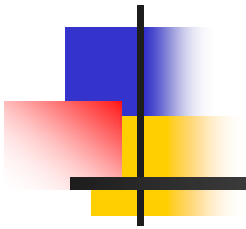
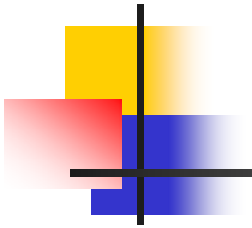


Deadlock and Starvation



Deadlock and Starvation

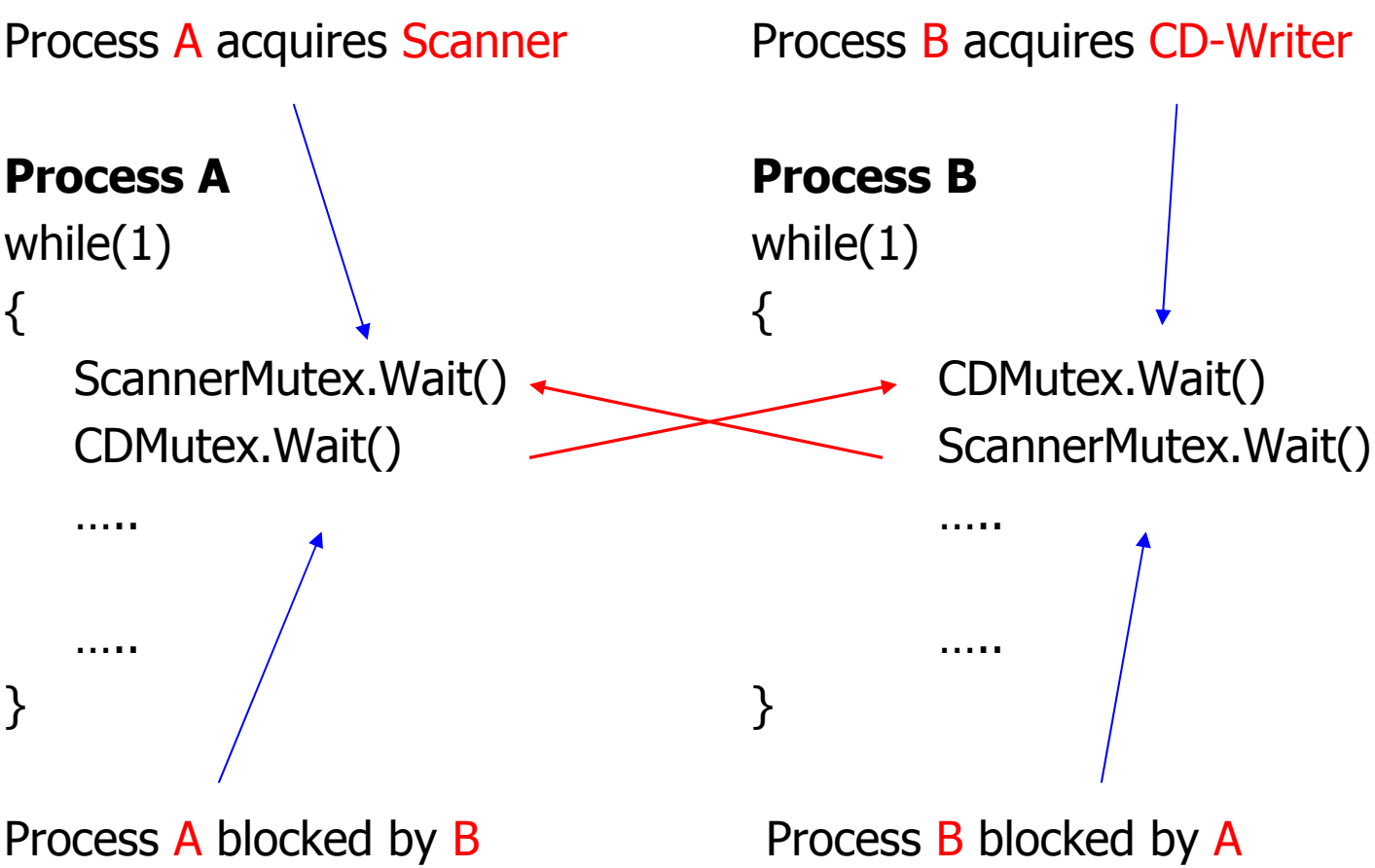
Introduction to Deadlock

- Computer systems as we know from discussions of mutual exclusion are full of resources that can only be accessed by one process at a time. Common examples include
  - Printers,
  - Tape Drives
  - Databases etc.
- Such problems are generally solved using variations of a Mutex or Semaphore to protect the resource as we have seen previously. However mutual exclusion on its own may not be enough, since there are many situations where a process may actually need exclusive access not just to one resource, but perhaps to several resources simultaneously and this is where the potential for deadlock and/or starvation arises.

Example

- As an example, suppose two processes A and B each require access to a scanner and to a CD-Writer to publish a document. Each of these resources has a mutex to enforce single thread access to it
- Let's imagine that A starts off by acquiring the scanner, by performing a Wait() on its mutex.
- At the same time B (which is programmed to acquire the resource in a different order to A), trys to acquire the CD writer, by performing a Wait() on its mutex.
- Process A, having got the scanner, now attempts to acquire the CD writer but is blocked when it performs the Wait() on the mutex acquired by B.
- Worse than that however, is the fact that Process B now attempts to acquire the Scanner and it too is blocked when it performs a Wait() on the mutex acquired by A.
- In other words both processes are waiting for the other process to release a resource it needs but neither process is prepared to give up the one resource it has – In summary we have a deadlock situation and potential starvation where neither process will ever get the resource it needs.

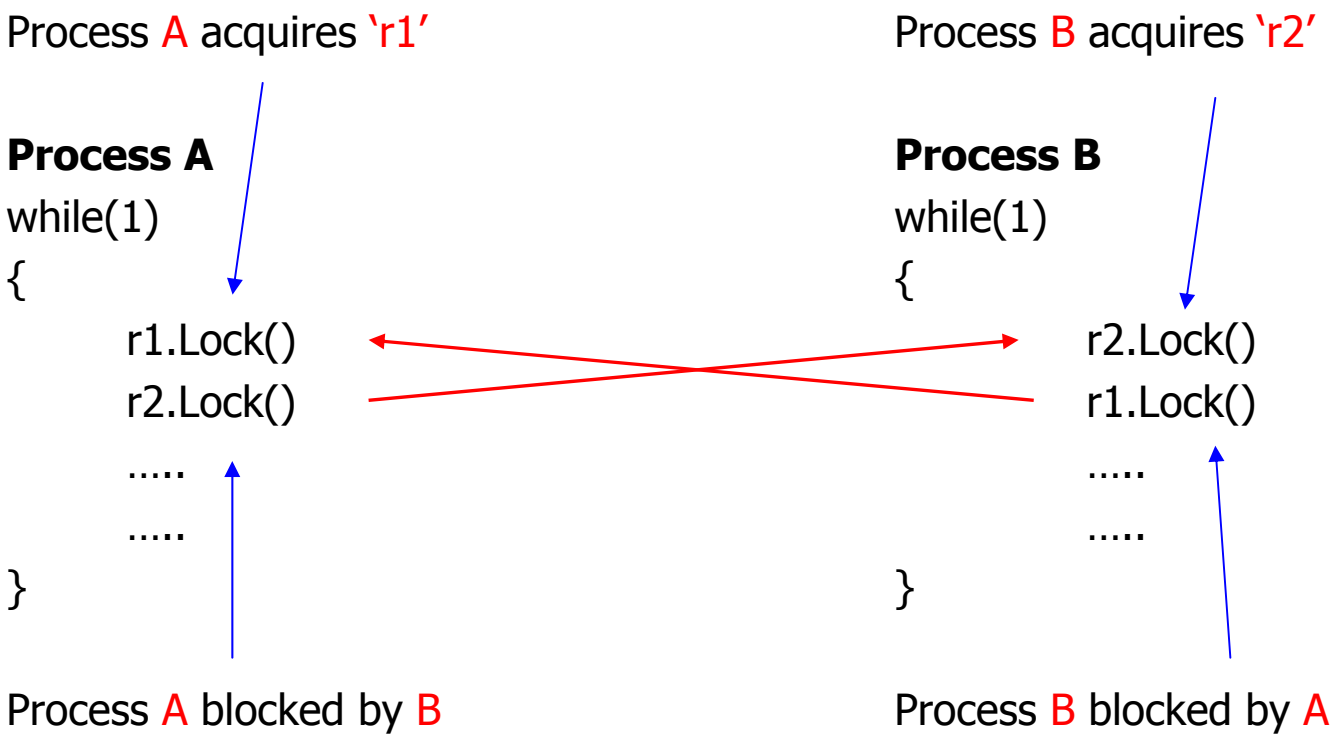
Deadlock and Starvation



Deadlock and Starvation

Deadlock in Data Base Applications

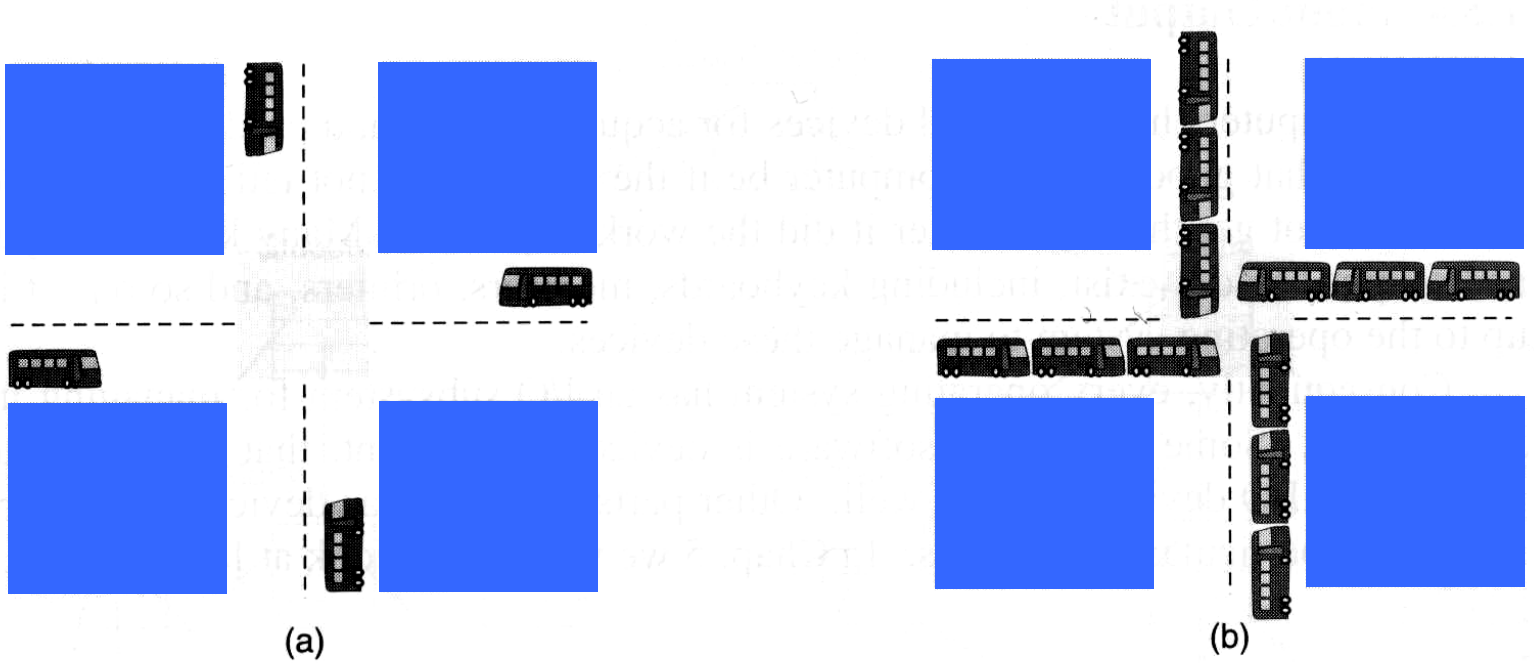
- The same problems can also arise in database systems.
- For example, in order to perform a database update, a process may have to lock several records to prevent them being changed by other processes running at the same time.
- If process A locks record r1 and process B locks record r2, and then each process tries to lock the other one's record, we also have a deadlock.



Deadlock and Starvation

An example of Real-life Deadlock - Gridlock

- As you can see from the diagram below, the potential for deadlock arises on today's overcrowded roads and is commonly referred to as gridlock.



(a) A potential deadlock. (b) An actual deadlock.

Conditions for Deadlock

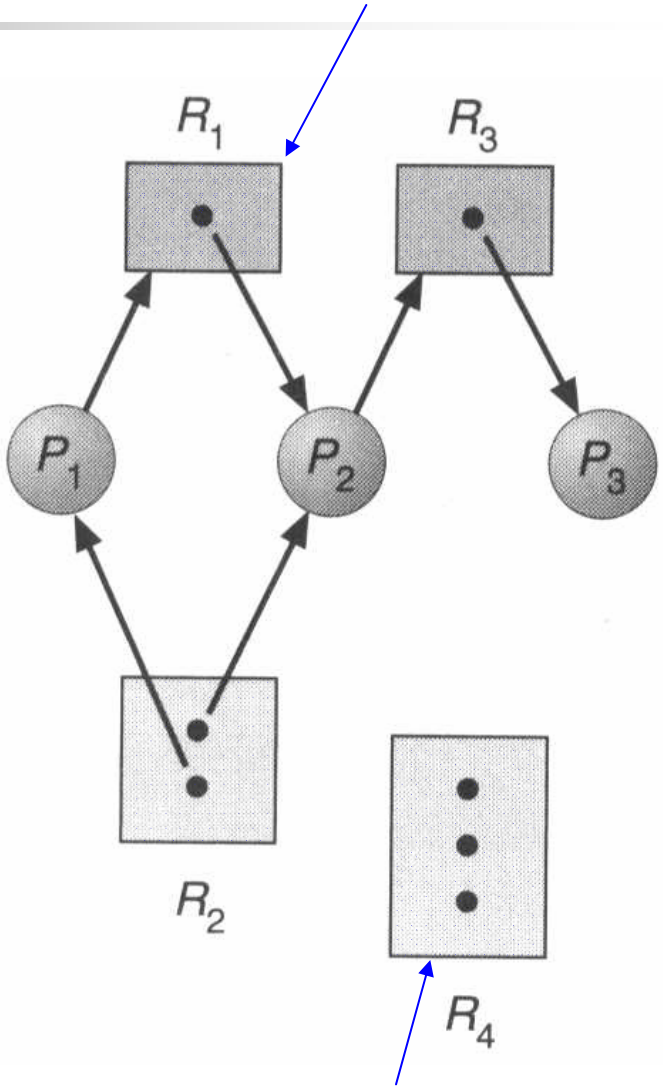
- Coffman and others showed in 1971 that four conditions must hold true for deadlock to arise.

1. At least one resource must be **non-sharable** amongst the competing processes; that is only one process can use that resource at any one time. Others wishing to use the resource will be blocked until it is released.
2. A process must be **holding** at least one resource and **requesting** another held by other processes.
3. No pre-emption is possible, that is, once a resource has been **granted** to a process, it cannot be **forcibly** taken away from that process.
4. There exists at the moment of deadlock, a circular wait condition among the processes and resources they are requesting, i.e. each process must be waiting for a resource held by another process and this queue of waiting dependencies forms a circular loop.

Deadlock and Starvation

Modeling Deadlock using Directed Graphs

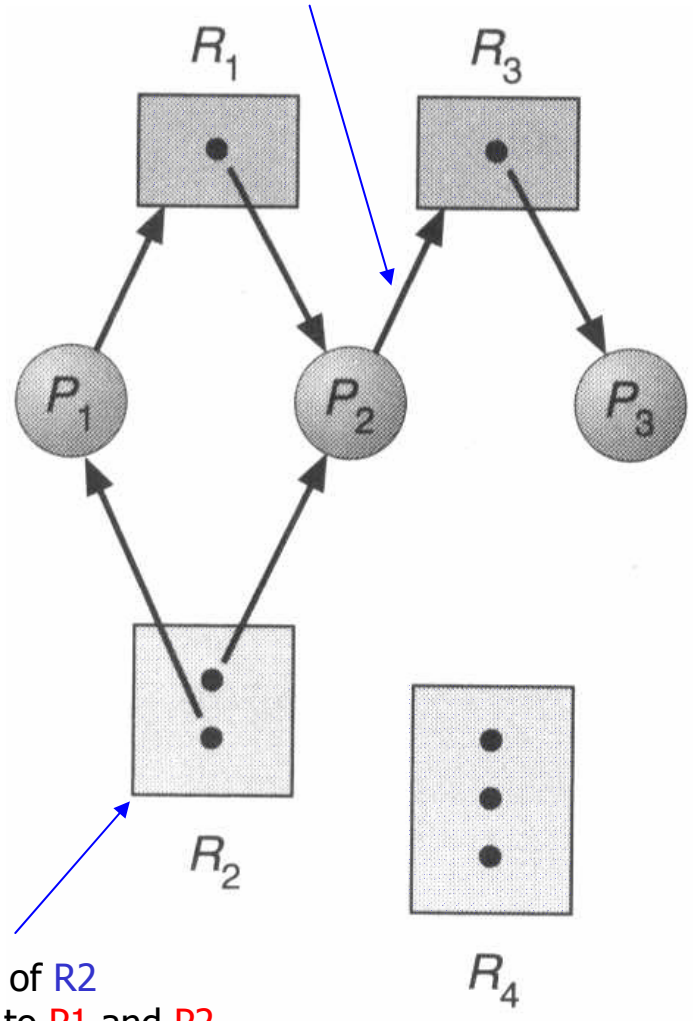
- Holt showed in 1972 that these four conditions can be modeled using a directed graph and used to predict if deadlock has or could potentially arise in a system.
- Such a graph has two elements
  - Processes shown as circles
  - Resources shown as rectangles
- Processes and resources are connected by arrows which show pending requests for and allocations of resources.
- Enhancements to this technique allow for multiple instances of a particular resource to exist using a small black dot within the resource rectangle.
- For example, the illustration opposite shows three processes P1, P2 and P3 and four resource types, R1, R2, R3 and R4.
- There is just one instance of the resource type R1 and R3 (shown by the single block dot inside the resource), while there are two and three instances respectively of resource type R2 and R4.



Deadlock and Starvation

Interpreting the Graph

- Such a diagram indicates which resources are currently allocated to processes and which processes are attempting to acquire those resources.
- For example
  - P1 has been allocated an instance of R2 and is requesting an instance of R1 (which cannot be granted as it has been allocated to P2)
  - P2 has been allocated an instance of R1 and R2 (the latter is not blocked by P1 because there are two instances of R2) and is requesting an instance of R3 (which cannot be granted as it has been allocated to P3)
  - P3 has been allocated an instance of P3 and is requesting nothing.
  - No instances of R4 have been requested or allocated
- We show these dependencies using the example notation below
  - R2->P1->R1->P2->R3->P3
  - R2->P2



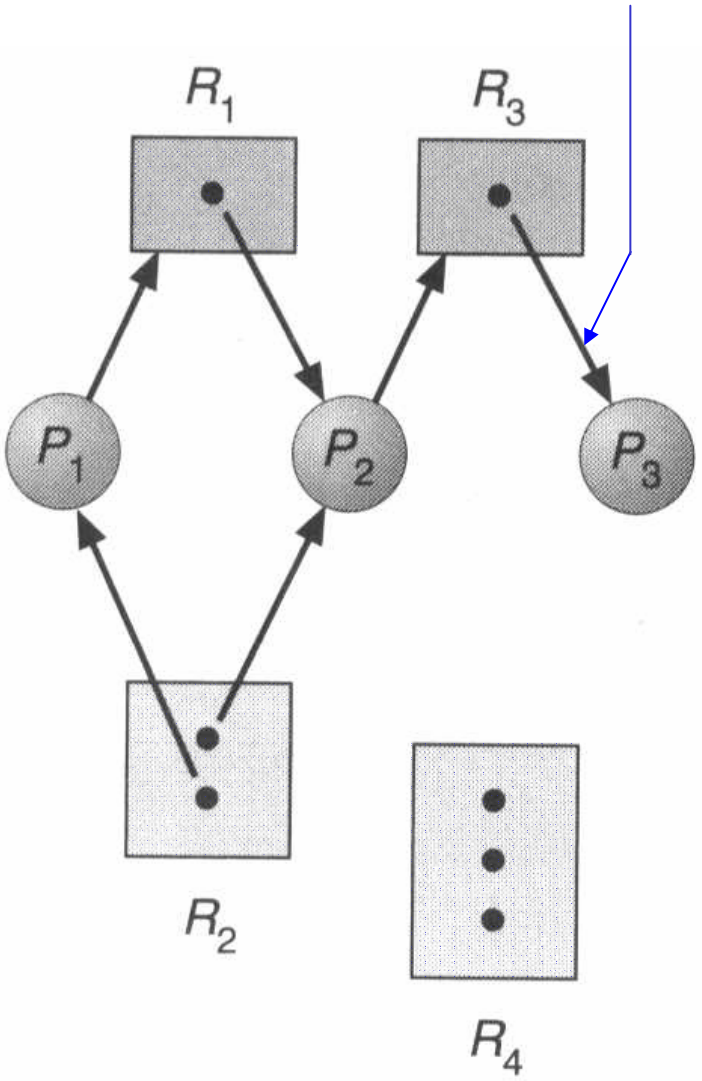
Deadlock and Starvation

Detecting Deadlock from a Graph

- In essence such graphs can be created and maintained by operating systems to enable them to detect deadlock (*though to be honest very few operating systems bother as we shall see later*).
- The question then is how do we tell from the graph when deadlock has occurred? In essence deadlock has occurred when there is a circular list of allocation/request dependencies.
- In the example opposite, deadlock has NOT occurred, since even though
  - P3 is blocking P2's request for R3,
  - and P2 is blocking P1's request for R1,

  P3 can release R3 when it has finished and this will propagate backwards allowing P2 and from there P1 to acquire the resource it needs to continue.

- That is, P1 is requesting R1 which has been given to P2 which in turn is requesting R3 which has been given to P3, but P3 has not completed a circular dependency since it is not blocked on a resource which has been allocated to any other process.



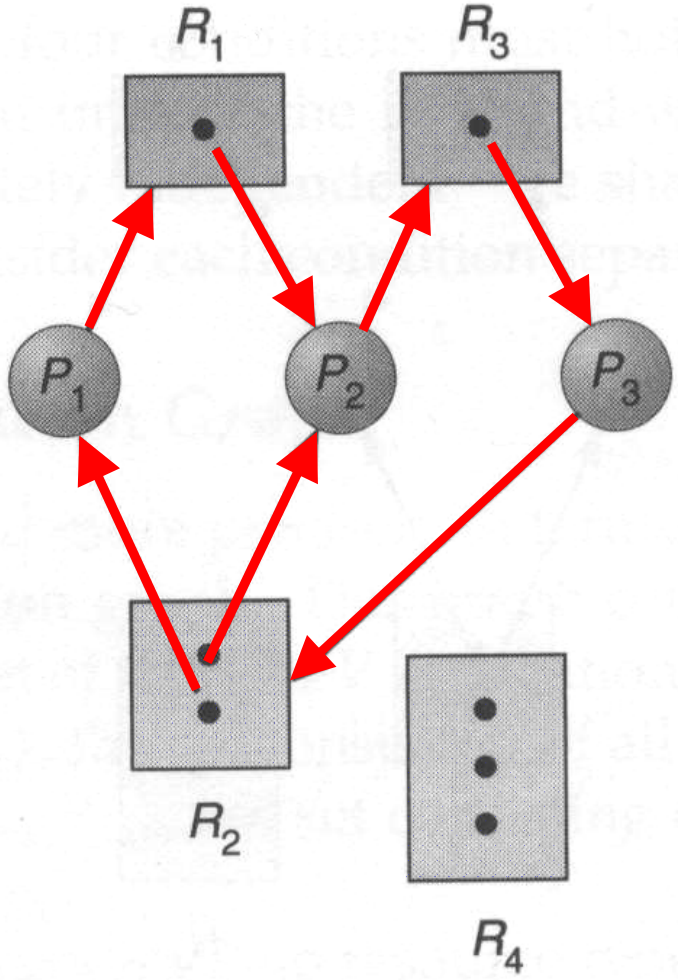
Deadlock and Starvation

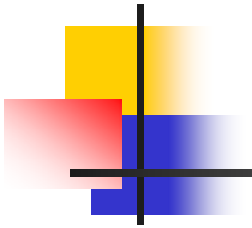
A Deadlock Situation

- The graph opposite is virtually identical to the previous graph, except for the inclusion of a request from P3 to acquire an instance of R2.
- As all instances of R2 have been allocated to P1 and P2, it follows that P3 is blocked and a circular list of dependencies arises as shown below

P1->R1->P2->R3->P3->R2

- That is, P1 is requesting R1 which has been given to P2 which in turn is requesting R3 which has been given to P3 which is requesting R2 which has been given to P1.
- The result of this is deadlock or stalemate.
- In fact there are two circular lists of dependencies in the graph opposite, can you see what they are?

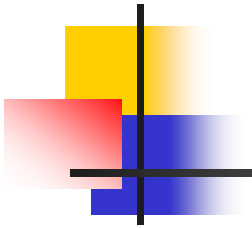




Dealing with Deadlock

- In general there are four strategies for dealing with Deadlock

1. Just ignore the problem altogether, i.e. do not bother to detect it or even try to correct it. Maybe if you ignore it, it will ignore you.
2. Anticipate that deadlock could arise and avoid it actually happening by carefully allocating the resources during process design (i.e. write your programs to avoid deadlock occurring)
3. Detection and Recovery. Acknowledge that deadlocks could occur outside of your control; thus detect them and take action to recover.
4. Prevent it happening by employing resource allocation algorithms at run-time.



Strategy 1 - Ignoring Deadlock: The Ostrich Algorithm

- The simplest approach to dealing with deadlock is to ignore it, i.e.

  "*bury your head in the sand like an Ostrich*".

- Different people react differently to this strategy. For example, if you are a Computer Scientist or Mathematician you would find this approach totally unacceptable and would probably explore all avenues in your attempt to come up with a solution regardless of cost and time.

- However, if you speak to an Engineer, they might take a more pragmatic approach. They might decide that if deadlocks are occurring only once every 3 months but the system is having to be rebooted due to crashes and bugs in operating system or application twice a week then it is not worth spending too much time, effort and money trying to solve the problem.

Deadlock and Starvation

Strategy 2 - Deadlock Avoidance: Solution 1 – Resource Allocation Ordering

- One simple solution to avoiding deadlock is to write all your process code such that it attempts to acquire multiple resources in exactly the same order, as shown below

Process A

```
while(1)
{
        ScannerMutex.Wait()
        CDMutex.Wait()
        …..
        …..
}
```

Process B

```
while(1)
{
        ScannerMutex.Wait()
        CDMutex.Wait()
        …..
        …..
}
```

- This will work in this particular situation because even if Process B is blocked waiting for the Scanner or CD Writer that has been granted to Process A, Process A will not be blocked and will eventually be able to release one or other of the resources that is blocking B allowing it to complete. That is, no circular queue of dependencies can exist.
- It is not a universal solution, since it requires a certain amount of 'insider' knowledge in that the designer of each process has to know about the existence of all other processes in the system using the same resources and make sure they agree on a common resource allocation order. This is practical only in a small embedded operating system application where the number of processes is fixed and known in advance. It would not work in a dynamic environment such as UNIX where new processes are written and run at will.

Deadlock and Starvation

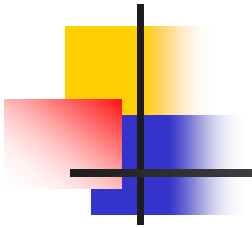
Strategy 2 - Deadlock Avoidance: Solution 2 – Voluntary Relinquishment of Resources

- An alternative solution to resource allocation ordering is to get a blocked process to voluntarily relinquish whatever resources that it has thus far acquired, if it fails to acquire all the resources it needs.
- For example if Process A requires resources r1 and r2 and, having successfully acquired r1 it then fails in its attempt to acquire r2 within a specified time period, it gives up r1 and tries to acquire both resources at some suitable time in the future.

Process A

```
while(1)    {                                          // try to acquire both resources
    ScannerMutex.Wait() ;                              // gain access to 1st resource
    if(CDWriter.Wait( 1000 ) = = WAIT_TIMEOUT)         // if cannot get 2nd resource within 1 Sec
    {
            ScannerMutex.Signal() ;                    // release 1st resource
            SLEEP(2000) ;                              // wait 2 seconds
            continue;                                  // try whole to acquire them again
    }
    …                                                  // got both resources so use them
}
```

- Such a scheme is frequently employed in today's networked environments where resources such as servers and web-pages may become temporarily unavailable. All the calls in rt.cpp have a timeout period associated with them which is set to infinity by default.
- However the problem with this solution is one of starvation.
- If the process gives up the resources it originally acquired, there is no guarantee it will ever get them back if another process acquires them in between.
- Thus it is possible that a process might never find **all** its resources free at the same time and **starve**.



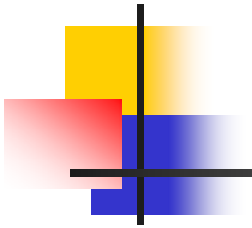
Deadlock and Starvation

Strategy 2 - Deadlock Avoidance: Solution 3 - Treat Multiple Resources as One

- A 3rd solution to avoiding deadlock involves treating all resources as one big single resource instead of several individual resources.
- In essence if a process wishes to acquire 'n' resources at the same time, then a single mutex is used to protect them all.

Problems with this approach

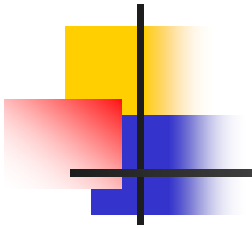
- The problem with this approach is that it once again requires co-operation between all the processes using any of the resources.
- Furthermore, it suffers in that other processes cannot acquire a sub-set of the resources.
- For example, to avoid deadlock using this scheme imagine two processes A and B that requires access to 5 resources r1-r5.
- If we create a single mutex to cover all 5 resources and allow A and B to perform Wait() and Signal() operations on that single mutex we avoid deadlock and starvation, but it means that a 3rd process C cannot acquire the individual resources such as r2 and r4 without upsetting processes A and B
- Furthermore, why should C have to acquire all five resources when it is only interested in using 2 of them?



Deadlock and Starvation

Strategy 3 - Deadlock Detection and Recovery

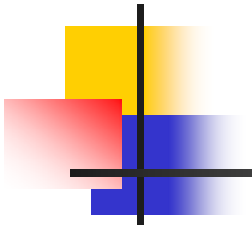
- This scheme requires that the host operating system maintain a directed graph recording processes and their requests for and allocation of resources.
- When a process requests a new resource (which it may or may not get), the operating system could subsequently inspect this graph and look for loops in the resource allocation and requests that would indicate a deadlocked system (as we did visually on Page 10)
- Having identified the deadlocked processes and the resources that are blocking them, the operating system could recover from the deadlock situation in one of three ways:
  - Recovery through Pre-emption
  - Recovery through Roll-Back
  - Recovery by Killing processes



Deadlock and Starvation

Strategy 3 - Deadlock Detection and Recovery through Pre-emption

- This solution is based on the idea that it may be possible to temporarily take back a resource from a process and give it to another process, thus breaking the circular dependency list in the graph.
- In some cases the operating system can do this automatically, in others, manual intervention might be required. For example, it would not be too tragic if a laser printer were taken away from a user, since they could collect their sheets of paper and come back for the rest later when the printer had been returned to them.
- Many GUI's (Graphical User Interfaces) can be pre-empted by the operating system. For example, if the user is currently interacting with a window, it is possible by clicking another window with the mouse for the other window to steal the 'input focus', in essence the resource the programs are fighting over is 'us', the human being interacting with it via the mouse and keyboard.
- Likewise important pop-up dialogue boxes can pre-empt an interactive window so the same technique could be applied to avoid deadlock. If this is the case then the pre-empted process will be put to sleep until its resources are returned.
- Another example is the Windows Offline files and folders. If the connection to the server is lost, then the client operating system uses a mirror of the real file structure so that it can continue to work off-line. When connection is re-established the real files are updated. Such a mirroring technique could be applied when it is necessary to take away a resource and update it later.
- In general this is not a good scheme as there is no guarantee the pre-empted process will return to the resource and find it in the same state as it was when it got taken away.



Deadlock and Starvation

Strategy 3 - Deadlock Detection and Recovery through Roll Back

- Here, a program can be written to anticipate that deadlock could occur and arrange for their program to issue checkpoints at periodic intervals, usually just before requests for resources are made.
- If a deadlock does arise and is can be detected by the Operating system, it can take back the resource the process is holding (thus breaking the circular dependency list) and the process can be 'rolled back' to a point in time when it did not have the crucial resource that was leading to deadlock. Once it had been rolled back, it could then try to acquire the resource again.
- This technique is frequently employed in database design systems and e-commerce applications, where transactions on several databases may need to be carried out simultaneously. If a transaction fails, (perhaps due to deadlock or other problems) then the transaction can be rolled back to a previous known state and retried later.
- Note however that rollback is quite a complex task and may involve undoing previous transactions that were successful (which can in turn could lead to deadlock as the system tries to re-acquire them).

Strategy 3 - Deadlock Detection and Recovery through Process Killing

- This is a brute force method whereby a process is killed thus releasing its resources.
- With a careful choice of 'victim' the other deadlocked processes may be able to continue.
- It is best to kill processes that can be re-started from the beginning with any adverse affects, such as a compiler.
- Killing a database or some e-commerce application is sure to end in tears !!!