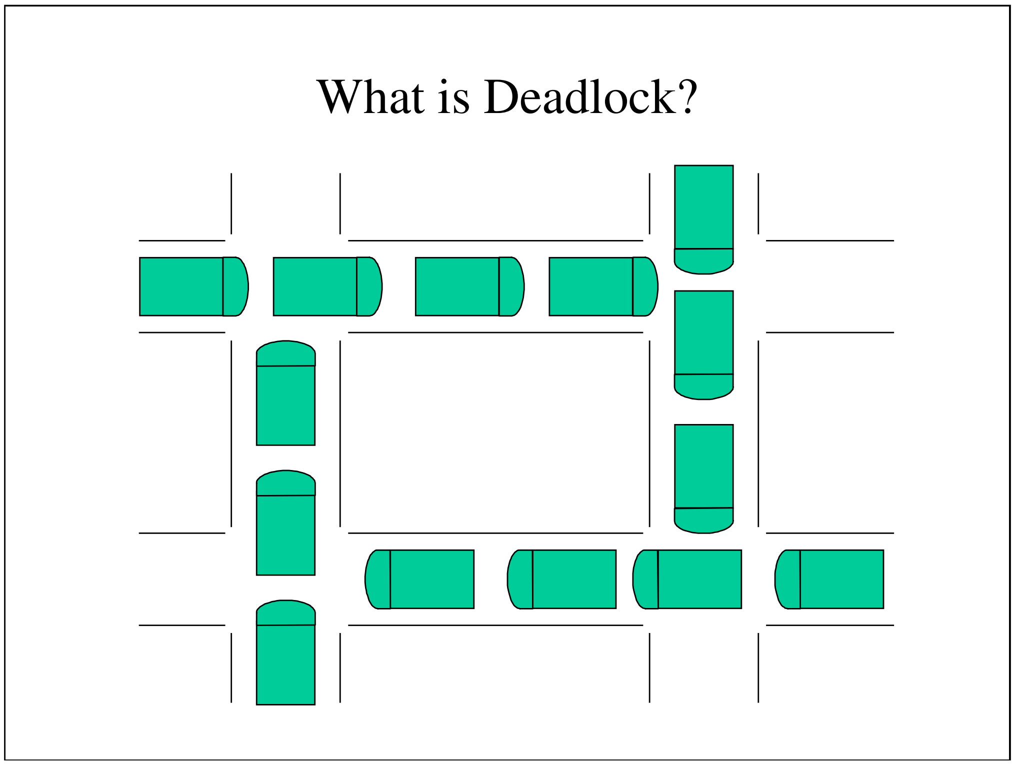
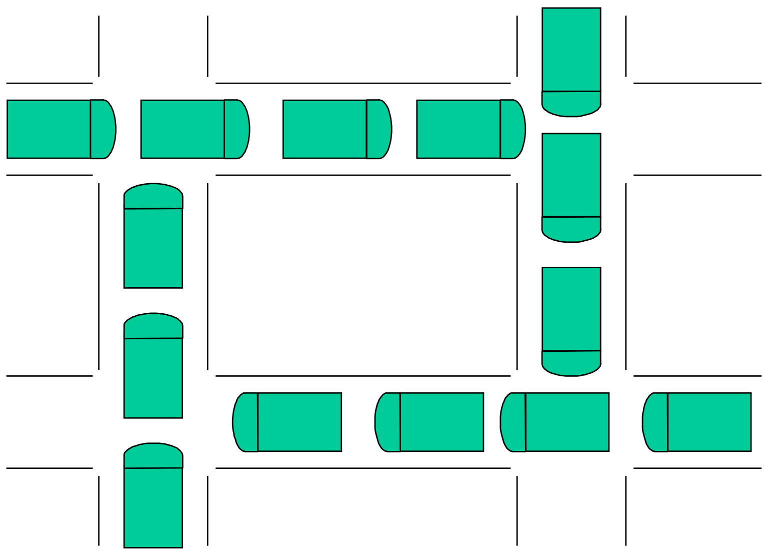
CS 453 Operating Systems

Lecture 7 : Deadlock

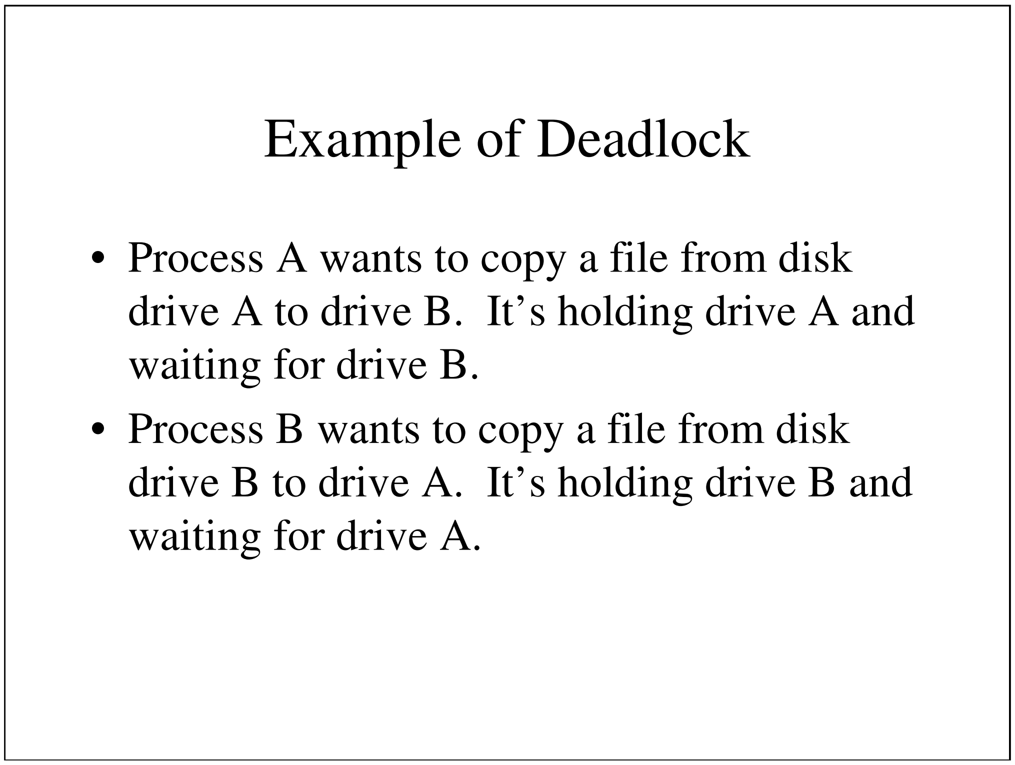


What is Deadlock?



Every New Yorker knows what a gridlock alert is - it’s one of those days when there is so much traffic that nobody can move. Everything just sits there! The REAL reason for gridlock is that there are so many cars getting stuck in the intersection that traffic on the cross streets can’t get through. As a result, that street backs up as well, and after a while, if there are enough cars on the street, no one can move because all the intersections are blocked and one way or the other.

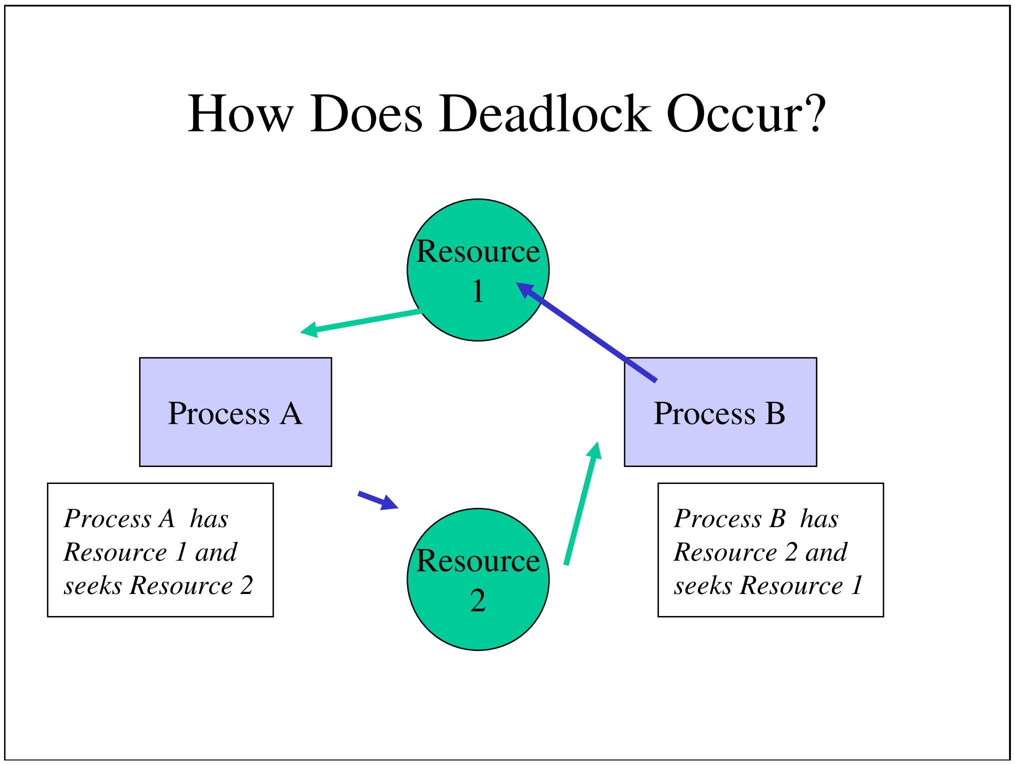
Essentially, this is what deadlock is. Processes effectively block each other, because each process is standing in the way of another process trying to finish its jobs. Each process is holding something that the other process wants, and as a result, neither can move foreward.



Example of Deadlock

- Process A wants to copy a file from disk drive A to drive B. It’s holding drive A and waiting for drive B.
- Process B wants to copy a file from disk drive B to drive A. It’s holding drive B and waiting for drive A.

Imagine two processes running on the same computer. Both are trying to copy files. One process is trying to copy from Drive A to Drive B. The other process is doing the reverse. And they both request (and get) the drive from which they are reading. The end result is that they are both waiting for the drive onto which they are going to write - which the other one is holding onto for dear life because they are going to read from it. Without some intervention, this standoff can continue forever - unless we unplug the computer first!



How Does Deadlock Occur?

Process A has Resource 1 and seeks Resource 2

Process B has Resource 2 and seeks Resource 1

This diagram illustrates the problem pretty well. The green circles represent the resources and the blue rectangles the processes. The green arrows show that Resource 1 belongs to Process A and that Resource 2 belongs to Process B. The blue arrows show that Process A is waiting for Resource 2 and that Process B is waiting for Resource 1. Unless they can share these resources, we have a 2-process deadlock.

We'll see quite soon that deadlocks do not necessarily involve two processes. This little standoff can involve a larger circle of processes, which actually makes it more dangerous. It isn't a matter of starving one or two processes - it can actually involve a fairly large number directly or indirectly!

Processes and Resources

A process using a resource goes through the following sequence of events:

- **Request**: The process asks the operating system for the resource. If it isn't available immediately, it waits until it is available.
- **Use**: The process does whatever operations it needs to do using the resources that it requested.
- **Release**: The process surrenders the resource back to the operating system

There are essentially 3 events that describe the role that a resource plays in the life of a process. The process must first request the use of the resource. Since it may very well not be available immediately, the process may have to wait. This is the stage where the process may be deadlocked. After securing the resource and whatever other resources it may need, the process uses the resource. After it's finished with its need of the resource, it releases it and now it can be used by other processes. Release is also important, because if processes do not release their resources, other processes can't use them.

The Necessary Conditions For Deadlock

- A deadlock can only happen when these four conditions are met:
  - **Mutual exclusion** – At least one resources can be used by only one process at a time
  - **Hold and wait** – A process will hold a resource while waiting for another one.
  - **No preemption** – A resource cannot be taken away from a process that has not released it voluntarily.
  - **Circular wait** – E.g, P_0 is waiting for a resource that P_1 is holding, P_1 is waiting for a resource that P_2 is holding, and P_2 is waiting for a resource that P_0 is holding.

There are four conditions that must be true for deadlock to occur. These are mutual exclusion, hold and wait, lack of pre-emption and circular wait.

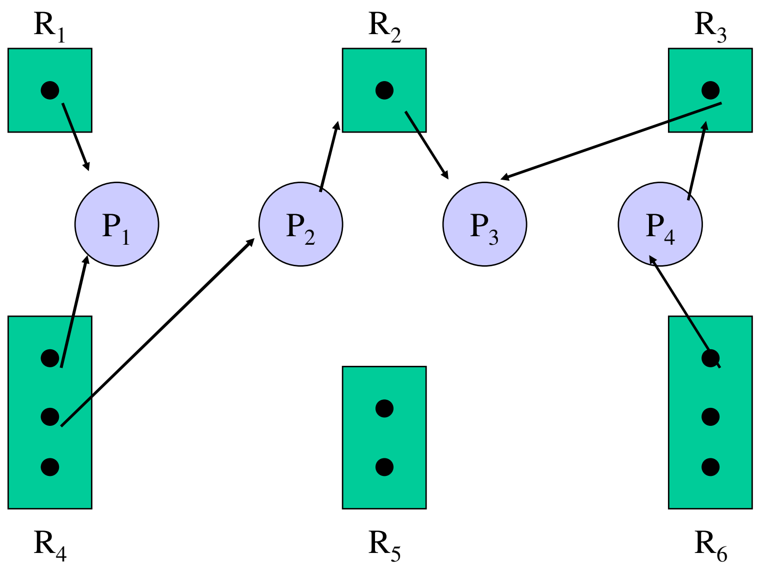
Mutual exclusion means that only one process can use a resource at a time. This is fairly common. Only one process can read from a keyboard at a time, for example. But even resources that can be shared may only allow x number of processes to use it at a time. The sign that barber shops used to have "8 chairs, no waiting" assumed that there would never be more than 8 customers in the shop at once to get their hair cut. If there were, then there would be some waiting.

Hold and wait means that a process can hold onto a resource while it waits for another. The assumption is that it is using the two resources in conjunction, such as copying or printing a file.

No pre-emption means that a process cannot be strip of a resource without surrendering it voluntarily. Pre-empting the resources of a process means that when it resumes, it must somehow replace the resource that it lost and find a reasonable place to resume execution.

Circular waiting means that A is waiting for B which is waiting for C which is waiting for A. This, taken together with hold and wait, means that they are each holding something that the other one wants. Mutual exclusion means that the very fact that they are holding it prevents the other from having it. And no preemption means that neither will give it up.

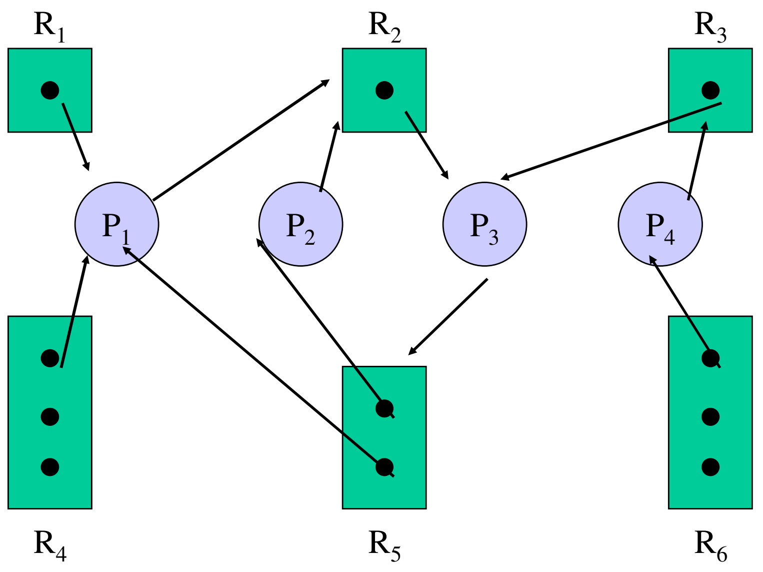
Resource-Allocation Graph



A resource-allocation graph is a directed graph that shows us each resource (along with each instance of it which can be reserved) with the processes that have requested them and are waiting for them, each process with the resources that are reserved them.

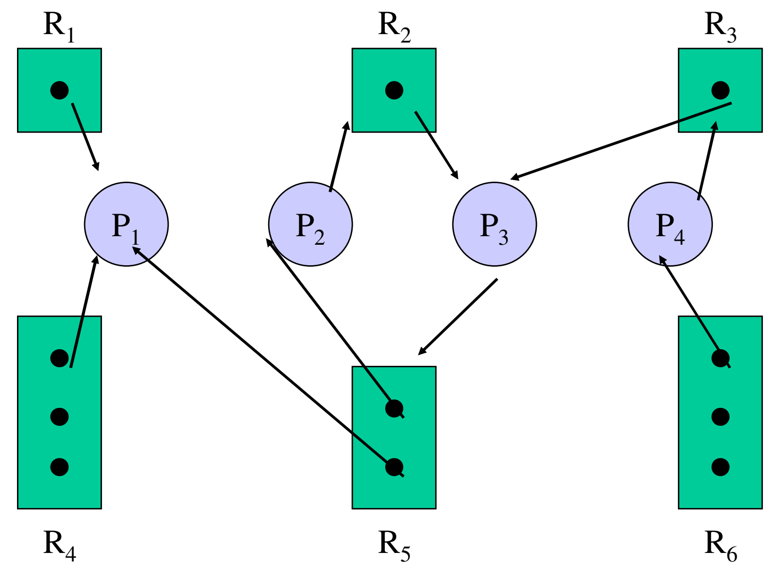
An arrow pointing from resource to process means that the process is holding the resources. The arrow pointing from resource to process means that the process is waiting for the resource. If any of these arrow completed a cycle or circuit, then that would indicate circular wait. You don't see any circuits here, so there is no circular wait, hence no deadlock.

Resource-Allocation Graph Showing Deadlock



A circuit does not necessarily mean a deadlock, although it will indicate the possibility of one. Here you'll notice that there is clearly a circuit from R_5 to P_2 to R_2 to P_3 back to R_5. That means that there is circular wait. But since R5 has two instances, each of which can be reserved, we need that second circuit from R_5 to P_1 to R_2 to P_3 back to R_5 . Since this completes a circuit which includes the same resource (R_2), we have a deadlock.

Resource-Allocation Graph Showing A Circuit But No Deadlock



This diagram includes a circuit, but there is no deadlock. R_5 has a second instances which was reserved by P_1. Since P_1 can complete its execution - it has all the resources it needs) - it will free R_5 and then P_3 can use R_5 and complete its own execution. Lastly, P_2 can finish. Since they all can finish, there is no deadlock.

How To Handle Deadlock

- We can prevent deadlock from ever happening.
- We can allow deadlock, detect it when it happens and recover from it.
- We can pretend that deadlock never happens.

How do we handle deadlock? For all practical purposes, we can either prevent, detect it whenever it happens and then recover from it, or simply pretend that it never happens. Many operating systems take this last approach, which is not without problems. There is a story of one system that was shut down and they discovered on it a process that had been waiting for ten years. Ignoring deadlock can potentially lead to situations like this.

Deadlock Prevention

Deadlock prevention means that we will eliminate one of the four necessary conditions for deadlock:

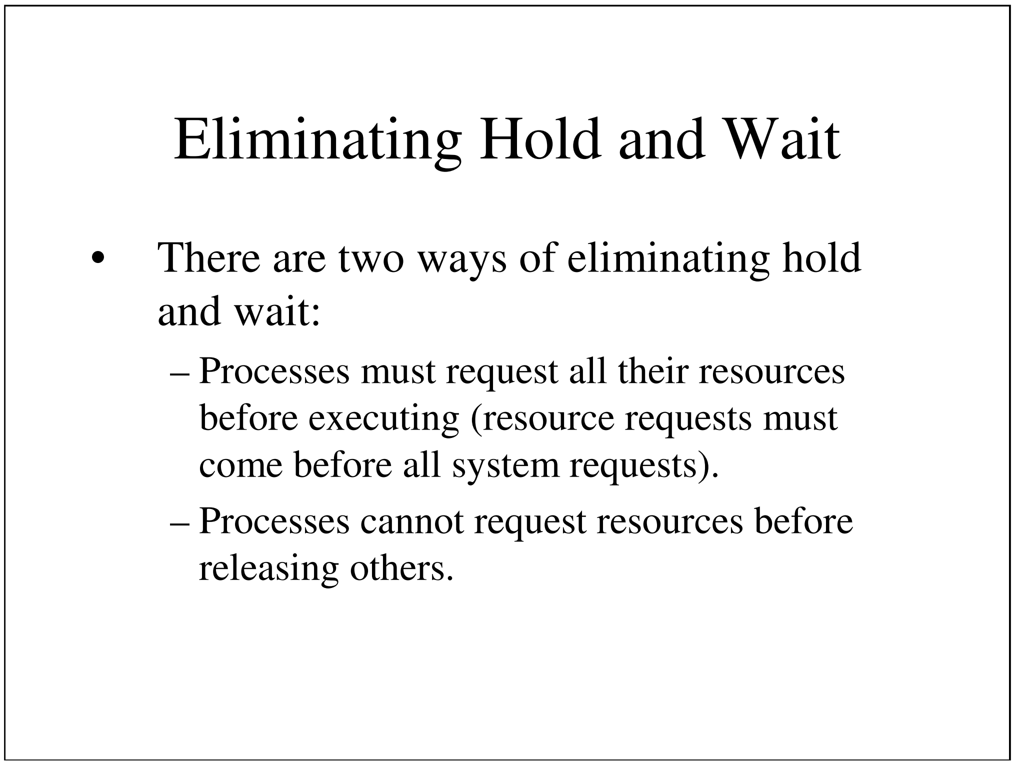
- We cannot eliminate mutual exclusion
- We can forbid processes to request resources while holding others before executing. *(Eliminates hold and wait)*
- We can require that a process denied a resource must give up the others. *(Eliminates no pre-emption)*
- We can require that resources be requested in a specific order. *(Eliminates circular waiting)*

Preventing deadlock, in theory, is simple: prevent one of the four necessary conditions from becoming true. The most immediate problem is that you cannot prevent the first condition in most instances where you find it. The only way to eliminate mutual exclusion is a hardware solution: increase the number of devices. But even that will always eliminate mutual exclusion.

Any possibility is to prevent processes from holding resources while they wait for other resources. This prevents the hold and wait condition.

We can require that a process that was denied a resource give up the ones that it is holding. This adds the possibility of pre-emption, preventing deadlock.

Lastly, we can require that resources be requested and allocated in a specific order. This eliminates the possibility of circular waiting.



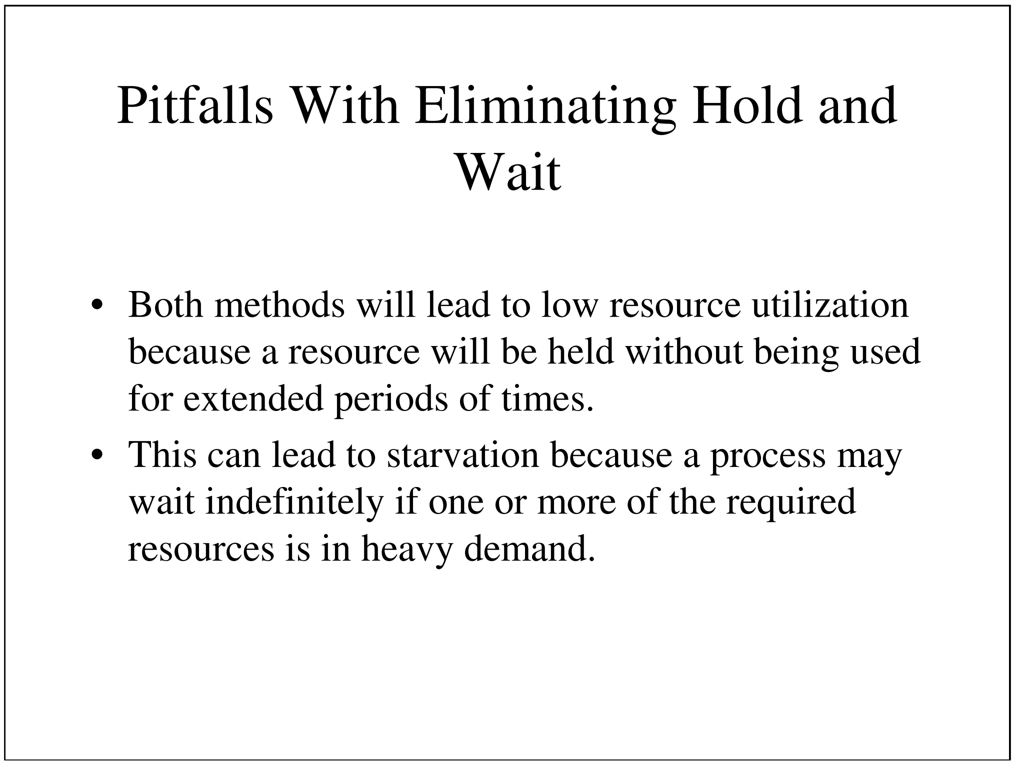
Eliminating Hold and Wait

- There are two ways of eliminating hold and wait:
  - Processes must request all their resources before executing (resource requests must come before all system requests).
  - Processes cannot request resources before releasing others.

There are two ways of eliminating hold and wait:

The first possibility is that process request all the resources before starting. This eliminates the scenario of a process holding one resource while waiting for another.

The other possibility is that process cannot request resources if it is holding any. This would require a process to surrender its resources before requesting any others.

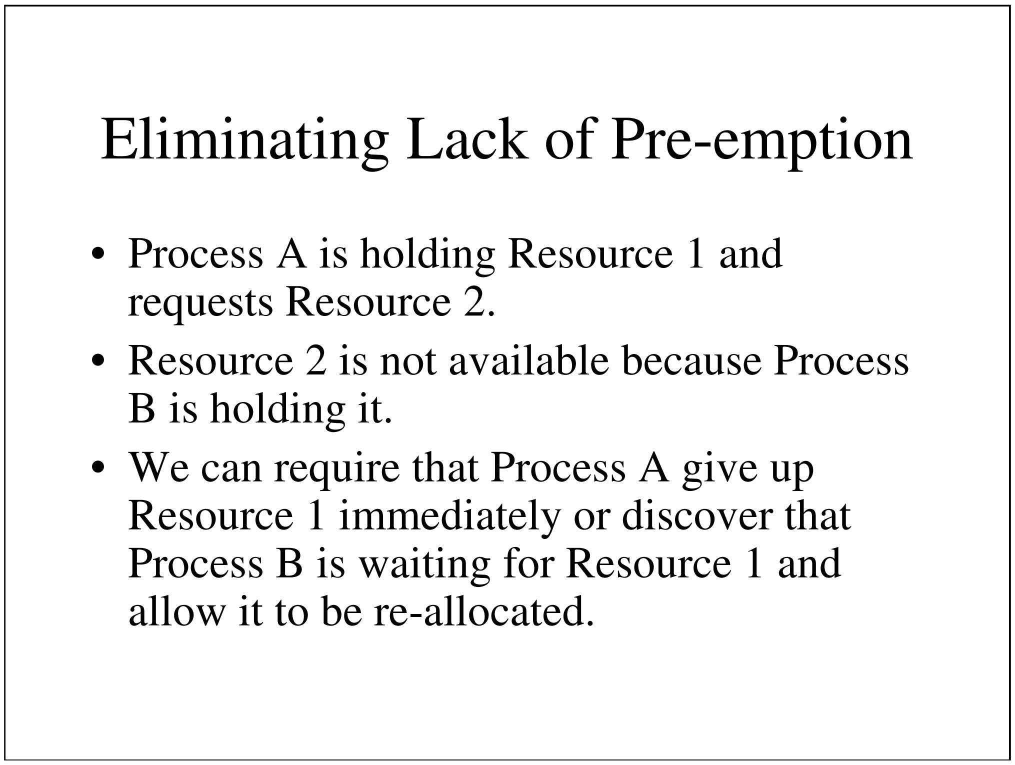


Pitfalls With Eliminating Hold and Wait

- Both methods will lead to low resource utilization because a resource will be held without being used for extended periods of times.
- This can lead to starvation because a process may wait indefinitely if one or more of the required resources is in heavy demand.

Both methods will lead to low utilization because resources will sit idle but reserved for long periods of time. A process would request everything at the start but not use one or more of these resources for quite some time.

Additionally, it opens up the possibility of starvation because a process may wait forever for the situation where all the resources it needs to all be available at the same time.



Eliminating Lack of Pre-emption

- Process A is holding Resource 1 and requests Resource 2.
- Resource 2 is not available because Process B is holding it.
- We can require that Process A give up Resource 1 immediately or discover that Process B is waiting for Resource 1 and allow it to be re-allocated.

We can also allow pre-emption to occur. This means that we can take a resource away from one process and give it to another that it waiting for it.

Eliminating Circular Wait

- We can establish a precedence in which resources must be requested.
- If a requested resource is not available, we wait until it is before requesting any other resources.
- This precedence should follow the normal order of usage for these resources.

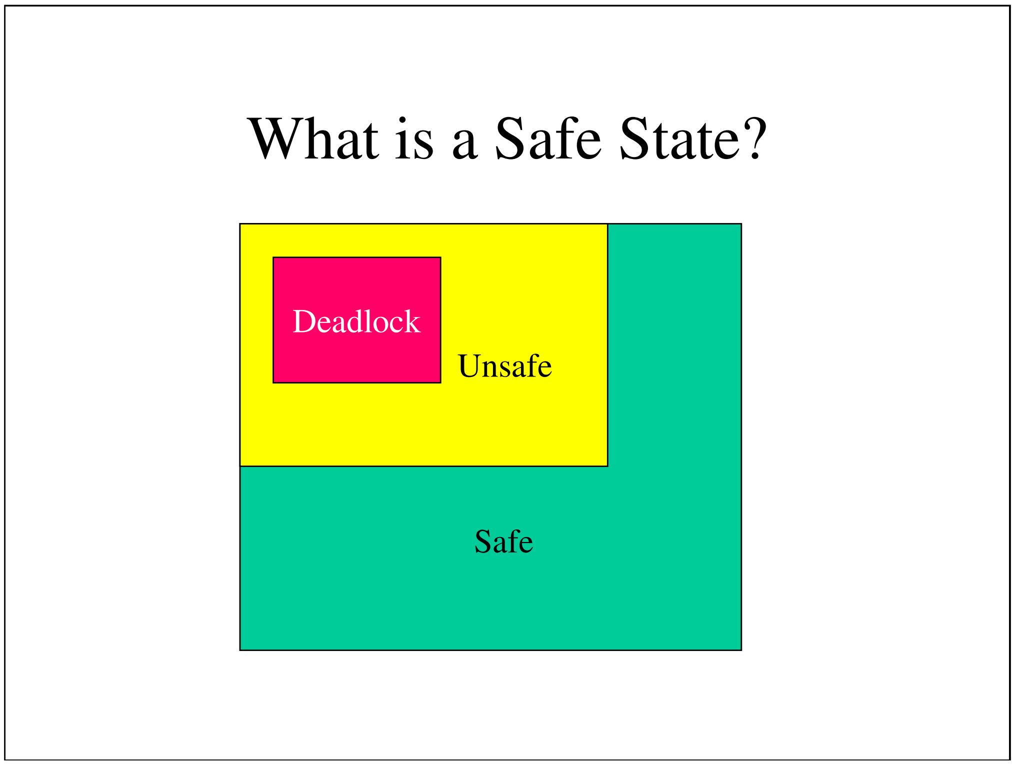
Eliminating circular wait requires us to establish some kind of order to the manner in which processes request resources. If processes must always get Resource 1 before waiting for Resource 2, then circular wait is impossible and therefore there can never be a deadlock.

Deadlock Avoidance

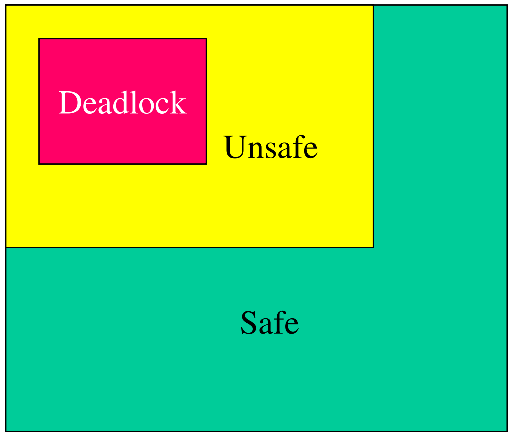
- Deadlock prevention algorithms reduce resource utilization and systems throughput.
- If the operating system knows in advance about a process's resource requests, it can determine whether or not they can be fulfilled without a risk of deadlock.
- This strategy is called ***deadlock avoidance***.

The problem with preventing deadlock is that it also makes our use of the system resources much less efficient. Our utilization of resources will be diminished as will our systems throughput. Since we want to maximize throughput and resource utilization and not minimize them, deadlock prevention may not be worthwhile design goals.

There is an alternative that may be more worthwhile. Let's give the operating system prior notice of what resources a process may request. Now we can determine the likelihood of deadlock and avoid situations where we are more likely to encounter it. This is what we mean by deadlock avoidance.



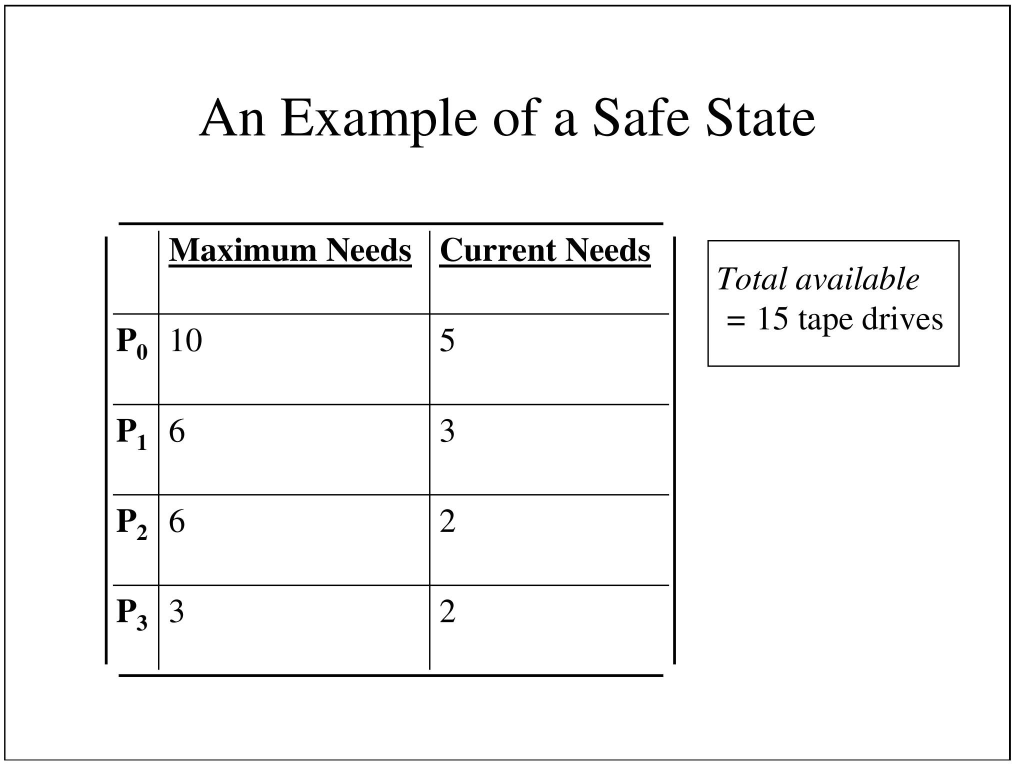
What is a Safe State?



A state is ***safe*** if the system can allocate resources to the various processes in some particular order and avoid deadlock in doing so.

An unsafe state is a state that may lead to deadlock. Not all unsafe states are actual deadlock.

Our goal in avoiding deadlock is to avoid the unsafe states. Since safe states cannot lead to deadlock, we can avoid it without incurring as high a cost as we would in eliminating the necessary conditions for deadlock.



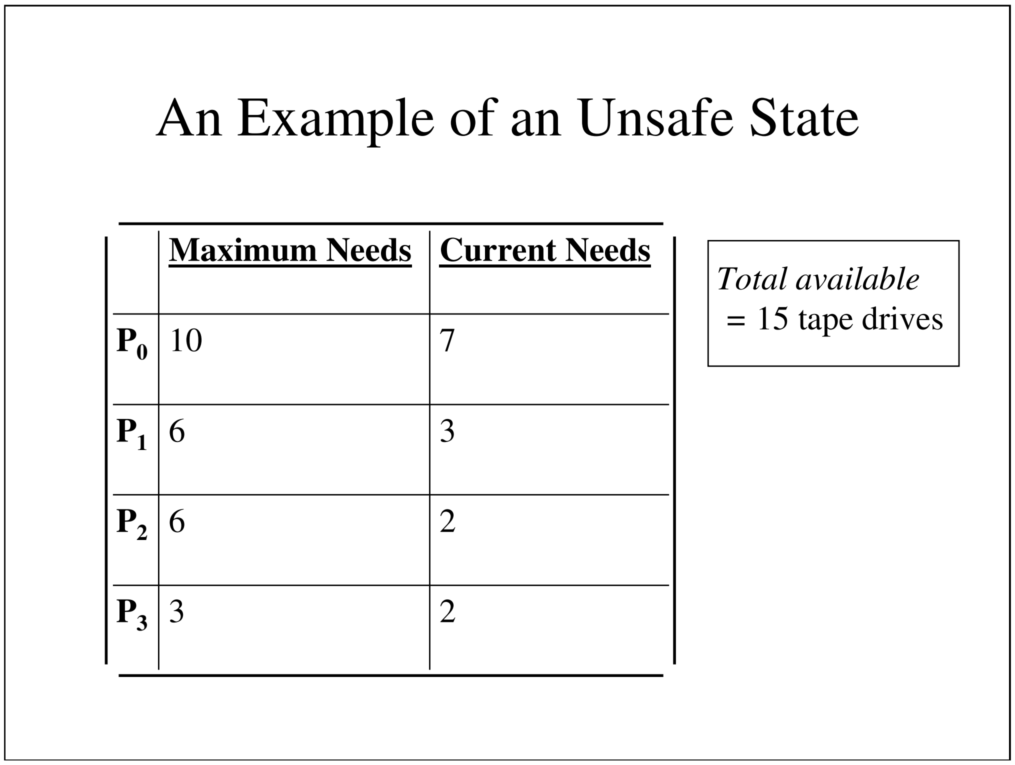
An Example of a Safe State

	Maximum Needs	**Current Needs**
P_0	10	5
P_1	6	3
P_2	6	2
P_3	3	2

Total available
= 15 tape drives

Imagine a situation where we have four processes that are looking to use some of a system's 15 tape drives. They each have indicated what their maximum needs could be. Although they could request a total of 25 tape drives on a system that only has 15, currently they have only requested 12 tape drives.

That leaves 3 tape drives available. With those three tape drives, P_1, P_2 or P_3 could finish their work. As each finishes, the drives that they have reserved become available for the remaining processes. Then there will be the extra tape drives available that P2 needs and eventually even the 5 tape drives that P_0 needs. Since all the processes can get the necessary resources and run through completion, this is a safe state.

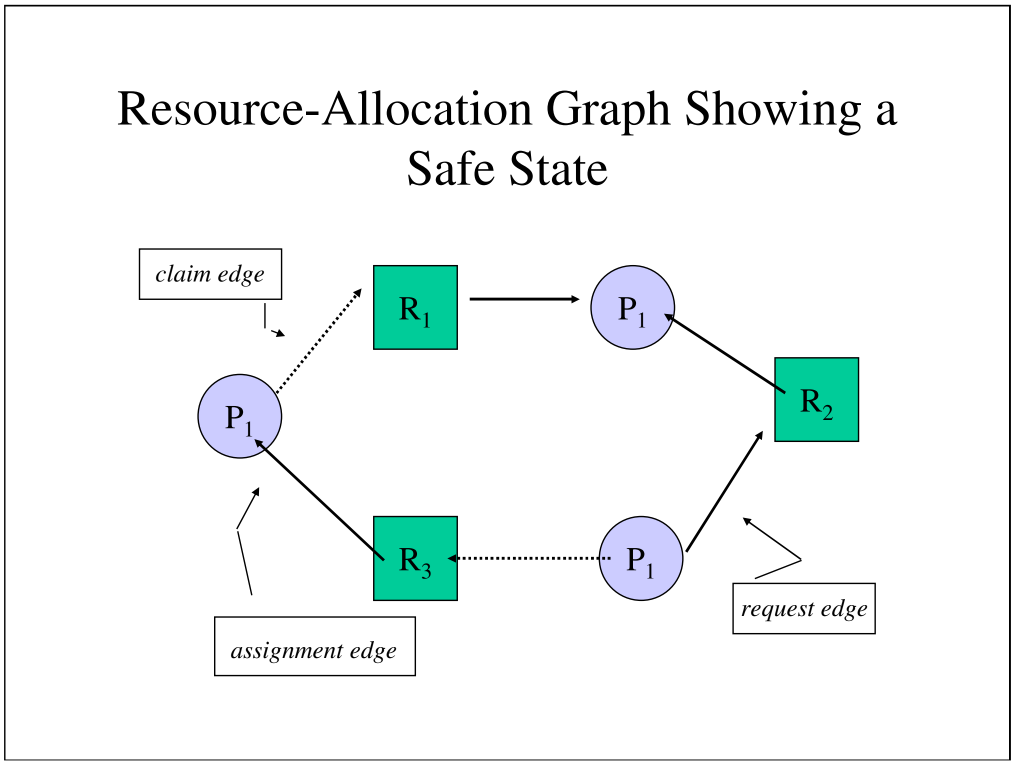


An Example of an Unsafe State

	Maximum Needs	**Current Needs**
P_0	10	7
P_1	6	3
P_2	6	2
P_3	3	2

Total available = 15 tape drives

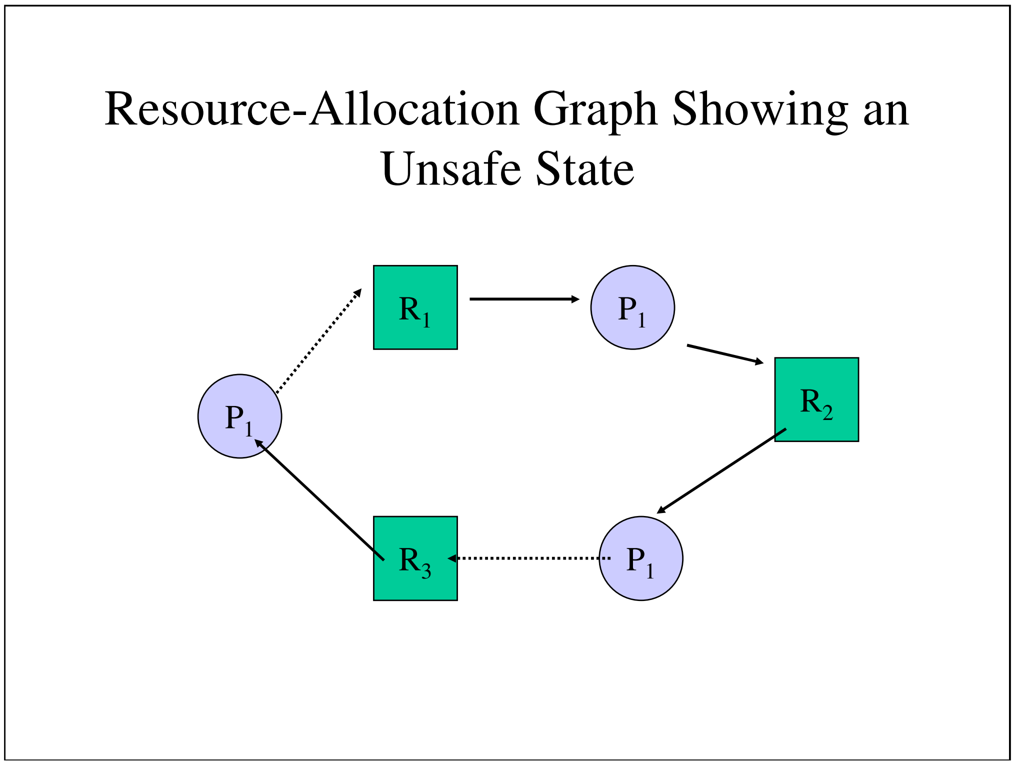
If any of these processes were to request another tape drive, the system would not be able to fulfill the request. This means that it is not a safe state

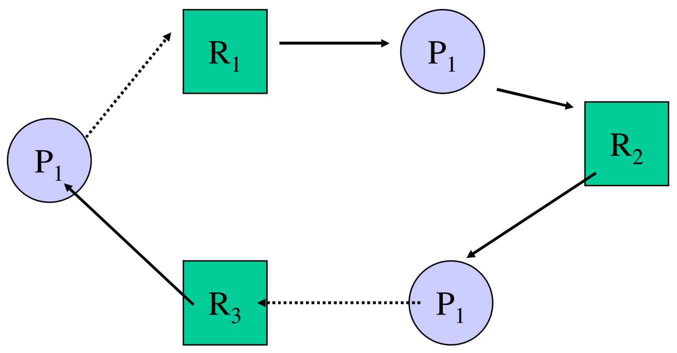


Resource-Allocation Graph Showing a Safe State

It is possible to use resource-allocation graphs to show whether a state is safe or unsafe if each resource has only one instance. In addition to the assignment edges and request edges that we discussed earlier, we add another type of edge: the claim edge. Processes must inform the operating system of the resources that they may request over their lifetime. These potential requests, or claims, are shown as claim edges.

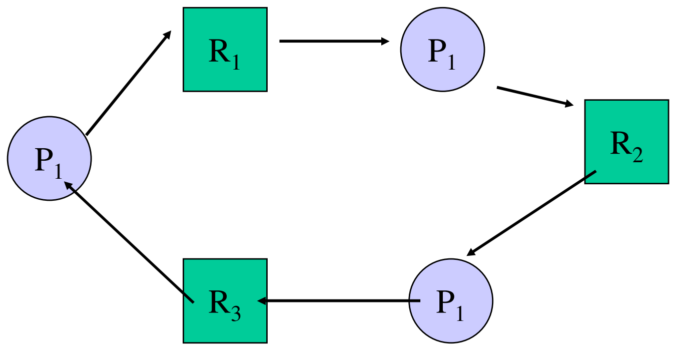
Claims are not requests, because the requests have not yet been made, but the process has the right to request them if necessary. We will now include them in our analysis. If we have a circuit, there is a possibility that there will be a circular wait. Since there is only one instance of each resource, there is mutual exclusion also, as well as hold and wait and no pre-emption. This could become a deadlock, although it may not. We will avoid these because without unsafe states, there is no deadlock.


Resource-Allocation Graph Showing an Unsafe State

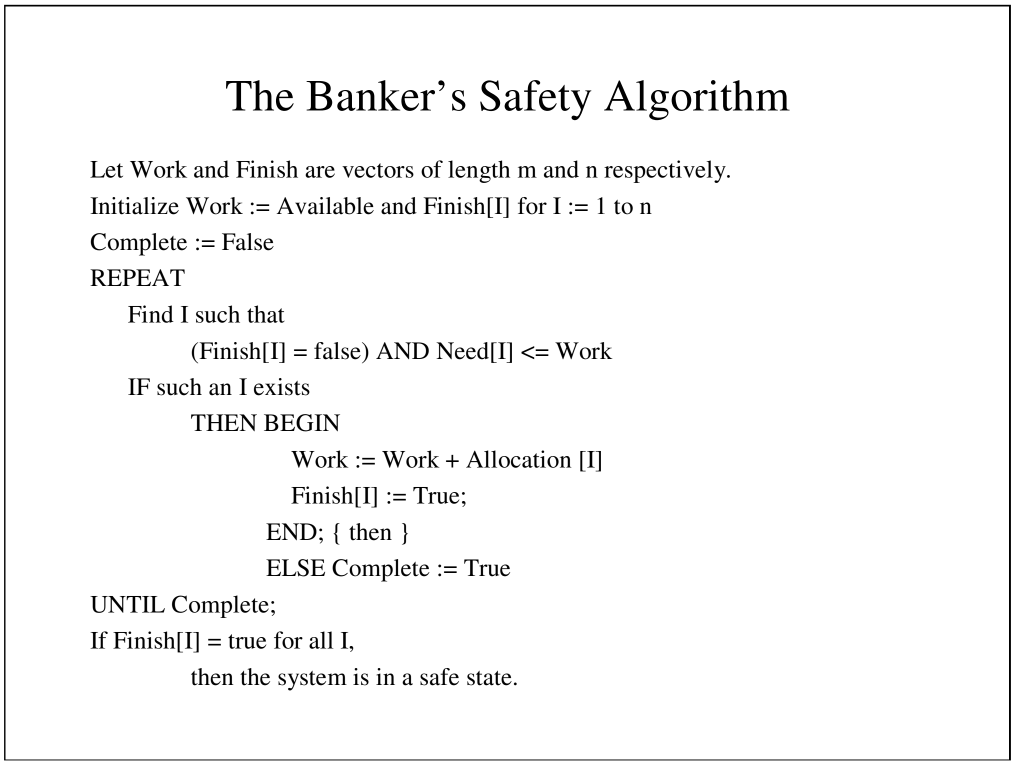


In this case, there is a circuit, so we are obviously in an unsafe state.

Resource-Allocation Graph Showing a Deadlock



This graph shows a state in deadlock. Like the unsafe state, there is a circuit, but what makes this different is that the claim edge is replaced by a request edge. Now our circuit indicates that all the conditions are present to create a deadlock.

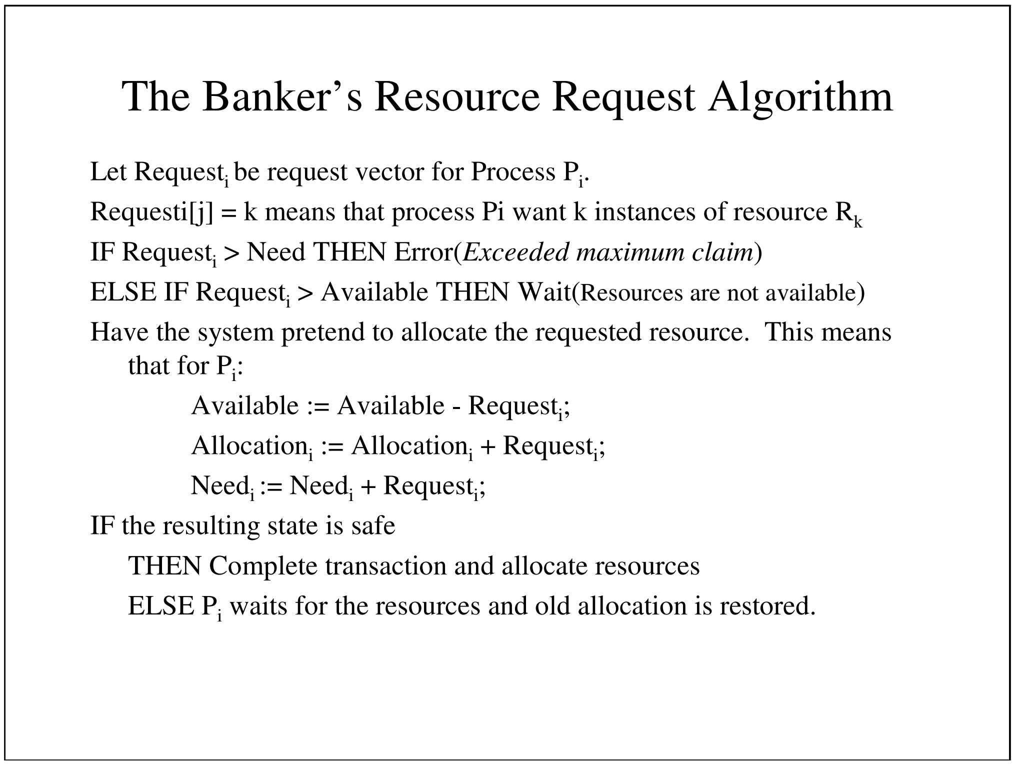


The Banker's Safety Algorithm

```
Let Work and Finish are vectors of length m and n respectively.
Initialize Work := Available and Finish[I] for I := 1 to n
Complete := False
REPEAT
   Find I such that
        (Finish[I] = false) AND Need[I] <= Work
   IF such an I exists
        THEN BEGIN
                Work := Work + Allocation [I]
                Finish[I] := True;
              END; { then }
              ELSE Complete := True
UNTIL Complete;
If Finish[I] = true for all I,
        then the system is in a safe state.
```

The problem with using resource-allocation graphs is that they are hard to turn into programs if there is more than one instance of a resource. In that event we will use something called the ***Banker's Algorithm***. A banker does not want to have too much cash on hand because it isn't earning interest or any other return on investment. But he or she must have at least enough to meet the demands of the day.In figuring out how much cash, he or she needs, a banker can take into account the amount of money coming in as well as how much likely to go out. The estimates should be as conservative as necessary because it is better to keep too much on hand than to run out.

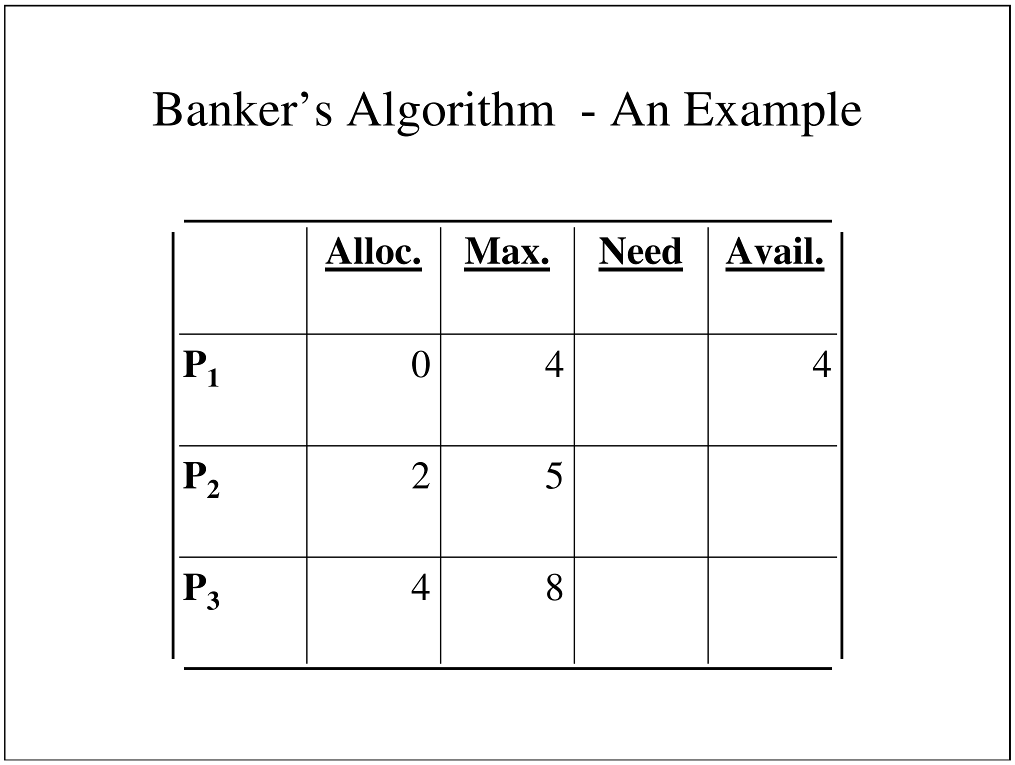
The algorithm that you see here tells us if a given state is safe. It requires that we must have at least one process that can get everything that it is allowed to reserve. Then we'll pretend to free its resources and see if there is a process which can get the rest of what it may need. We continue this until every process is able to complete its execution or until we are certain that they can't. In the first case, the system is safe and in the second case it is unsafe.



The Banker's Resource Request Algorithm

```
Let $Request_i$ be request vector for Process $P_i$.
Requesti[j] = k means that process Pi want k instances of resource $R_k$
IF $Request_i$ > Need THEN Error(*Exceeded maximum claim*)
ELSE IF $Request_i$ > Available THEN Wait(Resources are not available)
Have the system pretend to allocate the requested resource.  This means
    that for $P_i$:
        Available := Available - $Request_i$;
        $Allocation_i$ := $Allocation_i$ + $Request_i$;
        $Need_i$ := $Need_i$ + $Request_i$;
IF the resulting state is safe
    THEN Complete transaction and allocate resources
    ELSE $P_i$ waits for the resources and old allocation is restored.
```

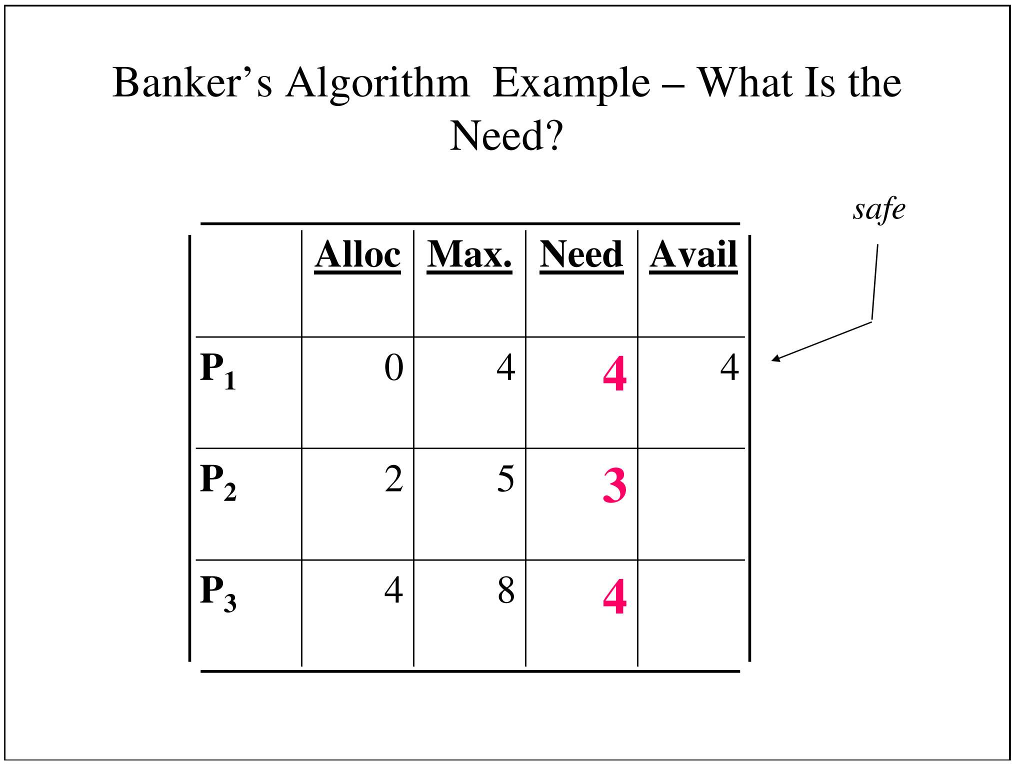
The last algorithm tells us if the system is safe. But will it be safe if we allocate a resource (or more than one resource) to a process? Let's pretend to allocate the resource or resources and see if the system is in a safe state. If it is, we'll do the allocation. If not, we won't.



Banker's Algorithm - An Example

	Alloc.	**Max.**	**Need**	**Avail.**
P_1	0	4		4
P_2	2	5		
P_3	4	8		

Let's look at a simple example. Imagine that there are three processes, which have been allocated our one resource as shown in the slide. They each have a maximum allowed allocation of as shown.



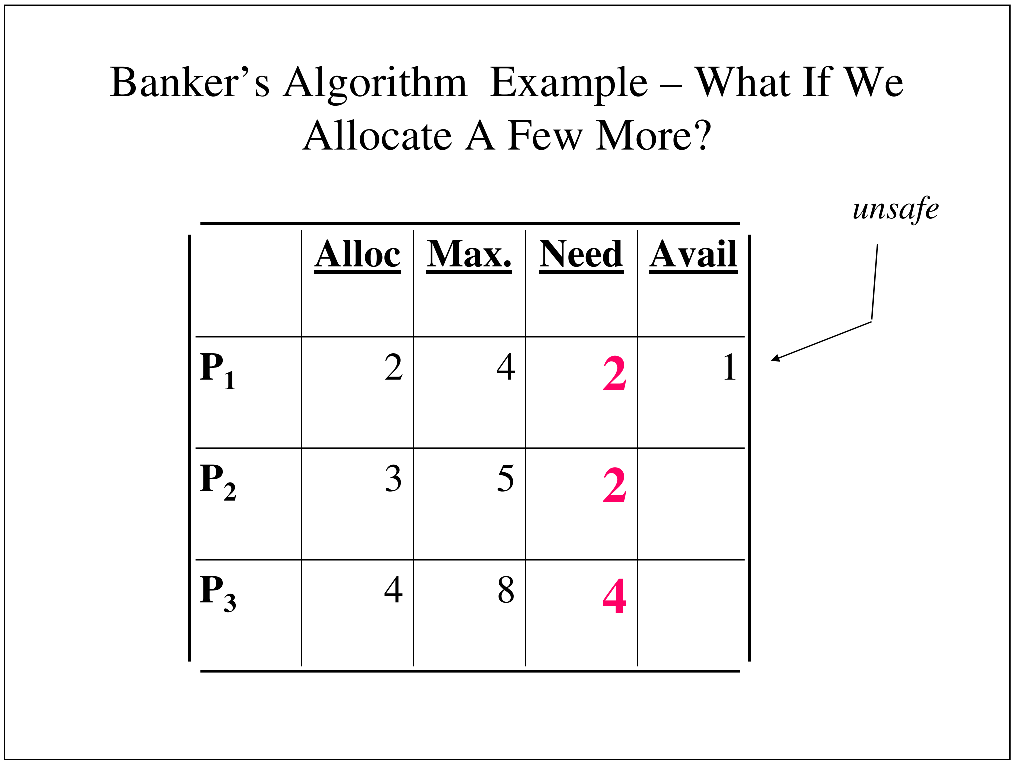
Banker's Algorithm Example – What Is the Need?

safe

	Alloc	**Max.**	**Need**	**Avail**
P_1	0	4	**4**	4
P_2	2	5	**3**	
P_3	4	8	**4**	

The first step is to calculate what they might need. For Process P_1, it is 4 because a maximum of 4 minus an allocation of 0 yields 4. For process P_2, it is 5 minus 2 yielding a need of 3. For process P_3, it is 8 minus 4 yielding a need of 4. With the sequence <P_2, P_1, P_3>, we have determined that the system is in a safe state.

With 4 instances of the resource available, we could complete any of the three processes; let's say that we complete P_2 first.Now we can free the 2 that P_2 is holding. With the 6 available, we can complete either P_1 or P_3; let's compete P_1 first. Since there is nothing to free we don't change the amount available and we complete P_3



Banker's Algorithm Example – What If We Allocate A Few More?

unsafe

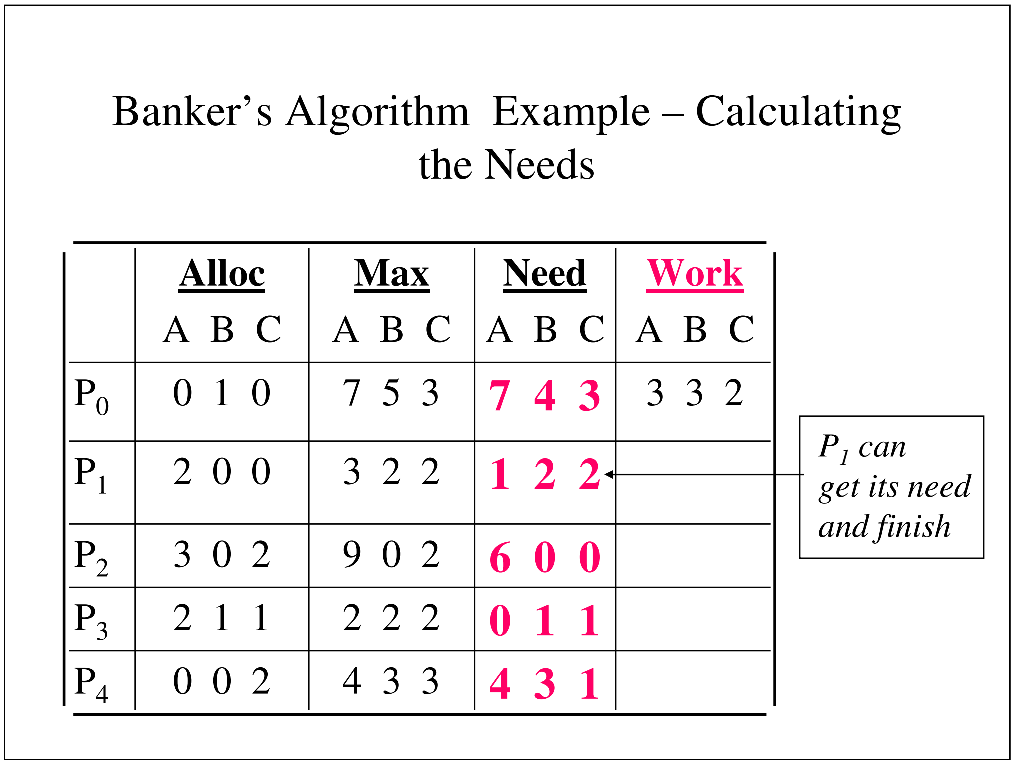
	Alloc	**Max.**	**Need**	**Avail**
P_1	2	4	**2**	1
P_2	3	5	**2**	
P_3	4	8	**4**	

Let's allocate 2 instances of our resource to P_1 and one more to P_2. With only one instance of the resource left, there is no process whose potential need can be met. So we cannot be certain that any will run to completion. The system is in an unsafe state and the allocation was not a prudent move.

Banker’s Algorithm Example – 3 Types of Resources

	Alloc	**Max**	**Need**	**Avail**
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3		3 3 2
P_1	2 0 0	3 2 2		
P_2	3 0 2	9 0 2		
P_3	2 1 1	2 2 2		
P_4	0 0 2	4 3 3		

Let’s take a look at a more typical situation. There are three different types of resources, A, B and C. Shown here is their current allocations, the maximum allocations that they may request and what remains available. Our first step is to calculate the needs.



Banker's Algorithm Example – Calculating the Needs

	Alloc	**Max**	**Need**	**Work**
	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 can get its need and finish

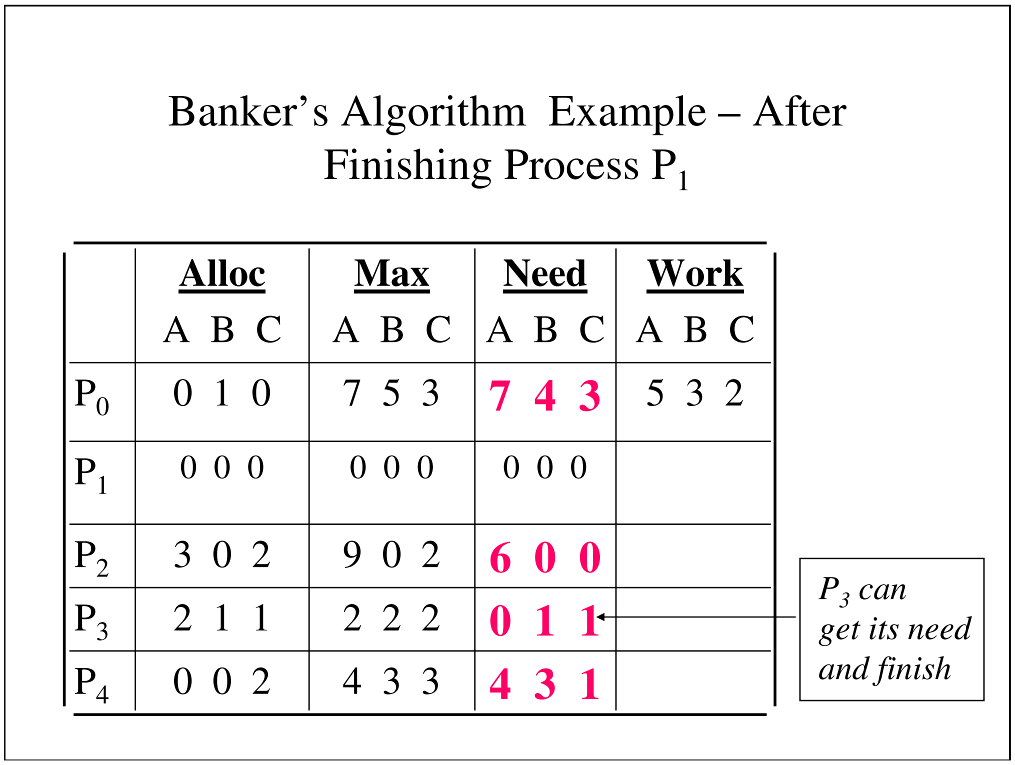
P_0 has 0 instances of A, 1 instance of B and 0 instances of C. It has the right to 7 instances of A, 4 instances of B and 3 instances of C.

7 instances of A minus 0 allocated instances of A equals a need of 7 instances of A.

5 instances of B minus 1 allocated instance of B equals a need of 4 instances of B.

3 instances of C minus 0 allocated instances of C equals a need of 3 instances of C.

We now do the same for processes P_1, P_2, P_3, and P_4. When we finish, we see that P_1 can get everything that it may need, so it will run through completion.

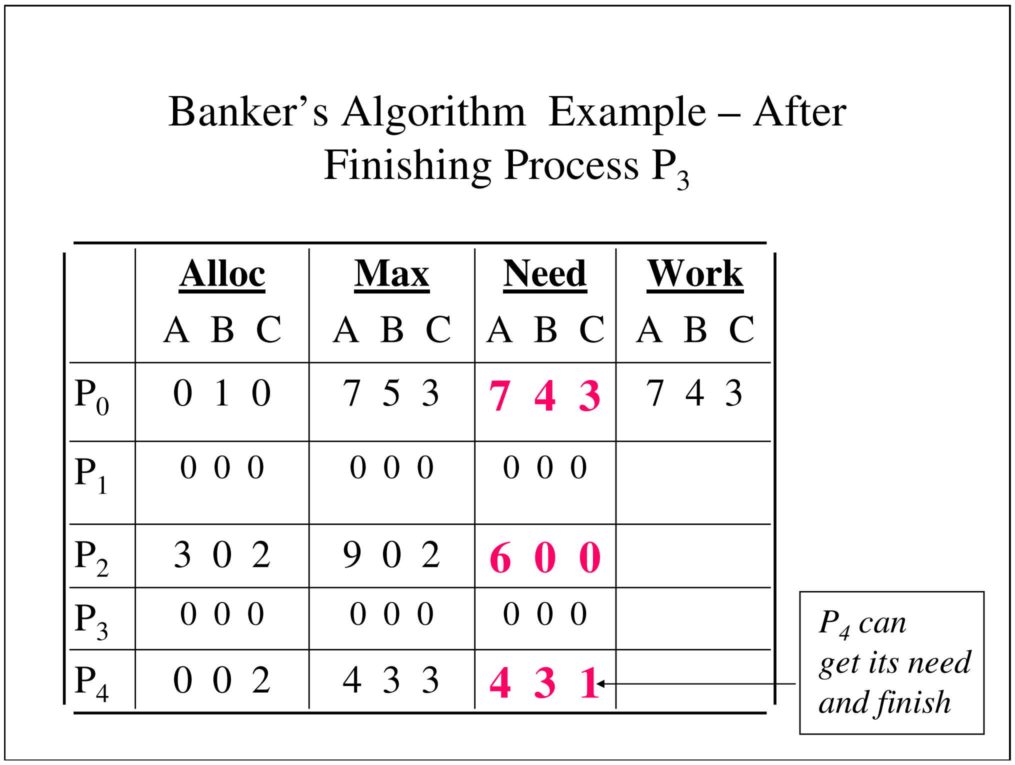


Banker's Algorithm Example – After Finishing Process P_1

	Alloc	**Max**	**Need**	**Work**
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	5 3 2
P_1	0 0 0	0 0 0	0 0 0	
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_3 can get its need and finish

After we re-allocate the 2 instances of A that P1 actually is holding, we have 5 A's, 3 B's and 2 C's. With this, we can satisfy P_3 and it can run through completion.

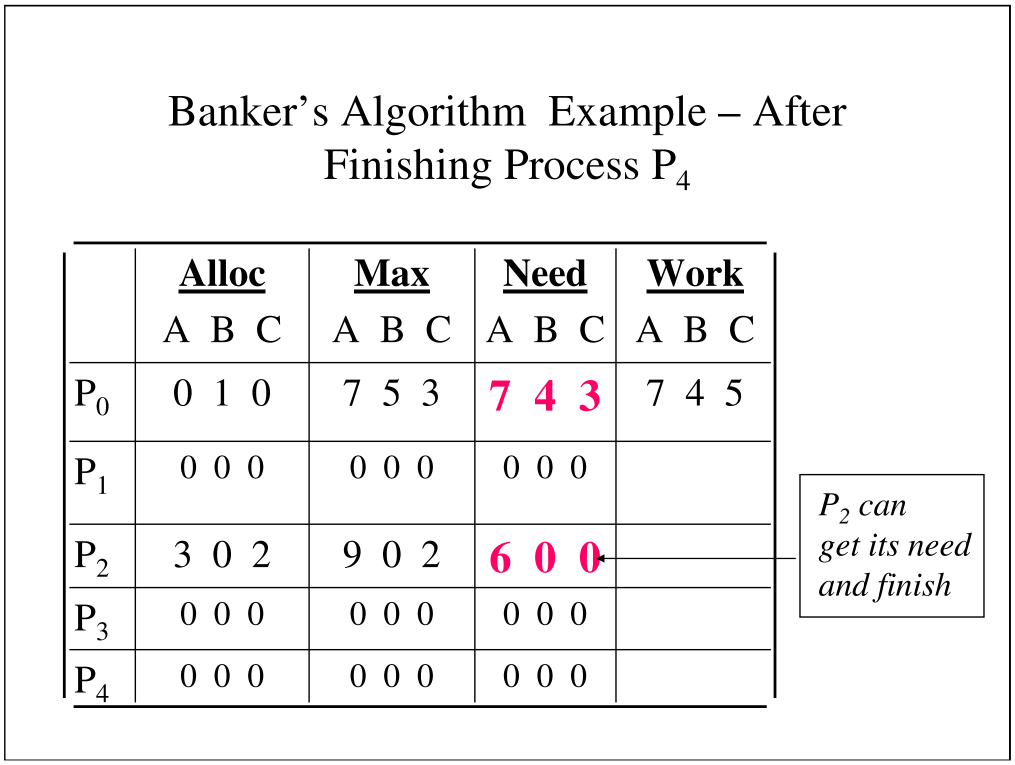


Banker's Algorithm Example – After Finishing Process P_3

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

P_4 can get its need and finish

Once P3 finishes, we re-allocate its 2 A's, 1 B and 1 C, making 7A's, 4 B's and 3 C's available for re-allocation. Now we can finish P_4.

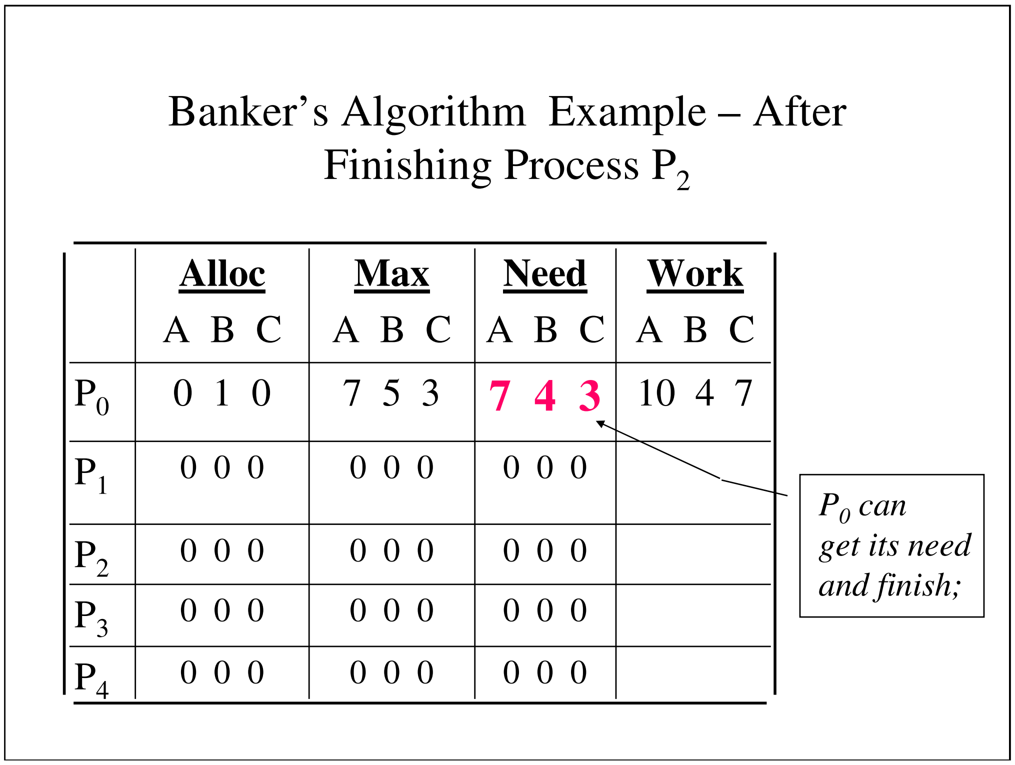


Banker’s Algorithm Example – After Finishing Process P_4

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

P_2 can get its need and finish

We can now free up the 2 C’s that belong to P_4 and and now we have 7 A’s, 4 B’s and 5 C’s. This is enough to allow us to complete P_2.

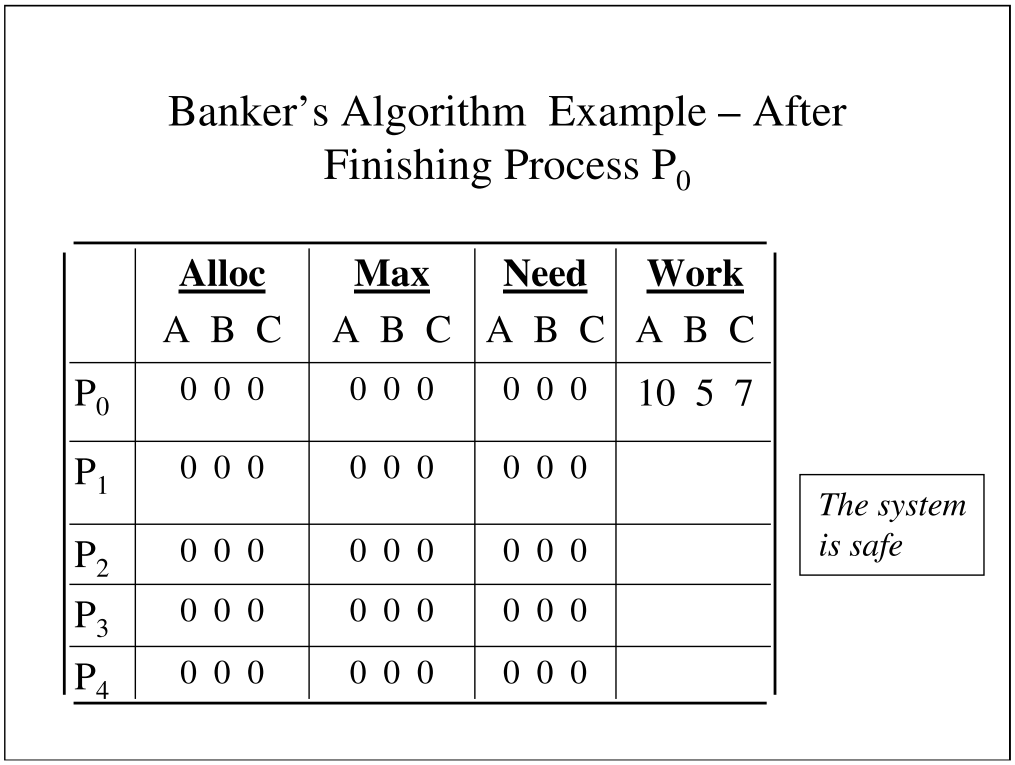


Banker's Algorithm Example – After Finishing Process P_2

	Alloc	**Max**	**Need**	**Work**
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	**7 4 3**	10 4 7
P_1	0 0 0	0 0 0	0 0 0	
P_2	0 0 0	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	0 0 0	

P_0 can get its need and finish;

Lastly, we free up the 3A's and 2 C's that P_2 has . This leaves us with enough to complete P_0.

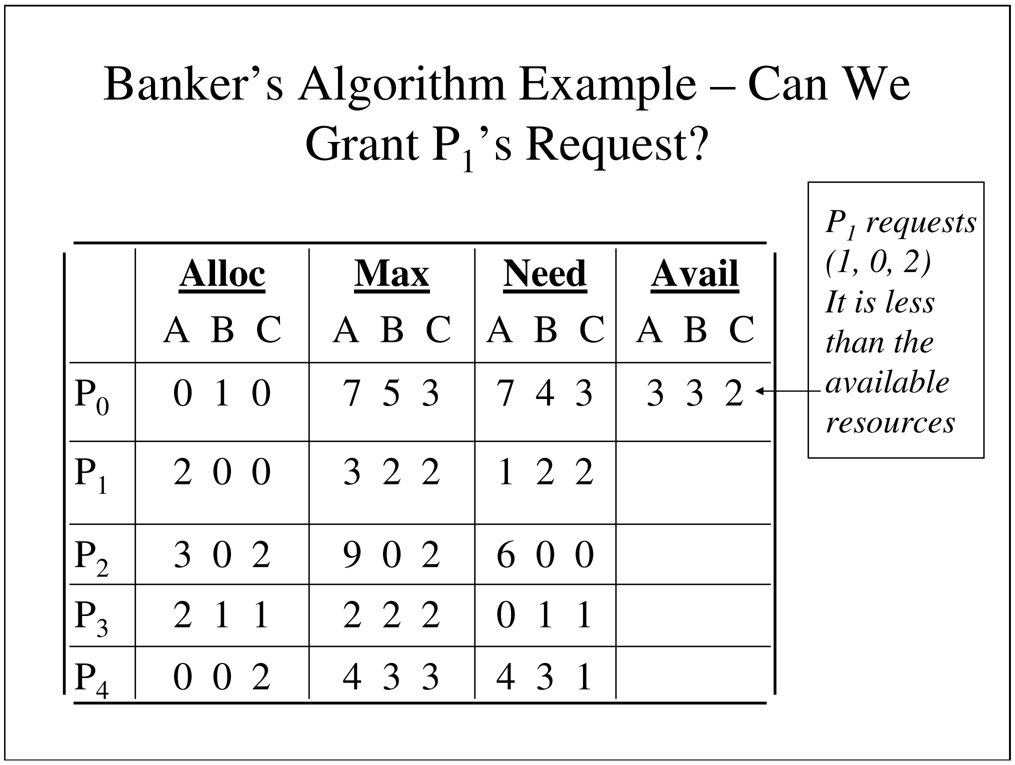


Banker's Algorithm Example – After Finishing Process P_0

	Alloc	**Max**	**Need**	**Work**
	A B C	A B C	A B C	A B C
P_0	0 0 0	0 0 0	0 0 0	10 5 7
P_1	0 0 0	0 0 0	0 0 0	
P_2	0 0 0	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	0 0 0	
P_4	0 0 0	0 0 0	0 0 0	

The system is safe

There is a sequence in which we can complete all the processes if they do need their maximum allocation. The system is safe.

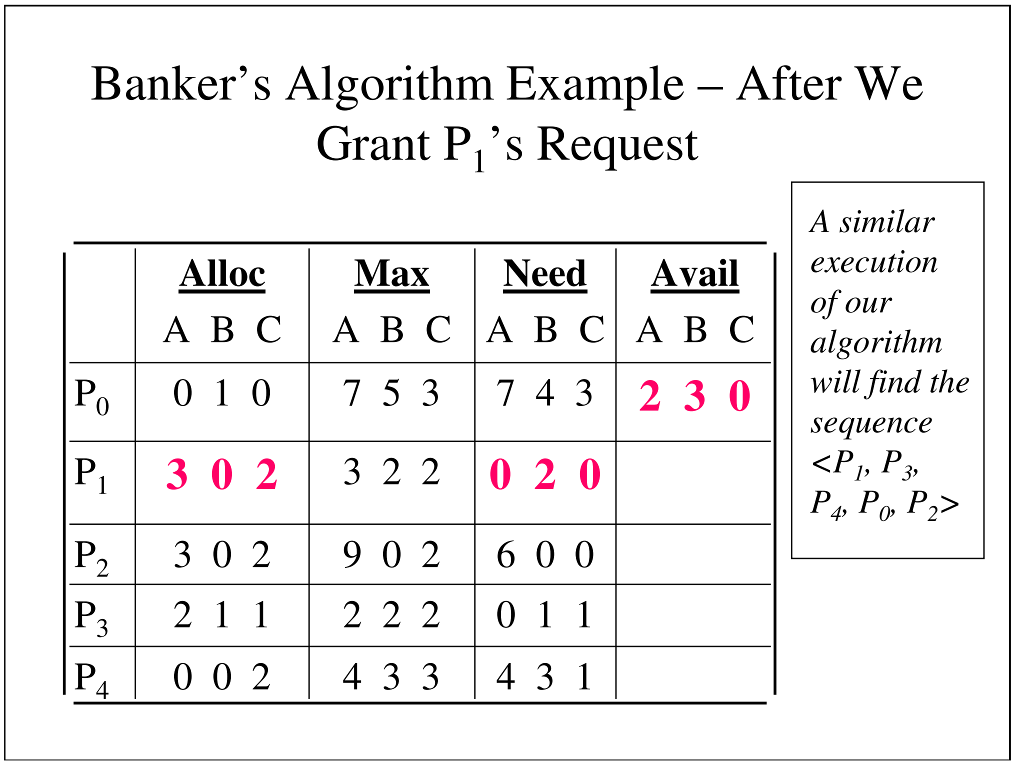


Banker's Algorithm Example – Can We Grant P_1's Request?

	Alloc	**Max**	**Need**	**Avail**
	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 requests (1, 0, 2) It is less than the available resources

Let's examine a request of process P_1 for 1 A, 0 B's and 2 C's. Since this is less than what is available, we have the resources available for the request, but will it leave us in a safe state?



Banker's Algorithm Example – After We Grant P_1's Request

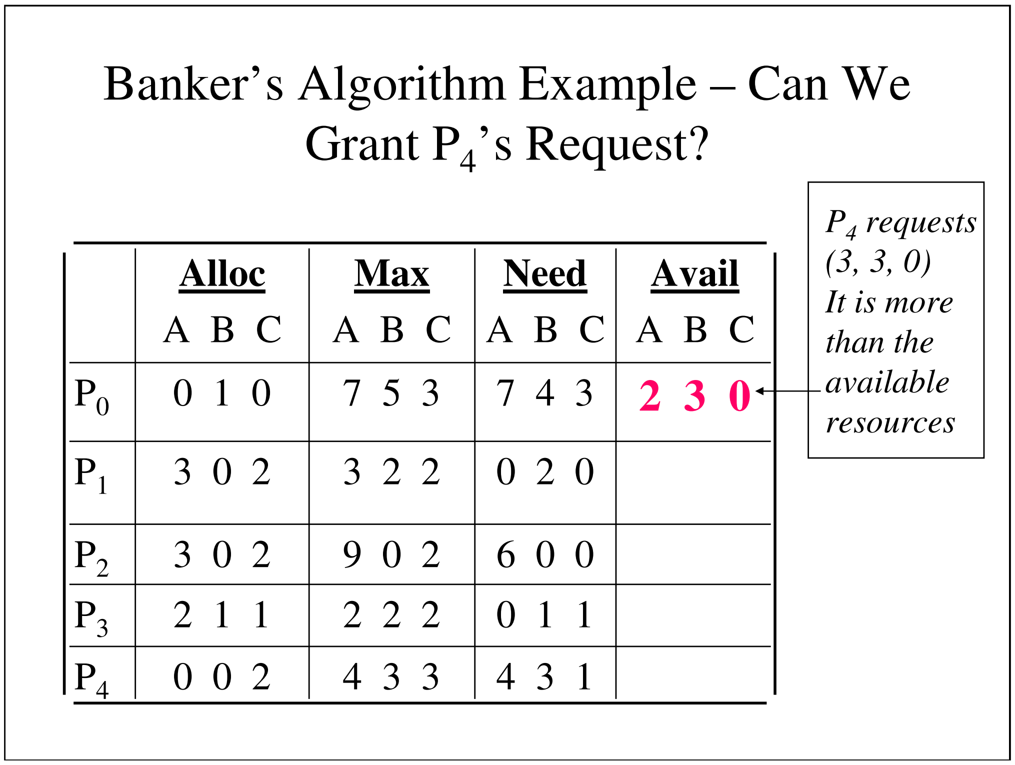
	Alloc			Max			Need			Avail		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	**2**	**3**	**0**
P_1	**3**	**0**	**2**	3	2	2	**0**	**2**	**0**			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

A similar execution of our algorithm will find the sequence $<P_1, P_3, P_4, P_0, P_2>$

Let's pretend to allocate it. This leave us with 2 A's, 3 B's and 0 C's available.

We have enough for P_1 to run to completion, even if it needed its maximum allocation. After de-allocating its 3 A's and 2 C's, we would have 5 A's, 3 B's and 2 C's available.

With this, we could run P_3 through completion. Then we could free its resources, which would give us 7 A's, 4B's and 3 C's. With this we could easily complete P_4, P_0 and P_2. Thus, this request can be made without placing the system in an unsafe state.

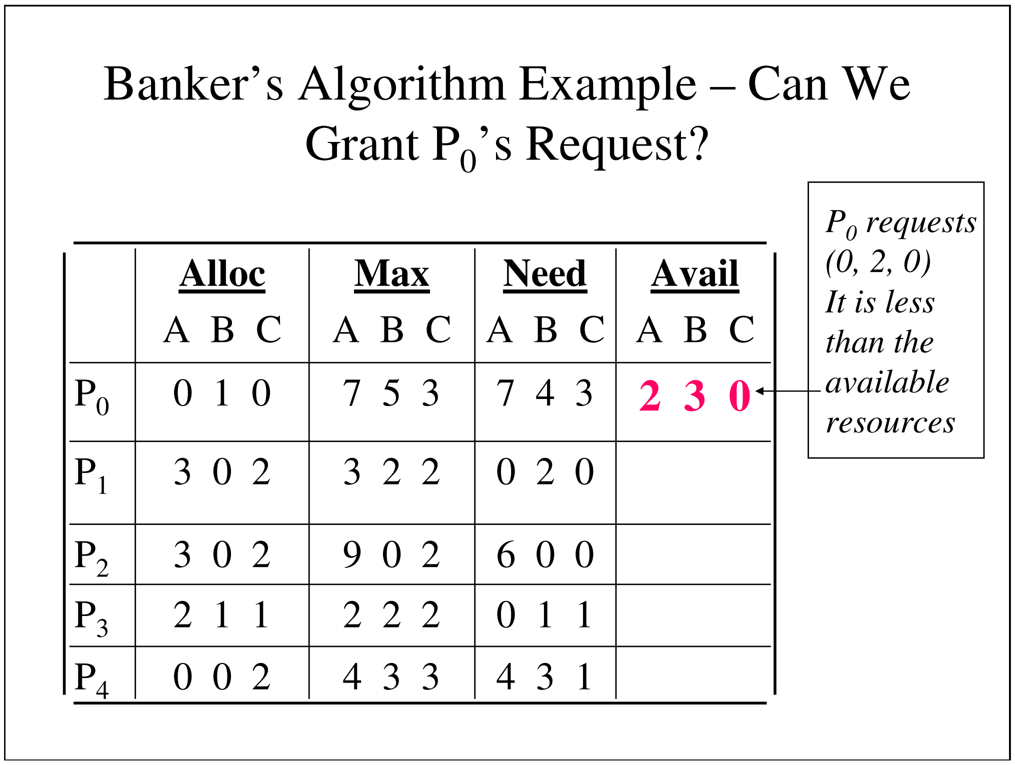


Banker's Algorithm Example – Can We Grant P_4's Request?

	Alloc	**Max**	**Need**	**Avail**
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	**2 3 0**
P_1	3 0 2	3 2 2	0 2 0	
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_4 requests (3, 3, 0) It is more than the available resources

If P4 were to request 3 A's and 3 B's, we would be unable to fulfill the request because we do not have the resources available - we are short 1 A.

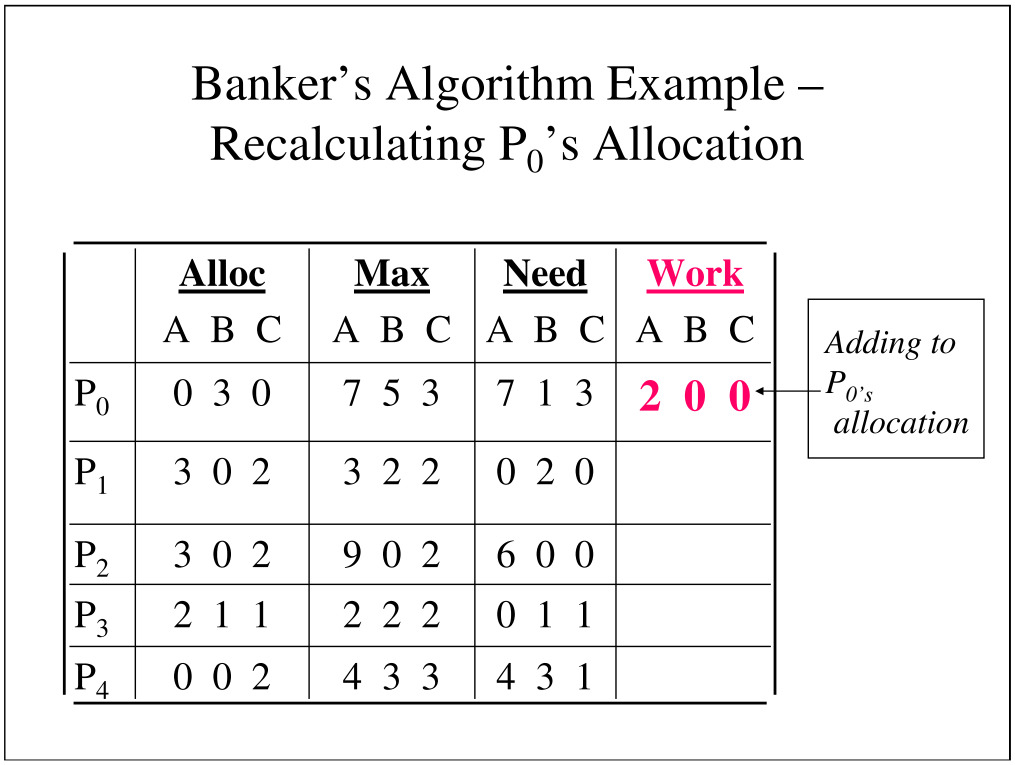


Banker's Algorithm Example – Can We Grant P_0's Request?

	Alloc	**Max**	**Need**	**Avail**
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	7 4 3	**2 3 0**
P_1	3 0 2	3 2 2	0 2 0	
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_0 requests (0, 2, 0) It is less than the available resources

Let's see what happens if P_0 requests 2 B's. It's less than the available resources, But does it place us in an unsafe state?



Banker's Algorithm Example – Recalculating P_0's Allocation

	Alloc	**Max**	**Need**	**Work**
	A B C	A B C	A B C	A B C
P_0	0 3 0	7 5 3	7 1 3	**2 0 0**
P_1	3 0 2	3 2 2	0 2 0	
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	

Adding to $P_{0's}$ allocation

Let's add the 2 B's to P_0's allocation and recalculate its need and what would be available.

Banker's Algorithm Example – Recalculating P_0's Allocation

	Alloc	**Max**	**Need**	**Avail**
	A B C	A B C	A B C	A B C
P_0	0 3 0	7 5 3	7 1 3	**2 0 0**
P_1	3 0 2	3 2 2	0 2 0	
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	

There is not enough left for anyone's need; it is not safe

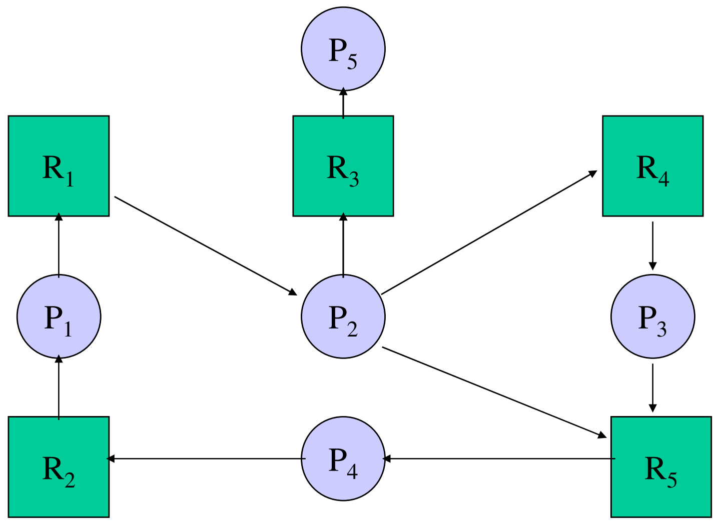
The problem is that 2 A's and 0 B's or C's leave us unable to meet any possible requests by other processes. This request won't be granted because it places us in an unsafe state.

Deadlock Detection

- If a system does not provide for deadlock avoidance or prevention, then it must provide:
  - an algorithm for deadlock detection
  - an algorithm for deadlock recovery

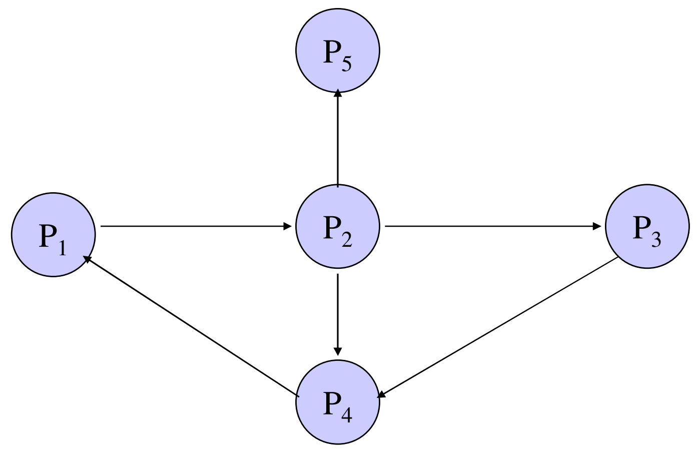
Some systems don’t avoid or prevent deadlock. What they do is detect it and recover from it. That means that such systems need an algorithm for each of these two tasks.

Detecting Deadlock With Single-Instances of Each Resource



In the case of single-instance resource, we can use a graphical approach. Let’s take another look at our resource allocation graph. What we will do is simplify this by redrawing the request edges that currently run from process to resource. Instead, let’s have them point directly to the process holding the requested resource.

Detecting Deadlock Corresponding Wait-For Graph



This is what the graph looks like after the redrawing the request edges. Now we know at a single glance that P_4 is waiting for a resource that P_1 is holding; which resource is not important. Similarly P1 is waiting for a resource that P_2 is holding and P_2 is waiting for a resource that P_4 is holding. It is fairly straightforward to write a program that can detect whether there is a circuit like this in such a graph.

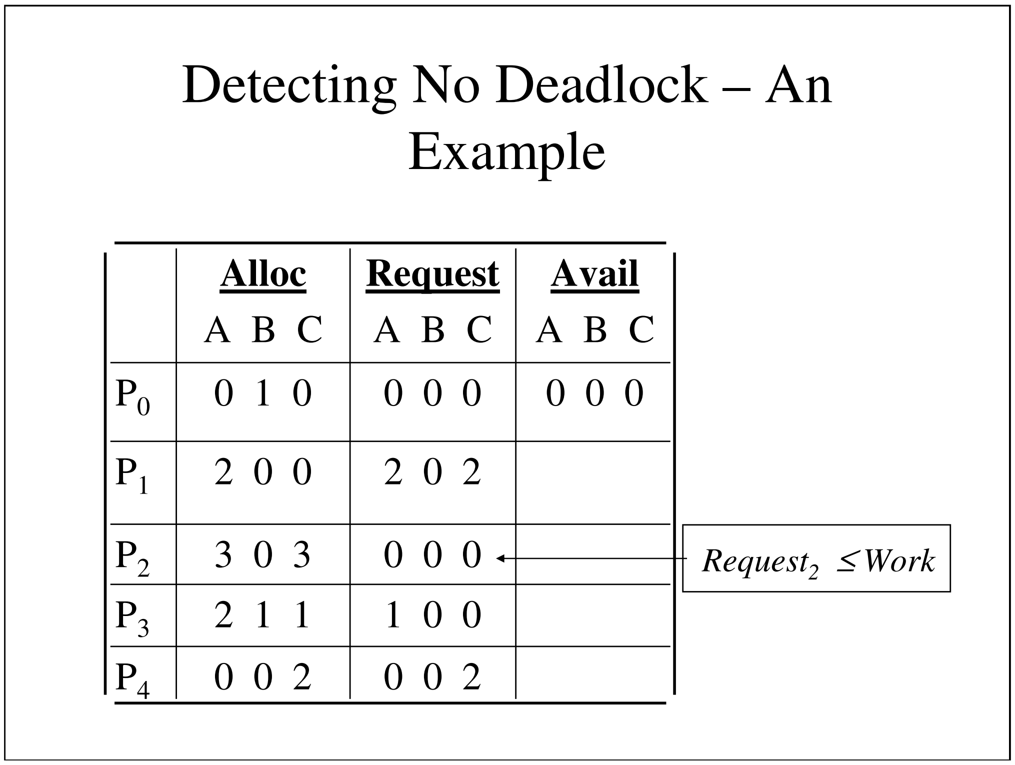
Detecting Deadlock With Multiple-Instances of Each Resource

```
Let Work and Finish be vector of length m and n respectively.
Initialize Work := Available
FOR I := 1 To n
    IF Allocation <> 0 THEN Finish[i] := False;
                            ELSE Finish[i] := True;
WHILE there exists an I such that
        (Finish[i] = False) AND (Request$_i$ <= Work) DO
   BEGIN
      IF such an I exists
         THEN  BEGIN
                  Work := Work + Allocation$_i$;
                  Finish[i] := True
               END;  { then }
   END;  { while }
IF Finish[i] = False then the system is in a deadlock state and Process P$_i$ is
    deadlocked.
```

This is similar to the Banker’s Algorithm that we looked at before. We assume that certain resources have been allocated and that others have been requested.

We’ll look to see if there are any requests that we can fulfill. If that’s the case, these processes can run through completion and then their resources can be re-allocated to other processes.

We continue this until there are no more cases like this. If we haven’t finished all our processes, there is a deadlock which includes this process.

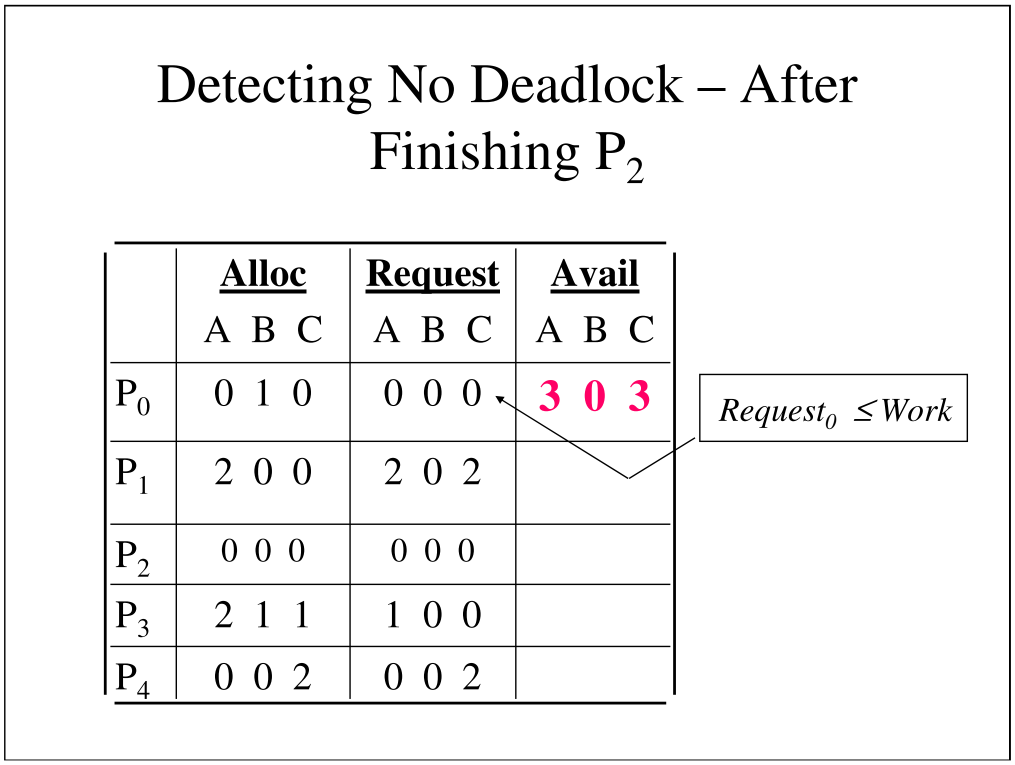


Detecting No Deadlock – An Example

	Alloc	**Request**	**Avail**
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

$Request_2 \leq Work$

Let’s look at an example. In this system, we have no remaining resources but there are processes with outstanding requests. Since P_2 doesn’t need anything else, it can run through completion and we can reassign its resources.

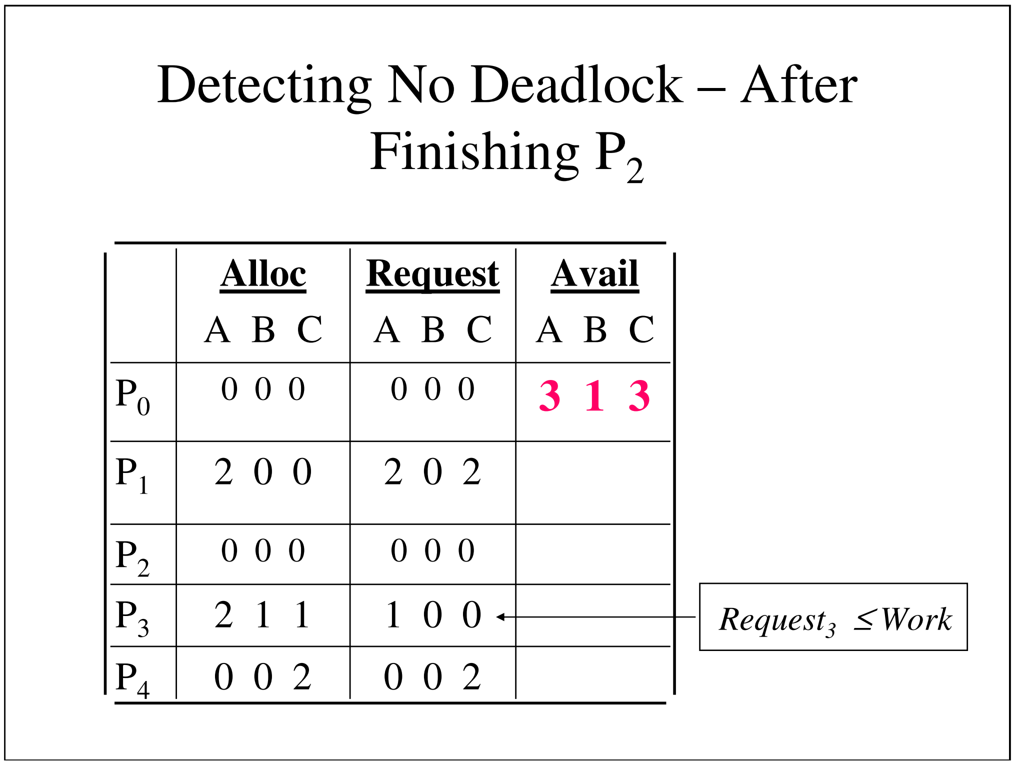


Detecting No Deadlock – After Finishing P_2

	Alloc	Request	Avail
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	**3 0 3**
P_1	2 0 0	2 0 2	
P_2	0 0 0	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

$Request_0 \le Work$

We now have 3 A's and 3 C's available, but since P_0 doesn't even need this, it can run through completion and we can reassign its resources.

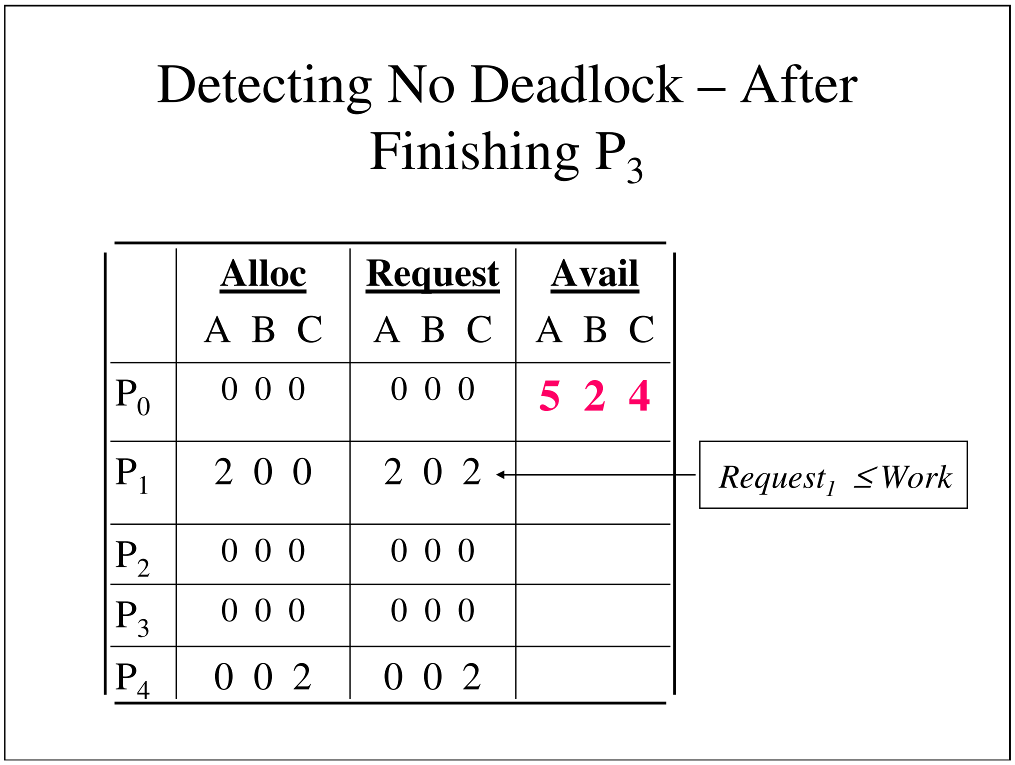


Detecting No Deadlock – After Finishing P_2

	Alloc	**Request**	**Avail**
	A B C	A B C	A B C
P_0	0 0 0	0 0 0	**3 1 3**
P_1	2 0 0	2 0 2	
P_2	0 0 0	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

$Request_3 \leq Work$

We now have 3 A's, 1 B and 3 C's. This is more than enough to meet P_3's request and it can run through completion. We can reallocate its resources.

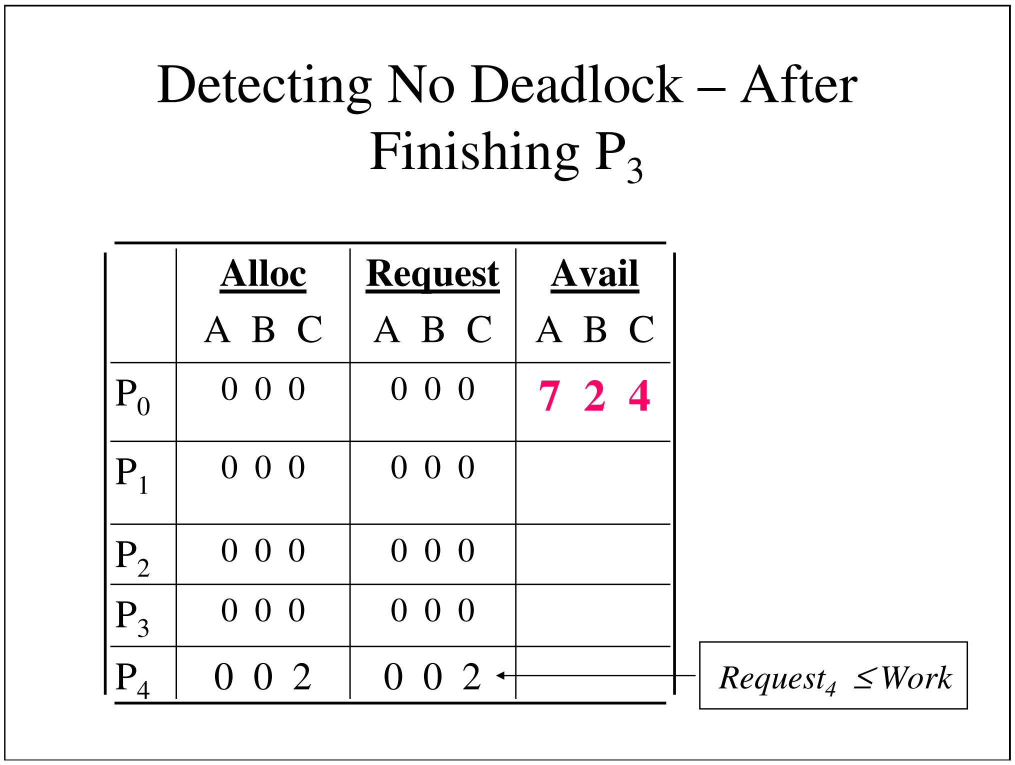


Detecting No Deadlock – After Finishing P_3

	Alloc	Request	Avail
	A B C	A B C	A B C
P_0	0 0 0	0 0 0	**5 2 4**
P_1	2 0 0	2 0 2	
P_2	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	
P_4	0 0 2	0 0 2	

$Request_1 \leq Work$

We now have 5 A's, 2 B's and 4 C's. This is more than enough to meet P_1's request and it can run through completion. We can reallocate its resources.



Detecting No Deadlock – After Finishing P_3

	Alloc	Request	Avail
	A B C	A B C	A B C
P_0	0 0 0	0 0 0	**7 2 4**
P_1	0 0 0	0 0 0	
P_2	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	
P_4	0 0 2	0 0 2	

$Request_4 \leq Work$

We now have 7 A's, 2 B and 4 C's. This is more than enough to meet P_3's request and it can run through completion. We can reallocate its resources.

Detecting No Deadlock – After Finishing P_4

	Alloc	**Request**	**Avail**
	A B C	A B C	A B C
P_0	0 0 0	0 0 0	**7 2 4**
P_1	0 0 0	0 0 0	
P_2	0 0 0	0 0 0	
P_3	0 0 0	0 0 0	
P_4	0 0 2	0 0 2	

Not deadlocked Our algorithm found the sequence $<P_0, P_2, P_3, P_1, P_4>$

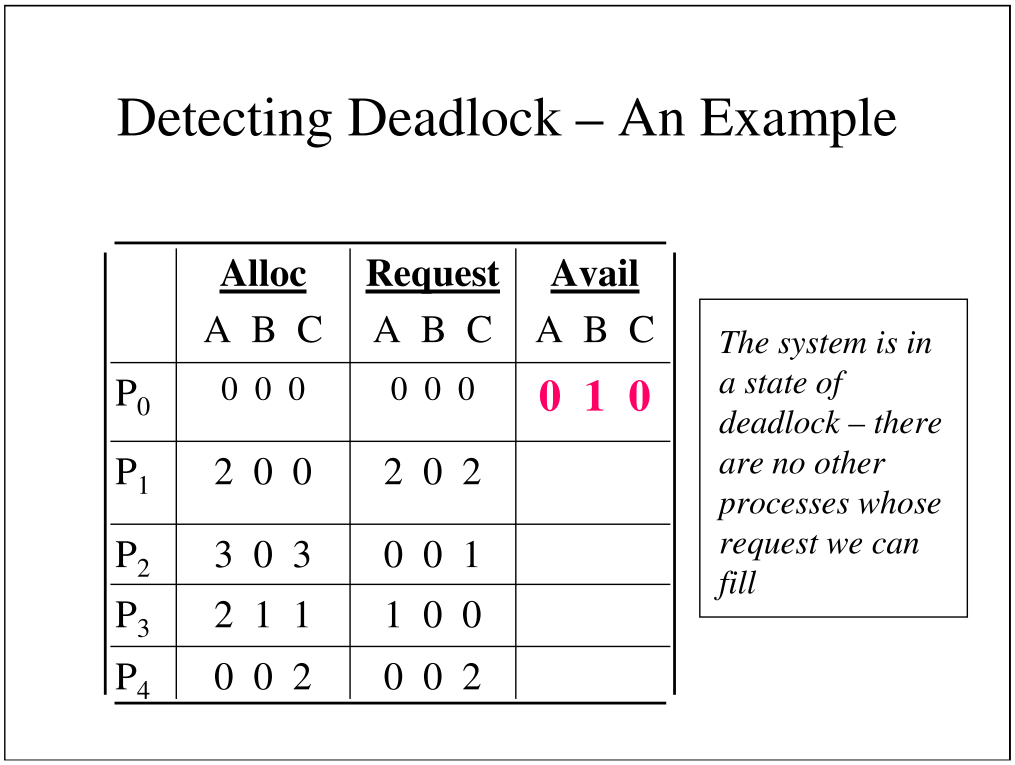
This is clearly NOT a deadlock because there is a sequence of processes that can be completed - that sequence is $<P_0, P_2, P_3, P_1, P_4>$

Detecting Deadlock – An Example

	Alloc	**Request**	**Avail**
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

$Request_0 \le Work$

Let's look at this system with one change - let P_2 request a single instance of Resource C. Since the only Process that can run to completion is P_0, let's assume that it does and we can re-allocate its resources.



Detecting Deadlock – An Example

	Alloc	Request	Avail
	A B C	A B C	A B C
P_0	0 0 0	0 0 0	**0 1 0**
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

The system is in a state of deadlock – there are no other processes whose request we can fill

We now have available a single instance of B - no A’s, no C’s. This is not enough to meet the demands of any of the other processes. Therefore, there is a deadlock between P_1, P_2, P_3, and P_4.

Recovery From Deadlock

- Once we detect deadlock, we have two alternatives for breaking it:
  - Terminating one of the deadlocked processes
  - Preempting the resources that belong to one of the deadlocked processes.

Detecting a deadlock is important, but it is even more important that we recover from it. We have two options for breaking a deadlock: we can terminate one or more of the processes in the deadlock or we can take back enough resources so that one of the deadlock processes can proceed to completion.

Process Termination

- If the operating system recovers from deadlock by terminating processes, there are two approaches that it can take:
  - Terminate all the deadlocked processes
  - Terminate deadlock processes until the deadlock is broken.
- If we use the second option, we have to determine the process whose termination incurs the minimum cost.

Terminating a process is not something that we really wish to do, but it's better to terminate a process than to leave several processes unfinished in a deadlock. Here, too, we have some alternatives: we can terminate all the processes in the deadlock or just enough to break the deadlock.

Obviously, terminating them all seems fair on one level, but at the same time it seem like overkill. We would prefer to kill only a few. Here what determines the processes that are terminated and the ones allowed to run is whatever constitutes the lowest cost. Such factors include:

Process priority

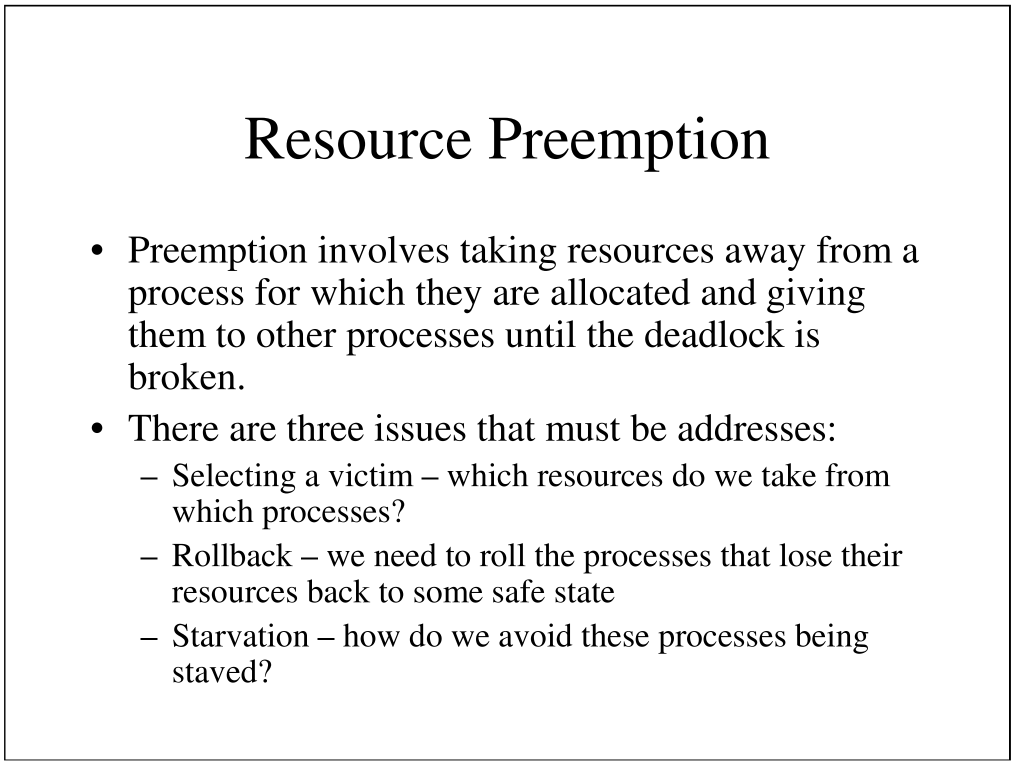
•How far the process has progressed and how far from completion it is?

•The resources that it has used

•The resources that it needs

•How many processes need to be terminated?

•Is it batch or interactive?



Resource Preemption

- Preemption involves taking resources away from a process for which they are allocated and giving them to other processes until the deadlock is broken.
- There are three issues that must be addresses:
  - Selecting a victim – which resources do we take from which processes?
  - Rollback – we need to roll the processes that lose their resources back to some safe state
  - Starvation – how do we avoid these processes being staved?

Preemption would seem to be preferable because it allows all the deadlocked processes to eventually run to completion. This also has a few issues that must be resolved:

Which resource or resources do we preempt from which process or processes?

•When we restore the process that has had its resources preempted, where do we pick up execution? In other words, how far do we roll back?

•How do we make sure that the process doesn't starve? We have to make sure that it can eventually reclaim the necessary resources.

Combined Approach

- Some researchers argue that using only prevention, avoidance or detection alone is not enough to handle the whole range of resource-allocation problems.
- An approach that combines these approaches will work best.
  - We could partition our resources into classes that are ordered in an hierarchical fashion, with each class subject to resource ordering.

Some researchers feel that the none of these three approaches work best on their own, that we really need an approach that combines elements of prevention, avoidance and detection/recovery. One possibility is to divide our resource into a set of classes, in which each class of resources is subject to resource ordering. This should allow us to avoid deadlock or minimize its impact.