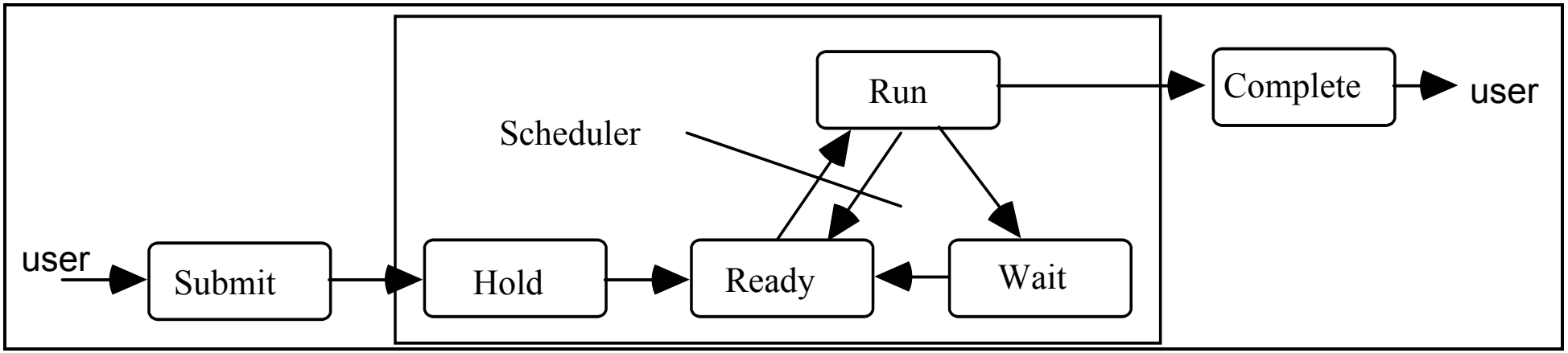
Deadlock

- **OS Scheduler determines which "ready" process runs and when it should run**
  - Programmer usually has no control over or visibility into the OS Scheduler



- **In a multiple-resource, multi-programmed system:**
  The interactions among processes and resources can cause problems
  As processes run, they request and (if granted by the OS) acquire resources
    Resources are shared among processes that are active in the system
    Conflicts can arise due to the sharing of finite resources.
    The most significant resource allocation problem is that of Deadlock

- **Deadlock is a Non-Deterministic Bug that can happen in Parallel Systems**
  Non-Deterministic bugs are extremely hard to test for and debug

- **Deadlock (Deadly Embrace, or Indefinite Postponement)**

  A system of resources can cause a system of processes to deadlock whenever any two or more processes are forced to wait (in a blocked state).

  It is possible that the waiting processes will never again become "ready" b/c the resources they have requested are held by other waiting procs.

  Deadlock can occur if a pattern of waiting processes is established

  Example:

  P2 is forced to wait for P1.

  P3 is forced to wait for P2.

  P1 is forced to wait for P3.

  In effect, P1 ends up waiting for itself.

  Since P1 is blocked, it cannot execute and remove itself from waiting

  That is, a cycle is established:

  P1 <- P2 <- P3 <- P1

  Another example:

  Assume a system with four tape drives and two processes.

  If each process has 2 tape drives and needs a third one in order to proceed, then each will wait for the other.

- **Deadlock scenarios are not unique to computing systems**

  Consider 2 people crossing a river from opposite sides

  At most, one foot can be on each stepping stone at a time

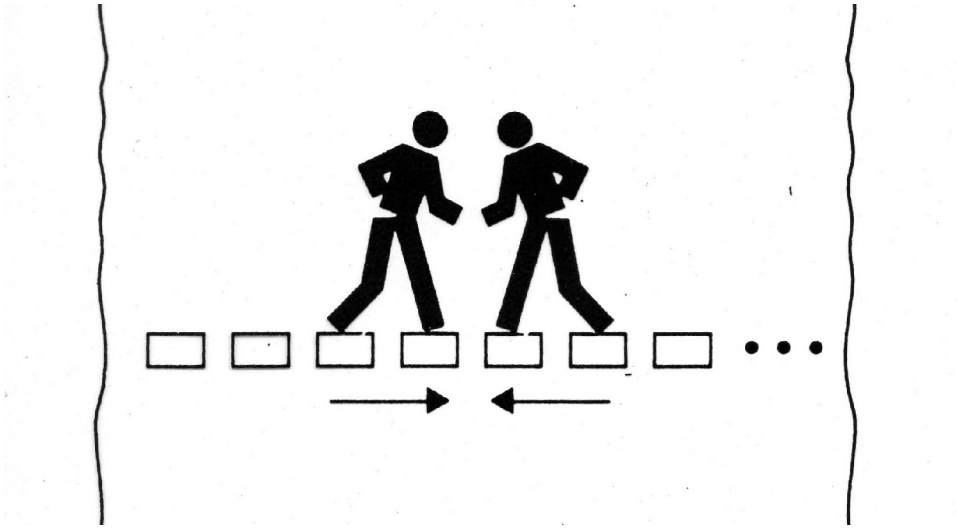
  To cross the river, a person must use each of the stepping stones

  Analogy: Each person crossing the river is a process, the stones are resources, and the act of stepping onto a stone mutually exclusively acquires that resource for the process.

  Deadlock can occur if:

  Person starts to cross river without first checking to see if someone else is trying to cross from the other side in the opposite direction.

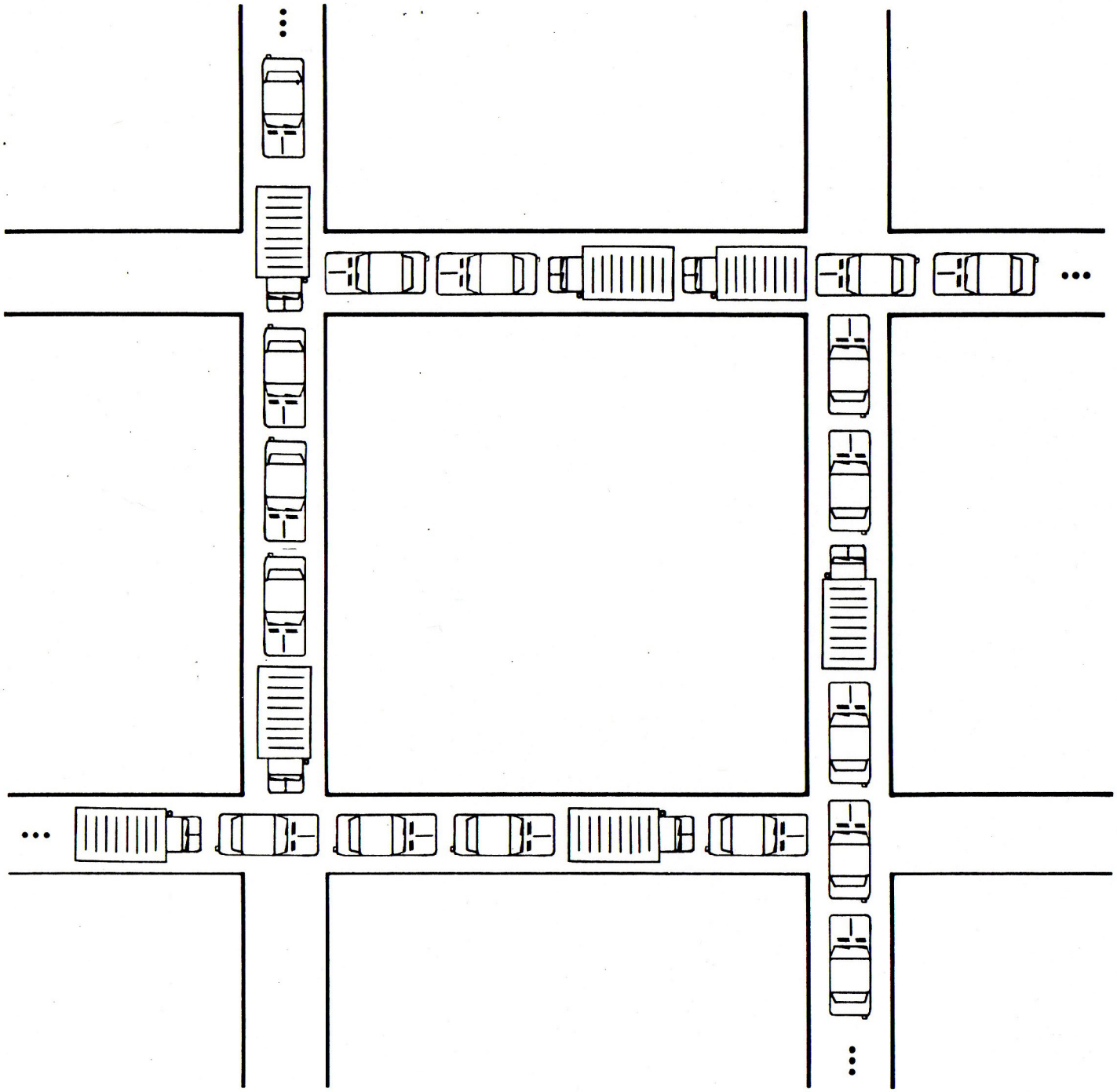
  Two people start crossing river from opposite sides and meet in the middle.



- **Avoiding deadlock requires each person crossing river to follow a protocol.**
  One such protocol might be to see if anyone is crossing from other side.
    If so, wait until they are done. Otherwise, it is safe to cross.
    Must also handle situation where 2 people want to cross at "same" time.
      If both go, deadlock occurs. (Both stuck in the middle of the river)
      If both wait, deadlock also occurs. (Both wait forever)
    One solution is to give one side of the river higher priority to break the tie.
      But, now given infinite supply from one side, deadlock can still occur.
      Modified solution: Periodically alternate the direction of crossing

- **A system has a finite number of resources to be shared b/w competing procs.**
  The sequence of events required for a process to use a resource is:
    1) Request the resource.
    2) Use the resource.
    3) Release the resource.
  If resource is not available when requested, requesting process must wait
  In a deadlock, processes never finish executing and never release resources
  This, in turn, prevents other jobs from obtaining their requested resources
  Eventually, no process can go, all work stops, and the system "hangs"

- **Another real-world, non-computer example of Deadlock: Traffic Gridlock**



- **Four Conditions must exist for deadlock to occur**

1) Mutual Exclusion Condition.

Only one process at a time can use a particular resource.

If another process requests that same resource, the process must wait until the resource is released by the first process.

2) Hold and Wait Condition.

Processes holding resources granted earlier can request new resources.

If process must wait for a new additional resource, it continues to hold onto the resources it currently has.

3) No Preemption Condition.

Resources previously granted to a process cannot be forcibly taken away from that process.

Resources can only be released voluntarily by the processes that are holding them.

4) Circular Wait Condition.

There must be a circular chain of two or more processes.

Each is waiting for a resource held by the next member of the chain.

- **Resource Graphs (A graphical method for Deadlock Detection)**

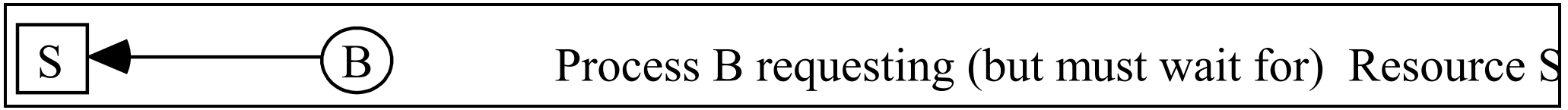
  Holt in 1972 modeled the four conditions for deadlock using directed graphs.

  Square nodes are used to represent resources.

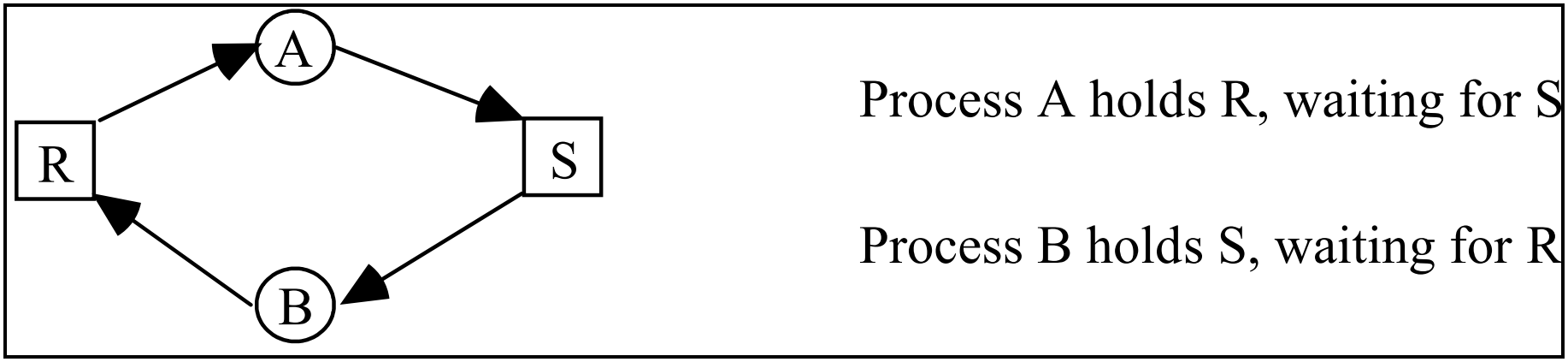
  Round nodes are used to represent processes.

  Arc from a resource to a process means that the resource is currently held (mutually exclusively) owned by that process.

  Arc from a process to resource means that the requesting process is currently blocked and needs to wait for that resource.



  If any cycles occur in the graph, then a deadlock situation exists.

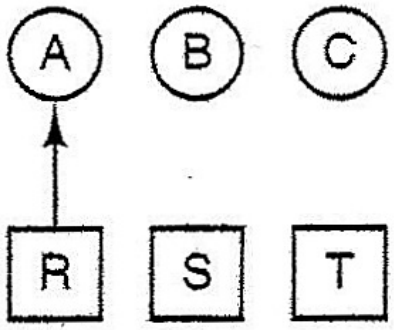


- **Example of Resource Graph Usage and Impact of Scheduling on Deadlock**

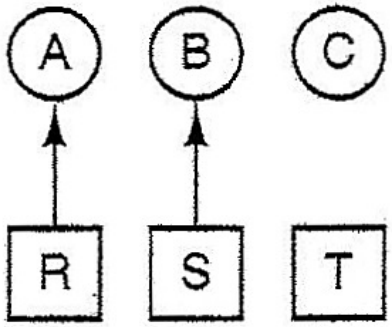
Assume a System of Three Processes (A, B, C) & Three Resources (R, S, T)

Process A	***Process B***	***Process C***
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

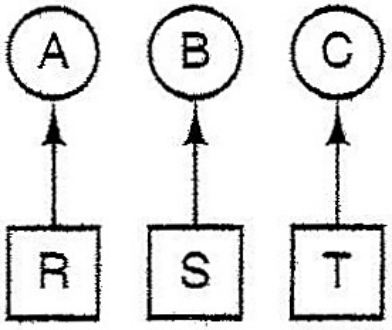


(d)

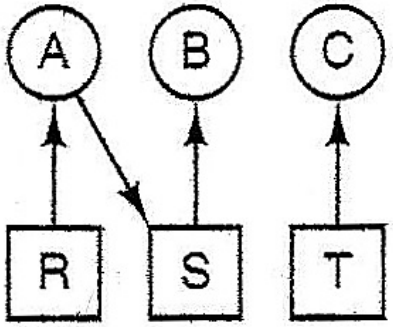


(e)

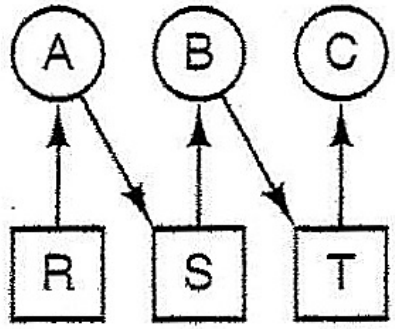
(f)



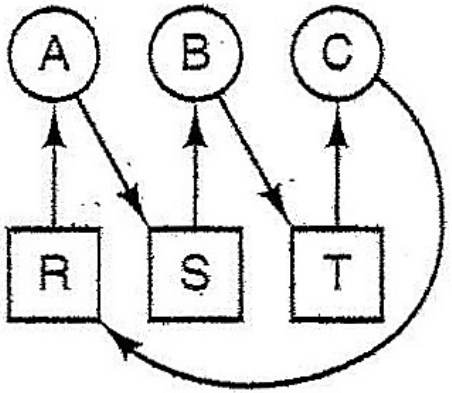
(g)



(h)



(i)



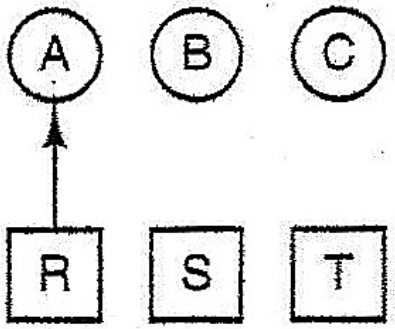
- **Same System of Three Processes (A, B, C) & Three Resources (R, S, T)**
  - **But different ordering (Schedule)**

Process A	***Process B***	***Process C***
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

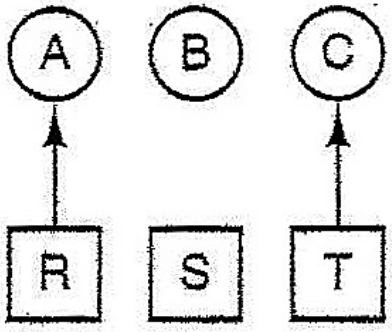
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

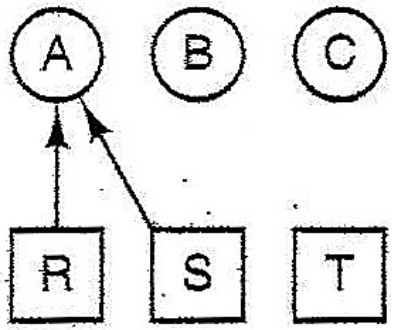
(k)



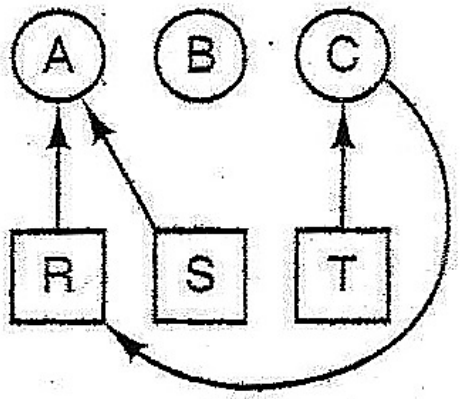
(l)



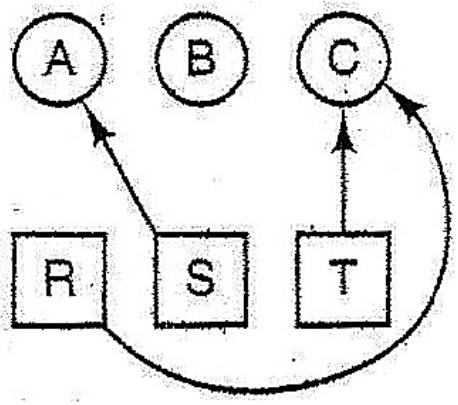
(m)



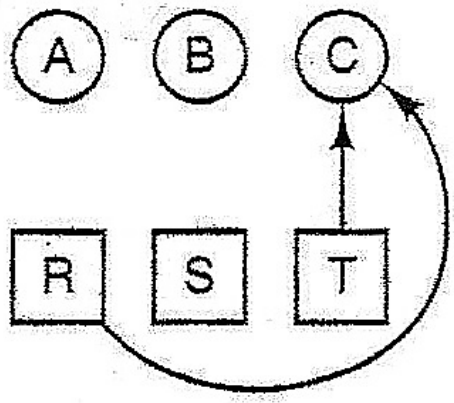
(n)



(o)



(p)



(q)

- **Prevention by negating one of the four necessary conditions for deadlock:**

  Approach: Examine each of the four necessary conditions for deadlock.

  Try to prevent at least one of the conditions from ever happening.

  - Prevention of Condition 1 (Mutual Exclusion)

  Difficult to prevent since some resources are inherently non-sharable.

  e.g.) Card Reader, Writeable file

  Not practical to try to eliminate this condition.

  Mutual exclusion limitation must remain for some resources.

  - Prevention of Condition 3 (No Preemption)

  Require a process that is refused its request to free all of its resources.

  That is, all the resources it has are preempted.

  It would then need to request them again later.

  Preemption cost of most resources would be too high and impractical.

  (e.g. How can you take a line printer away from a job that's printing ?)

  CPU preemption has relatively low overhead cost, So:

  Task switching the CPU is reasonable (and done in real life).

  But, Task switching other resources is generally not feasible.

- Prevention of Condition 2 (Hold and Wait)

  Can be prevented by using Preallocation.

  Force all processes to request all resources initially, all at once.

  Process cannot begin execution until all resource requests are granted.

    Advantage:

      Easy to implement.

    Problems:

      Substantial cost in terms of resource utilization efficiency & sharing

      Each process must wait until all of its resources are available.

        (Even if a resource is not needed until very late in its execution)

      Wasteful to commit resource if likelihood it will be needed is small.

        Resources allocated may not even be used.

        e.g.) Printer for core dumping (May not even arise if no errors).

      Resources used only for short periods of time must be allocated

        (mutually exclusively) to the process for the job's entire life.

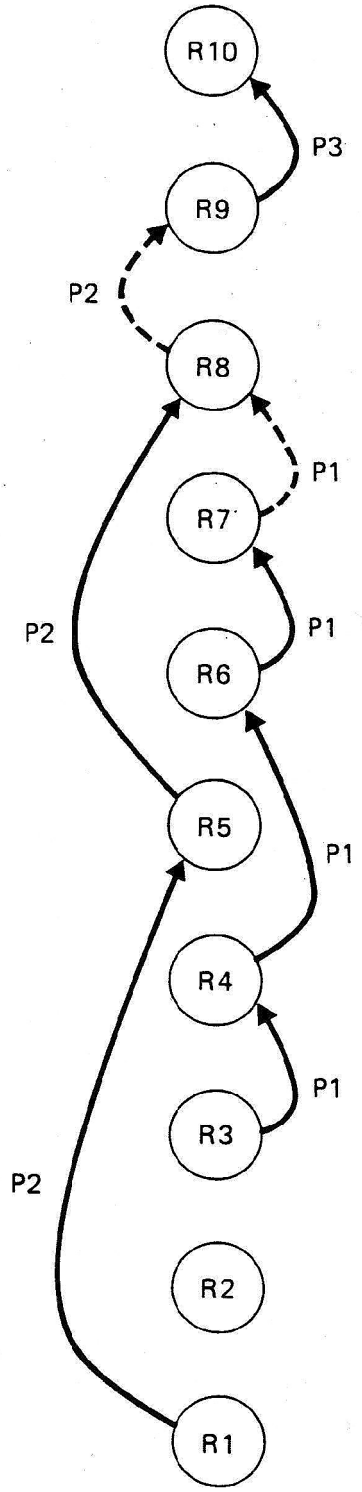
      Resource is therefore inaccessible for long periods of time,

        effectively making it not shareable by other processes.

      Difficult to formulate and predict full (and contingent) resource

        requirements before execution starts.

- Prevention of Condition 4 (Circular Wait)
  - Cycles can be avoided by imposing an order on resource types.
  - This Method is called Standard Allocation Pattern or Hierarchic Allocation
    - Resources are assigned a level (order) in a hierarchy.
    - All resource allocation requests must be made in ascending order.
    - If a process has resource k, it can only request resources at level > k
    - Advantages:
      - Less restrictive than full preallocation.
        - Doesn't require all resources at once; just in a certain order.
      - Does not require knowledge about maximum needs in advance.
    - Problems:
      - More costly to implement than full preallocation.
      - Constraint is imposed on the natural order of resource requests.
        - Imposes a burden on the application programmers.
      - Order in which process needs resources could differ from hierarchy
        - e.g.) Process needs plotter early, tape drive at end of execution.
          - But tape is at lower order in hierarchy than plotter.
          - So tape is allocated early and remains idle until end.
      - Hard to find an ordering that satisfies all processes & programmers



Process P1 has resources R3, R4, R6 and R7, and is requesting resource R8 (as indicated by the dotted line). No circular wait can develop because all arrows must point upward.

- **Resource Trajectories: A graphical method of visualizing Deadlock**

  Statements 1 - 4 correspond to instructions of P2; Statements 5 - 8 to P1.

  Vertical Y axis shows progress of P1; horizontal X axis progress of P2.

  On a uniprocessor, trajectory is limited to horizontal and vertical directions

  Instantaneous status of a computation is represented by a point in the plane.

  e.g.) Point W shows that:

  P2 has started but not yet reached statement 1 and

  P1 has passed statement 6 but not yet reached 7.

  Path through plane represents execution order of statements in P1 and P2.

  e.g.) Trajectory A shows the sequence of statements 5, 6, 7, 8, 1, 2, 3, 4.

  That is, P1 is run for four statements (5 - 8), then P2 run for four (1 - 4)

  Unsafe regions in trajectory can be identified by overlapping resource needs

  Assume: P1 requests R1 at 6 and releases it at 7.

  P2 requests R1 at 1 and releases it at 3.

  Shaded rectangle bounded by 6, 7, 1, 3 cannot be crossed in trajectory.

  P1 & P2 cannot both have R1 at same time due to mutual exclusion on R1.

Also assume: P1 requests R2 at 5 and releases it at 8.

P2 requests R2 at 2 and releases it at 4.

Therefore, shaded region bounded by 5, 8, 2, 4 also cannot be crossed.

P1 & P2 cannot both have R2 at same time due to mutual exclusion on R2.

Trajectory B shows a safe execution order of 1, 2, 5, 3, 4, 6, 7, 8.

When P1 requests R2 at 5 (point X), it is unavailable and P1 must wait for it.

P1 cannot cross 5 until it acquires and can use R2.

P2 continues to execute past stmt 4, releases R2 allowing P1 to continue.

Trajectory C shows an unsafe execution path for statement order 5, 1, 6, 2.

P1 becomes blocked at 6 (point Y) since it must wait for R1 to become avail.

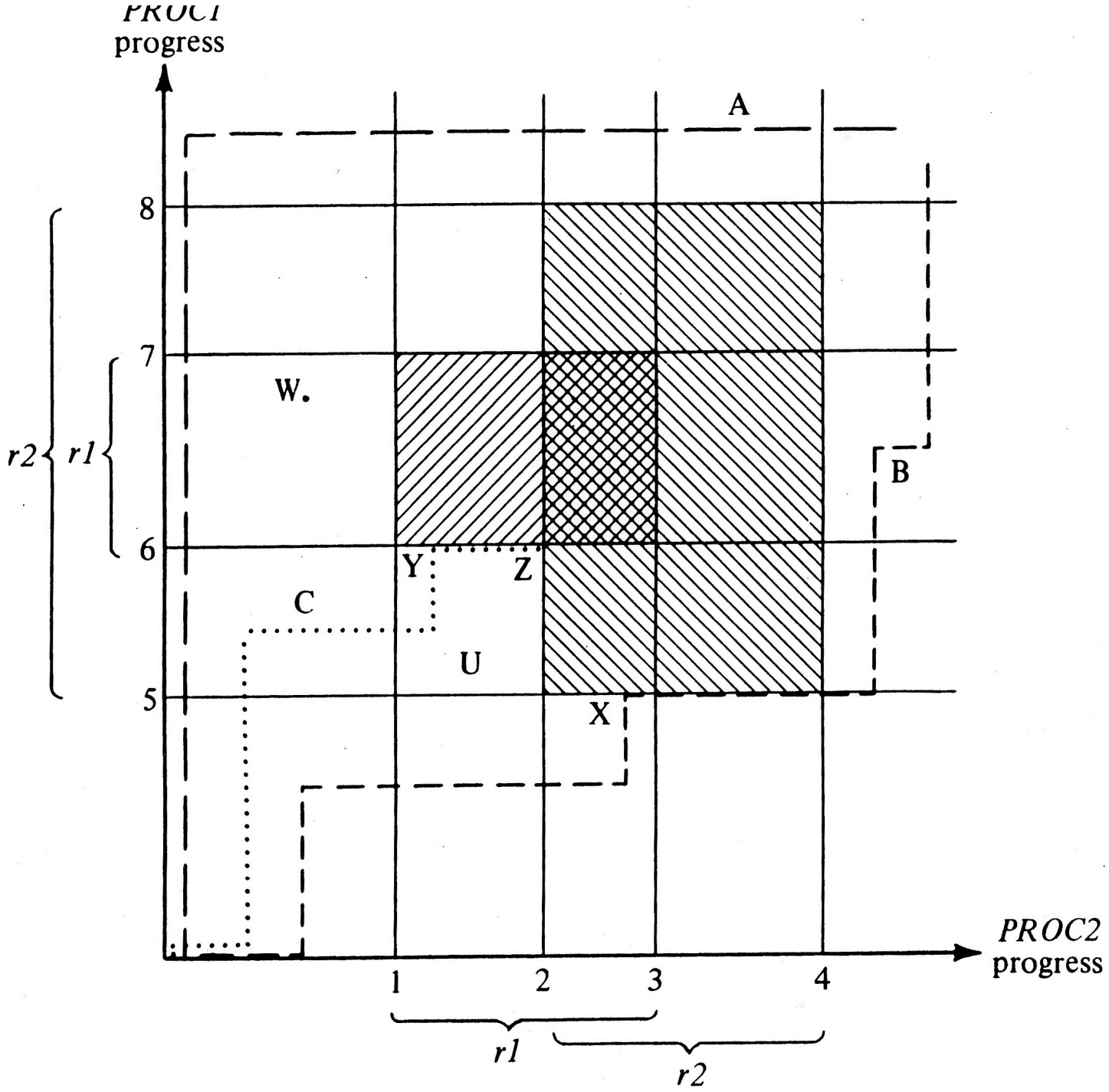
P2 becomes blocked at 2 (point Z) since it must wait for R2 held by P1 after 5

Both processes are now deadlocked:

P1 is waiting for P2 to reach statement 3 and P2 waiting for P1 to finish 8.

Some states may appear to be “OK” because processes are still running, but

Deadlock is inevitable once computation enters rectangle U, an unsafe state.


PROC1
progress
A
8
7
W.
r2
r1
B
6
Y
Z
C
U
5
X
PROC2
progress
1
2
3
4
r1
r2