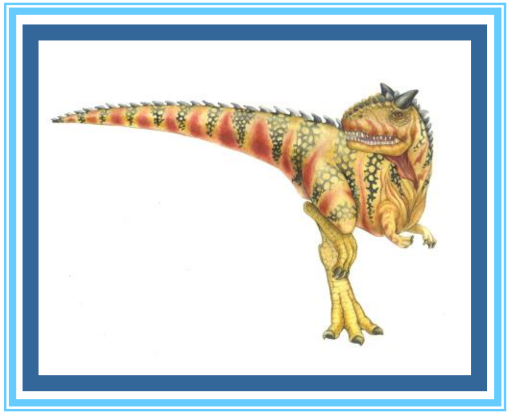
Chapter 8: Deadlocks

Chapter 8: Section 8.1-8.6



System Model

- System consists of resources
- Resource types R_1, R_2, . . ., R_m
  - Physical resource types: CPUs, memory, I/O devices
  - Logical resource types: files, mutex locks, semaphores
- Each resource type R_i has one or more identical instances
- Each thread utilizes a resource as follows:
  - **Request –** If the request cannot be granted immediately, then the thread must wait until it can acquire the resource
  - **Use**
  - **Release**

What is a Deadlock?

- A **deadlock** is a situation involving a set of threads in which each thread waits for an event that can be caused only by another thread in the set

- Deadlock example when using semaphores

  Two threads share semaphores A and B, both initialized to 0

T_1	T_2
wait (A);	wait(B);
…	…
signal(B);	signal(A);

8.3 3

Deadlock when Using Mutex Locks

- Two mutex locks are created and initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

- Deadlock is possible if thread 1 acquires **`first_mutex`** and thread 2 acquires **`second_mutex`**. Thread 1 then waits for **`second_mutex`** and thread 2 waits for **`first_mutex`**.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

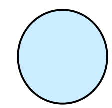
Deadlock Characterization

Deadlock occurs when four conditions hold simultaneously

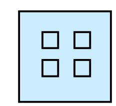
- **Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is, only one thread at a time can use the resource
- **Hold and wait:** A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads
- **No preemption:** A resource can be released only voluntarily by the thread holding it, after that thread has completed its task
- **Circular wait:** There exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting threads such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, …, T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Resource-Allocation Graph

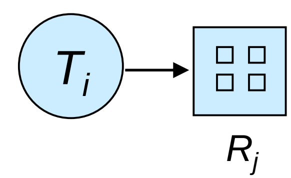
- We can use resource-allocation graphs to study deadlocks
- A resource-allocation graph has a set of vertices V and a set of edges E
- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all threads in the system



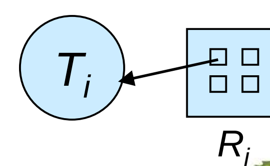
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system



- E is partitioned into two types:
  - **Request edge** – directed edge $T_i \rightarrow R_j$ indicating thread T_i has requested an instance of resource type R_j

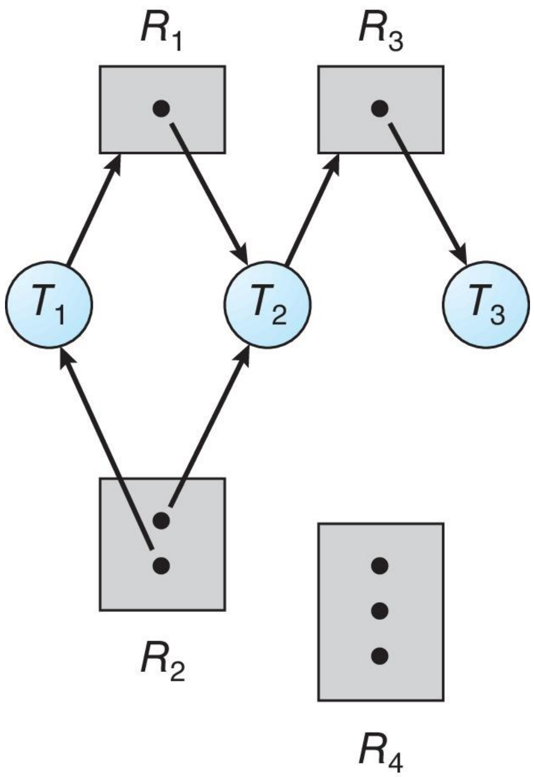


  - **Assignment edge** – directed edge $R_j \rightarrow T_i$ indicating an instance of resource type R_j has been allocated to thread T_i

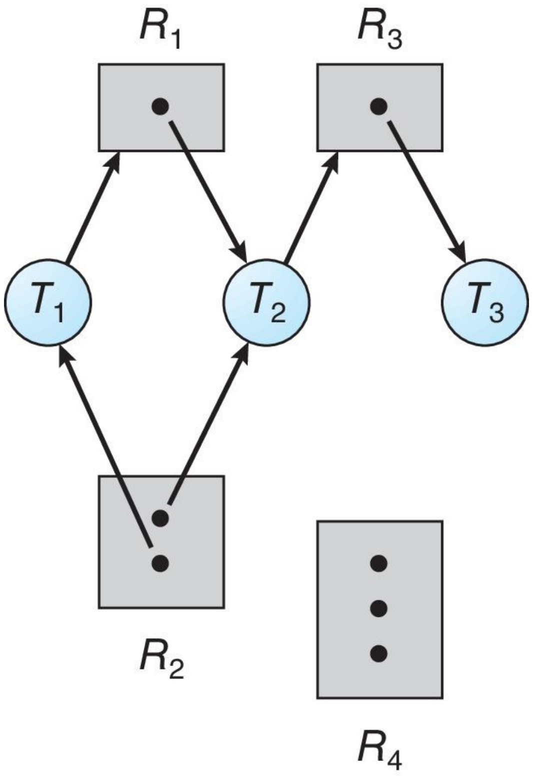


Resource Allocation Graph Example

- 1 instance of R1
- 2 instances of R2
- 1 instance of R3
- 3 instances 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

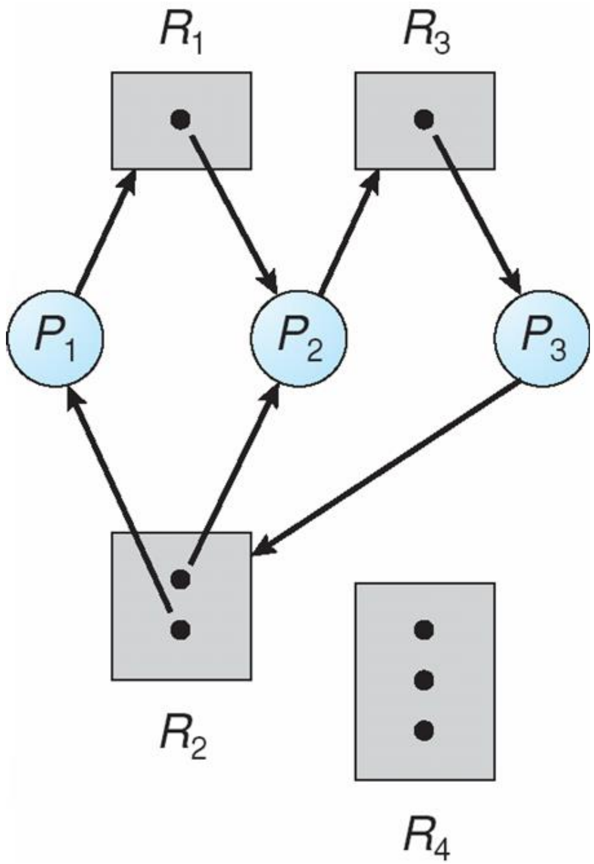


Resource Allocation Graph with No Deadlock



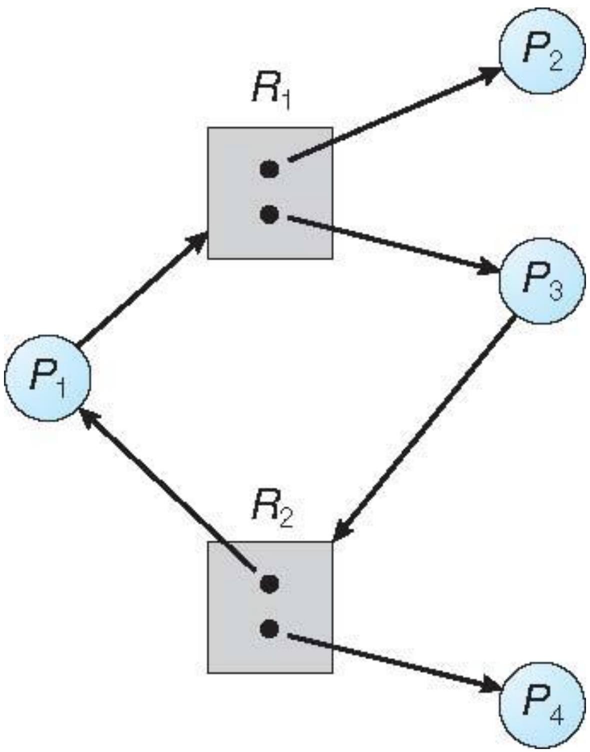
If graph contains no cycles, then there is no deadlock

Resource Allocation Graph With A Deadlock



- Processes P1, P2, and P3 are deadlocked
- Does a cycle imply a deadlock?
  Answer is No!

Graph With A Cycle But No Deadlock



When P_4 releases an instance of R_2, that resource can be allocated to P_3, breaking the cycle

Basic Facts

- If a resource-allocation graph contains no cycles $\Rightarrow$ no deadlock
- If a resource-allocation graph contains a cycle, then the system **may** or **may not** be in a deadlocked state
  - If there is only one instance per resource type $\Rightarrow$ deadlock
  - If there are several instances per resource type $\Rightarrow$ possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Deadlock detection and recovery:
  - Allow the system to enter a deadlock state
  - Abort a process or preempt some resources when a deadlock is detected
- Ignore the problem all together
  - Simply let deadlocks happen and reboot as necessary
  - Used by most operating systems, including Linux and Windows

Deadlock Prevention

Ensure that one of the four necessary conditions for deadlock cannot hold

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources (e.g., mutex locks)
- **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
  - Drawbacks: low resource utilization, starvation possible

Deadlock Prevention (Cont.)

- **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 preempted
  - 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
  - Some resources can be preempted (e.g., CPU, memory space); some resources cannot be preempted (e.g., mutex locks)

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Denying Circular Wait

- Simply assign each resource a unique number
- Resources must be acquired in an increasing order of enumeration
- For example:

F(first_mutex) = 1
F(second_mutex) = 5

code for **thread_two** cannot

be written as this:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Deadlock Avoidance

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

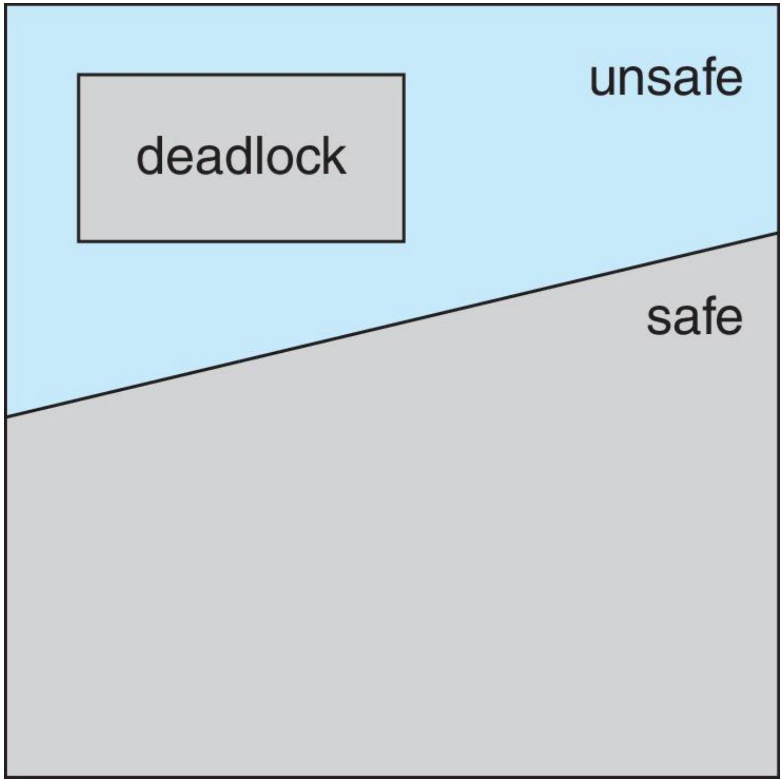
- Each process declares the ***maximum number*** of resources of each type that it may need
- Whenever a process requests a resource that is currently available, the system decides whether the resource can be allocated immediately or whether the process must wait
- The system makes the decision by considering the resource-allocation state of the system
  - **Resource-allocation state** is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- 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 safe sequence
  - A sequence of processes <$P_1, P_2, \ldots, P_n$> is a **safe sequence** for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all P_j, with $j < i$
- That is:
  - If P_i's 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 $\Rightarrow$ 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



Safe State Example

- Consider a system with 12 resources and 3 processes
- System state at t_0:

	Maximum Needs	Current Allocation	Available = 3
P_0	10	5	
P_1	4	2	
P_2	9	2	

The system is in a safe state at t_0 because < P_1 , P_0 , P_2 > is a safe sequence

- Suppose at t_1 P_2 requests and is allocated one more resource

	Maximum Needs	Current Allocation	Available = 2
P_0	10	5	
P_1	4	2	
P_2	9	3	

The system is no longer in a safe state at t_1

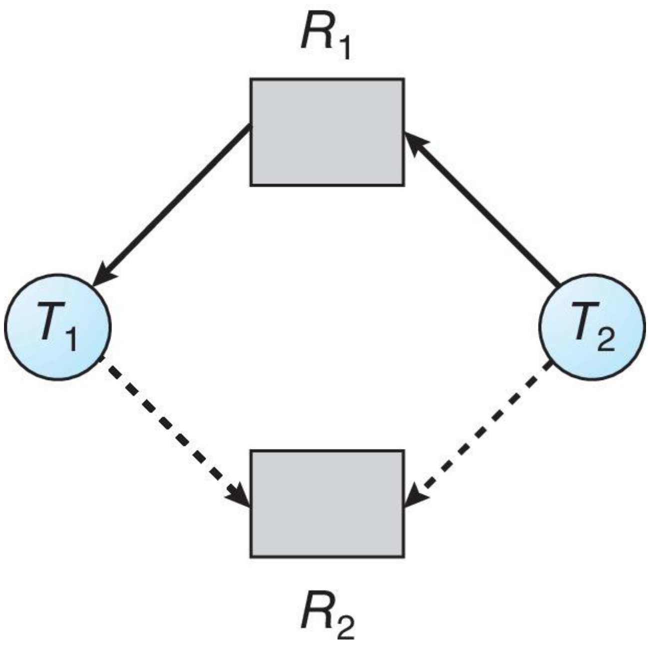
Deadlock Avoidance Algorithms

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

- Multiple instances of a resource type
  - Use the Banker's Algorithm

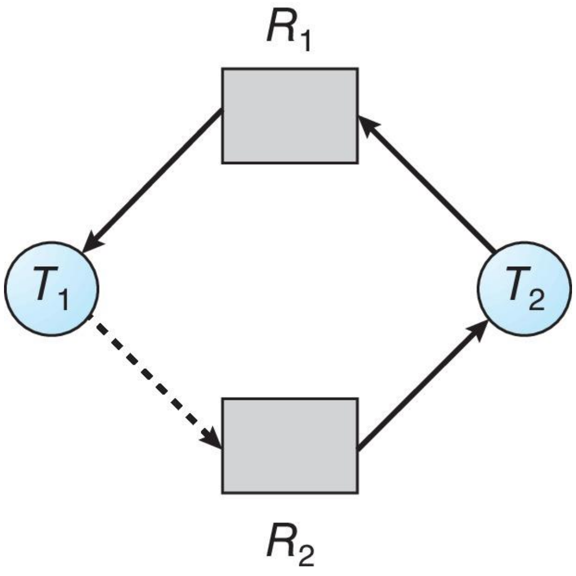
Resource-Allocation Graph Scheme

- **Claim edge** $T_i \rightarrow R_j$ indicates that thread T_i may request resource R_j at some time in the future
  - Claim edge represented by a *dashed line*
- Claim edge is converted to request edge when a thread requests a resource
- Request edge is converted to an assignment edge when the resource is allocated to the thread
- When a resource is released by a thread, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



Resource-Allocation Graph Algorithm

- Suppose that thread T_i requests a resource R_j
- 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
- Example: Suppose T_2 requests R_2
  - Allocating R_2 to T_2 will create a cycle in the graph. Thus, the request cannot be granted.



Banker's Algorithm

- Applicable to a system with multiple instances of each resource type
- Each process must a priori claim maximum use of each resource type
- When a process requests a resource that is available, it may have to wait

Data Structures for the Banker's Algorithm

Let n = number of processes, m = number of resource types.

- **Available**: Vector of length m
  - If available [j] = k, there are k instances of resource type R_j available

  Example: *Available*

A	*B*	*C*
3	5	4

- **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

  Example: *Max*

	A	*B*	*C*
*P*0	3	4	0
*P*1	2	0	3

Data Structures for the Banker's Algorithm

- **Allocation**: n x m matrix.
  - If Allocation[i,j] = k then P_i is currently allocated k instances of R_j

Example: *Allocation*

	A	*B*	*C*
*P*0	2	3	0
*P*1	2	0	1

- **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]

Example: $\text{Need}_0 = \text{Max}_0 - \text{Allocation}_0 = (3,4,0) - (2,3,0) = (1,1,0)$

Safety Algorithm

The algorithm determines whether or not a system is in a safe state.

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

   Work = Available

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

2. Find an ***i*** such that both:

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

   (b) $\boldsymbol{Need_i \leq Work}$

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

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

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Example of Safety Algorithm (1)

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5 instances), and C (7 instances)

- Snapshot at time t_0:

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- The system is in a safe state since there exists a safe sequence < P_1, P_3, P_4, P_2, P_0>

Example of Safety Algorithm (2)

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

< P_1, P_3, P_4, P_2, P_0> is a safe sequence as shown below:

Work = Available = (3, 3, 2)

1. P_1 can finish because $Need_1$=(1,2,2) ≤ Available = (3,3,2). When P_1 finishes, Available = (3,3,2) + (2,0,0) = (5,3,2)
2. P_3 can finish because $Need_3$=(0,1,1) ≤ Available = (5,3,2). When P_3 finishes, Available = (5,3,2) + (2,1,1) = (7,4,3)
3. P_4 can finish because $Need_4$=(4,3,1) ≤ Available = (7,4,3). When P_4 finishes, Available = (7,4,3) + (0,0,2) = (7,4,5)
4. P_2 can finish because $Need_2$=(6,0,0) ≤ Available = (7,4,5). When P_2 finishes, Available = (7,4,5) + (3,0,2) = (10,4,7)
5. P_0 can finish because $Need_0$=(7,4,3) ≤ Available = (10,4,7). When P_0 finishes, Available = (10,4,7) + (0,1,0) = (10,5,7)

Resource-Request Algorithm

- Let $\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$
  - e.g., $\mathbf{Request_1} = (1,0,2)$
- When a request is made by $\boldsymbol{P_i}$, the following actions are taken to determine whether the request can be safely granted
  1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
  2. If $\boldsymbol{Request_i \leq Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
  3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

     - If new state is safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
     - If new state is unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Resource-Request Algorithm Example

- Suppose the current state of the system is the following:

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

- We have shown that the system is in a safe state
- Suppose now that process P_1 makes a request **$Request_1$** = (1,0,2)

Example: P_1 Requests (1,0,2)

- Check that $\text{Request}_1 \leq \text{Need}_1$, that is, $(1,0,2) \leq (1,2,2) \Rightarrow$ true
- Check that $\text{Request}_1 \leq$ Available, that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true
- Pretend that the request is granted, then we arrive at the new state:
  - Available = Available – Request = (3,3,2) – (1,0,2) = (2,3,0)
  - $\text{Allocation}_1 = \text{Allocation}_1$ + Request = (2,0,0) + (1,0,2) = (3,0,2)
  - $\text{Need}_1 = \text{Need}_1$ – Request = (1,2,2) – (1,0,2) = (0,2,0)

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Requests (1,0,2)

- The new system state is the following:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- The new state is safe as we can find a safety sequence < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> as shown below. Thus, we can grant the request.

Work = Available = (2,3,0)

1. P_1 can finish because $Need_1$=(0,2,0) ≤ Available = (2,3,0). When P_1 finishes, Available = (2,3,0) + (3,0,2) = (5,3,2)
2. P_3 can finish because $Need_3$=(0,1,1) ≤ Available = (5,3,2). When P_3 finishes, Available = (5,3,2) + (2,1,1) = (7,4,3)
3. P_4 can finish because $Need_4$=(4,3,1) ≤ Available = (7,4,3). When P_4 finishes, Available = (7,4,3) + (0,0,2) = (7,4,5)
4. P_0 can finish because $Need_0$=(7,4,3) ≤ Available = (7,4,5). When P_0 finishes, Available = (7,4,5) + (0,1,0) = (7,5,5)
5. P_2 can finish because $Need_2$=(6,0,0) ≤ Available = (7,5,5). When P_2 finishes, Available = (7,5,5) + (3,0,2) = (10,5,7)

Example: P_4 Requests (3,3,0)

- Suppose process P_4 makes a request **Request**$_4$= (3,3,0) when the system is in the following state. Can the request be granted?

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Check that Request$_4$ $\leq$ Need$_4$, that is, (3,3,0) $\leq$ (4,3,1) $\Rightarrow$ true
- Check that Request$_4$ $\leq$ Available, that is, (3,3,0) $\leq$ (2,3,0) $\Rightarrow$ false
- Thus, the request cannot be granted as there are not enough resources to satisfy the request

Example: P_0 Requests (0,2,0)

- Suppose process P_0 makes a request **Request**$_0$= (0,2,0) when the system is in the following state. Can the request be granted?

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Check that Request$_0$ $\le$ Need$_0$, that is, $(0,2,0) \le (7,4,3) \Rightarrow$ true
- Check that Request$_0$ $\le$ Available, that is, $(0,2,0) \le (2,3,0) \Rightarrow$ true
- Next, we pretend that the request is granted, then we arrive at the new state:
  - Available = Available – Request = (2,3,0) – (0,2,0) = (2,1,0)
  - Allocation$_0$ = Allocation$_0$ + Request = (0,1,0) + (0,2,0) = (0,3,0)
  - Need$_0$ = Need$_0$ – Request = (7,4,3) – (0,2,0) = (7,2,3)

Example: P_0 Requests (0,2,0)

- The new state is the following:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Now let's try to find a safe sequence:

  Work = Available = (2,1,0)

  We cannot find a process P_i such that $Need_i \leq Available$. Thus, no safe sequence exists.

  Therefore, the system state is unsafe and the request cannot be granted!