CSCI 340 Operating Systems

Chapter 8: Deadlock

Stewart Weiss

Table of Contents

Prerequisite Reading
About This Chapter
Chapter Objectives
An Example of Deadlock
Deadlock Visualized
Description of the Model
Modeling How Processes Use Resources
Reusable Resource Graphs (RR-Graphs)
Examples of RR-Graphs
Examples of RR-Graphs
Deadlock State
Characterizing Deadlock: Four Necessary Conditions
About Circular Waiting
Circular Waiting and Cycles
Cycles Do Not Imply Deadlock
A Theorem About Cycles
Another Theorem About Cycles
Using RR-Graphs
Example Leading to Deadlock
Other Types of Waiting
Dealing With Deadlock

Table of Contents

Prevention
Prevention: Removing Mutual Exclusion
Prevention: Removing Hold-and-Wait
Prevention: Removing Non-Preemption
Prevention: Removing Non-Preemption
Prevention: Removing Circular Waiting
Example of Ordered Requests
Avoidance Algorithms
Avoidance Algorithms Using Maximum Needs
Representing Maximum Needs
Use of Maximum Need
Safe (and Unsafe) System States
Avoiding Deadlock Using Safe States
The Banker's Algorithm
Banker's Algorithm Data Structures
The Safety Algorithm
The Safety Algorithm
The Request-Simulation Algorithm
Safety Algorithm Example
Request-Simulation Algorithm Example
Deadlock Detection
Deadlock Detection Algorithms
Deadlock Detection Algorithm

Table of Contents

[Deadlock Recovery](#)
[References](#)

Prerequisite Reading

Before reading these slides, you should be familiar with

- Processes and concurrency (Chapter 3)
- Threads and the POSIX Threads API (Chapter 4)
- Process synchronization tools and primitives (Chapter 6)

About This Chapter

Chapter 6 introduced deadlock in an informal way and gave an example of it, and in Chapter 7, a solution to the Dining Philosophers Problem was shown to cause deadlock.

This chapter examines deadlock in greater depth, presenting formal models of systems of processes that make it possible to reason about deadlock and prove theorems about it.

In particular, it explores the most important issues related to deadlock, which are:

- Deadlock Characterization
- Deadlock Prevention and avoidance
- Deadlock Detection
- Recovery from Deadlock

As we did in Chapters 6 and 7, we use the terms "process" and "thread" interchangeably,

Chapter Objectives

Having read and understood the content of this chapter, you should be able to

- provide concrete examples of how deadlock can occur when mutex locks are used for synchronizing access to shared resources;
- state the four conditions that must exist for deadlock to occur;
- explain what a resource allocation graph is and determine whether or not deadlock is present in a system modeled by one;
- describe and compare the four strategies for handling the possibility of deadlock;
- apply the Banker's Algorithm to a system of processes;
- apply the deadlock detection algorithm to a system of processes;
- compare different methods of recovering from deadlock.

An Example of Deadlock

Suppose that each of two processes, P1 and P2, needs to modify a file on disk and one on a USB drive. The disk file is named R1 and the one on the USB drive is named R2. (Each file is a **resource** and our convention henceforth is to name resources using uppercase R.)

Files can only be modified by one process at a time and a process can only acquire files one at a time, but each needs to have both files open to perform its updates.

An Example of Deadlock

Suppose that each of two processes, P1 and P2, needs to modify a file on disk and one on a USB drive. The disk file is named R1 and the one on the USB drive is named R2. (Each file is a **resource** and our convention henceforth is to name resources using uppercase R.)

Files can only be modified by one process at a time and a process can only acquire files one at a time, but each needs to have both files open to perform its updates.

Therefore, each process must acquire both files, one after the other, perform its update, and then release the two files. Suppose their code is something like the following.

```
P1:
    request(R1);
    request(R2);
    perform update;
    release(R2);
    release(R1);
```

```
P2:
    request(R2);
    request(R1);
    perform update;
    release(R1);
    release(R2);
```

An Example of Deadlock

Suppose that each of two processes, P1 and P2, needs to modify a file on disk and one on a USB drive. The disk file is named R1 and the one on the USB drive is named R2. (Each file is a **resource** and our convention henceforth is to name resources using uppercase R.)

Files can only be modified by one process at a time and a process can only acquire files one at a time, but each needs to have both files open to perform its updates.

Therefore, each process must acquire both files, one after the other, perform its update, and then release the two files. Suppose their code is something like the following.

```
P1:
    request(R1);
    request(R2);
    perform update;
    release(R2);
    release(R1);
```

```
P2:
    request(R2);
    request(R1);
    perform update;
    release(R1);
    release(R2);
```

If they are each granted their first request, i.e., P1 gets R1 and P2 gets R2, before either requests the second file (in the highlighted lines above), then each process will be holding a file the other needs and will be in a **blocked** state, waiting for the file held by the other.

This is an example of **deadlock**.

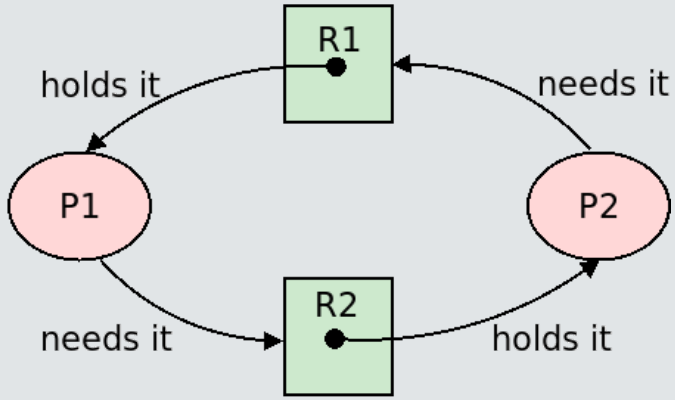
Deadlock Visualized

Deadlock can be visualized with a type of **directed graph** called a **reusable resource graph** (**RR-graph**). We illustrate it informally using the preceding example and formalize it later.

- We represent each process by a node drawn as an **ellipse**.
- We represent each resource by a **rectangular node** with a "dot" inside to represent the actual file.
- We represent the fact that a process **is holding** (has acquired and not released) a resource with a **directed edge from the dot inside the resource node to the process**. This is called an **assignment edge**.
- We represent the fact that a process **is waiting for** (has requested but not acquired) a resource with a **directed edge from the process to the entire resource node**. This is called a **request edge**.

Using these rules, the deadlocked state of the system from the preceding example is depicted in the next slide.

Deadlock Visualized



In the figure, the edge from `R1` to `P1` is drawn from the small dot representing the single file, but the edge from `P2` to `R1` is drawn just to the rectangle itself, for reasons to be explained later.

Notice that this graph has a **cycle**. This is not a coincidence.

A System Model

We describe an abstract model of a collection of processes and resources that they share that will allow us to reason about deadlock.

Description of the Model

A **resource allocation system** is a collection consisting of processes and resources. The resources in this system **can only be held by one process at a time**[1]. (There are resources, such as read-only files, that can be shared; these are excluded.)

We define it as follows:

1 Technically, these notes describe a **serially reusable resource allocation system**.

Description of the Model

A **resource allocation system** is a collection consisting of processes and resources. The resources in this system **can only be held by one process at a time**[1]. (There are resources, such as read-only files, that can be shared; these are excluded.)

We define it as follows:

- There is a fixed, finite number N of processes represented by the symbols P_1, P_2, ...,P_N. The process set is static - no processes are created or destroyed.

1 Technically, these notes describe a **serially reusable resource allocation system**.

Description of the Model

A **resource allocation system** is a collection consisting of processes and resources. The resources in this system **can only be held by one process at a time**[1]. (There are resources, such as read-only files, that can be shared; these are excluded.)

We define it as follows:

- There is a fixed, finite number N of processes represented by the symbols P_1, P_2, ...,P_N. The process set is static - no processes are created or destroyed.
- There is a finite number t of **resources types** represented by the symbols R_1, R_2, ...,R_t.
    - Resource types are things such as memory units, disk blocks, network interfaces, files, and mutex locks.

1 Technically, these notes describe a **serially reusable resource allocation system**.

Description of the Model

A **resource allocation system** is a collection consisting of processes and resources. The resources in this system **can only be held by one process at a time**[1]. (There are resources, such as read-only files, that can be shared; these are excluded.)

We define it as follows:

- There is a fixed, finite number N of processes represented by the symbols P_1, P_2, ...,P_N. The process set is static - no processes are created or destroyed.
- There is a finite number t of **resources types** represented by the symbols R_1, R_2, ...,R_t.
    - Resource types are things such as memory units, disk blocks, network interfaces, files, and mutex locks.
- Each resource type consists of a fixed number of **identical, interchangeable units**. The number of units of type R_k is denoted W_k. For example:
    - If primary memory is resource type R_2 and it consists of 4,194,304 4096-byte blocks, then W_2=4,194,304.
    - If a tertiary storage system used for backup consists of 16 identical tape drives and the tape drive type is R_5, then W_5=16.

1 Technically, these notes describe a **serially reusable resource allocation system**.

Modeling How Processes Use Resources

In this model, for a process P_i to use a resource R_j, it must perform 3 actions in sequence:

Modeling How Processes Use Resources

In this model, for a process P_i to use a resource R_j, it must perform 3 actions in sequence:

1. It **requests** a number of units of the resource less than W_j, and if they are not available, it **blocks** itself to wait for them. Because units are identical and interchangeable, a request is not for specific units; any of the available ones will satisfy the request.

Modeling How Processes Use Resources

In this model, for a process P_i to use a resource R_j, it must perform 3 actions in sequence:

1. It **requests** a number of units of the resource less than W_j, and if they are not available, it **blocks** itself to wait for them. Because units are identical and interchangeable, a request is not for specific units; any of the available ones will satisfy the request.

2. When all units are available, it **acquires and uses** the resource. The units that it acquires are specific units.

Modeling How Processes Use Resources

In this model, for a process P_i to use a resource R_j, it must perform 3 actions in sequence:

1. It **requests** a number of units of the resource less than W_j, and if they are not available, it **blocks** itself to wait for them. Because units are identical and interchangeable, a request is not for specific units; any of the available ones will satisfy the request.

2. When all units are available, it **acquires and uses** the resource. The units that it acquires are specific units.

3. It **releases** all units of the resource that it holds. (Implicitly it has done all of the work that it needed to do with these resources.)

Modeling How Processes Use Resources

In this model, for a process P_i to use a resource R_j, it must perform 3 actions in sequence:

1. It **requests** a number of units of the resource less than W_j, and if they are not available, it **blocks** itself to wait for them. Because units are identical and interchangeable, a request is not for specific units; any of the available ones will satisfy the request.

2. When all units are available, it **acquires and uses** the resource. The units that it acquires are specific units.

3. It **releases** all units of the resource that it holds. (Implicitly it has done all of the work that it needed to do with these resources.)

Usually steps 1 and 3 above are **system calls**, in which case the operating system must verify and service these calls.

Modeling How Processes Use Resources

In this model, for a process P_i to use a resource R_j, it must perform 3 actions in sequence:

1. It **requests** a number of units of the resource less than W_j, and if they are not available, it **blocks** itself to wait for them. Because units are identical and interchangeable, a request is not for specific units; any of the available ones will satisfy the request.

2. When all units are available, it **acquires and uses** the resource. The units that it acquires are specific units.

3. It **releases** all units of the resource that it holds. (Implicitly it has done all of the work that it needed to do with these resources.)

Usually steps 1 and 3 above are **system calls**, in which case the operating system must verify and service these calls.

Although we are primarily interested in the request and release of resources managed by the kernel, programmers in general must be concerned about the request and release of non-kernel managed resources as well:

- Deadlock can occur within multi-threaded applications, and steps 1 and 3 may not be system calls in that case. For example, an application might use the Pthreads library and the resources might be mutex locks provided by the library. The acquisition and release of these locks are not system calls.

Reusable Resource Graphs (RR-Graphs)

A **reusable resource graph** (**RR-graph** for short) represents the state of a resource allocation system **at a given moment in time**.

It is a directed graph whose vertices are partitioned into two types of nodes: **resource nodes** and **process nodes**.

Reusable Resource Graphs (RR-Graphs)

A **reusable resource graph** (**RR-graph** for short) represents the state of a resource allocation system **at a given moment in time**.

It is a directed graph whose vertices are partitioned into two types of nodes: **resource nodes** and **process nodes**.

- There is a resource node for each resource type in the system. Each resource node contains a dot for each unit of the resource type. Resource nodes are drawn as **rectangles**.

Reusable Resource Graphs (RR-Graphs)

A **reusable resource graph** (**RR-graph** for short) represents the state of a resource allocation system **at a given moment in time**.

It is a directed graph whose vertices are partitioned into two types of nodes: **resource nodes** and **process nodes**.

- There is a resource node for each resource type in the system. Each resource node contains a dot for each unit of the resource type. Resource nodes are drawn as **rectangles**.
- There is a process node for each process in the system. Process nodes are drawn as **ellipses**.

Reusable Resource Graphs (RR-Graphs)

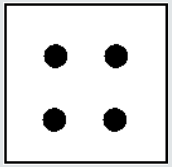
A **reusable resource graph** (**RR-graph** for short) represents the state of a resource allocation system **at a given moment in time**.

It is a directed graph whose vertices are partitioned into two types of nodes: **resource nodes** and **process nodes**.

- There is a resource node for each resource type in the system. Each resource node contains a dot for each unit of the resource type. Resource nodes are drawn as **rectangles**.
- There is a process node for each process in the system. Process nodes are drawn as **ellipses**.
- If a process is **holding** (has acquired and not released) a unit of a resource, there is a **directed edge from the unit of the resource to the process**. This is called an **assignment edge**. Note that processes hold specific units and the edges start at the unit and are directed to the process node.
- If a process is **waiting for** (has requested but not acquired) a unit of a resource, there is a **directed edge from the process to the resource node**. This is called a **request edge**. Because a request is not for specific units, a request edge is directed at the entire resource node. There is a request edge for each unit requested.

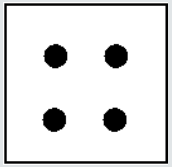
Examples of RR-Graphs

Example. The node for a resource with four units is drawn like this:



Examples of RR-Graphs

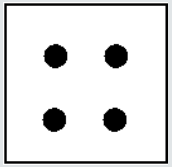
Example. The node for a resource with four units is drawn like this:



A system in which, at a given moment, process P1 requests two of these units would be drawn like this:

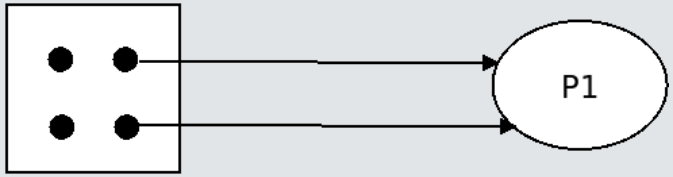
Examples of RR-Graphs

Example. The node for a resource with four units is drawn like this:



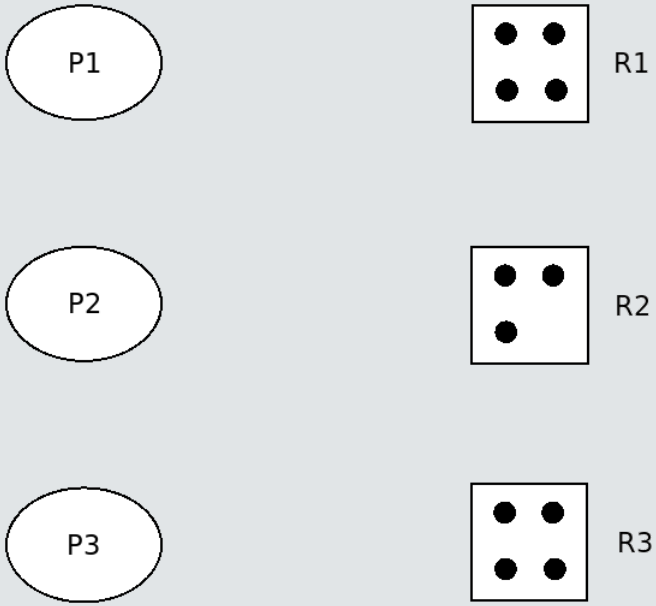
A system in which, at a given moment, process P1 requests two of these units would be drawn like this:

A system in which, at a given moment, process P1 holds two of these units would be drawn like this:



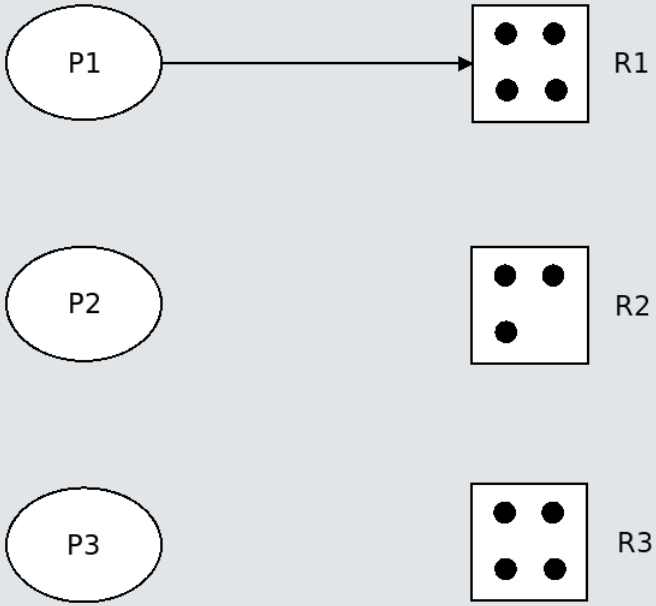
Examples of RR-Graphs

The following figure represents an RR-graph for a 3-process, 3-resource system in which no activity has taken place yet.



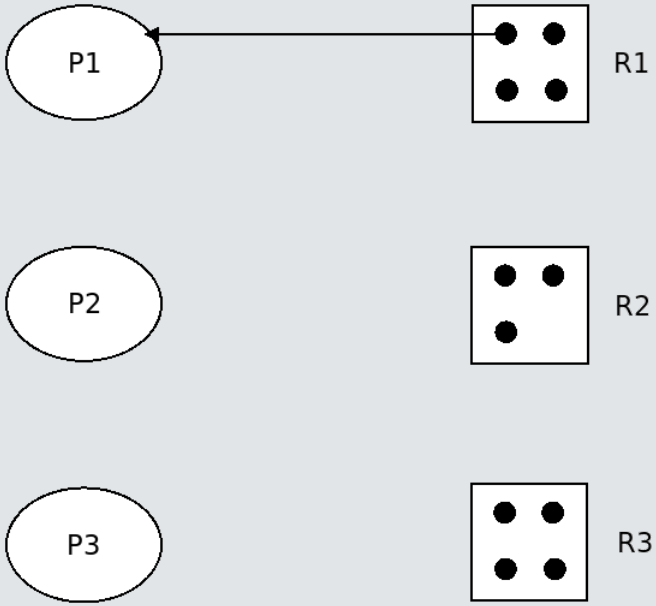
Examples of RR-Graphs

P1 requests a unit of R1:



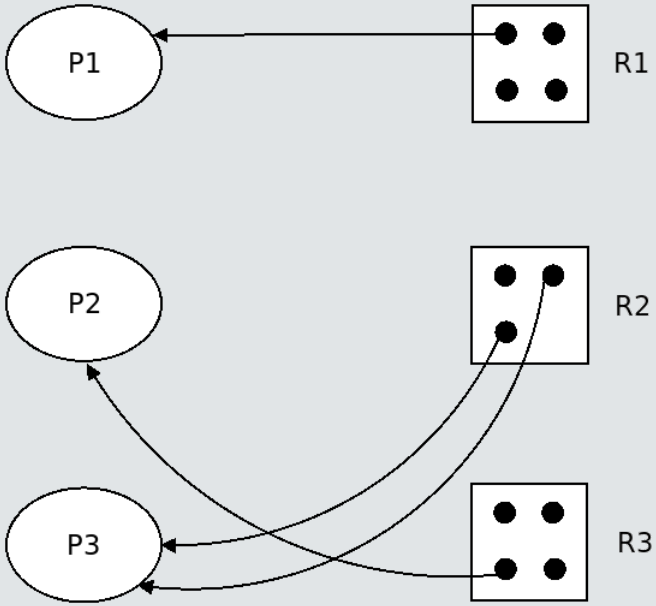
Examples of RR-Graphs

P1 acquires a unit of R1:



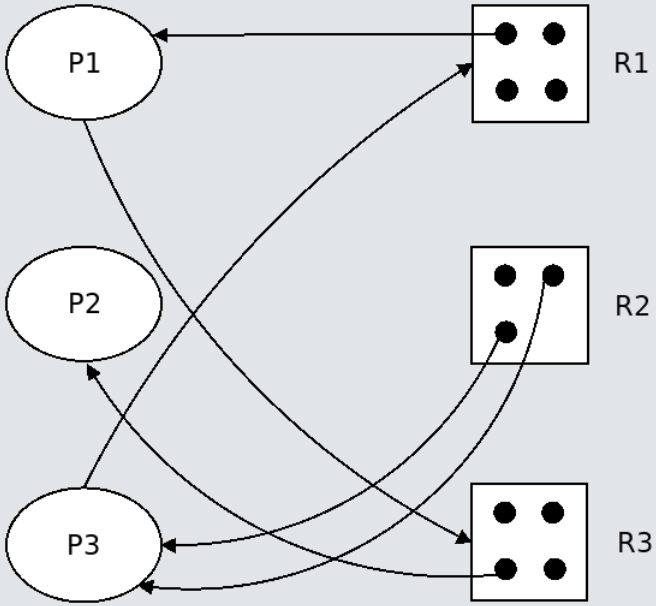
Examples of RR-Graphs

The graph below represent a few actions that took place. P3 requested and acquired 2 units of R2 and P2 requested and acquired 1 unit of R3.



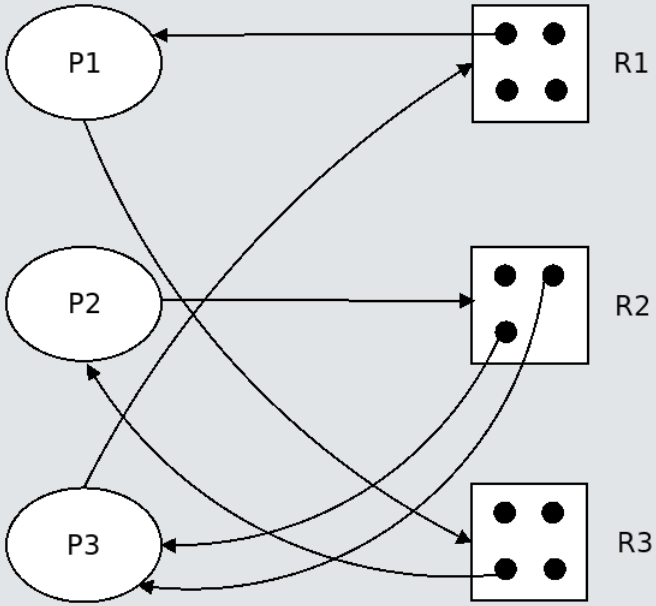
Examples of RR-Graphs

The graph below represent two actions that took place. `P3` requested 1 unit of `R1` and `P1` requested 1 unit of `R3`. Notice that there is no cycle in this graph.



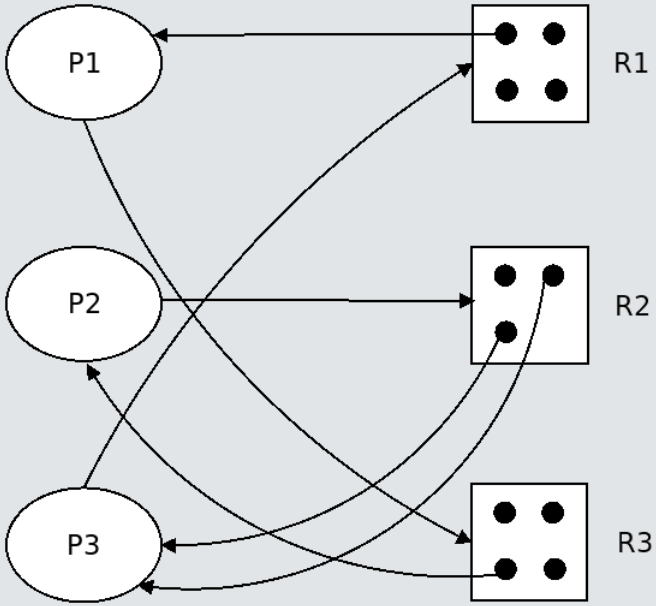
Examples of RR-Graphs

Now P2 requests 1 unit of R2. Notice that there is now a cycle in this graph. Where?



Examples of RR-Graphs

Now P2 requests 1 unit of R2. Notice that there is now a cycle in this graph. Where?



We will re-examine this graph a bit later. It is time to formalize deadlock.

Deadlock State

A set of processes is in a **deadlock state** when every process in the set is **waiting for an event that can be caused only by another process in the set**.

Example. A system has three tape drives. Three processes each need two tape drives for reading input data.

1. Each process requests and acquires its first tape drive. (All three drives are now held.)
2. Now each process requests its second tape drive. (No tape drives are available, so each process blocks.)
3. Each process is now waiting for an event, the release of a tape drive, that can only be caused by one of the other two processes.

They are now in a **deadlock** state.

Characterizing Deadlock: Four Necessary Conditions

Deadlock can arise in a resource allocation system only if all four of the following conditions are true of that system simultaneously.

1. **Mutual Exclusion**. At least one resource must be held in a non-shareable mode.
    - This means that only one process can hold the resource at a time. If another process requests it, it will be blocked until the resource is released.
2. **Hold-and-Wait**. There exists at least one process that is holding a resource and waiting to acquire another resource which is held by some other process.
3. **Non-Preemption**. No resources can be taken away from any process. Put another way, a resource can only be released voluntarily by the process that holds it when that process has finished using it.
4. **Circular Waiting**. There exists a set of processes, P_0, P_1, P_2,...,P_n such that for each k, for $0 \leq k < n$, P_k is waiting for a resource held by P_{k+1}, and P_n is waiting for a resource held by P_0.

   This last condition can only be true if **Hold-and-Wait** is also true, but not conversely.

About Circular Waiting

The definition of deadlock implies that circular waiting must be true, which implies that there must be a cycle in the resource allocation graph that represents the deadlock state.

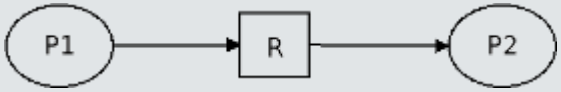
We illustrate this with an example. Suppose a set of five processes is in a deadlock state.

This means that each process is waiting for an event that can only be caused by another of the five processes.

This implies that each process is waiting for a resource that can only be released by one of the others.

For simplicity, assume that the event for which each process waits is the release of a resource `R` held by another process.

Instead of drawing the two edges representing one waiting for the resource held by the other in the way we did before:



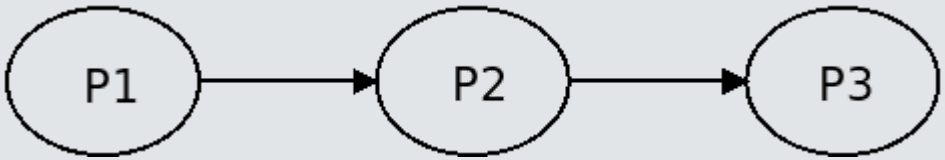
for simplicity we omit the resource from the graph, and combine two edges into one:

Circular Waiting and Cycles

Beging by picking any one of those processes, call it `P1`. Let `P2` be the process it waits for. Then there is an edge from `P1` to `P2`:

Circular Waiting and Cycles

We name the process that P2 waits for, P3 and draw an edge from P2 to P3:



Circular Waiting and Cycles

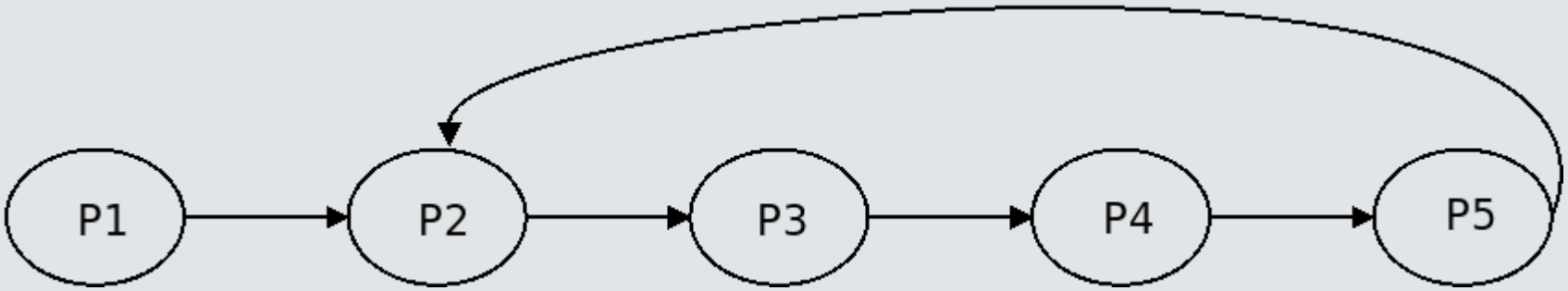
Next, we name the process that P3 waits for, P4 and draw an edge from P3 to P4:

Circular Waiting and Cycles

We name the process that P4 waits for, P5 and draw an edge from P4 to P5:

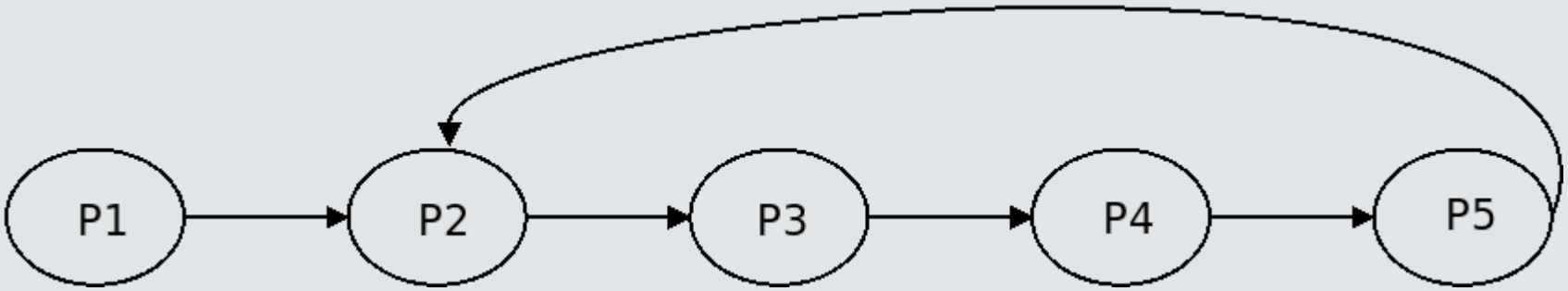
Circular Waiting and Cycles

Because all five processes are involved in this deadlock, P5 must be waiting for one of P1,P2,P3, or P4. While it is possible that it could be waiting for any of them, no matter which it waits for, there is a cycle in this graph. To illustrate we arbitrarily assume it is waiting for P2, which holds the resource it needs, and draw an edge from P5 to P2:



Circular Waiting and Cycles

Because all five processes are involved in this deadlock, P5 must be waiting for one of P1,P2,P3, or P4. While it is possible that it could be waiting for any of them, no matter which it waits for, there is a cycle in this graph. To illustrate we arbitrarily assume it is waiting for P2, which holds the resource it needs, and draw an edge from P5 to P2:

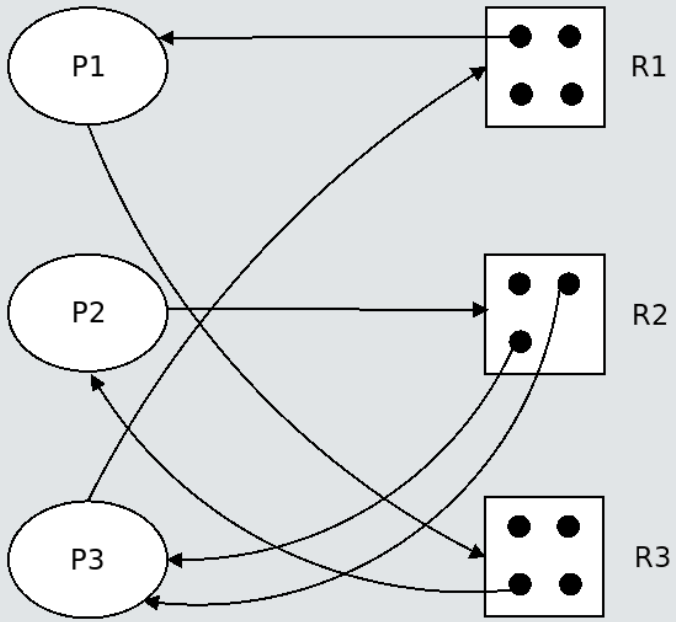


Notice that this graph has a **cycle**, a sequence of nodes and edges such that starting in any node and following the edges, we get back to that same node. The graphs that represent deadlock states in our model will always have cycles. The cycle in this case does not include all of the deadlocked processes.

Does the presence of a cycle in an RR-graph imply that it represents a deadlock state?

Cycles Do Not Imply Deadlock

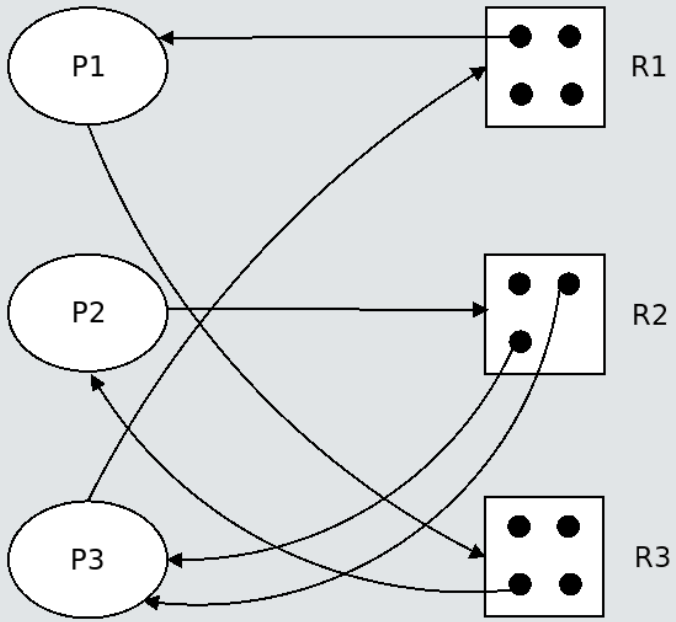
Consider the RR-graph from the earlier example:



In this graph, P2 is free to acquire a unit of R2, since P3 holds only two of them. Similarly, P1 can acquire a unit of R3, since three are available. Then all processes hold the resources they need, none are waiting, and so all release their units eventually, leading to a graph with no edges in it.

Cycles Do Not Imply Deadlock

Consider the RR-graph from the earlier example:



In this graph, P2 is free to acquire a unit of R2, since P3 holds only two of them. Similarly, P1 can acquire a unit of R3, since three are available. Then all processes hold the resources they need, none are waiting, and so all release their units eventually, leading to a graph with no edges in it.

Therefore, this graph does not represent a deadlock state.

A Theorem About Cycles

We can now state the following theorem:

Theorem. A cycle in a reusable resource graph is a necessary but not sufficient condition for it to represent a deadlock.

In other words, if it is a deadlock state, it must have a cycle, but having a cycle does not imply that it is a deadlock state.

A Theorem About Cycles

We can now state the following theorem:

Theorem. A cycle in a reusable resource graph is a necessary but not sufficient condition for it to represent a deadlock.

In other words, if it is a deadlock state, it must have a cycle, but having a cycle does not imply that it is a deadlock state.

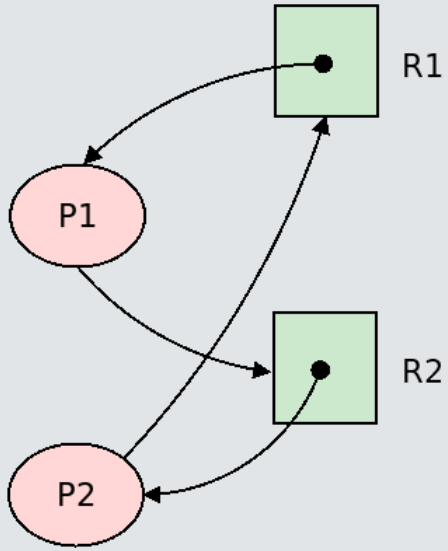
The preceding RR-graph provided the "witness" to the claim that graphs with cycles are not necessarily deadlock states.

The reason that the preceding RR-graph did not represent a deadlock state was that `P2` was free to acquire a unit of `R2` because `R2` had multiple units.

Another Theorem About Cycles

If every resource node contains just a single unit, then an outgoing edge implies that the resource is in use. If there is also an incoming edge, that resource is also requested. The process that requested it is blocked.

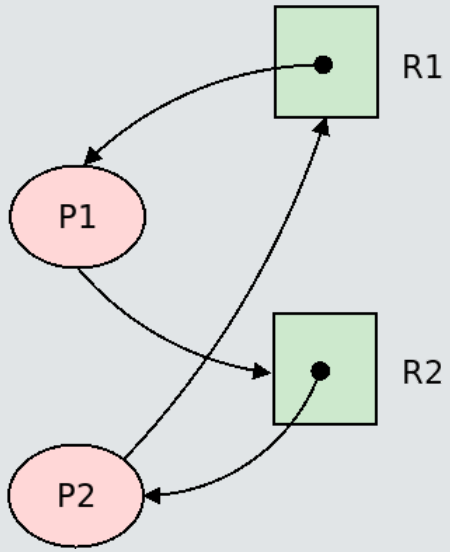
The graph to the right illustrates this. If there is a cycle, every process in that cycle must be blocked. In this graph both `P1` and `P2` are in the cycle and both are blocked. Neither will ever run again. This leads to the following:



Another Theorem About Cycles

If every resource node contains just a single unit, then an outgoing edge implies that the resource is in use. If there is also an incoming edge, that resource is also requested. The process that requested it is blocked.

The graph to the right illustrates this. If there is a cycle, every process in that cycle must be blocked. In this graph both `P1` and `P2` are in the cycle and both are blocked. Neither will ever run again. This leads to the following:



Theorem. A cycle in a reusable resource graph is a necessary and sufficient condition for it to represent a deadlock if every resource has just a single unit.

Using RR-Graphs

One reason to use RR-graphs is that they make things visual and therefore help us to understand deadlock in a more intuitive way.

For example, when a process P requests a resource R, we create a request edge P→R. When the resource is granted to the process, the request edge is removed and immediately replaced by an assignment edge P←R.

Using RR-Graphs

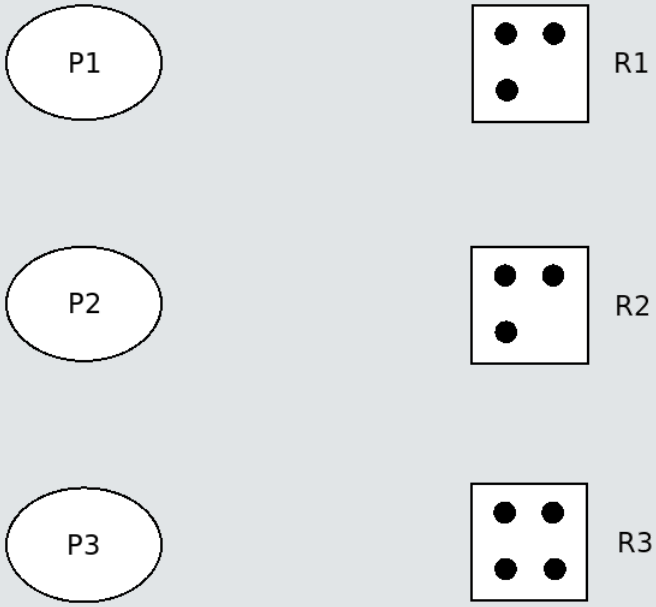
One reason to use RR-graphs is that they make things visual and therefore help us to understand deadlock in a more intuitive way.

For example, when a process P requests a resource R, we create a request edge P→R. When the resource is granted to the process, the request edge is removed and immediately replaced by an assignment edge P←R.

We can determine when a system enters a deadlock state by simulating the actions using this graph, as we illustrate next.

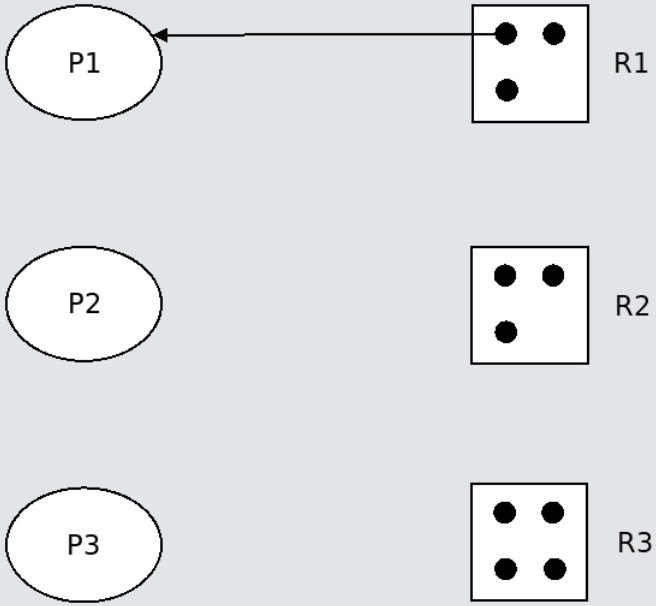
Example Leading to Deadlock

The initial state:



Examples of RR-Graphs

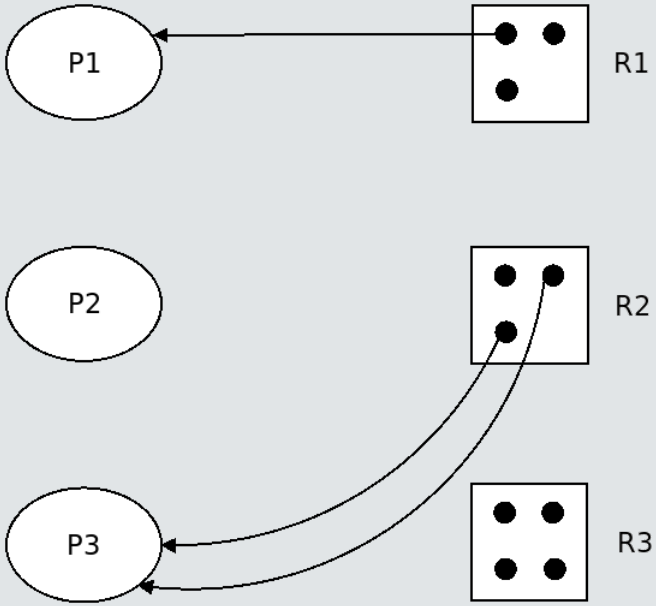
P1 requests requests and acquires 1 unit of R1:



No deadlock here.

Examples of RR-Graphs

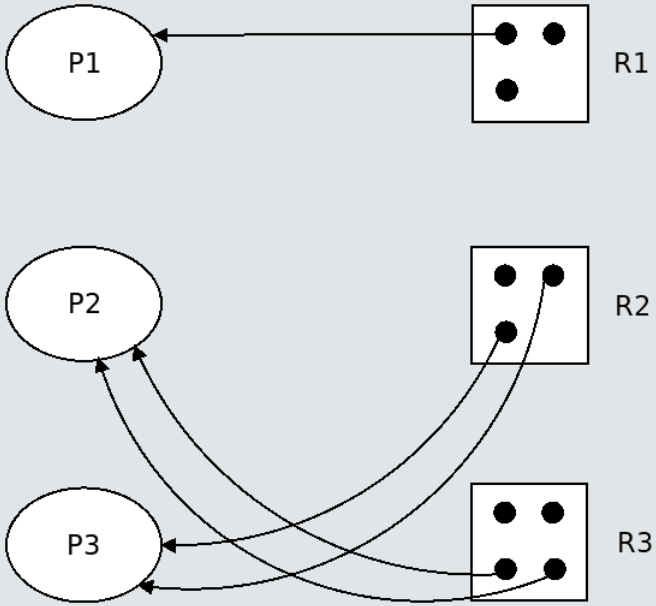
P3 requests and acquires 2 units of R2:



Still no deadlock here.

Examples of RR-Graphs

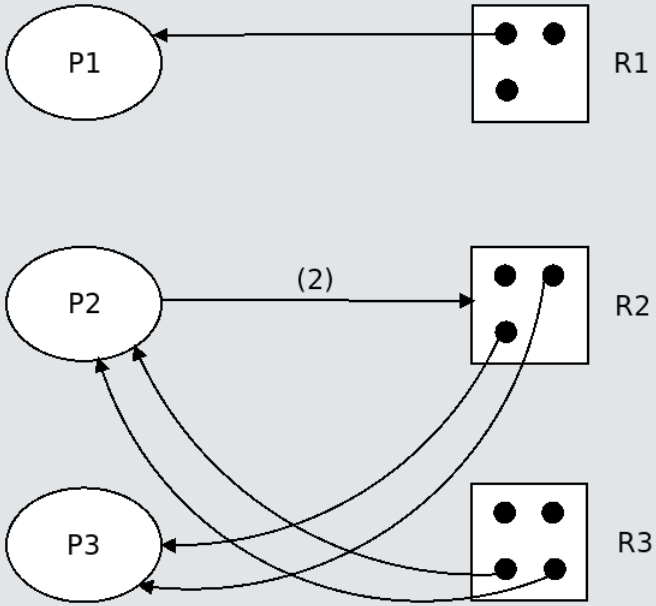
P2 requests and acquires 2 units of R3:



Still no deadlock here.

Examples of RR-Graphs

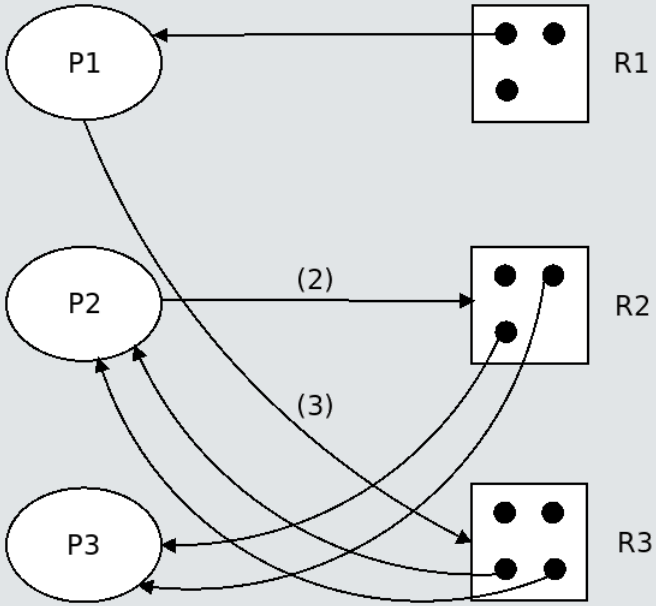
P2 requests 2 units of R2, which are unavailable, so it blocks. Instead of drawing multiple request edges, we draw a single edge labeled by how many units are requested.



Still no deadlock here.

Examples of RR-Graphs

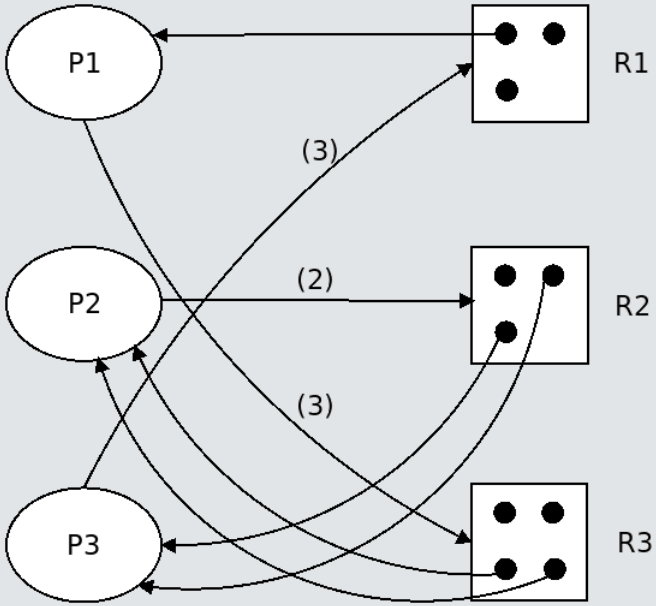
P1 requests 3 units of R3, which are unavailable, so it blocks.



Still no deadlock here.

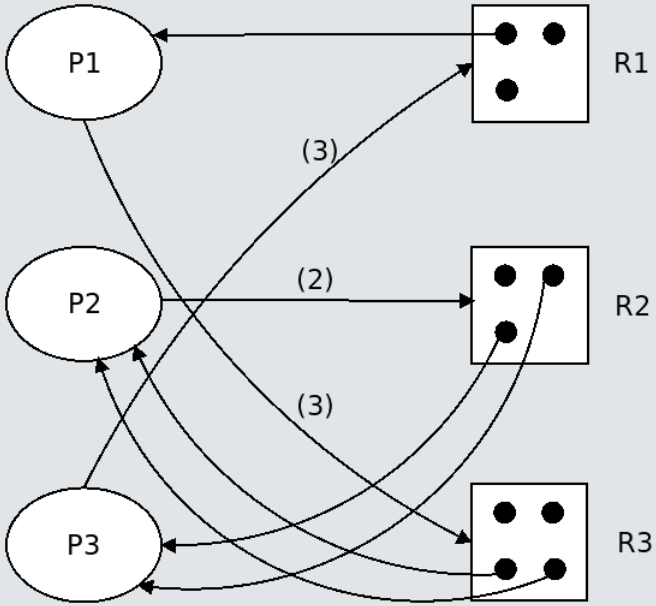
Examples of RR-Graphs

P3 requests 3 units of R1, which are unavailable, so it blocks.



Examples of RR-Graphs

P3 requests 3 units of R1, which are unavailable, so it blocks.



All processes are blocked waiting for resources held by the others, so this is a deadlock state.

Other Types of Waiting

With respect to the model we just described, the **event** for which a process waits is the release of a resource by the process that holds it.

Events can be other things such as the arrival of a message through a channel.

- One process can be waiting for a message that another process must send, and the second process is waiting for the first's message:

```
P1:
    receive(P2,m);
    send(P2,m1);
```

```
P2:
    receive(P1,m);
    send(P1,m2);
```

This is also deadlock. Messages can be treated as resources also.

Dealing With Deadlock

There are three general ways in which to deal with deadlock:

- **Prevention/Avoidance**. Some people distinguish between prevention and avoidance. The distinction is a subtle one.
    - Prevention is the act of making sure that one or more of the necessary conditions cannot exist, ever. It enforces control over how a process can request resources.
    - Avoidance methods are dynamic algorithms that are applied when the system has to decide whether to grant requests for resources to prevent a deadlock state from being reached. Avoidance algorithms generally control allocation as opposed to limiting requests.

  Both prevent deadlock from occurring.

- **Detection**. Detection is required when no attempt is made to prevent or avoid deadlock. In systems using detection the system periodically runs algorithms to detect whether deadlock exists.

- **Recovery**. If the system discovers that deadlock has occurred, this refers to the means by which it removes that deadlock.

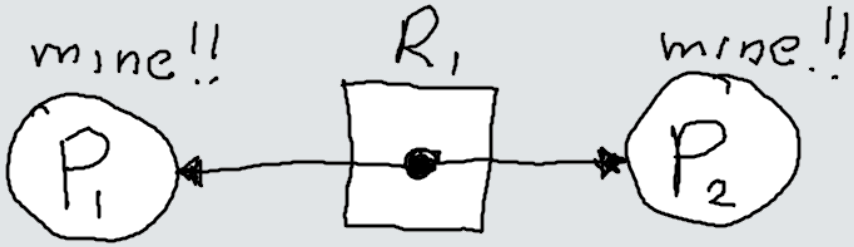
Prevention

Prevention must remove one of the necessary conditions. Recall that these are

1. Mutual Exclusion
2. Hold-and-Wait
3. Non-Preemption
4. Circular Waiting

Prevention: Removing Mutual Exclusion

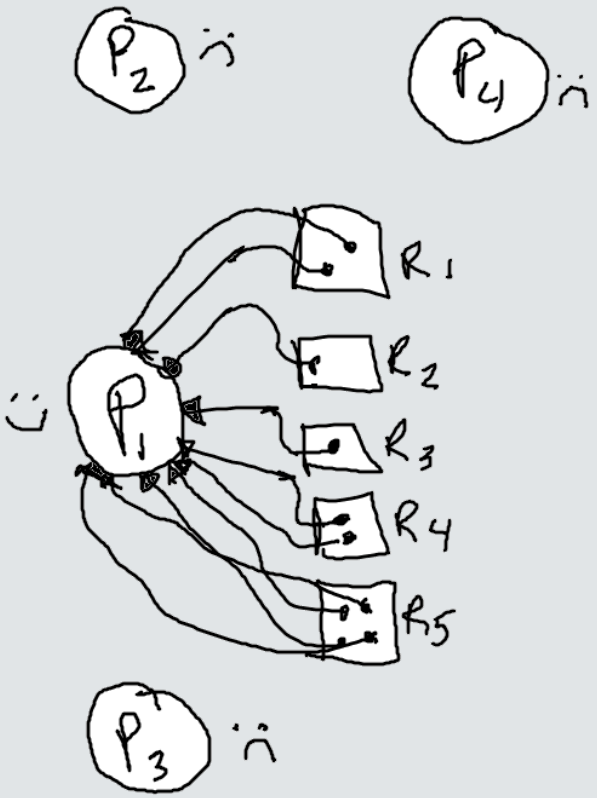
Mutually exclusive acquisition of resources cannot be removed for the types of resources that we study here - reusable, non-shareable resources. By definition they are held in mutual exclusion. Removing mutual exclusion would mean processes could simultaneously send files to a printer, or write to the same disk block, etc.



Prevention: Removing Hold-and-Wait

Removing Hold-and-Wait implies every process must acquire all resources it needs all at once. This results in

- **poor resource utilization**, since processes would hold resources they do not use for a long time, and
- potential **starvation** of processes that never get the resources they need.
- To the right, P_1 grabbed all resources and now no other process has any!



Prevention: Removing Non-Preemption

Removing non-preemption means **allowing preemption**. That means being able to forcibly take resources away from a process.

One protocol to do this is:

- If a process that is holding resources requests a resource that cannot be allocated immediately to it, then all of its currently held resources are released.
- The preempted resources are added to the list of resources for which the process is waiting and the process is blocked.
- The process is released to run only when it can acquire its old resources together with the ones that it requested after.

Prevention: Removing Non-Preemption

Removing non-preemption means **allowing preemption**. That means being able to forcibly take resources away from a process.

One protocol to do this is:

- If a process that is holding resources requests a resource that cannot be allocated immediately to it, then all of its currently held resources are released.
- The preempted resources are added to the list of resources for which the process is waiting and the process is blocked.
- The process is released to run only when it can acquire its old resources together with the ones that it requested after.

An alternative protocol is:

- When a process requests a resource, if it is available it is granted. If not and it is held by a blocked process, the kernel preempts the resource from that blocked process and gives it to the requesting process.
- If it is unavailable and not held by a blocked process, the requesting process is blocked. If another process requests any of the resources it holds, they are taken away from it. The blocked process is not restarted until it has all resources it needs, including any taken away and the ones it requested but was not granted.

Prevention: Removing Non-Preemption

These protocols can be used only for resources whose state can be saved and restored, such as hardware in general. It is not usually applied to soft resources such as locks, semaphores, etc.

Prevention: Removing Circular Waiting

In 1968, J. W. Havender proposed a method of deadlock prevention for the IBM)S/360 operating system by removing the circular waiting condition. (In fact he called it a deadlock *avoidance* method.)

Resource types are put into an ordering relation $R_1 < R_2 < \cdots < R_k$ [1] and processes adhere to a set of rules for making requests for resources.

Specifically, a process can request resources only in increasing order of its resource type:

- If a process holds R_i it can only request R_j if $i < j$.
- To request R_j, it must first release all R_k such that $j \leq k$; i.e., it cannot request a resource of a lower number than any it currently holds, so it must first release all resources of a higher number in the order.
- If a process violates these rules, it is terminated.

This makes circular waiting impossible, but it still means that sometimes resources are held that are not needed until later, resulting in reduced resource utilization.

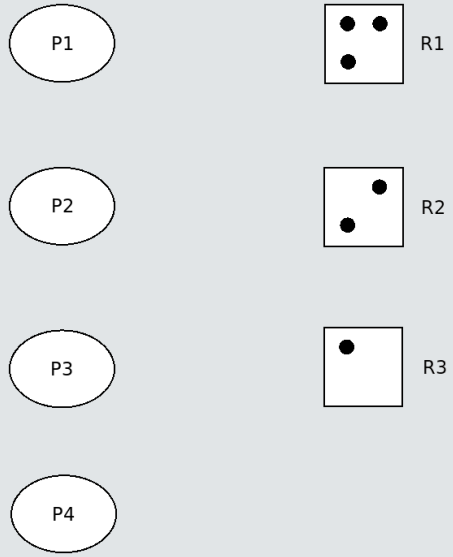
1 We describe this ordering in reverse to be consistent with the textbook's description.

Example of Ordered Requests

A system has three resource types:

- R1 has 3 units (files)
- R2 has 2 units (memory blocks)
- R3 has 1 unit (I/O device)

There are four processes, which make a sequence of requests adhering to the Havender rules. Initially there are no requests.



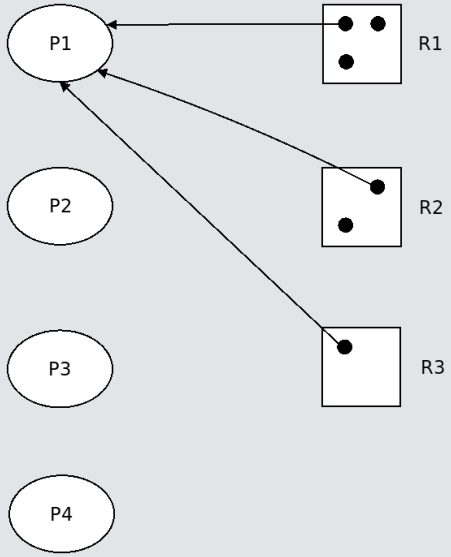
Example of Ordered Requests

A system has three resource types:

- R1 has 3 units (files)
- R2 has 2 units (memory blocks)
- R3 has 1 unit (I/O device)

There are four processes, which make a sequence of requests adhering to the Havender rules. Initially there are no requests.

1. P1 requests and acquires 1 unit each of R1, then R2, then R3.



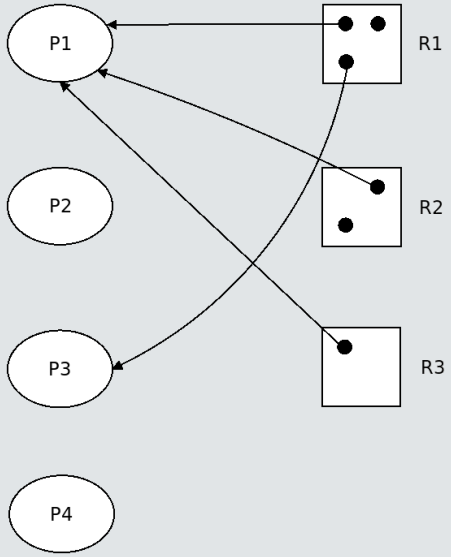
Example of Ordered Requests

A system has three resource types:

- R1 has 3 units (files)
- R2 has 2 units (memory blocks)
- R3 has 1 unit (I/O device)

There are four processes, which make a sequence of requests adhering to the Havender rules. Initially there are no requests.

1. P1 requests and acquires 1 unit each of R1, then R2, then R3.
2. P3 requests and acquires 1 unit of R1.



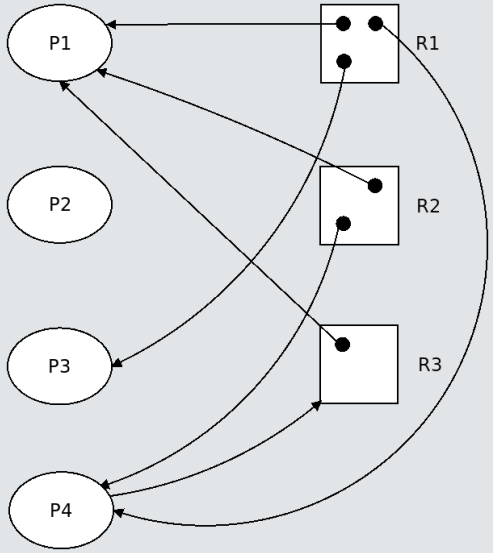
Example of Ordered Requests

A system has three resource types:

- R1 has 3 units (files)
- R2 has 2 units (memory blocks)
- R3 has 1 unit (I/O device)

There are four processes, which make a sequence of requests adhering to the Havender rules. Initially there are no requests.

1. P1 requests and acquires 1 unit each of R1, then R2, then R3.
2. P3 requests and acquires 1 unit of R1.
3. P4 requests and acquires 1 unit of R1 and 1 unit of R2. It then requests 1 unit of R3 but it is now blocked.



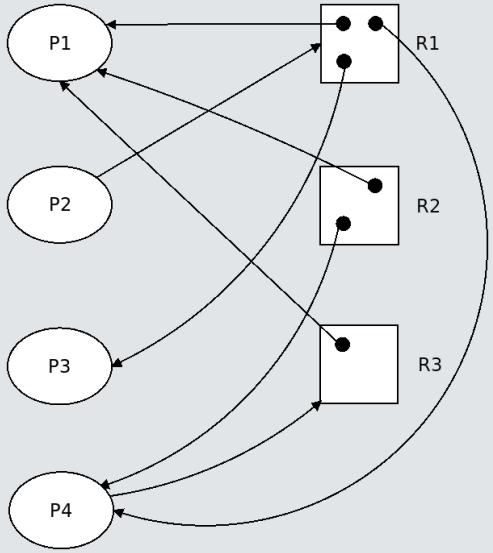
Example of Ordered Requests

A system has three resource types:

- R1 has 3 units (files)
- R2 has 2 units (memory blocks)
- R3 has 1 unit (I/O device)

There are four processes, which make a sequence of requests adhering to the Havender rules. Initially there are no requests.

1. P1 requests and acquires 1 unit each of R1, then R2, then R3.
2. P3 requests and acquires 1 unit of R1.
3. P4 requests and acquires 1 unit of R1 and 1 unit of R2. It then requests 1 unit of R3 but it is now blocked.
4. P2 requests 1 unit of R1 and is blocked (all 3 are held).



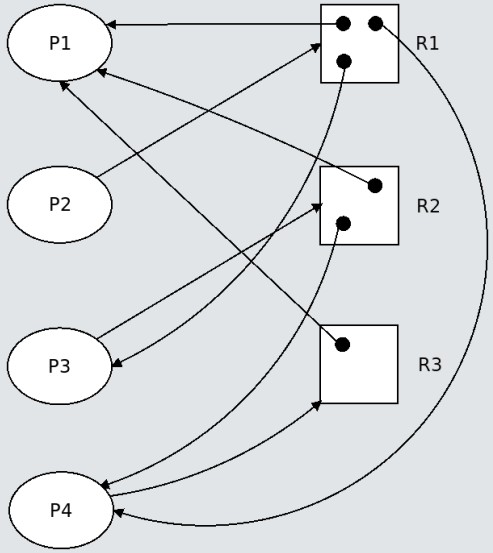
Example of Ordered Requests

A system has three resource types:

- R1 has 3 units (files)
- R2 has 2 units (memory blocks)
- R3 has 1 unit (I/O device)

There are four processes, which make a sequence of requests adhering to the Havender rules. Initially there are no requests.

1. P1 requests and acquires 1 unit each of R1, then R2, then R3.
2. P3 requests and acquires 1 unit of R1.
3. P4 requests and acquires 1 unit of R1 and 1 unit of R2. It then requests 1 unit of R3 but it is now blocked.
4. P2 requests 1 unit of R1 and is blocked (all 3 are held).
5. P3 requests 1 unit of R2 and is blocked.



Avoidance Algorithms

Up until now we did not consider the possibility of not granting a request for a resource if there were units available.

We assumed that when a process made a request for a resource, if the resource were available, the request was granted immediately.

When this is a standard policy of the operating system, then all states are **expedient states**. In an **expedient state**, any process with an outstanding request is in a blocked state, because all requests that could be satisfied have been satisfied.

Avoidance Algorithms

Up until now we did not consider the possibility of not granting a request for a resource if there were units available.

We assumed that when a process made a request for a resource, if the resource were available, the request was granted immediately.

When this is a standard policy of the operating system, then all states are **expedient states**. In an **expedient state**, any process with an outstanding request is in a blocked state, because all requests that could be satisfied have been satisfied.

One way to prevent deadlock is to use *a priori* additional information to allow the system to decide when and whether to grant requests instead of granting them automatically. This is called **deadlock avoidance**.

Avoidance Algorithms Using Maximum Needs

One type of information that makes deadlock avoidance possible is the **maximum need** of each process for every resource type. If this is known in advance, deadlock is avoidable.

The maximum need of a process for a single resource type is the largest number of units of that resource that it will need to hold **at the same time**.

> The **maximum need** of a process is the maximum number of units of each resource type that it will hold **at the same time**.

We can represent the maximum need of a process `P` by a vector of length `t` such as this one:

	R1	R2	R3	R4
P	7	5	3	5

in which the process will need to hold 7 units of `R1`, 5 units of `R2`, 3 units of `R3`, and 5 units of `R4` all at the same time.

Each process must declare its needs in advance.

Representing Maximum Needs

The maximum needs of all processes can be represented by a two-dimensional matrix such as the following, for a system with five processes and three resource types.

	R1	R2	R3
P1	7	5	3
P2	3	6	3
P3	9	0	2
P4	2	3	2
P5	4	3	1

The matrix shows that process P3 will never need any units of R2 and will never need more than 9 units of R1 and 2 units of R3 at any given time.

Use of Maximum Need

If we know a process's maximum need, and all of the resources are available to satisfy that need, then if we grant that process all of its needed resources, it will complete its computation using those resources and then release them, freeing up all resources that it currently holds plus those it was just granted.

In effect, by granting resources to processes that can finish up their work if they are given those resources, we increase the available resources in the system

This is the strategy of the Banker's Algorithm that we present shortly.

Safe (and Unsafe) System States

To prevent deadlock by using the maximum needs of all processes, the resource allocation state of a system must also include information about the maximum need of each process. If it has this information, it can distinguish between **safe** and **unsafe** states.

A state is **safe** if it is possible for the system to allocate resources to each process in such a way as to avoid deadlock. It is an **unsafe state** if no such sequence exists. Unsafe states can lead to deadlock.

Safe (and Unsafe) System States

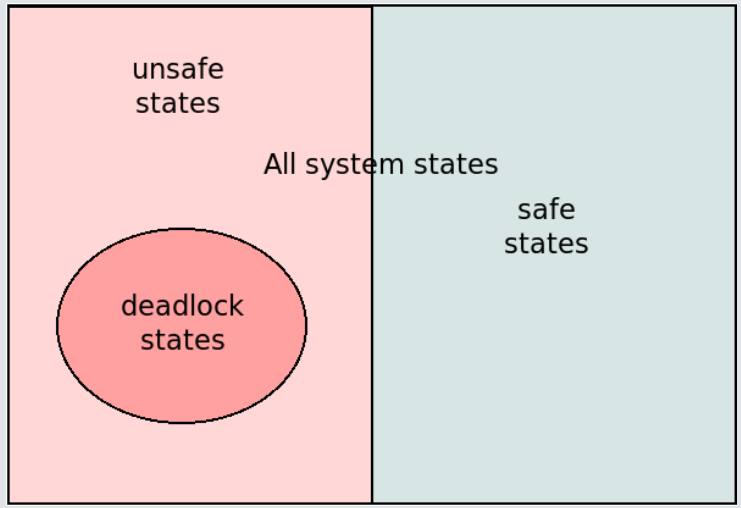
To prevent deadlock by using the maximum needs of all processes, the resource allocation state of a system must also include information about the maximum need of each process. If it has this information, it can distinguish between **safe** and **unsafe** states.

A state is **safe** if it is possible for the system to allocate resources to each process in such a way as to avoid deadlock. It is an **unsafe state** if no such sequence exists. Unsafe states can lead to deadlock.

This implies that:

- No safe state is a deadlock state.
- Unsafe states are not necessarily deadlock states but might be, and could lead to deadlock by the wrong sequence of allocations.
- All deadlock states are unsafe.

The figure to the right illustrates the relationship.



Avoiding Deadlock Using Safe States

A resource allocation algorithm can avoid deadlock using the following strategy:

- When a process requests a resource, the system decides if granting the request would leave the system in a **safe state**:
  - If so, it grants the request, because from this state **it can still avoid deadlock**.
  - If not, it does not grant the request, because from this state, deadlock might be inevitable.

Avoiding Deadlock Using Safe States

A resource allocation algorithm can avoid deadlock using the following strategy:

- When a process requests a resource, the system decides if granting the request would leave the system in a **safe state**:
    - If so, it grants the request, because from this state **it can still avoid deadlock**.
    - If not, it does not grant the request, because from this state, deadlock might be inevitable.

This leads to a concept of **safe sequences**.

Avoiding Deadlock Using Safe States

A resource allocation algorithm can avoid deadlock using the following strategy:

- When a process requests a resource, the system decides if granting the request would leave the system in a **safe state**:
  - If so, it grants the request, because from this state **it can still avoid deadlock**.
  - If not, it does not grant the request, because from this state, deadlock might be inevitable.

This leads to a concept of **safe sequences**.

A **safe sequence** is a sequence < P1, P2,..., PN > of **all processes** in the system such that for each Pi, the resources that Pi requests can be satisfied by the available resources at the time of the request together with the resources held by all of the Pj, for which j < i.

In essence, a safe sequence is a sequence of processes to which resources should be granted to avoid deadlock and that leads to termination of all processes.

The Banker's Algorithm

The **Banker's Algorithm** is an example of an algorithm using this strategy. It was invented by Edsger Dijkstra in 1965 and is modeled on the way banks make loans[1].

In the Banker's Algorithm, when a process makes a request, it is granted only if the resulting system state is a safe state, meaning that there is a way to avoid deadlock in this state while still continuing to allocate resources to each process.

It has two parts - a **safety checking** algorithm that checks if a state is safe and a **request-simulation** algorithm that pretends to grant a request and computes the resulting state.

1. E. Dijkstra, "Cooperating Sequential Processes," Technological University, Eindhoven, The Netherlands, 1965.

Banker's Algorithm Data Structures

Assume `N` is the number of processes and `t` is the number of resource types.

The algorithm uses four data structures:

Name	Meaning
`int Available[t]`	`Available[j] == k` if and only if `k` instances of resource type `Rj` are available.
`int Max_Claim[N][t]`	`Max_Claim[i,j] == k` if and only if `Pi` may hold at most `k` instances of resource type `Rj`
`int Allocation[N][t]`	`Allocation[i,j] == k` if and only if `Pi` is currently allocated `k` instances of `Rj`
`int Need[N][t]`	`Need[i,j] == k` if and only if `Pi` may need `k` more instances of `Rj` to complete its task.

The `Need` matrix is related to the `Max_Claim` and `Allocation` matrices as follows:

```
Need[i,j] = Max_Claim[i,j] - Allocation[i,j]
```

The Safety Algorithm

Given a state characterized by the contents of `Available`, `Allocation`, and `Need`, the safety algorithm determines whether the state is safe or not.

The algorithm uses the following variables, initialized as shown. We use vector operators to simplify the code. An assignment `A = B` where `A` and `B` are vectors of length `m`, is short for

```
for ( i = 0; i < m; i++) A[i] = B[i];
```

Similar shorthand is used for relational operators.

Initializations:

```
int Work[t]   = Available;
int Finish[N] = {false};
int safe = true;
int found, done = false;
```

The Safety Algorithm

Given a state characterized by the contents of `Available`, `Allocation`, and `Need`, the following algorithm tries to find a running process `Pi` whose maximum need can be satisfied by the available resources. It returns `true` if one is found and `false` otherwise.

```
while ( ! done ) {
    found = false;
    for ( i = 0; i < N; i++) {
        if ( Finish[i] == false && Need[i] ≤  Work ) {
            // Process Pi meets the criteria, so can be granted its request
            found = true;
            break;
        }
    }
    if ( found ) {
        Work = Work + Allocation[i];
        Finish[i] = true;
    }
    else
        done = true;
 }

 for ( i = 0; i < N, i++) {
     safe = safe &&  Finish [i];  // return true iff all processes terminated
 }
 return safe;
```

The Request-Simulation Algorithm

Let `Request[i]` be the request vector for process `Pi`. If `Request[i][j] == k`, then `Pi` requests `k` instances of resource type `Rj`. When `Pi` makes a request for resources, the following algorithm is run:

```
if ( Request[i] > Need[i] )
    raise an error condition and exit; // the process has exceeded its maximum claim

if ( Request[i] > Available )
    make Pi wait until the resources are available;

// pretend to allocate the requested resources to Pi 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 algorithm on the resulting state:
safe = is_safe(Available, Allocation, Need);
if ( safe )
    grant the request to Pi ;
else {
    make Pi wait for Request[i] to be satisfied;
    restore the state to the original state;
}
```

Safety Algorithm Example

Assume there are five processes and three resource types and that the total resources of each type are

R1	R2	R3
10	5	7

Safety Algorithm Example

The current maximum claims and allocations are:

Max_Claim

	R1	R2	R3
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocation

	R1	R2	R3
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Total	7	2	5

Safety Algorithm Example

The current maximum claims and allocations are:

Max_Claim

	R1	R2	R3
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocation

	R1	R2	R3
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2
Total	7	2	5

Since `Need = Max_Claim - Allocation`, the `Need` matrix is

	R1	R2	R3
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

and `Available = [10,5,7] - [7,2,2] = [3, 3, 2]`

We check whether this is a **safe state** using the safety algorithm.

Safety Algorithm Example

`Work` is initially a copy of `Available` and `Finished[i]` is initially false for all processes:

```
Work = [ 3, 3, 2 ];

Finished = [ false, false, false, false, false ];
```

Safety Algorithm Example

`Work` is initially a copy of `Available` and `Finished[i]` is initially false for all processes:

```
Work = [ 3, 3, 2 ];

Finished = [ false, false, false, false, false ];
```

The `Need` matrix is examined to find a process that is not finished and whose `Need` is less than `Work`. In this case either `P1` or `P3` satisfies the condition. We choose `P1` and update the vectors:

Need

	R1	R2	R3
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Allocation

	R1	R2	R3
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Its allocation is added to `Work` and we mark it as finished.

Safety Algorithm Example

`Work` and `Finished` are now:

```
Work = [ 5, 3, 2 ];

Finished = [ false, true, false, false, false ];
```

The `Need` matrix is examined to find another process that is not finished and whose `Need` is less than `Work`. `P3` is chosen.

Need

	R1	R2	R3
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Allocation

	R1	R2	R3
P0	0	1	0
P1	0	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Its allocation is added to `Work` and we mark it as finished.

Safety Algorithm Example

`Work` and `Finished` are now:

```
Work = [ 7, 4, 3 ];

Finished = [ false, true, false, true, false ];
```

The `Need` matrix is examined to find another process that is not finished and whose `Need` is less than `Work`. `P0` is chosen.

Need

	R1	R2	R3
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Allocation

	R1	R2	R3
P0	0	1	0
P1	0	0	0
P2	3	0	2
P3	0	0	0
P4	0	0	2

Its allocation is added to `Work` and we mark it as finished.

Safety Algorithm Example

`Work` and `Finished` are now:

```
Work = [ 7, 5, 3 ];

Finished = [ true, true, false, true, false ];
```

The `Need` matrix is examined to find another process that is not finished and whose `Need` is less than `Work`. `P2` is chosen.

Need

	R1	R2	R3
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Allocation

	R1	R2	R3
P0	0	0	0
P1	0	0	0
P2	3	0	2
P3	0	0	0
P4	0	0	2

Its allocation is added to `Work` and we mark it as finished.

Safety Algorithm Example

`Work` and `Finished` are now:

```
Work = [ 10, 5, 5 ];

Finished = [ true, true, true, true, false ];
```

The `Need` matrix is examined to find another process that is not finished and whose `Need` is less than `Work`. Only `P4` remains and it satisfies the condition.

Need

	R1	R2	R3
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Allocation

	R1	R2	R3
P0	0	0	0
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Its allocation is added to `Work` and we mark it as finished.

Safety Algorithm Example

All processes are finished and so this state is safe:

```
Work = [ 10, 5, 7 ];

Finished = [ true, true, true, true, true ];
```

We see that <P1, P3, P0, P2, P4> is a safe sequence in this state.

Request-Simulation Algorithm Example

Suppose that, in this state, P1 requests one additional unit of R1 and two of R3. Then Request[1] = [1,0,2]. To decide whether this request can be granted immediately, we run the simulation algorithm.

The first two steps check whether Request[1] > Need[1] and then whether Request[1] > Available.

Since [1,0,2] ≤ [1,2,2], Request[1] ≤ Need[1]. Since [1,0,2] ≤ [3,3,2], Request[1] ≤ Available.

Therefore the request is valid and we pretend to grant it by changing Available to [2,3,0] to reflect that we allocated [1,0,2] to P1 and changing Need[1] and Allocation:

Need

	R1	R2	R3
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Allocation

	R1	R2	R3
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2

Request-Simulation Algorithm Example

We now check whether this state is safe, using the safety algorithm.

`Work` is initially a copy of `Available` and `Finished[i]` is initially false for all processes:

```
Work = [ 2,3,0 ];
```

```
Finished = [ false, false, false, false, false ];
```

In this state only `P1` and `P3` have needs that can be granted immediately. We combine two steps and assume `P1` and then `P3` have the requests granted.

Their combined allocation is `[5,1,3]`, which is added to `Work`.

Request-Simulation Algorithm Example

The resulting state is

```
Work = [ 7,4,3 ];

Finished = [ false, true, false, true, false ];
```

Need

	R1	R2	R3
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Allocation

	R1	R2	R3
P0	0	1	0
P1	0	0	0
P2	3	0	2
P3	0	0	0
P4	0	0	2

It is not hard to see that any of the remaining processes can have its request granted and so there are several different safe sequences in this case. For example, <P1, P3, P0, P2, P4> is a safe sequence.

It is therefore safe to grant the request to P1.

Deadlock Detection

Most systems do not do anything about deadlock. They just allow the system to enter a deadlock state if it happens.

Such systems can use a **deadlock detection algorithm** to discover when deadlock has occurred.

If deadlock is detected, the system can then use a **deadlock recovery algorithm** to undo the deadlock.

Deadlock Detection Algorithms

Deadlock detection algorithms check whether the circular wait condition exists.

- It simulates the most favorable execution of each unblocked process, by granting it all resources for which it has outstanding requests if they are available.
- Specifically:

    1. It searches for an unblocked process that can acquire all of its needed resources; if no such process exists, it jumps to step 6.
    2. Such process "runs" and then releases all resources it holds (reducing the RR-graph).
    3. And then becomes dormant.
    4. The released resources increase the total available and may as a result wake up some previously blocked process.
    5. It then repeats starting with step 1.
    6. If any blocked processes remain, they are deadlocked. If all processes have finished, no deadlock exists.

Deadlock Detection Algorithm

The preceding algorithm is similar to the safety algorithm but it does not require maximum claim information.

But it is $O(t \cdot N^2)$ where t is the number of resource types and N is the number of processes and is therefore costly to run.

When, and how often, to invoke it depends on:

- How often a deadlock is likely to occur?
- How many processes will need to be rolled back? one for each disjoint cycle?
- If a deadlock detection algorithm is run at random times, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
- It is better to run it only when a process has made a request for resources, since it is only new requests that can cause deadlock, not after releases.

Deadlock Recovery

There are various strategies for recovering from deadlock.

- The simplest - abort all deadlocked processes. This is very costly though easy to implement.
- More complex - abort one process at a time until the deadlock cycle is eliminated But, in which order should they be aborted? Choices:

    1. Priority of the process
    2. How long process it has computed, and/or how much longer to completion
    3. How many resources the process has acquired
    4. How many resources the process needs to complete
    5. How many processes will need to be terminated if this one is terminated (children, and other processes communicating with process)
    6. Is the process interactive or batch?

References

1. J. W. Havender, “Avoiding Deadlock in Multitasking Systems”, IBM Systems Journal, Volume 7, Number 2 (1968), pages 74-84.

2. Richard C. Holt, "Some Deadlock Properties of Computer Systems", ACM Computing Surveys, Volume 4, Number 3 (1972), pages 179-196.

3. E. Dijkstra, "Cooperating Sequential Processes," Technological University, Eindhoven, The Netherlands, 1965.

4. Abraham Silberschatz, Greg Gagne, Peter B. Galvin. **Operating System Concepts**, 10th Edition. Wiley Global Education, 2018.