Deadlock

Topics

- Deadlock
- Necessary conditions for deadlock
- Strategies for dealing with deadlock
- Strategies for detecting deadlock
- Deadlock in different models of resource allocation

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- Involves conflicting needs for resources by two or more processes
- No efficient general solution

Example

![Dining Philosophers Problem](image)

We have separately discussed the Dining Philosophers problem, demonstrated that it can result in deadlock, and shown two ways to avoid deadlock in solutions to that problem. We will now look at some more examples, analyze more carefully what deadlock means, and look at a broader range of solutions.

Traffic Deadlock Example from Stallings' Text
The figure, which is of a traffic deadlock, makes the nature of deadlock very clear. None of the cars can pass unless at least one of them backs up or is removed.

**Types of Resources**

- Reusable
- Consumable

Deadlock comes from contention between processes (or threads) for resources. So, to understand deadlock we need to understand resources.

One author of an operating systems textbook has proposed that a resource be defined as "something a process can wait for".

A first step in understanding resources is to recognize that they can be divided into two kinds, according to whether they can be reused or not.

**Reusable Resources**

- Fixed number of units
- Neither created nor destroyed
- Used by one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- several different possible modes of access:
  - exclusive
  - read/write (can be modeled by multi-unit)
  - shared (not treated by text)
- operations: request, grant/acquire, release
- Deadlock occurs if each process holds one resource and requests the other
The most is known about reusable resources. Deadlock is easy to analyze because of there being a known finite number of resources.

You can think of reusable resources as collections, each of which contains one or more units. At a high level of abstraction, all the different modes of access can be modeled in terms of exclusive access to different numbers of units of a resource.

In the simplest case, a single-unit resource, the collection has just one unit. An example of a single-unit resource is a mutex. Only one thread may hold a mutex at a time. The only mode of access is exclusive.

We have previously seen the readers-writers synchronization problem. Reader-writer access to a resource can be modeled by pretending there are "tickets" for the resource, sufficiently many that every reader can have a ticket. These tickets are the logical units of the resource. For a reader to access the resource it needs only one ticket. For a writer to access the resource, it needs all the tickets. Assuming the tickets are accessed in exclusive mode, this models gives the correct behavior for reader-writer locks. That is, any number of concurrent readers can obtain tickets to access the resource, but cannot succeed in getting all the tickets so long as another reader or another writer is holding a ticket.

Unrestricted sharing can also be modeled, by pretending there is a very large number of tickets for the resource, and only a single ticket is required to access it. However, unrestricted sharing means no waiting, and so is not a mode we have to consider when we study deadlock.

The steps by which a process (or thread) gets a resource acquires and releases a resource involve an alternation of control between the process and the operating system (or resource manager). The first step is for the process to ask for (request) the resource. This is at the initiative of the process. The second step is for the operating system (or resource manager) to grant the resource. This is the only step that is under control of the operating system, and the only aspects that the operating system can control is when (if ever) to grant the request. After the resource has been granted, the process is said to have acquired it. The third step is for the process to release the resource. This is again under the control of the process.

This alternation can be viewed like a game, in which the players alternate moves. In the deadlock game the operating system loses if a deadlock develops.

Questions:

What are examples of reusable resources?

Processors, I/O channels, main and secondary memory, files, databases, and semaphores

Consumable Resources

- Created (produced) by a process
- Destroyed (consumed) by a process
- Operations: request/receive (implies consume), produce/send
- Deadlocks may occur if the receive operation can block
- Potential deadlock situations are more difficult to detect, and reproduce, by testing

Deadlocks involving consumable resources are harder to analyze than those involving only reusable resources. The main problem is that the conditions under which a reusable resource may be produced can depend in an arbitrarily complex way on the internal logic of processes. Given the operating system has to deal with an unknown and arbitrary collection of user processes, cannot see into the processes, and could not predict their actions even if it could see into them (remember the Halting Problem), there is no general way for the operating system to prevent, avoid, or even recognize the existence of a deadlock involving consumable resources. For this reason, dealing with consumable resource deadlocks is outside the scope of
our present study of operating systems.

By the way, this difficulty with deadlock is arguably one of the disadvantages of message-based synchronization. By comparison, if we can solve a problem using simple mutual exclusion alone (that is, single-unit reusable resources) there are simple design techniques that guarantee there is no possibility of deadlock.

Questions:

What are examples of consumable resources?

Hardware interrupts, Unix signals, pthread_cond_signal calls, messages, and information in I/O buffers.

**Deadlock Example with Reusable Resources**

Process P1:
request(R1); ... request(R2); ... release(R2); ... release(R1);
   R1 critical section
   R2 critical section

Process P2:
request(R2); ... request(R1); ... release(R1); ... release(R2);
   R1 critical section
   R2 critical section

The pseudocode above shows two processes, each of which has nested critical sections for the same pair of resources. They request the resources in a different order, so there is a possibility of deadlock. Suppose P1 requests R1, and that is granted. Then P2 requests R2, and that is granted. Then P1 requests R2, and that cannot be granted because R2 is held by P2, so P1 blocks. Then P2 requests R1, and that cannot be granted because R1 is held by P1, so P2 blocks. The two processes are not deadlocked.

**Resource Allocation Graph of a Deadlocked System**

![Resource Allocation Graph](image)

The figure shows the deadlock that is possible for the two pseudocode processes above, modeled by a (single-unit) resource allocation graph. This is the simplest of several graph models for resource allocation we will consider below. (The others involve multiunit and consumable resources.) There are two types of nodes: process nodes (show as squares) and resource nodes (show as circles). There are two types of edges: held-by edges (shown as solid arrows) go from a resource to the process that is holding it; wait-for edges (shown as dashed arrows) go from a process to the resource it is waiting for. Notice that there is a cycle in the
graph. A cycle in the resource allocation graph is a necessary condition for deadlock (in every model). For single-unit resources (but not in the more general models) it is also a sufficient condition.

### Deadlock Trajectory Diagram

![Deadlock Trajectory Diagram](image)

The figure shows a trajectory diagram for the same pair of processes P1 and P2 as the pseudocode and resource allocation graph shown above.

A trajectory diagram is useful for visualizing how a system can get into a deadlock. We can draw such a diagram if we have just two processes and the steps in the execution of each process progress can be viewed as a simple sequence of steps \(i.e.,\) we don't have to consider condition control flow or looping. The combined state of the two processes corresponds to a point \((X, Y)\) in the plane, where \(X\) is the state of one process and \(Y\) is the state of the other process. In our example, the progress of one process P1 is plotted on the Y (vertical) axis and the progress of process P2 is plotted on the X (horizontal) axis.

The pink shaded regions correspond to states that are inside the critical region for resource R1. The darker pink region covers the states in which both processes would need to hold R1, which would violate mutual exclusion, and so that region is forbidden.

The grey shaded regions correspond to states that are inside the critical region for resource R2. The darker grey region covers the states in which both processes would need to hold R2, which would violate mutual exclusion, and that region is forbidden.

The white dots correspond to states whose entry is under the control of the OS, and black dots represent states whose entry is under the control of the application. The system start state is shown at the lower left corner. In the start state, neither P1 nor P2 has started execution, so neither is holding any resources.

The green lines show two possible system trajectories. The solid green line shows a trajectory that leads to deadlock. It is deadlock because there is no way the system can make progress without entering one of the forbidden regions. Here, progress in process P1 means movement upward and progress in process P2 means movement toward the right.

Notice that the deadlock state is in the corner of a concave pocket that is formed by the intersection of the two forbidden regions. This pocket is a trap. Once we get into it we cannot get out without backing up. Since preemption is not allowed, the system cannot force a process to back up.

The problem here can be viewed as a board game in which the processes are playing against the operating system. The goal of the operating system is to not get into a deadlock state. The goal of the processes is to try to force the operating system into a deadlock. The moves of the processes are resource requests and resource releases. The moves of the operating system are resource grants. The only choice of the operating system is
how long it postpones granting a request.

The dashed green line shows an alternate trajectory, which enables both processes to complete without deadlock. The critical difference between the two trajectories is at the point where P1 requests R1. If the system grants the request immediately (which it can, because P2 is not using R1 at the time), ultimate deadlock is unavoidable. If the system postpones granting P1's request for R1, P2 is given time to release R2, and there will be no deadlock. A good resource manager can force the system along the safe trajectory, since the decision when to allocate a resource is up to the resource manager.

---

**Example of Deadlock with Consumable Resources**

```
P1                      P2
...                    ...
receivefrom(P2, &M2);  receivefrom(P1, &M1);
...                    ...
sendo(P2, M1);         sendo(P1, M2);
```

Deadlock occurs if the `receivefrom` operation is blocking.

---

The pseudocode above shows how deadlock is possible with message passing. Process P1 and P2 each wait for a message from the other, and while they are waiting neither is able to send the message that the other is waiting for. This kind of deadlock is difficult to prevent unless one imposes some constraints (protocols) on the sending and receiving of messages.

---

**Natural Questions about Deadlock**

What are the fundamental causes of deadlock?

How can we deal with deadlock?

---

Now that we have looked at examples of some specific cases of deadlock, it is time to look at deadlock from a general point of view.

---

**The 4 Necessary Conditions for Deadlock**

1. Exclusive access (mutual exclusion)
   - only one process may use a resource at a time
2. Wait while holding (hold-and-wait)
   - A process can continue to hold a resource while requesting another
3. No preemption
   - A process cannot be forced to give up resources before it chooses to give them up
4. Circular wait
   - There is a cycle of hold-and-wait relationships
In order for there to be a deadlock, all of the above conditions must be true. You can observe that they are true in the examples we considered. Richard Holt, in a PhD dissertation published in the 1970's, showed that these conditions must apply in any deadlock. You can verify that informally, by looking at each condition and convincing yourself that if the condition is not true, there is no deadlock.

The 3 Approaches/Strategies for Dealing with Deadlock

- Prevention - apply design rules to insure it can never occur
- Avoidance - dynamically steer around deadlocks
- Detection - hope deadlocks will not occur, but recover when one does

Questions:

How can we deal with deadlock?

What is the difference between prevention and avoidance?

What are examples of strategies for prevention?

There are only three approaches to dealing with deadlock. We can allow it to happen. In that case we need to be able to detect it when it has happened, and then deal with it. Alternatively, we can try to make sure deadlock does not happen. There are two general approaches to that. One is called prevention and the other is called avoidance. Prevention means always restricting what we do in such a way that there is no possibility of deadlock. Avoidance means waiting to impose restrictions until we are close to a deadlock, and then "steering around" the deadlock.

Experience has shown that some people have difficulty distinguishing prevention form avoidance. The technical way to understand the difference is in terms of safe and unsafe states, which we will get to a bit later. A more intuitive way is by analogy to other situations in which a person needs to deal with a potential hazard.

Analogies:

Suppose there is an open hole in the middle of a street. One way to prevent people from falling into the hole is to cover it. Another is to put up a barricade around it. A third avoidance strategy would be to never use that street. A way to avoid falling into it is to keep your look where you are going, and steer around the hole.

Suppose a person does want to get a dangerous communicable disease, such as AIDS. A prevention strategy would be to not engage in any of the activities by which the disease can be transmitted. An avoidance strategy would be to engage in the risky activities, but try not to engage in them with anyone who is a carrier of the disease.
In general, hazard prevention is safer and simpler than avoidance, but it is more limiting. In designing software systems, it is a good idea to design in a way that prevents hazards, to the extent the system requirements allow you to do so.

With deadlock, the prevention approach is to impose some restriction on how resources are used, that assures there can never be a deadlock. The avoidance approach is to impose no restrictions on resource usage, up to a point where it there is a danger the system will deadlock, and then guide the system around the deadlock, by postponing some allocations.

**Deadlock Prevention Approaches**

*All* of the four conditions are necessary for deadlock to occur. Hence, by preventing any one of them we prevent deadlock.

1. Exclusive access (mutual exclusion)
   - redesign to eliminate the need for mutual exclusion
     *(Can you think of an example?)*
2. Wait while holding (hold-and-wait)
   - If a process holding resources is denied a further request, the process must release all its resources and rerequest them
   - Require that a process request all of its required resources at one time
3. No preemption
   - If a process requests a resource that is currently held by another process, the OS preempts the second process and requires it to release its resources
4. Circular wait
   - Define a linear ordering of resources and require allocations be requested only in this order

If we look at each of the four necessary conditions for deadlock, we can see how deadlock might be prevented by denying that condition. Whether one of these prevention strategies can apply to a given resource depends on the nature of the resource and how it is used. For example, ordered allocation works well for mutexes, but preemption of a mutex is not acceptable.

**Questions:**

What happens to a process if resources it is holding are preempted?

Is it safe for the process to just go on executing?

Short of killing the process that is preempted, how might this problem be addressed?

**Cycle Implies Different Allocation Orders**
For a hold-and-wait cycle to exist, one process must request a pair of resources in a different order than another. In the diagram, P1 requested R1 (which it is holding) and then requested R2, but P2 requested R2 (which it is holding) and then requested R1.

In general, if we number the resource nodes and walk around a cycle in the resource allocation graph, there must come a point where we go from a higher-numbered resource to a lower numbered resource. It follows that if we never request a resource while holding a higher-numbered resource, there can be no cycle.

### Ordered Allocation

<table>
<thead>
<tr>
<th>Process P1:</th>
<th>Process P2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>request(R1); ... release(R1);</td>
<td>request(R1); ... release(R1);</td>
</tr>
<tr>
<td>R1 critical section</td>
<td>R1 critical section</td>
</tr>
<tr>
<td>R2 critical section</td>
<td>R2 critical section</td>
</tr>
</tbody>
</table>

If all processes always request resources in a fixed order, there can be no hold-and-wait cycle.

This is usually not a burdensome restriction. It still allows us to have critical sections that involve more than one resource, and use simple lock and unlock operations on individual resources to implement them.

### Trajectory Diagram for Ordered Allocation

The trajectory diagram shows that there are no "concave" regions formed by the intersections of the forbidden regions, so there are no accessible deadlock states.

### Release Before Request

Process P1:
request(R1); ... release (R1); request(R2); ... release(R2);
If processes always release one resource before requesting another, there can be no hold-and-wait

This is a tougher restriction. It rules out critical sections that involve more than one resource.

Request All at Once

If processes always request all resources they will need at once, there can be no hold-and-wait

This allows critical sections that involve more than one resource, but requires the operating system (or resource manager) to provide a way to atomically request several resources at once. Some batch and real-time operating systems support this, but it is not supported, for example, by the POSIX thread mutex API.

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of worst-case future process requests
- Approaches:
  1. Postpone starting a process if its demands might lead to deadlock, 
     \textit{i.e.} while resources it may need are held by others
  2. Postpone granting an incremental resource request to a process if granting the allocation might lead to deadlock

Deadlock avoidance requires foreknowledge of the worst-case resource requests a process may make.

The two logical points at which a deadlock avoidance decision can be made are the point of process admission and the point of an incremental resource request.

Requirements for Deadlock Avoidance

- Maximum resource requirement must be stated in advance
Processes under consideration must be independent; no synchronization requirements
There must be a fixed number of resources to allocate
No process may exit while holding resources

Maximum resource requirement information may not be available, except for certain kinds of resources and certain restricted types of processes. The availability of this kind of information was typical of traditional batch processing systems, and is true for some real-time operating systems.

The kinds of resources this method may be applied to are also limited. It has been used for allocating tape drives to jobs in a batch-oriented operating system. Another example of an application is to the allocation of signal processors to a main-CPU process in a real-time signal processing system.

### Banker's Algorithm

An example of a deadlock avoidance algorithm.

- Avoids deadlock by postponing incremental requests
- State of the system is the current allocation of resources to processes
- Safe state is one from which there is no way processes can force the system into deadlock
- Unsafe state is a state that is not safe


The banker's algorithm postpones deadlock avoidance decisions until the point of an incremental resource request, which if granted would take the system into an unsafe state.

The following slides contain an example from Stallings' text, which we walk through in class.

It is helpful to also go through some other examples, to make certain you understand the algorithm. One way to do this is using the applet to which link is given above.

### System with Potential for Deadlock

**Process P1:**
```
request(R1); ... request(R2); ... release(R2); ... release(R1);
```
R1 critical section
R2 critical section

**Process P2:**
```
request(R2); ... request(R1); ... release(R1); ... release(R2);
```
R1 critical section
R2 critical section

We will review the two-process two-resource deadlock example given in pseudocode above, to see how deadlock could be avoided.

### Trajectory to Unsafe State
This trajectory diagram is identical to the one given for the same example further above, except that one region of the diagram has been colored gold. That region contains the \textit{unsafe states} of the system. Notice that the unsafe states are in a concave pocket, formed by the intersection of the two forbidden regions (the dark pink and dark grey regions). Once the system enters this pocket, it cannot get out without either backing out (violating nonpreemption) or entering a forbidden zone (violating mutual exclusion).

Notice that there is only one deadlocked state, in the upper right corner of the unsafe region. The other states in this region are unsafe, but not deadlocked.

The green trajectory in the diagram above corresponds to the following, which one many possible sequences of moves:

- P2 requests R2
- The OS grants the request of P2
- P1 requests R1
- The OS grants the request of P1
- P2 requests R2
- The OS cannot grant the request of P2, because R2 is locked.
- P1 requests R2
- The OS cannot grant the request of P1, because R1 is locked.

No further progress is possible, \textit{i.e.} we have a deadlock.

The system made a fatal error when it granted P1's request for R1 while P2 was holding R2. This took the system into an unsafe state, from which the processes P1 and P2 were finally able to drive the system into deadlock.

\textbf{Questions:}

What is the difference between a safe state and a deadlock-free state?

\textbf{Resource Allocation State Transition Diagram}
Another way of looking at safety is in terms of a state transition diagram. The figure shows the states and transitions of the example system considered further above. The red state is the deadlock state. The shaded yellow states are unsafe states. The black and green states are safe states. The black transitions and states are under control of the processes, and the green transitions and states are under control of the operating system (or resource manager).

The property that makes the unsafe region unsafe is that there are no transitions out of the unsafe region *that are controlled by the operating system*.

In this diagram, the system will eventually reach deadlock, once it enters an unsafe state, since all the transitions lead toward the deadlock state.

In this regard the example gives a false impression, that there can be no transitions out of an unsafe region, at all. That is because the example has no conditional branching in the code. In a more general example there might be transitions out of the unsafe region that are under control of the processes (but none under control of the operating system).

### How Deadlock Is Avoided

Deadlock avoidance algorithms keep track of the system state, and avoid making transitions that lead from safe states to unsafe states.
The green path on the diagram above shows how the system can steer itself around unsafe states, for the same sequence of requests from P1 and P2 as in the deadlock example considered previously. The difference is that the system does not grant every request immediately, just because the resource is available. Instead, the system blocks P1 at its request for R1 until after P2 releases R2.

P2 requests R2
The OS grants the request of P2
P1 requests R1
The OS postpones granting the request of P1, because it would lead to an unsafe state.
P2 requests R1
The OS grants the request of P2
P2 releases R1
The OS grants the request of P1 for R1
P1 requests R2
The OS cannot grant the request of P1, because R2 is still locked by P2.
P2 releases R2
The OS grants the request of P1 for R2
P1 releases R2
P1 releases R1

How do we know a state is safe?

- Start in the given state
- Simulate running each process to completion,
  by allocating its maximum resource requirements,
  and then releasing all resources
- If all processes can complete, the state is safe

Representing State as Set of Tables

- Claim matrix [Process][Resource]: Integer
  maximum requirement of each resource type for each process
- Allocation matrix [Process][Resource]: Integer
  current allocation of each resource type for each process
- Resource vector [Resource]: Integer
  total number in system of each resource type
- Allocation vector [Resource]: Integer
  number currently available of each resource type

The banker's algorithm represents the state of the system by a set of tables, which are shown above. This example comes from Stallings' text. In class, we walk through the example, step by step, explaining each step. Stallings gives the same explanation in his text, so it is not repeated here.

Determination of a Safe State: initial state
### Determination of a Safe State: P2 runs to completion

#### Claim Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Allocation Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Available Vector

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

### Determination of a Safe State: P1 runs to completion

#### Claim Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Allocation Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Available Vector

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Determination of a Safe State: P3 runs to completion

#### Claim Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Allocation Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Available Vector

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

### An Unsafe State: initial state
### An Unsafe State: P1 requests one unit each of R1 and R3

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

### Claim Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

### Allocation Matrix

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Resource Vector

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### Available Vector

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

### Deadlock Detection with Tables

#### How to Detect Deadlock?

- Similar to detecting an unsafe state
- Simulate execution of unblocked processes, assuming they will complete and release all resources

It would be a good idea here to jump ahead momentarily to look at example of **graph reduction** much further below. Some people may find it easier to grasp quickly than the example with tables.

---

#### Deadlock Detection with Tables

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Request Matrix Q**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Allocation Matrix A**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Resource Vector**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Available Vector**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Mark P4.
Select P3 because it is unblocked.

---

#### Deadlock Detection with Tables (Step 2)
Complete and mark P3.
No other process is unblocked, so we have detected a deadlock.

---

**What to do when Deadlock is Detected?**

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart it from the checkpoint
  - original deadlock may reoccur
- Kill deadlocked processes until deadlock no longer exists
- Preempt resources until deadlock no longer exists

---

**How to Choose which Process to Abort?**

- Least amount of processor time consumed so far
- Least number of lines of output produced so far
- Most estimated time remaining
- Least total resources allocated so far
- Lowest priority

---

**Graph Models of System Resource Allocation States**

- Wait-for graphs (WFG)
- Single-unit resource allocation graphs
- Multi-unit resource allocation graphs
- General resource allocation graphs (GRG)

---

**Wait-For Graphs (WFG)**

Nodes correspond to *processes (only).*
There is an edge from process $P_1$ to process $P_2$ iff $P_1$ is blocked waiting for $P_2$ to release some resource.

This type of graph models the wait-for relationships among processes, at a high level of abstraction. A cycle in this graph is a necessary condition for deadlock. Whether existence of a cycle is sufficient for deadlock depends on the kinds of wait-for relationships among processes we allow.

The following forms of graphs provide more detail, showing enough about the specifics of the wait-for relation that we can talk about deadlock detection.

**Single-Unit Resource Allocation Graphs**

In this model there are two types of nodes, corresponding to processes and resources, respectively. Each resource is a single-unit reusable resource.

There is a *request* edge from process $P$ to resource $R$ iff $P$ is blocked waiting for an allocation of $R$.

There is an *assignment* edge from resource $R$ to process $P$ iff $P$ is holding an allocation of $R$.

In this model, the existence of a cycle is a necessary *and sufficient* condition for deadlock.

*How do we model multi-unit resources?*

**Multiunit Resource Allocation Graphs**
In this form of graph the resource nodes are complex. There is an outer-level node, which represent a collection of resources, and one or more nodes within, which represent units of that resource. A request is represented by an edge from a process to a resource collection. An allocation is represented by an edge from a unit of a resource to a process.

The graph shown in the figure is a multi-unit reusable resource graph. There are three processes (P₁, P₂, and P₃) and three collections of resources (R₁, R₂, and R₃). Resource collections R₂ and R₃ each have only a single unit. Resource collection R₁ has two units. Process P₁ is holding one unit of resource R₁ and one unit of resource R₃ and waiting on a request for the second unit of resource R₁. Process P₂ is holding one unit of resource R₁ and is waiting on a request R₂. Process P₃ is holding one unit of resource R₂ and is waiting on a request R₂. This is a deadlock, since every process is blocked on a resource request, and none of the requests can be satisfied.

The example can be modified slightly and used to show that the existence of a cycle in a multiunit resource graph is not a sufficient condition for deadlock. Just remove the edge that goes from P₁ to R₁, i.e., suppose process P₁ has not (yet) made its request for a second unit of R₁. There would still be a cycle in the graph, but since P₁ would not be blocked, there would be no deadlock.

---

**Advanced Topics**

Undergraduate students are not expected to go beyond this point in class, but graduate students should know this material.

---

**General Resource System**

A system state consists of:

- set of processes Π = {P₁, P₂, ..., Pₙ}
- set of resources Γ = {R₁, R₂, ..., Rₘ}
- for every reusable resource Rᵢ, a nonegative integer tᵢ, the total number of units present in the system
- for every consumable resource Rᵢ, a nonempty set of producers of Rᵢ
The general resource graph adds the capability of modeling reusable resources, such as messages. This allows us to model the kinds of deadlock that arise in message-passing systems.

**General Resource Graph**

Bipartite directed graph

- nodes are in sets P and G
- \((r_1, r_2, \ldots, r_m)\) = available units vector
- \((P,R)\) is a request edge
- \((R,P)\) is an assignment edge if R is reusable
- \((R,P)\) is a producer edge if R is consumable

**Questions:**

What is a bipartite graph?

Can you give examples of graphs, operations on graphs, deadlock, and safe/unsafe states, in terms of graphs?

**Request Models**

- single-unit
- AND request model
- OR request model
- AND-OR request model
- P-out-of-Q request model

**Questions:**

What is a sufficient condition for deadlock in each of these models?

**Deadlock Conditions in WFG**

Cycle is always a *necessary* condition for deadlock, but it is not sufficient in all models.

- single-unit -- cycle is sufficient
- AND request model -- cycle is sufficient
Insufficiency of Cycle with Multi-unit Resources

$P_j$ is a producer of $R_i$

The notation $\#(R_i, P_j)$ denotes the number of units of resource $R_i$ that are currently held by process $P_j$, and $\#(R_i, *)$ denotes the total number of units of resource $R_i$ that are currently allocated to all the processes in the system.

Questions:

What is the relationship between a system state and a GRG?

Can you describe an algorithm for detecting a deadlock on a GRG?

Operations on General Resource Graph

- request: add request edges (requires $P$ was not blocked)
- grant: replace reusable request edges by assignment edges, and decrement $r_j$ (requires the resource was available)
- produce: delete consumable request edges (requires the producer was not blocked)
- release: delete assignment edges, and increment $r_j$ (requires $P$ was not blocked)

Graph Reduction

Models a most favorable possible (optimistic) continuation of execution from the current system state.

while there is an unblocked process $P_i$:

1. For each request edge $(P_i, R_j)$, remove the request edge and decrement the corresponding count $r_j$. This corresponds to satisfying the request.
2. For each allocation edge $(R_j, P_i)$, remove the allocation edge and increment the corresponding count $r_j$. This corresponds to releasing the allocation.
3. For each consumable resource production edge $(P_i, R_j)$, remove the producer edge from the graph, and set $r_j$ to infinity. This corresponds to assuming $P_i$ produces as many units of $R_j$ as may be required by other processes.

There is deadlock iff the remaining graph is non-null

Example: the state in question
P1 is not blocked.

Example: after completion of P1

P3 is not blocked.

Example: after completion of P3

P2 is not blocked.
Example: after completion of P2

All processes have completed.

For systems with consumable resources, the final outcome depends on the sequence in which the processes have been reduced. Hence, we must try all possible sequences of reductions. That could be prohibitively expensive. However, for systems with only reusable resources one can prove that if a graph can be reduced completely, the order in which the reductions are performed does not matter.

Practicality of GRG Reduction

- Useful for characterizing properties of deadlock systems
- Not practical for implementation
- Reducibility is dependent on the order of the reductions
- Worst time complexity is $O(mn!)$ where $m$ is number of resources and $n$ is number of processes

Note: We are just talking about graphs that have consumable resources here. Graph reduction is practical for reusable resource graphs.

Expedient States

A state is expedient iff all processes having outstanding requests are blocked, i.e. all grantable requests have been granted.

Note: A system must eventually get to one of these states if it is deadlocked.

Theorem In a GRG,

- A cycle is a necessary condition for deadlock
- If the graph is expedient, then a knot is a sufficient condition for deadlock

This still leaves a gap. That is, the existence of a cycle is necessary but not sufficient, and the existence of a knot is sufficient but not necessary.