**Chapter 07: Deadlocks**

Table of Contents

[System Model 4](#_Toc83032926)

[Deadlock Characterizations 5](#_Toc83032927)

[Resource Allocation Graphs 6](#_Toc83032928)

[Handling Deadlocks 8](#_Toc83032929)

[Deadlock Prevention 9](#_Toc83032930)

[Preventing Mutual Exclusion 9](#_Toc83032931)

[Preventing Hold and Wait 9](#_Toc83032932)

[Preventing No Pre-Emption 10](#_Toc83032933)

[Preventing Circular Wait 10](#_Toc83032934)

[Side Effects 10](#_Toc83032935)

[Deadlock Avoidance 11](#_Toc83032936)

[Safe State 11](#_Toc83032937)

[Resource Allocation Graph Algorithm 12](#_Toc83032938)

[Banker’s Algorithm 13](#_Toc83032939)

[Safety Algorithm 14](#_Toc83032940)

[Resource-Request Algorithm 18](#_Toc83032941)

[7.6 Deadlock Detection 19](#_Toc83032942)

[Single Instance for Each Resource 19](#_Toc83032943)

[Several Instance of a Resource Type 20](#_Toc83032944)

[Detection-Algorithm Usage 20](#_Toc83032945)

[7.7 Recovery from Deadlock 21](#_Toc83032946)

A **deadlock** is a set of **blocked processes**, each holding one **resource** while waiting to acquire another resource, which is in turn held by a different process in the set.

For example, say a system has two tape drives and each one is being held by a different process. If both processes need both tape drives to continue, we have a deadlock.

Deadlocks can cause **starvation**.

## System Model

For the different deadlock situations we will be examining next, we will be using a specific system model, as defined here.

A system consists of a **finite** number of **resources** that are to be **distributed** among several computing processes. The resource types could be , , , . Examples of different resource types are CPU cycles, memory space, I/O devices, etc.

Each resource type in turn has several identical **instances**. The th resources has instances.

A process may request as many resources as it requires, given that this number is not greater than the total number of resources available in the system. Each process utilizes a single resource as follows:

1. Request
2. Use
3. Release

## Deadlock Characterizations

For a deadlock to occur, **four conditions** must be met **simultaneously**:

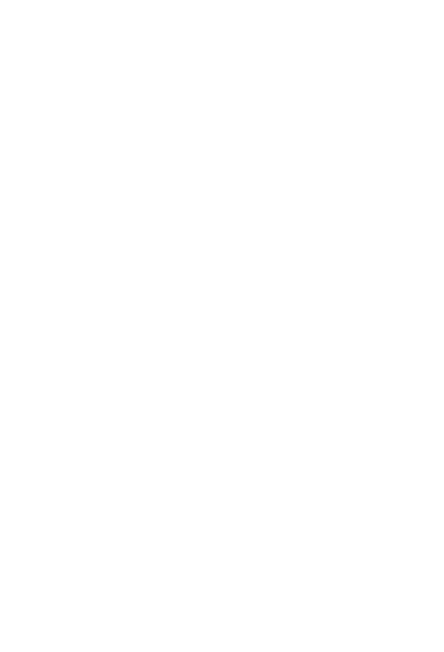
1. **Mutual Exclusion** – A single resource can only be used by a single process at a time.
2. **Hold and Wait** – A process that is holding at least one resource must be waiting to acquire other resources that are held by other processes.
3. **No Pre-emption** – A resource can only be released voluntarily by the process holding it, after that process has completed its task. This means the system cannot force a process to release the resource.
4. **Circular Wait** – There exists a set of processes, , , , , , such that the th process is waiting for a resource held by the th process. Thus, is waiting for a resource held by , is waiting for a resource held by and so on, until we reach , which is waiting for a resource held by .

## Resource Allocation Graphs

A **Resource Allocation Graph** is a set of vertices, , and edges, .

is partitioned into two types, , the set consisting of all the **processes** in the system, and , the set consisting of all the **resources** in the system.

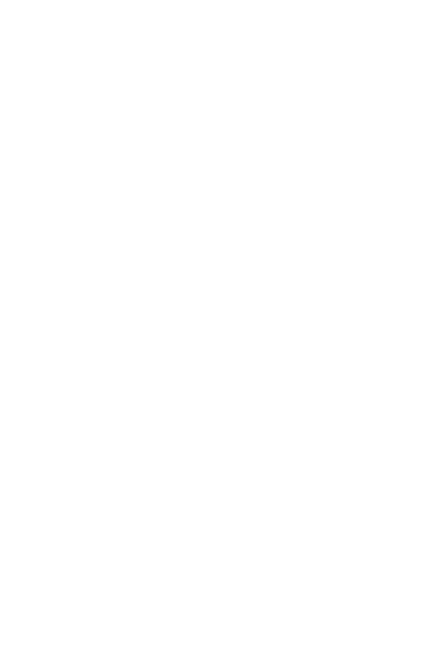
can also be of two types, a **request edge**, which is a directed edge from , and an **assignment edge**, which is a directed edge from .



The **dots** inside the resources indicate **instances**. Notice that request edges go from the process to the resource, but assignment edges go from the instance to the process.

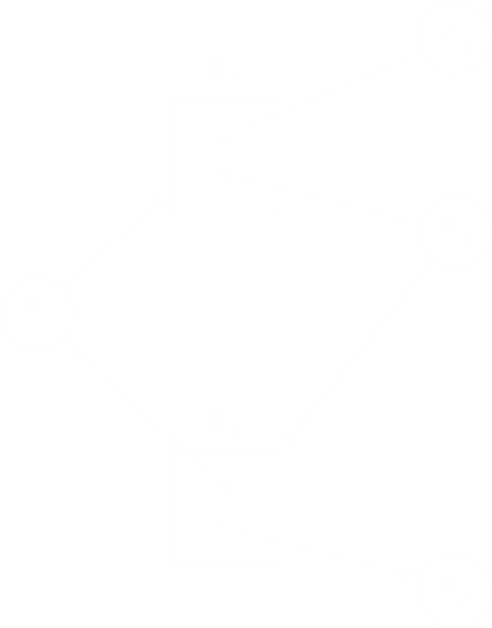
In the graph above, there is no deadlock, even though and are both still waiting. will eventually release , which will allow to execute, which will eventually cause the release of , allowing to execute.

However, we only need to make a small change to cause a deadlock.



Now we will have a deadlock. will keep waiting for , but will never be able to get it, which will lead to a deadlock.

The way to identify a deadlock is to look for **cycles** in the graph. However, just having a cycle is not enough to cause a deadlock. Consider this scenario:



In this case, will eventually release an instance of , which will allow to execute, thus breaking the cycle. Similarly, may release first, which will allow to execute, also breaking the cycle.

## Handling Deadlocks

There are three methods to handling deadlocks:

1. **Deadlock Prevention** – We can ensure that the system never enters a deadlock state at all.
2. **Deadlock Avoidance** – We can pretend that deadlocks never occur. This is used by most operating systems.
3. **Deadlock Detection and Recovery** – We can allow a deadlock to occur and then recover from it once it does.

## Deadlock Prevention

In order to prevent deadlocks from occurring, we must restrain the ways in which requests can be made. All we need to do is ensure that one of the **four conditions** required to cause a deadlock is **not met**.

### Preventing Mutual Exclusion

It is **not possible** to completely prevent deadlocks by denying mutual exclusion. This is because some resources are inherently **non-shareable**.

### Preventing Hold and Wait

We can remove the hold-and-wait condition by ensuring that, whenever a process **requests** a resource, it **does not hold** other resources.

One approach to doing this would be to ensure that a process has requested and has been allocated **all the resources** it requires before it begins execution.

Another approach is to allow a process to request resources only when it **has none**. This would mean that a process can request some resources, use them and then be forced to release them before it can request more resources.

Consider that a process copies data from a CD to a disk file and then prints the file with a printer. Under the first approach, it would take up the CD drive, the disk file and the printer throughout the whole execution. Under the second approach, it would take up the CD drive and disk file initially, and the disk file and the printer later on.

The two approaches have two types of problems. For the first, it is that there is **low resource utilization**. Processes are essentially being forced to take up resources for longer than they need them. For the second, **starvation** could occur. A process that needs several resources simultaneously may end up waiting forever.

### Preventing No Pre-Emption

If a process is **holding** resources and **requests** a resource that cannot immediately be allocated to it, all held resources are **released**. The released resources become part of the list of resources the process is waiting for. The process will be started only when it can get a hold of all the resources it needs.

### Preventing Circular Wait

To prevent circular wait from occurring, we can impose an **order** for resources by assigning a numeric value to each resource type. Each process will then be allowed to request resources in an **increasing order** only, i.e. if the process requests resource 5, it can then only go on to request resource 1, 2, 3 or 4.

### Side Effects

Side effects to using deadlock prevention mechanisms include:

* **Low Device Utilization**
* **Reduced System Throughput**

## Deadlock Avoidance

Deadlock avoidance is by far the most sophisticated method of handling deadlocks and is the one used by most operating systems. To be able to avoid deadlocks, it is required that the system have some **a priori information**. For example, if the system knows beforehand that one process will request the CD drive and then the printer and another process will request the printer and then the CD drive, it can make the second process wait in order to avoid a deadlock.

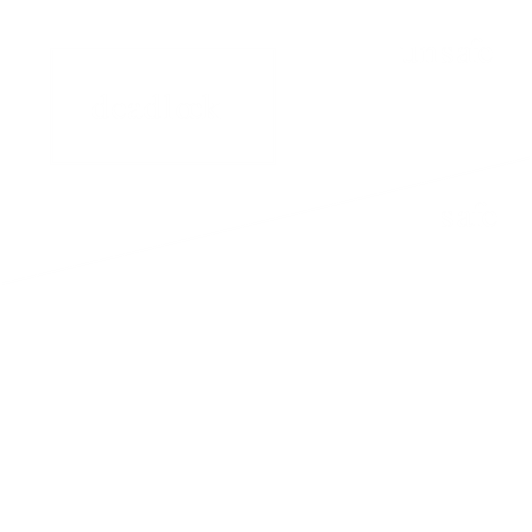
There are multiple algorithms for deadlock avoidance. The simplest and most useful model requires each process to declare the **maximum number** of each resource type it will require. The **deadlock avoidance algorithm** then dynamically **examines** the **resource allocation state** to ensure that a **circular wait condition** can never occur. The resource allocation state is defined by the number of **available and allocated resources** and the **maximum demands** of the process.

### Safe State

When a process requests an available resource, the system must decide if allocating the resource leaves the system in a **safe state**. The system is in a safe state if there is a **safe sequence of all processes**.

A sequence , , , , is a safe sequence of processes if for each process , the resources it might still request can be satisfied using the **available resources** and the resources currently held by the **processes before it**. If a resource requested by the process is not available, it can simply wait until some process before it releases it.

If a system is in a safe state, deadlocks will not occur. If it is not, there is a possibility of deadlocks. Avoidance essentially ensures that the system never enters an unsafe state.

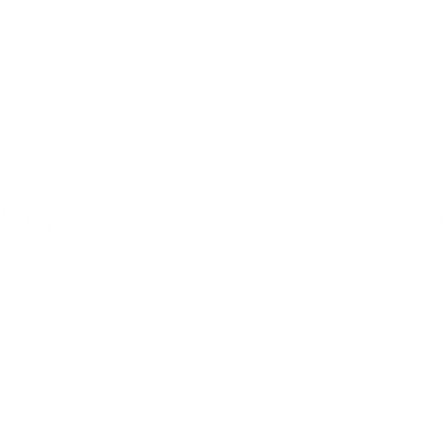


### Resource Allocation Graph Algorithm

If each resource in our system has only **one instance**, we can use the **Resource Allocation Graph Algorithm** for deadlock avoidance. This uses the Resource Allocation Graph as described earlier, but introduces a new edge, a **claim edge**.

A claim edge goes from the process to the resource , but uses a **dashed arrow**, as in . It represents the fact that the process **may request** the resource in the future. When the actual request is made, the claim edge becomes a request edge. When a resource is released by a process, the assignment edge becomes a claim edge.

Note that before a process begins execution, all its claim edge must be present, i.e. resources must be claimed a priori.



If a process makes a request, the request will only be granted if **converting** the claim edge to an assignment edge does not put the system in an **unsafe state**, i.e. a cycle is not produced.

In the above graph, even if requests , we cannot allocate it, since we will be put in a cycle. If makes a request for later on then we will enter a deadlock.

### Banker’s Algorithm

We already know that each resource can have **multiple instances** and each process must declare **how many instances** of each resource it will use at most. When a process requests a resource, it **may have to wait** and when it does end up getting all the required resources, it must **return** them in within a **finite time**.

In the **Banker’s Algorithm**, we make use of a few data structures, which we will look at first. Here, keep in mind that is the number of processes and is the number of resource types.

* **Available** – This is a **vector** of length . indicates that the th resource has instances available.
* **Max** – This is an matrix. indicates that the th process may request at most instances of the th resource.
* **Allocation** – This is also an matrix. indicates that the th process has currently been allocated instances of the resource.
* **Need** – This is also an matrix. indicates that the th process may still need instances of the th resource to complete its task.

Inconclusion, .

There are several parts to the Banker’s algorithm, each of which can be defined as their own algorithms.

### Safety Algorithm

The **Safety Algorithm** determines whether or not the system is currently in a safe state. In this algorithm, we try to find a **safety sequence**.

*// i = process number, j = resource number  
// Step 1: Initialization*for all i, j:  
 work[i, j] = available[i, j] *// we don't want to edit actual data* finish[i] = false *// keeping track of processes that can be finished  
   
// Step 2: Find a process that can be finished*  
find i such that:  
 finish[i] == false *// process is running* AND

for all j:  
 need[i, j] < work[i, j] *// but the process CAN be finished*  
   
*// Step 3: Take the process's resources*   
finish[i] = true; *// pretend process has finished*for all j:  
 work[i, j] += allocation[i, j] *// take its resources  
// go back to Step 2*

*// Step 4: cannot find process to end*for all i:  
 if finish[i] == true *// safe state* else:  
 *// not all processes are finished  
 // but we cannot find a process that CAN be finished  
 // thus, deadlock*

PSEUDOCODE

First, we create a vector, which is just a copy of the vector. We also create a new vector that will hold the status of each of the processes. As such, every index of the vector is initialized to , to indicate that none of the processes have finished execution.

Next, we try to find a process such that it is **unfinished** and the number of resources of each type it **needs** is less than the number of resources available as defined in the vector.

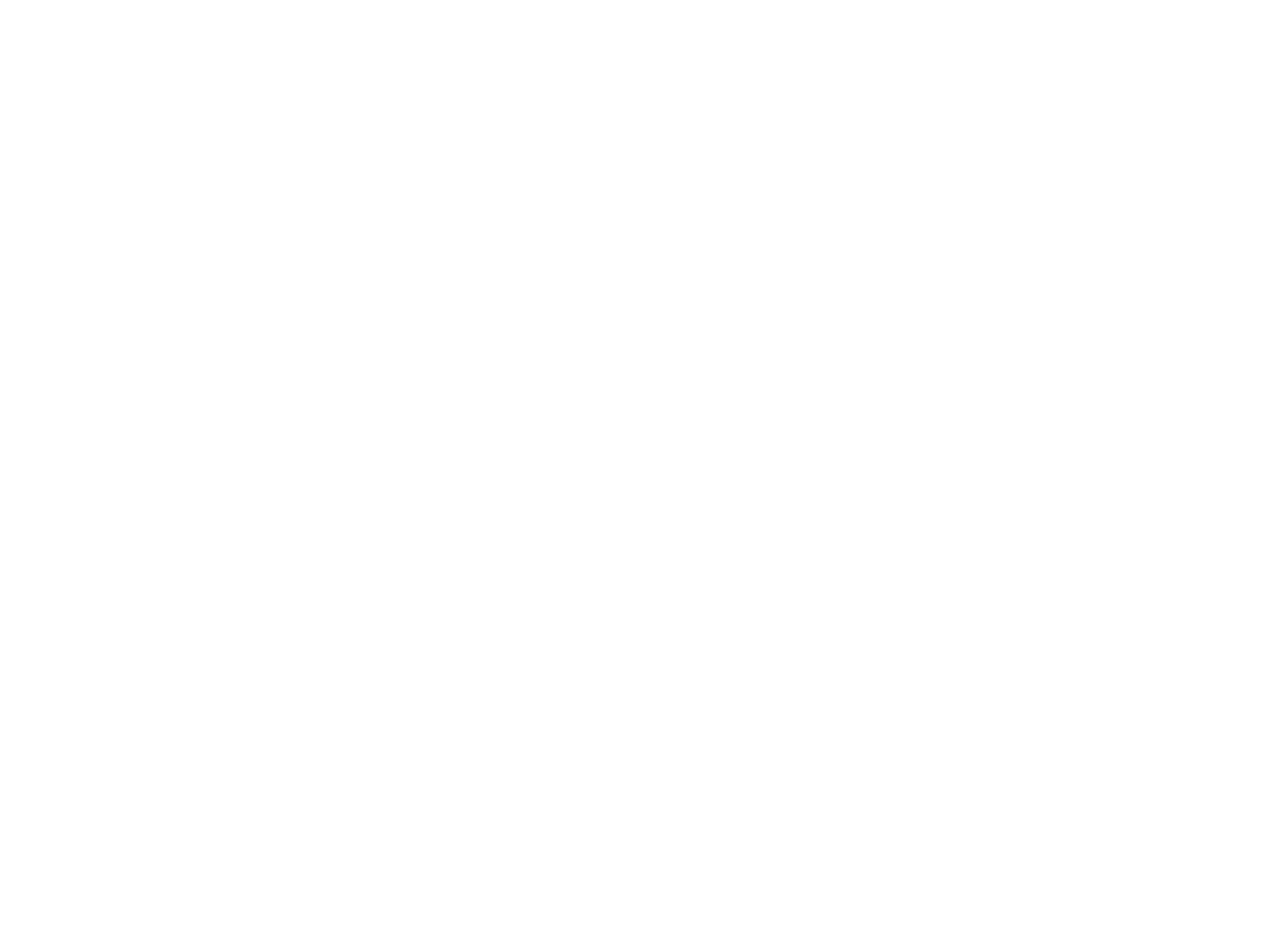
Say we find a process that matches this criteria. We can then claim the resources it is currently using, as defined in the matrix, and add it to the matrix. We can then mark that process as **finished**.

Consider why this makes sense. If we are trying to determine if the system is **safe**, we need to check that all processes currently executing can **finish executing**. This means ensuring that there are as many resources **available** as each process **requires**.

If we find a process that we know can **successfully finish**, we can **add** its resources to the **working set**, since any other processes that require resources will **eventually** get those resources, once this process has finished executing.

We keep **looping** and checking the different processes until we find that all the processes can **successfully finish**. If we have a **deadlock**, then at some point, we will be forced to go through **step 2 completely** and be unable to enter **step 3** for any of the processes. At this point, we will know that the system is not in a safe state. Otherwise, if no deadlock exists, we will find that all the processes are **finished**. Then, we can say that the system is in a **safe state**.

The graph below shows an example usage. The **nodes** contain the **current values**, while the **edges** indicate different **processes** that we can consider to have finished. Each process has its  **value** marked on the edge, and we can only follow an edge if the  **value** for the process on that edge is **less than** the **current value**. If we follow an edge, we can add the  **value** for that process to the current value.



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Allocation | | | Max | | | Available | | | Need | | |
|  | A | B | C | A | B | C | A | B | C | A | B | C |
|  | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 | 7 | 4 | 3 |
|  | 2 | 0 | 0 | 3 | 2 | 2 |  |  |  | 1 | 2 | 2 |
|  | 3 | 0 | 2 | 9 | 0 | 2 |  |  |  | 6 | 0 | 0 |
|  | 2 | 1 | 1 | 2 | 2 | 2 |  |  |  | 0 | 1 | 1 |
|  | 0 | 0 | 2 | 4 | 3 | 3 |  |  |  | 4 | 3 | 1 |

Thus, for each node, we can have **multiple edges** to go forward, which will give us **multiple different paths**. Some, but not all, possible paths for this scenario are shown. Notice how regardless of which path we follow, we always end up at the **same result**. The actual system will only follow **one** of these paths. Indeed, even the algorithm will only follow one of these paths.

Note that the **actual system** is **not changed** in this algorithm, which is why we are creating a copy of the vector. None of the processes are actually finishing either. We are simply checking the different possibilities and ensuring that there is at least one sequence of events that leads to a deadlock not arising.

### Resource-Request Algorithm

The **Resource-Request Algorithm** is fairly simple, since it utilizes the **Safety Algorithm**.

We have a vector, which is the vector containing the resources being requested.

We first check that that , to ensure that the process is not requesting **more resources** than it said it would.

Next, we check that . If it is not, then the process must **wait**.

Finally, we can conclude that the required resources **can be allocated**, but we still need to check that the system will not go into an **unsafe state** by doing this. For this, we allocated the resources in the form of a **transaction**. We update the vectors and matrices as required, and then we check whether the system is still in a safe state. If it is not, we **rollback** the changes.

if request <= need[i] && request <= available  
 available = available - request  
 allocation[i] += request  
 need[i] -= request  
 // check safe state

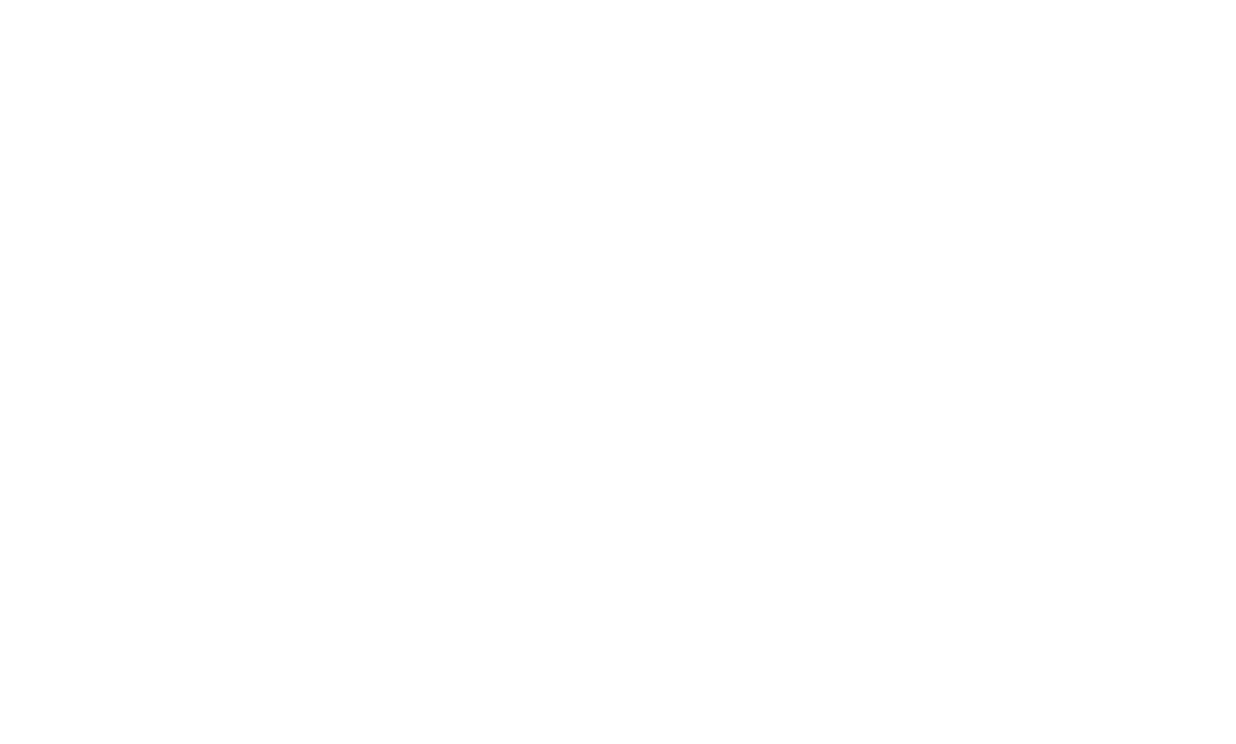
PSEUDOCODE

## 7.6 Deadlock Detection

If we do not **prevent** or **avoid** deadlocks, then we will eventually enter a **deadlock state**. In this case, we must provide a **deadlock detection algorithm** and also a **recovery scheme**.

### Single Instance for Each Resource

If we consider that each resource only has a **single instance**, then we develop a deadlock detection algorithm that uses a variant of the **resource-allocation graph**, called a **wait-for** graph. We take the resource-allocation graph, **remove** the **resource nodes** and finally **collapse** the **edges**. Thus, an edge from process to indicates that is waiting for some resource that is held by .



As before, a **deadlock** exists if there is a **cycle**. Thus, we will maintain this graph and periodically invoke an algorithm to check for cycles.

### Several Instance of a Resource Type

We cannot use the previous algorithm if we have **several instances** of each resource. In this case, we need to make use of a few **data structures**, similar to the ones we used with Banker’s Algorithm. These include the vector, the matrix and the matrix. Note that the matrix is different from the vector we saw previously, in that we are not accounting for all processes.

This algorithm is exactly the same as the **Banker’s Algorithm**, except that instead of checking that , we check that . If, at the end of it, we find some value of such that , then there is a deadlock.

### Detection-Algorithm Usage

When and how often we should invoke the detection algorithm depends on two things:

* How often is a deadlock likely to occur?
* How many processes are affected by a deadlock?

Invoking the algorithm for **every request** will cause **significant overhead**. We could invoke it at **intervals**, but then it is possible that there will be multiple cycles and we will be unable to tell **which process** caused the deadlock.

## 7.7 Recovery from Deadlock

There are two options to **break** a deadlock:

* Simply **abort** one of more processes to break the circular wait
* **Pre-empt** some resources from one or more of the deadlocked processes

For **abortion**, we could just abort all the deadlocked processes, or we could abort processes one by one until the deadlock breaks. For the latter case, the **order** of abortion could depend on:

* Priority of processes
* How long the process has been running and how far it is from completion
* Resources the process has used
* Resources the process needs to finish execution
* How many processes will need to be terminated
* Whether the process is interactive or batch

For **resource pre-emption**, we need to address three issues:

* Selecting a victim – We need to select victims in a way so as to minimize the cost.
* Rollback – We need to return the process to some safe state and restart it from that state.
* Starvation – We need to ensure we are not starving a process by repeatedly pre-empting from it.