CPS221 Lecture: Deadlock

last revised July 30, 2014

Objectives

1. To introduce the concept of deadlock
2. To discuss approaches to deadlock prevention, avoidance, and detection/ resolution

Materials:

1. Projectable of simple example of a self-deadlocking program
2. Projectable of Figure 7.9 in Silberschatz, Galvin and Gagne 8th ed
3. Demo of Lab 6 software; basic synchronization plus version using lookahead to prevent deadlock
4. Projectable of example trajectory
5. Projectable of trajectory space annotated with resource needs
6. Projectable and handout of banker's algorithm
7. Projectable of three RRAG's
8. Projectable of deadlock detection algorithm

I. **Introduction**

A. One of the major responsibilities of any multi-user operating system is managing the sharing of resources by the various processes.

We now want to look at one of the problems that can easily arise when multiple processes compete for the same resource: The problem of deadlock.

B. Definition

1. A process is said to be deadlocked if it is waiting for an event that can never occur.

2. A system is said to be deadlocked if it contains one or more deadlocked processes. The deadlock may be partial (some processes can still proceed) or total (no process can proceed.)

C. Deadlock can arise in many different ways

1. The following Java program illustrates how a process can deadlock itself:

PROJECT

```
class DeadlockDemo
{
        public static void main(String [] args)
        {
             Object foo = new Object();
             while (true)
             {
                  try
                  {
                       synchronized(foo)
                       {
                            foo.wait();
                       }
                  }
                  catch(InterruptedException e) {}
             }
        }
}
```

The program waits for the object foo to be notified - but since too is a local variable in main, no other code even sees it! (Of course, one would hardly write a program so blatantly incorrect; but similar problems can easily be embedded unwittingly in a more complex program.).

2. Often, deadlock arises when two processes compete for resources. As an example, the suppose a computer has a DVD drive and a printer. One process is using the DVD drive and needs to use the

printer before it is through with DVD tape drive. (Perhaps it needs to print something from the DVD). The other process is using the printer and needs to use the DVD drive before it is through with the printer. (Perhaps it needs to include something from the DVD in what it is printing).

3. Deadlock can arise when processes communicate by passing messages. For example, suppose two processes at different locations are communicating over a channel that is subject to interference. Process B is currently waiting for a message from process A. Suppose A sends a message to B and then goes into a wait state awaiting a reply from process B. If the message is somehow lost, then B will wait forever for A's message that was lost and A will wait forever for B's reply to the message which B will never receive.

4. Another place where deadlocks can occur is in connection with shared access to common data structures. For example, suppose that a bank has various ATM's scattered throughout the city, and suppose that each ATM is handled by its own process on a central computer. When an individual starts to perform a transaction on a particular account, the record for that account must be locked until the transaction is complete. Competition for access to a record can occur if a husband and wife simultaneously use different ATMs to access a joint account, or if an ATM access occurs while a central batch process is posting the day's checks to the account in question. We could envision a deadlock occurring as follows:

    a) Mr. Jones arrives at an ATM desiring to transfer money from his savings account to his checking account.

    b) As his request is being processed, the ATM first obtains a lock on the record for his savings account and is now ready to request a lock on the checking account.

c) At that moment, on another ATM, Mrs. Jones requests a transfer from checking to savings. The process for her ATM gets a lock on the checking account record and now attempts to get a lock on the savings account record.

   It might be argued that a scenario like the above is highly improbable. That is certainly true. But it is the very improbability of such a scenario that makes it so insidious. If the potential for a deadlock such as this were present in a system, it would probably not manifest itself during system testing unless the testers were specifically conscious of looking for such a problem. But, given enough time, the undetected bug could show up, causing two ATM's to lock up. In this case, the cause of the problem could probably be readily found if the operator who clears it knows what to look for; but imagine a similar problem occurring in a process control system running a nuclear power plant. (The deadlock may never be discovered if the computer is ruined by the resulting meltdown!)

5. Deadlocks can also arise in connection with the invisible resources of the system.

   a) Example: When a physical IO operation to a device like a disk is done, a buffer must be set aside in memory to contain the data being transferred. For various reasons, many systems use a pool of buffers in system memory space for this. Thus, a user request to perform an IO operation (say write) is implemented as follows:

      - Allocate a buffer in system space.
      - Copy the data from the user's memory to the system buffer.
      - Start the IO transfer.

      Unfortunately, this operation, while not involving the explicit allocation of a device, does involve an implicit allocation of a buffer from a pool of buffers maintained by the system. If no

buffer is available, then the process must wait for one. Thus, in a system that performs IO through system buffers a process can become deadlocked waiting to perform an IO operation to a device it already owns.

b) Example: Many operating system calls - such as opening a file - require that an appropriate entry be made in some system table. (In the case of opening a file, the entry gives the name of the file and the process which has it open to prevent possible access conflicts if another process tries to access the same file.)

   The system tables are often implemented as linked lists of small buffers obtained from a system pool. Again, if no space is available to store the new table entry, then the process must wait.

c) Example: Deadlock can occur without any specific activity by user processes. For example, one early multi-processor operating system handled accounting as follows:

   (1) When a CPU finished performing some service for a user process, it would write a record to an internal accounting log stored in shared memory.

   (2) When the accounting log became full, a request would be generated for the first available processor to write it out to a disk file.

   (3) Deadlock occurred in this system (as originally designed) if all of the CPU's were working on user processes when the internal accounting log became full. As each CPU finished its task for its user, its last step would be to write the accounting data, and then it would become free to take on some other task. But if the log was full, it would have to wait until some other free CPU could write the data out to disk. But if all the CPU's were serving users, none could ever become free to flush the internal log until it could write its user accounting data to the internal log!

D. The initial work on deadlock was done in conjunction with the requesting of resources like card readers, tape drives, and printers in a multi-user computer. While many of these resources are no longer present in modern computers, some of the classic examples of deadlock are couched in terms of them - but the same principles can be used in dealing with deadlock in other situations, as you will discover in lab and as we will discuss in conjunction with databases.

   In general, theoretical work on deadlock views a system as consisting of a set of resources which are partitioned into classes of identical resources, such that any resource of a given class can satisfy any request for a resource of that class.

   1. Examples: CPU's, blocks of memory, physical devices such as printers, etc.

   2. It is important that all the resources in a class be regarded as interchangeable. If this is not the case, then we have multiple classes.

      Example : suppose a system has two printers, one in the basement and one on the 9th floor. These would probably be considered as belonging to two distinct resource classes, rather than as being two instances of the one class "printer".

   3. In many of our examples we will work with situations in which each class contains only a single resource; but this is only for clarity of example and is not generally true.

E. We need to carefully distinguish deadlock from a related problem: starvation.

   1. A process is said to be starving if it waits forever for a resource that is denied it due to some bias in the allocation policies of the system - i.e. what it is waiting for <u>could</u> happen, but never does.

2. Poorly-designed deadlock avoidance schemes can inadvertently cause a process to starve. For example, if a certain scheme arbitrarily gives priority for some resource to processes with an odd process number, and there is much competition for this resource, then an even-numbered process might never be able to obtain it. The process is not deadlocked, because the event it is waiting for CAN occur - it just never does occur. But the effect on the starving process is the same as if it were deadlocked.

II. **Four conditions necessary for deadlock**

A. Deadlock involving shared resources can only occur if four conditions are true. Three of these are preconditions that must be present in the system design, and the fourth is the actual condition that constitutes deadlock:

1. Mutual exclusion: The resources in question can each be used by only one process at a time. (Precondition)

2. Hold and wait: A process that is waiting for a resource that it needs may hold onto other resources it has been granted, making them unavailable to other processes. (Precondition)

3. No pre-emption: Once a process has been allocated a shared resource, it cannot be forced to yield the resource until it has finished using it. (Precondition).

4. Circular wait: there exists a set of processes p0 .. pn such that:

   a) p_0 is waiting for a resource held by p_1.
   b) p_1 is waiting for a resource held by p_2.
   c) ...
   d) p_n is waiting for a resource held by p_0.

   (When this condition actually arises, the system is deadlocked. The processes $p_0..p_n$ are each deadlocked, though any other processes on the system may proceed.)

B. The three preconditions are related to the nature of the resources themselves:

1. Resources can be classified into three broad types:

   a) Shareable resources can be used by more than one process at the same time.

      (1) Examples: read-only disk files, shared code.

      (2) Note that shareable resources cannot give rise to deadlock because condition 1 does not hold for them.

   b) Pre-emptible resources cannot be used by more than one process at a time, but a pre-emptible resource can be taken away from one process and granted to another with minimal cost

      (1) The classical example is the CPU, which can be given to a new process by storing the registers in the PCB for the current process and loading the registers from the PCB of the new process. Pre-emption of the CPU is the basis for multiprogramming.

      (2) Main memory is generally pre-emptible by copying an entire process, or a portion of it (to disk and loading a new process or portion of it in its place.

      (3) Since a pre-emptible resource does not satisfy condition 3, it cannot give rise to deadlock. (There is a subtlety here, though, in connection with a resource like memory. If no space is available in the swapping/paging file, then an attempt to pre-empty memory may have to wait until a slot becomes free! The resource being waited for is a slot in the file, but the immediate cause of the wait is the preemption attempt.)

c) Serially re-usable resources can be used by only one process at a time. Only when one process has finished its use of the resource can another process use it.

(1) Example: sequential IO devices such as printers,

(2) Example: internal buffers maintained by the operating system.

(3) Example: Writeable disk files, or portions of disk files. (In general, only one process may have write access to a given record in a file at any one time. But in practice it is not always possible to lock just one record, so when a process is granted write access, the OS may have to lock out all write access by other processes to the entire block or cluster containing the record or - in some systems - may have to lock out write access to the entire file.

(4) These are the resources that can participate in deadlocks.

2. One approach to preventing deadlocks involves an attempt to convert a serially reusable resource into some other type. For example, many systems maintain a queue of print requests. Though the printer can only service one request at a time, multiple processes can submit print requests, where they wait their turn until they can be printed. The effect is that the printer looks to processes like a shareable "virtual printer".

C. Schemes for dealing with deadlock attack one or the other of these conditions.

1. Deadlock prevention schemes operate by ensuring that deadlock can never occur. They do so by making one of the three preconditions false, or by guaranteeing that the fourth condition cannot possibly arise.

2. Deadlock avoidance schemes operate by making sure that, while deadlock could occur in principle, the fourth condition does not occur in practice. This is done by recognizing and avoiding situations that would lead to a circular wait happening.

3. Deadlock detection schemes operate by looking for the occurrence of the fourth condition and taking measures to break up the deadlock.

D. We can illustrate these schemes in terms of a more familiar situation - traffic deadlock.

PROJECT Figure 7.9 in Silberschatz, Galvin, and Gagne 8th ed and/or DEMO Deadlock Lab software

1. What could we do in this situation to prevent deadlock?

   ASK

   A simple solution would be to build an overpass for one of the intersections. (Just building <u>one</u> would take care of the problem!)

2. What could we do in this situation to avoid deadlock?

   ASK

   We could require that a vehicle not enter an intersection until it can see the road clear its full length on the other side.

   Though this would probably work in the physical world, it could fail if vehicles arrive from all four directions simultaneously, see the road ahead clear, and enter the intersection simultaneously - or if all four vehicles wait for another to first!

   DEMO using Lab software

3. Deadlock detection and recovery could be handled by a traffic policeman coming out and do something like making one of the deadlocked trucks drive off the road.

III. Deadlock prevention schemes

A. Conceptually, the cleanest way to deal with deadlock is to prevent it by ensuring that one of the conditions for deadlock does not hold.

B. In practice, however, this may impose restrictions that may not be tolerable - so working systems often combine deadlock prevention for some resources with one of the other two schemes for others.

C. Theoretically, deadlock can be prevented by denying the mutual exclusion pre-condition. This is usually not possible in practice - though the use of print queues to convert serially reusable devices into seemingly shareable virtual devices could be regarded as an attack at this point.

   (We could also regard it as a multiplication of the number of printers to the point where each process can have all it needs.)

D. Deadlock can be prevented by denying the "hold and wait" precondition.

   1. One approach is as follows: Require that a process request all the resources that it ever needs in one single request at one time. The system will not grant any resource in the list until it can grant all of them.

   2. A less restrictive approach is to allow a process to request resources only when it is currently holding no resources. Thus, if a process needs a new resource, it must first yield all the resources it has and then put in its request (which might include a request for the reallocation of a resource it just gave up.)

   3. Problems with this approach:

a) It can lead to processes holding resources when they don't need them, thus reducing resource utilization. This is especially serious if a process does not know what resources it will actually need for a given run until it has started working on the data. With this scheme, it must request up front all the resources it MIGHT need.

b) A process that needs several "popular" resources might starve while processes that need a smaller number of these resources keep taking them away.

E. Deadlock can be prevented by denying the "No Pre-emption" condition.

1. One way to implement this is to stipulate that any process that is forced to wait for some resource will have any resources it already possesses taken away from it. The wait for a single resource is then converted into a wait for a list of resources including both those it had and the one it now needs.

2. Again, this can be made somewhat less severe. A process that is waiting for some resource can hold them as long as another process does not need them. But if another process should request a resource held by the waiting process, the resource is pre-empted and the waiting process must now wait for both the original resource it wanted and the resource that was taken away.

3. This scheme also has problems:

a) It only works if the resources are pre-emptible. If a process has printed output on a printer and is waiting for some other resource before it can generate more output, then the printer really cannot be taken away without messing up the output.

b) This scheme can also lead to starvation for a process that needs several "popular" resources at the same time, since it may keep losing the resources it gets because they don't all become available at the same time.

F. Deadlock can be prevented by making circular wait impossible.

1. One approach that is often practical relies on something called resource ordering or ranking. With each resource, we associate a unique number. We require that a process request resources in increasing order of resource number - i.e. if the process needs resources 3 and 5, it must request them in the order 3, 5.

   Example: This was the approach we used in the Dining Philosophers problem. In the version that did not deadlock, chopsticks were requested in chopstick number order - which meant that most philosophers could request them in the left then right, but the last philosopher had to request them in the order right then left because his right was 0 and his left was 4.

2. When there are multiple similar resources, an entire class of resources may be given the same number, with the additional provision that if a process needs multiple resources of the same type it must request them all at once - it cannot ask for a second "type N" resource if it already has a "type N" resource.

3. Again, this restriction can be loosened; if a process releases all the high-numbered resources it holds it may be allowed to request a lower-numbered resource. It still cannot request a resource of a given type if it already holds one or more resources of that type. That is, the number of the resource it requests must be strictly greater than the number of any it holds.

4. To see that this protocol does, in fact, make circular wait impossible, we use a proof by contradiction. Assume that the resource ordering protocol is being used, and a circular wait has resulted. This means that we have a process p_0 waiting on a resource held by p_1, and p_1 is waiting on a resource held by p_2 ... process p_n waiting on a resource held by p_0

a) Let r_0 be the resource held by p_0 which p_n is waiting for, r_1 be the resource held by p_1 that p_0 is waiting for ... rn be the resource held by p_n that $p_{(n-1)}$ is waiting for.

b) Let f_0 be the number associated with resource r_0, f_1 be the number associated with r_1 etc.

c) Now since p_0 is waiting for a resource r_1 while holding a resource r_0, it must be the case that $f_0 < f_1$. In like manner, $f_1 < f_2$ etc. So we have $f_0 < f_1 < f_2 \ldots < f_n$ - and therefore, by transitivity, $f_0 < f_n$.

d) But we also have process p_n requesting a resource r_0 held by p_0, while holding a resource r_n. Therefore, it must be that $f_0 > f_n$.

e) Since this is a contradiction, our assumption that circular wait could arise is false. QED

5. Of course, this scheme has problems too:

   a) The order of resource numbering may prove arbitrary and inconvenient. This is not necessarily too serious of a problem, since there are often natural ways of numbering resources. For example, processes generally use input devices (such as a DVD reader) before output devices (such as printers), etc.

   b) The order of numbering can force processes to request resources before they need them, thus reducing resource utilization.

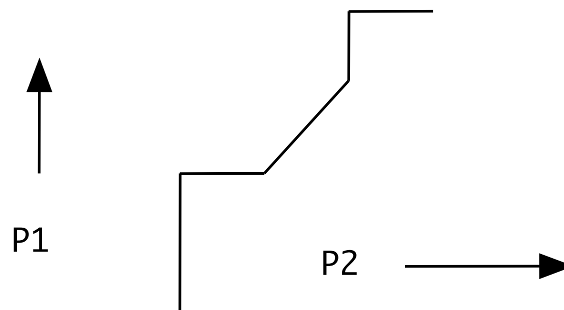
IV. Deadlock Avoidance Schemes

A. Another approach to deadlock is to configure a system so that deadlock could theoretically occur, but then avoid deadlock by not granting any resource request that could ultimately lead to deadlock. The basic idea is this: ordinarily, the operating system grants resource requests from processes on the basis that if the requested resource is available, then the process gets it, otherwise it waits. We modify this so that, in some cases, a process is forced to wait for a requested resource even though the resource is available.

B. One way to see how this works is with trajectories.

1. The following example shows the evolution of two processes over time:

PROJECT



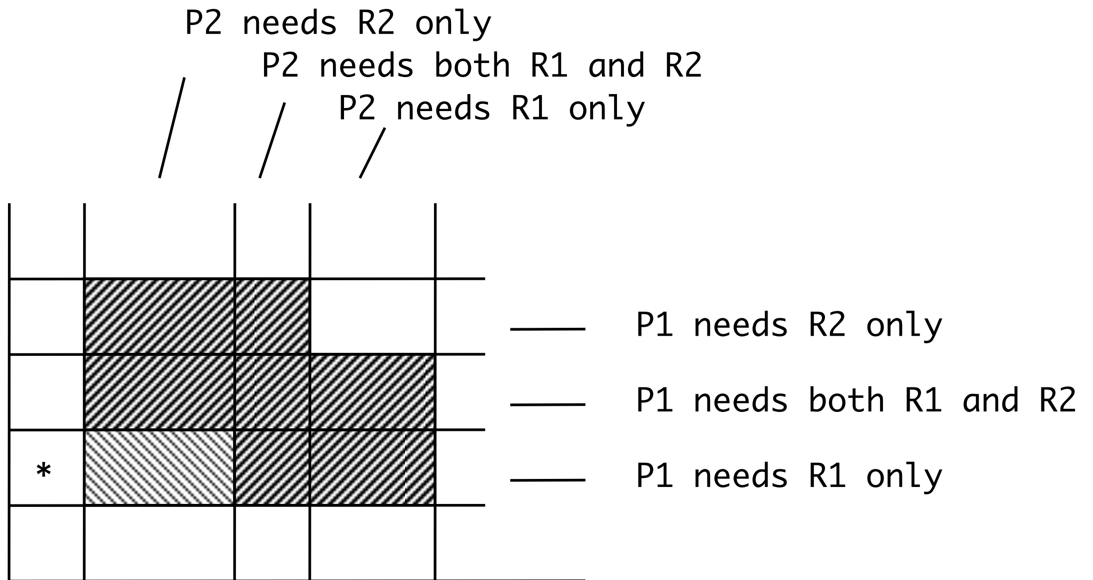
a) When P1 is running on the CPU, the trajectory moves up; when P2 is running, it moves to the right. When both are running at the same time (multiple CPU's or one doing IO while the other computes) it moves diagonally.

b) The trajectory can never move to the left or down.

2. We could construct such trajectories for any number of processes - but the drawing would have one dimension per process. Thus, with three processes we would have a 3D drawing etc.

3. We can now notate on the drawing the resources each process needs at various points in its history:

PROJECT



a) The region shaded ▨ is an impossible region. The trajectory cannot enter this region, because the resource requests of both processes cannot simultaneously be satisfied within it.

b) The region shaded ▧ is an unsafe region. If the trajectory enters this region, the system must deadlock. Note, however, that the unsafe region is not itself deadlocked. Both processes can progress until the trajectory reaches the boundary with the impossible region. At this point only one can progress, until the trajectory reaches the corner where both must stop.

c) There are two broad ways open for deadlock free trajectories - one to the left of the unsafe/impossible regions, and one below them.

d) A deadlock avoidance algorithm, would anticipate the inevitable deadlock resulting from entrance into the unsafe region, and would prevent entrance into it. In particular, if the processes were at the point labeled by the '*' (with P_1 holding

R_1), a request from P_2 for R_2 would be delayed even though R_2 is available, since that would setup inevitable deadlock.

C. The above has illustrated the general approach taken by all deadlock avoidance algorithms: some knowledge about the future behavior of a process is used to prevent disastrous choices. In this case, the knowledge is that P_1, once holding R_1, would need R_2 before it could progress far enough to release R_1.

D. The best-known deadlock avoidance algorithm is the banker's algorithm.

PROJECT and HANDOUT - Walk through

1. The advance knowledge required is the maximum number of units of each type of resource that the process will claim at any one time.

    a) This can be declared explicitly up front by the programmer; or it may be determined implicitly from the job control language in a batch environment.

    b) Any process which requests an allocation beyond its pre-declared maximum will be aborted.

    c) Of course, it cannot be the case that the declared maximum for resource for any process is greater than the total number of that resource available on the system to begin with!

2. Walk through how the system decides whether to grant a request coming in from some process. The basic approach is to check three things:

    a) Check to be sure that the process does not request more resources than its declared maximum. (If so, the process must be aborted)

    b) Check to be sure that the process is not requesting more resources of any type than are currently available. (If so, the request must be delayed.)

c) Check for safety by seeing if, even with the request having been granted, all processes can complete in some order by repeating the following until all processes have been shown to be able to complete.

   (1) Find a process that can surely complete (its outstanding needs is not greater than the number of unallocated resources for any resource type)

   (2) Pretend it gives back all the resources it has to the system (which makes more resources available for some other process)

d) Grant the request only if safe to do so.

3. Example: A system has 2, 5, and 1 unit respectively of resource types 0, 1, 2. Three processes are running, but as yet hold no resources. Their declared maximum needs are 1,2,1; 1,4,1, and 1,1,1.

max	1 2 1
	1 4 1
	1 1 1
allocated	0 0 0
	0 0 0
	0 0 0
(So need = max)	
available	2 5 1

Consider the following series of requests:

a) P0 requests	1 2 0
(legal and possible)	
Simulate allocation	1 2 0
	0 0 0
	0 0 0
Need becomes	0 0 1

	1 4 1
	1 1 1
Work	1 3 1

All processes can complete in the order P0, P1, P2

Grant the request

Allocated resources becomes	1 2 0
	0 0 0
	0 0 0
Available becomes	1 3 1

b) P1 requests 1 2 0
(legal and possible)

Simulate allocation	1 2 0
	1 2 0
	0 0 0
Need becomes	0 0 1
	0 2 1
	1 1 1
Work	0 1 1
- P0 can complete - so Work becomes	1 3 1
- P1 can complete - so Work becomes	2 5 1
- P2 can complete	
grant the request	
Allocated resources becomes	1 2 0
	1 2 0
	0 0 0
Available becomes	0 1 1

c) P1 requests 0 2 0
(legal but not possible, so delay the request - allocated and available do not change)

d) P2 requests 0 0 1
(legal and possible)

Simulated allocation	1 2 0
	1 2 0
	0 0 1
Need becomes	0 0 1
	0 2 1
	1 1 1
Work	0 1 0

No process can be guaranteed to complete, so this request is unsafe and so must be delayed. (allocated and total available do not change)

E. Problems with deadlock avoidance schemes:

1. Knowledge of the future behavior of a process is required. This may be hard to come by in some cases - especially interactive systems.

2. This approach can be unduly conservative - making processes wait because a deadlock might occur though, in fact, none would even if the request were granted.

3. Deadlock avoidance depends on a knowledge of how many resources of each type are available. Should a resource unexpectedly fail, a deadlock could still occur.

4. The deadlock avoidance algorithm is time-consuming. If the number of processes is n and the number of resources is m, then the banker's algorithm is $O(mn^2)$: The loop in the check for safety will have to be executed n times if the allocation is safe. On each iteration, we have to find a process that could complete - which potentially requires examining all n and may typically require looking at n/2 if there is only one that can complete. Examining a process to see if it could complete requires looking at its need for

all m resources. Since the banker's algorithm must be run for each resource request generated, the overhead can be quite high on a large system. (Note: there is a simpler algorithm that can be used if all the resources are unique - i.e. each "class" contains exactly one resource. This is still $O(n^2)$, however.

5. In some cases, deadlock avoidance algorithms can lead to starvation.

V. **Deadlock Detection**

A. We have seen that both deadlock prevention and deadlock avoidance schemes have a price associated with them that may not be acceptable.

B. Another approach to deadlocks is possible if, in a given system, their occurrence is determined to be relatively rare. We can allow deadlocks to occur from time to time, detect the fact that a deadlock has occurred, and undo it. This approach raises two issues:

1. How to we determine that a deadlock has occurred

2. How to recover from it.

C. One way to visualize whether a deadlock exists in a system is through the use of a Resource Request and Allocation Graph (RRAG).

1. In such a graph:

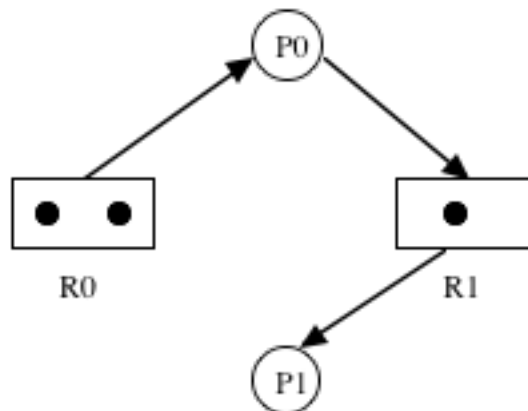
a) Processes are represented by circles.

b) Resource types are represented by rectangles. The number of instances of a given type of resource is represented by a dot within the rectangle.

c) There is an edge from a process to a resource type if the process is currently requesting a resource of that type. (This is called a request edge)

d) There is an edge from a resource to a process if the process currently has a resource of that type. (This is called an allocation edge)

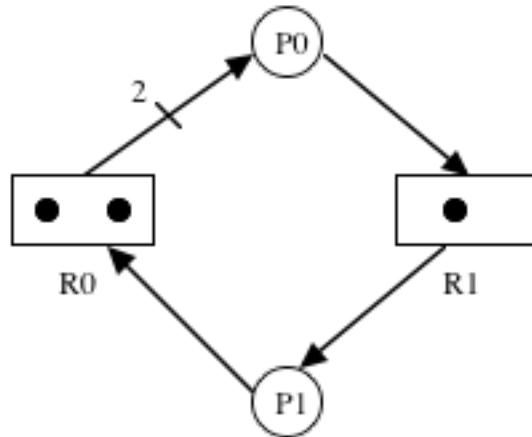
Example: The following RRAG depicts a system with two instances of resource type R0 and one instance of resource type R1, plus two processes. P0 has one instance of resource type R0 and is requesting resource type R1. P1 has the instance of resource type R1.
PROJECT



2. A cycle in such a graph may indicate the presence of deadlock.

Example: Suppose P0 obtains one more instance of R0 before requesting an instance of R1, and then P1 requests an instance of R0. The graph would now look like this; the cycle involving P0 and P1 indicates a deadlock.

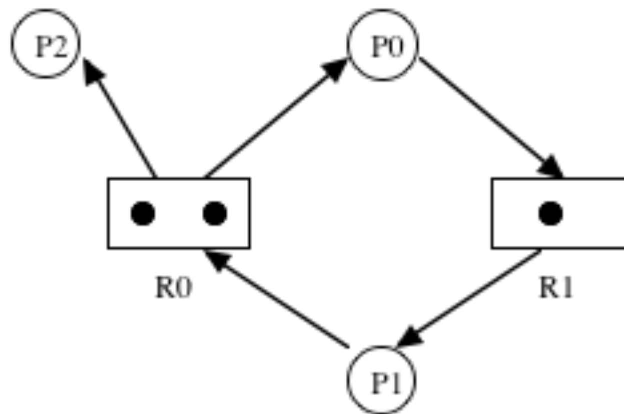
PROJECT



3. On the other hand, the presence of a cycle in the graph does not necessarily indicate a deadlock if there is some process not part of the cycle that can release a resource that breaks up the deadlock.

   Example: Suppose there is a third process in the system (P2) and it is the one having the second instance of R0, rather than P0:

   PROJECT



   Now, though P0 and P1 are currently deadlock, the deadlock will be broken if P2 releases its instance of R0

D. The existence of deadlock can also be determined by an algorithm that is similar to the Banker's algorithm, but simpler

   PROJECT Deadlock Detection Algorithm

1. There is no need for prior knowledge about what a process might do.

2. Like the banker's algorithm, this algorithm is $O(mn^2)$. But unlike the banker's algorithm, we do not have to run it for every resource request. We may schedule it to run periodically, or run it when any process has waited more than a specified interval for a resource.

3. As before, there is a simpler algorithm $O(n^2)$) that can be used if all resources are unique.

E. Deadlock Recovery

1. This is the more difficult problem. One or more processes are going to have to be forced to yield resources involuntarily.

2. The simplest approach is to abort one or more processes and then restart them. Of course, the work they have done is lost. Also, this may not be possible if a process has made changes to a database.

   (e.g. suppose a process is posting checks against checking accounts. Restarting the process from the beginning could lead to the same check being debited to your account twice!)

3. A less severe approach is to make use of checkpoints in the run of a process. Many large computations incorporate the notion of dumping the state of the process at periodic checkpoints so that, in the event of a system crash, the process can be safely restarted from the last checkpoint rather than from scratch. In such a situation, it may be possible to roll a process back to a checkpoint.

4. One other issue that is important is that if deadlocks are frequent then care must be taken that the algorithm for selecting a victim process results in starvation of some process.

VI. **Combined approaches**

A. We have seen that, in practice, no one method of dealing with deadlocks is without problems.

B. Therefore, most practical systems use a combined approach.

1. For example, resource ordering may be used with resources for which it is possible to establish a natural order.

2. Deadlock detection and recovery may be used with resources that rarely result in deadlock.

3. Etc.