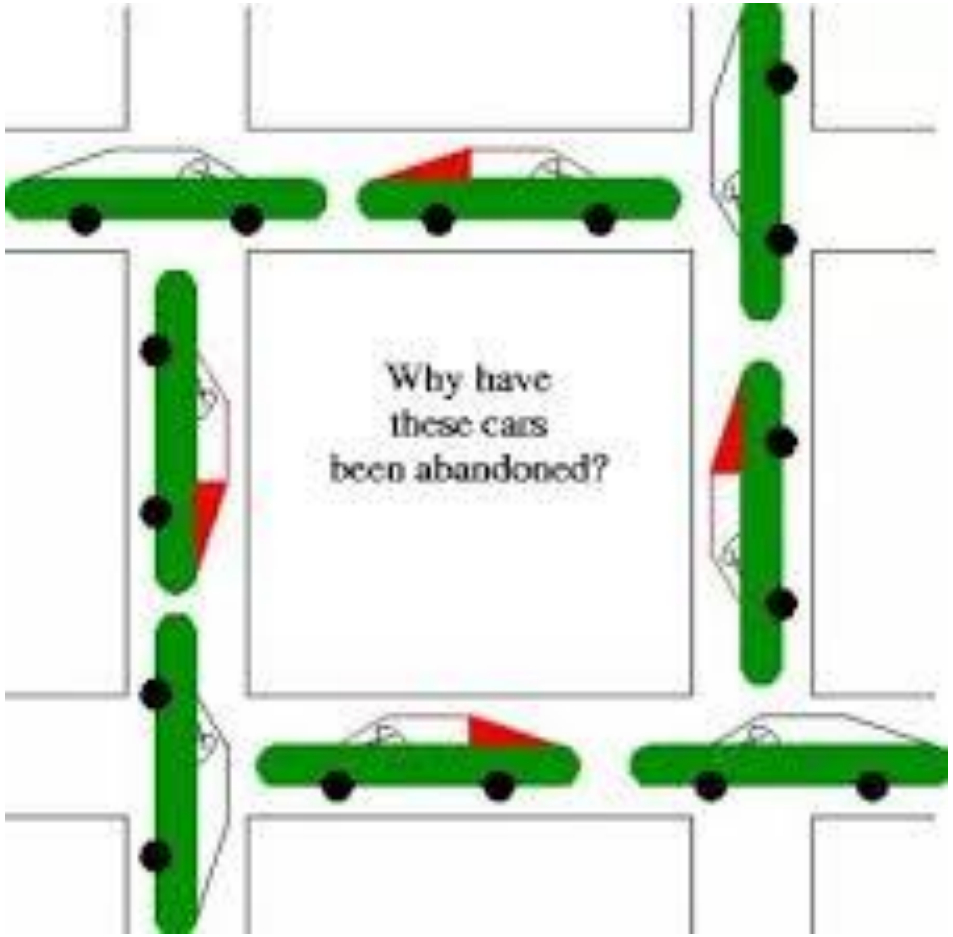
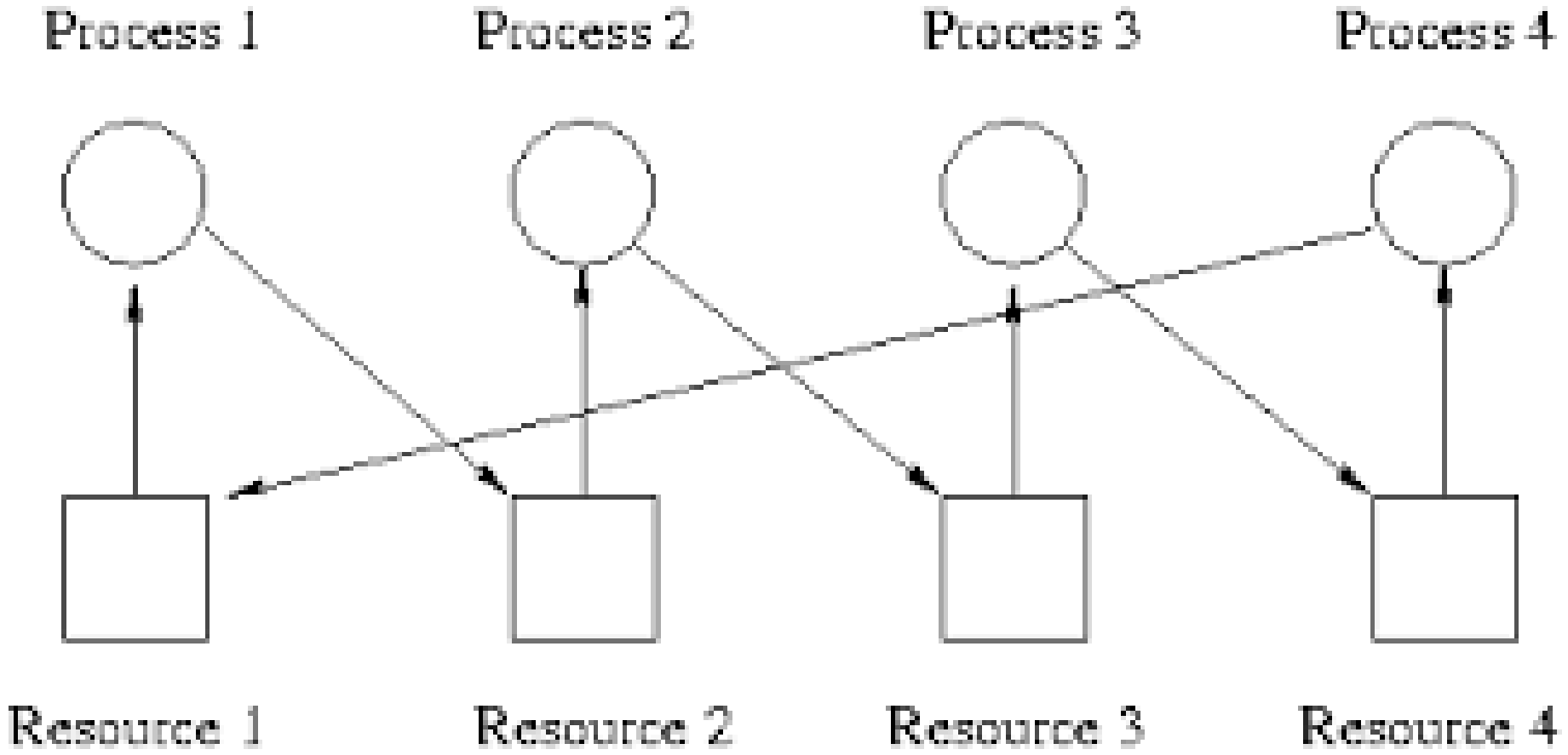
Deadlock: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.



Resource Deadlocks: most common deadlocks,

P_1 has R_1 and requests R_2, P_2 has R_2 and requests R_3,
P_3 has R_3 and requests R_4, P_4 has R_4 and requests R_1

Preemptable resource: can be taken from the process without damage



Conditions for resource deadlocks
Necessary & Sufficient Conditions

- 1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
- 2. Hold-and-wait condition. Processes currently holding resources that were granted earlier can request new resources.
- 3. No-preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
- 4. Circular wait condition. There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain (Directed Graph with a *Cycle*).

Dealing with Deadlocks

- 1. Just ignore the problem. Maybe if you ignore it, it will ignore you.

  The Ostrich Algorithm.
- 2. Detection and recovery. Let them occur, detect them, and take action.
- 3. Dynamic avoidance by careful resource allocation.
- 4. Prevention, by structurally negating one of the four conditions.

Detection and Recovery

- ***Detection:*** Keep a Directed Graph for the processes and the resources.
- Algorithm for detecting a Cycle in the processes Graph.
  - 1. For each node, *N*, in the graph, perform the following five steps with *N* as the starting node.
  - 2. Initialize *L* to the empty list, and designate all the arcs as unmarked.
  - 3. Add the current node to the end of *L* and check to see if the node now appears in *L* two times. If it does, the graph contains a cycle (listed in *L*) and the algorithm terminates.
  - 4. From the given node, see if there are any *unmarked* outgoing arcs. If so, go to step 5; if not, go to step 6.
  - 5. Pick an unmarked outgoing arc at random and *mark* it. Then follow it to the new current node and go to step 3.
  - 6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end (does not lead any further). Remove it and go back to the previous node, that is, the one that was current just before this one, take that one the current node, and go to step 3.

Recovery from Deadlock

- Recovery through ***Preemption***. Take away a resource from a process and when the others have proceeded (finished) give it back.
- ***Rollback***
  - Keep the history of each process at ***checkpoints*** (regular time period): variable values, commands, etc.
  - Start again from the previous checkpoint. The OS is NOT deterministic and processes are expected to run differently that before
- ***Killing Processes***
  - Rerun when possible
  - In cases with no ill effects, e.g. recompilation.
  - Damaging for processes updating resources, e.g. updating database files twice.

Deadlock Avoidance

- Keep data structures that present the system's status.
- Each time there is a process request for using a resource: run an algorithm to check whether it is ***safe*** to give the resource.
- **Banker's** Algorithm data structures:
  - ***Available:*** 1-D Array of ***m*** entries***;*** Available(i)=k means that the system has available to use k (it can grant these to the processes) instances of the resource type i.
  - ***Max:*** 2-D Array ***m x n*** entries; Max(i,j)=k means that the process j will use up to k instances of the resource type i.
  - ***Allocation:*** 2-D Array ***m x n*** entries; Allocation(i,j)=k means that the process j *currently* uses k instances of the resource type I (they were given by the system).
  - ***Need:*** 2-D Array ***m x n*** entries; Need(i,j)=k means that the process j, in order to complete its execution, will need k instances of the resource type i , which have to be granted by the OS.

The Banker's Algorithm

- Process j: ***Request***(i)=k a 1-D array conveying that process **j** asks for **k** instances of the resource **j** to continue its execution.
- The Algorithm examines the following:
    - 1. If Request(i) ≤ Need(i,j) then goto 2 else error (for each i requested by process j).
    - 2. f Request(i)≤Available(i) then goto 3 else wait (for each i requested by process j).
    - 3. If the above checks OK: the algorithm assumes that provides the resources to process j and computes the new status (the data structures):
        - Available(i)=Available(i)-Request(i);
        - Allocation(i,j)=Allocation(i,j)+Request(i);
        - Need(i,j)=Need(i,j)-Request(i);

The Banker's executes the *Safety* subroutine

- **Safety**

1. For j=1 to n do Finish(j)=false;
2. Find j such that
   - Finish(j)=false, and
   - Need(i,j) ≤ Available(i) for all i.

   If there is no such j then goto 4.
3. Execute:
   - Available(i)=Available(i)+Allocation(i,j);
   - Finish(j)=true;
   - Goto step 2.
4. If Finish(j)=true for all j then the system is in a SAFE state.

- Complexity: N^2
- Iff the state is SAFE the OS grants the process request.

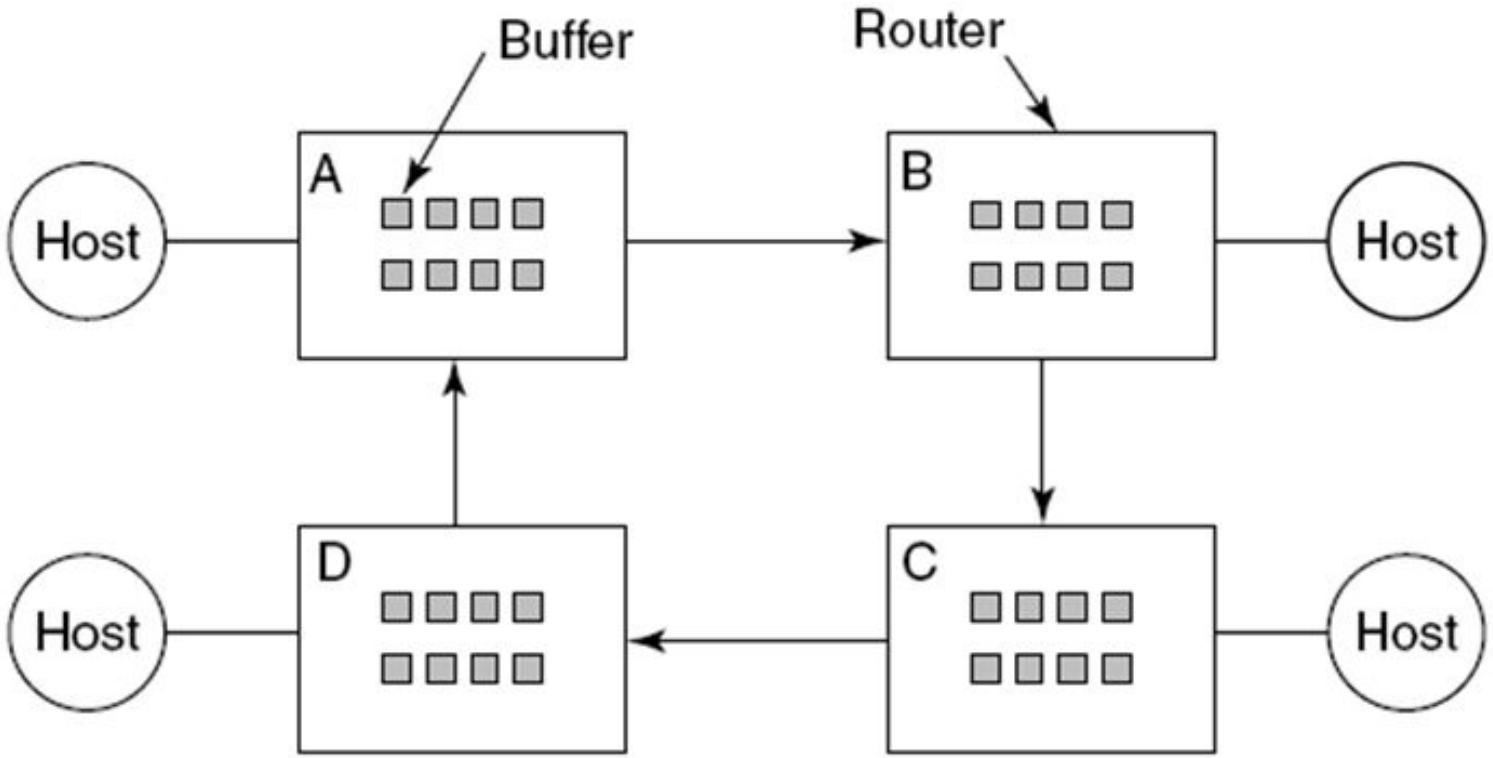
Deadlock Prevention

Attacking one of the four necessary & sufficient conditions

- Mutual Exclusion: by using SPOOL (System Peripheral Operation Off-Line).
- Hold & Wait: request all the resources initially.
- No preemption: take the resources away
- Circular wait: order the resources by numbers.

Avoidance and prevention are not widely used in OS, but have special purpose applications.

Communication Deadlocks



- In a communication system a node is either computer or Router having finite buffers (8 in example)
- If no buffer available we have a resource deadlock in a network.