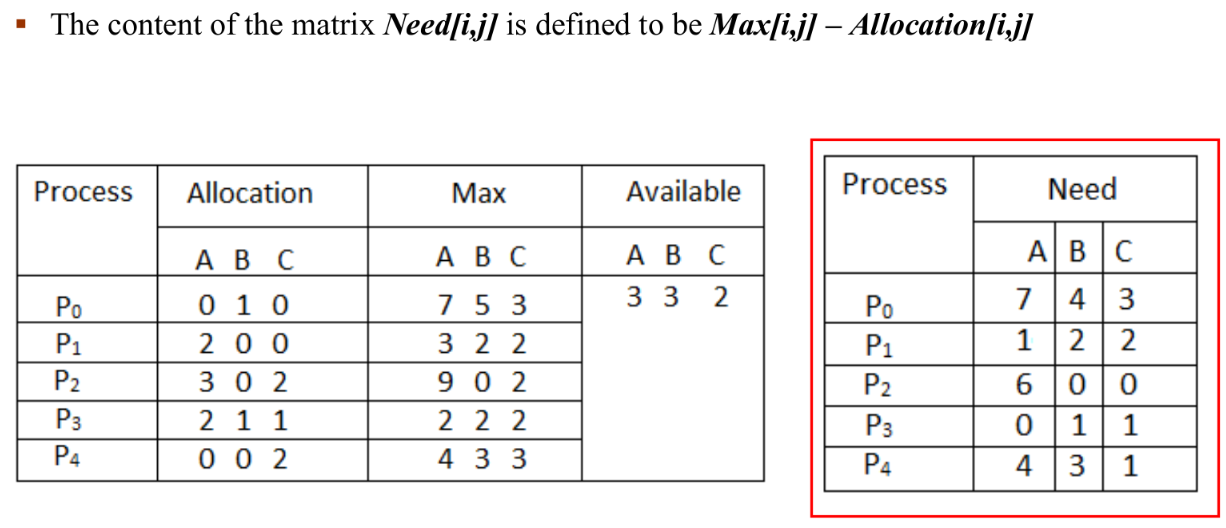
2. If **Request [i] ≤ Available**, go to step 3. Otherwise **Pi** must wait, since resources are not available.

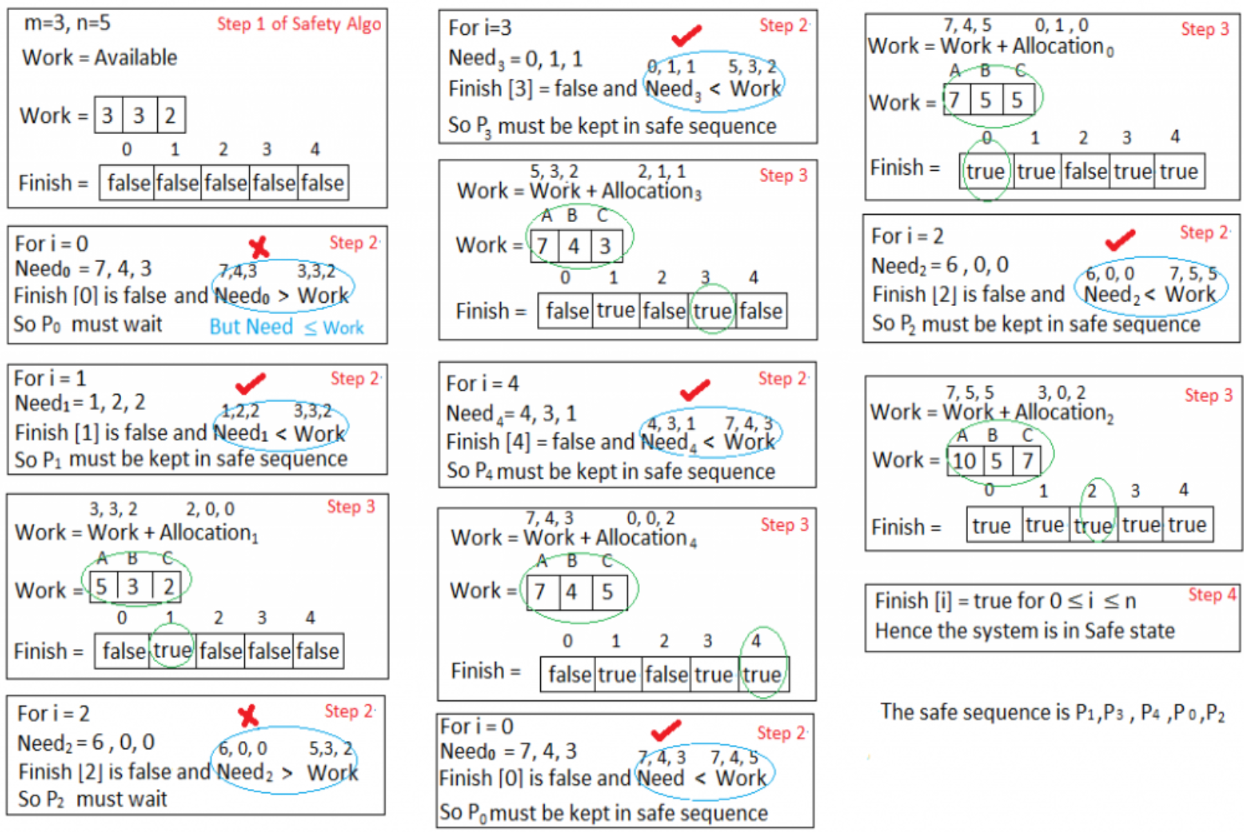
3. Pretend to (tentatively) allocate requested resources to **Pi** by modifying the state as follows:

* **Need [i] = Need [i] – Request [i];**
* **Available = Available – Request [i];**
* **Allocation [i] = Allocation [i] + Request [i];**
* Applying safety algorithm, if safe ⇒ the resources are allocated to **Pi**
* If unsafe ⇒ **Pi** must wait. and the **old resource-allocation state is restored**. The requested resources are not allocated to **Pi**.

**-- Safety Algorithm Problems --**

****

****

****

**-- Resource-Request Algorithm Problems --**

**Question:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Allocation** | **Max** | **Need** | **Available** |
|  | **ABC** | **ABC** | **ABC** | **ABC** |
| **T0** | **010** | **753** | **743** | **332** |
| **T1** | **200** | **322** | **122** |  |
| **T2** | **302** | **902** | **600** |  |
| **T3** | **211** | **222** | **011** |  |
| **T4** | **002** | **433** | **431** |  |

We claim that the system is currently in a safe state. Indeed, the **sequence <T1, T3, T4, T2, T0>** satisfies the safety criteria. Steps to fulfill a process request:

1. Suppose now that thread T1 requests one additional instance of resource type A and two instances of resource type C, so **Request1 = (1,0,2)**.
2. First check that **Request1 ≤ Available**, then **Request1 ≤ Need1.**
3. We then **pretend** that this request has been fulfilled, and we arrive at the following new state:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Allocation** | **Max** | **Need** | **Available** |
|  | **ABC** | **ABC** | **ABC** | **ABC** |
| **T0** | **010** | **753** | **743** | **230** |
| **T1** | **302** | **322** | **020** |  |
| **T2** | **302** | **902** | **600** |  |
| **T3** | **211** | **222** | **011** |  |
| **T4** | **002** | **433** | **431** |  |

1. We must determine whether this new system state is safe. To do so, we **execute our safety algorithm** and find that the sequence <T1, T3, T4, T0, T2> satisfies the safety requirement. Hence, we can **immediately grant the request** of thread T1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by T4 cannot be granted, since the resources are not available. Furthermore, a request for (0,2,0) by T0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

**A diagram of a work flow

AI-generated content may be incorrect.**