University of Nottingham
UK | CHINA | MALAYSIA

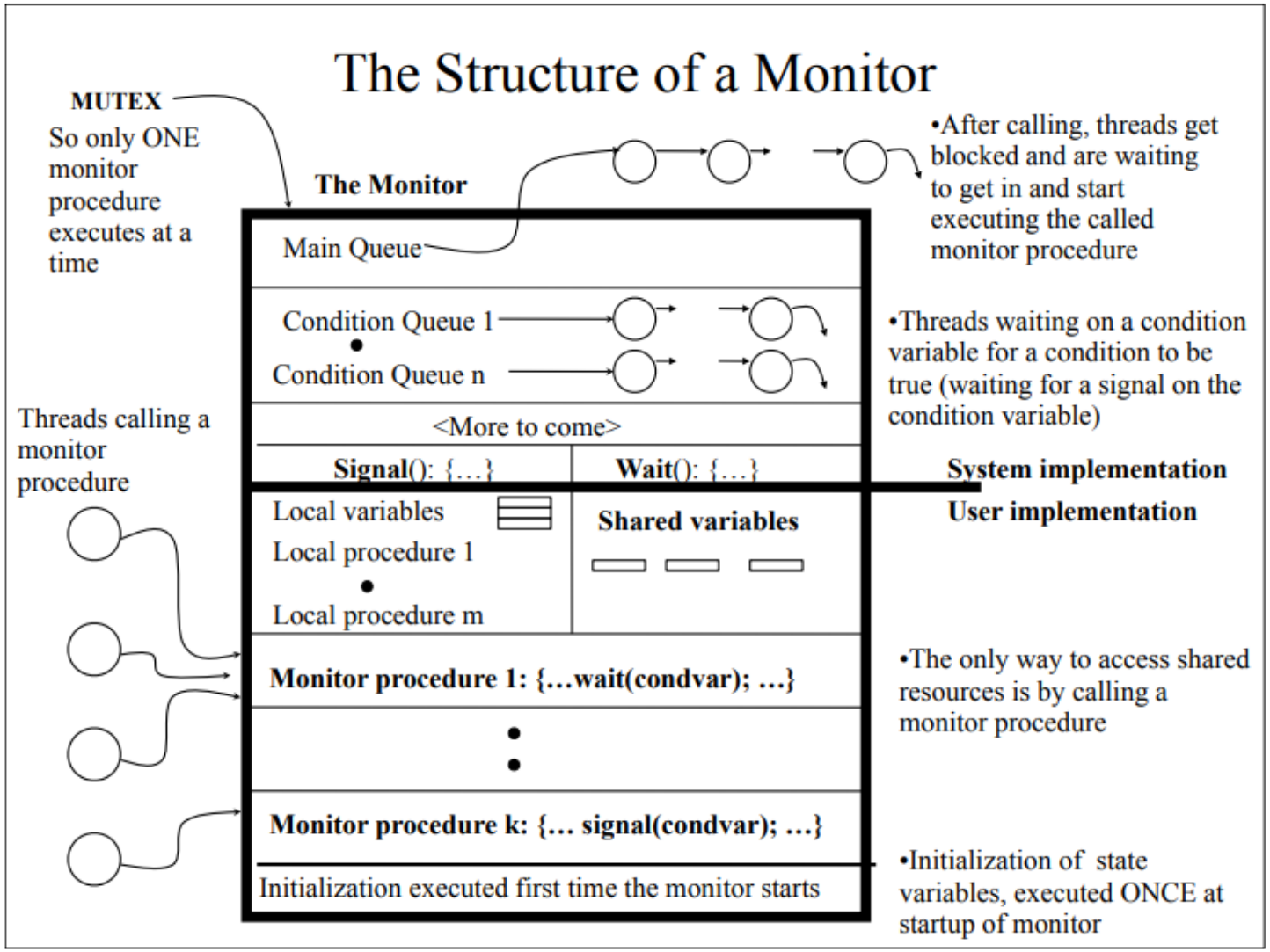
Operating Systems and Concurrency

Lecture 13: Deadlocks

University of Nottingham,
Ningbo China, 2024

Recap
Last Lecture

Monitor



Overview
Today class

- Resource
- Definition of deadlocks
- Deadlock occurrence
  - Minimum conditions
- Approaches to dealing with deadlocks
  - Just ignore the problem
  - Deadlock detection and recovery approaches

Deadlocks
Resources

- In operating System(OS), a resource is anything that must be acquired, used, and released over the course of time.
  - e.g. a device, a data record, file, semaphore

- Can be two types: Preemptable & non-preemtable
  - A **preemptable**, i.e., it can be forcefully taken away from the process without permanent adverse effect.
    - CPU cycles: The operating system can interrupt a process and reassign CPU cycles to another process without affecting the original process's state.
    - Memory on a PC with Swapping: On a system with memory swapping, memory can be taken from a process and stored on disk, then reallocated to another process. When needed, the original memory can be restored.

Deadlocks
Resources

- A **non-preemptable**, i.e., it cannot be taken away from a process without permanent adverse effect. E.g.
  - If a process start burn on a Blu-Ray, suddenly taking the Blu-ray record away from it and giving to another process will result in a garbled Blu-ray.
  - File Locks: If a process locks a file for exclusive access, the lock cannot be taken by another process until the initial process releases it.
  - Hardware Devices: If a printer is assigned to a process, it cannot simply be preempted and reassigned to another process without disrupting the operation.
  - Memory on Non-Swapping Systems (e.g., Some Smartphones): In systems without memory swapping, memory cannot be reclaimed easily without terminating the process.

- *Note: Deadlocks only occur for non-preemtable resources since preemptable resources can be temporarily taken away to recover from the deadlock.*

- *Thus, our discussion will focus on **non-preemtable resources**.*

Deadlocks

Resource: Sequences of event required to use a resource

1. Request the Resource (R):
    - The process requests access to resource R.
    - If R is available, it is allocated to the process immediately.
    - If R is unavailable (because another process holds it), there are two main ways the system can handle this:
        - Blocking: In many OS, the process is automatically placed in a blocked or waiting state until the resource becomes available. The OS handles the wait and then "wakes up" the process when R is released.
        - Error Code: In other systems, the resource request might fail immediately with an error code (e.g., EAGAIN or EWOULDBLOCK in POSIX systems). In this case, the process must handle the wait itself, often by implementing a loop that periodically retries the request.

2. Use the Resource: Once the resource is acquired, the process proceeds to use it.

3. Release the Resource: After using the resource, the process releases it, making it available for other waiting processes.

Deadlocks
Definition

- **Deadlock** in concurrent systems occurs when a group of processes is in a state where none of them can proceed because each one is waiting for a resource held by another process in the same group.

- In this state, the processes are effectively "stuck" since none can complete their tasks or release resources, causing a system halt for those processes.

Deadlocks
How do They Occur?

- On occasions, **multiple processes** will **require access** to **multiple mutually exclusive resources**

- Consider the following **sequence of events** in a **multi-programmed system**:

```
Process A:                  Process B:
Request resource X          ...
Acquire resource X          ...
...                         Request resource Y
...                         Acquire resource Y
Request resource Y          ...
...                         Request resource X
                            ...
```

Process A and B request the resources in opposite orders and end up in deadlock.

Deadlocks
Minimum Conditions

- Coffman et al. (1971) showed that **Four conditions** must hold for deadlocks to occur :
  1. **Mutual exclusion**: Each resource is either assigned to exactly one process or is available.
     - If mutual exclusion does not hold (resources are shareable), deadlock cannot occur, because processes are not restricted from accessing resources due to exclusive access.

  2. **Hold and wait condition**: A process currently holding at least one resource can request additional resources.
     - If the hold and wait condition does not hold, deadlock cannot occur because processes are not holding resources while waiting for others.

Deadlocks
Minimum Conditions

3. No preemption: Resources cannot be forcibly taken away from a process.

- If no preemption does not hold (resources can be preempted), deadlock cannot occur because resources can be reallocated as needed, avoiding indefinite waiting.

4. Circular wait: There exists a circular chain of two or more processes where each process is waiting for a resource held by the next process in the chain.

- If circular wait does **not hold** (e.g., resource ordering prevents cycles), deadlock cannot occur because there will be no circular dependency among processes.

- All four of these conditions must be present for a deadlock to occur. If one them is absent, no deadlock is possible.

Deadlocks

Deadlock Modeling

- Deadlocks can be modelled using **directed graphs(resource-allocation graph)**:
  - **Resources** are represented by **squares** and **processes** are represented by **circles**
    - A directed arc from a **square** (resource) to **a circle** (process) means that the resource was **requested and granted** (fig a), i.e. is allocated to the process.
    - A directed arc from a **circle** (process) to a **square** (resource) indicates that the process has requested the resource and is **waiting to obtain it** (fig b).
    - A **cycle** in the graph means that a **deadlock** occurs for the respective resources and processes (fig c)

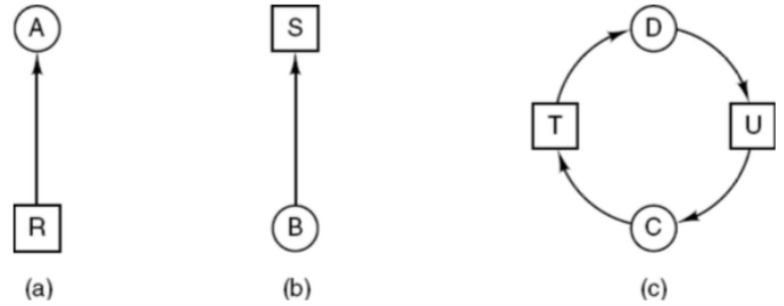


Figure: Modelling of Deadlocks (Tanenbaum)

Dealing with Deadlocks

Approaches to deal with deadlocks

1. Just ignore the problem: Ostrich algorithm (hide your self in to the sand and avoid the danger)
2. Detection and recovery: let it occur, detect them and take action (cut the graph)
3. Prevention, by structurally negating one of the four conditions
4. Dynamic avoidance by careful resource allocation.

Dealing with Deadlocks
Just ignore it: Ostrich Algorithm

- Stick your head in the sand and pretend there is no problem at all, this method of solving any problem is called Ostrich Algorithm.
- But the Engineers that deal with the system believe that deadlock prevention should be paid less attention as there are very less chances for deadlock occurrence.
- Mathematician find it unacceptable and say that deadlock must be prevented at all cost.
- Used by most operating systems, including UNIX.

Dealing with Deadlocks

Detection and Recovery

- **Detect** when the system **is deadlocked** and recover from them (i.e., **no deadlock prevention**)
  - Allow system to enter deadlock state=>Detection algorithm=>Recovery scheme
- Techniques to detect deadlock
  - Single Instance of Each Resource Type
    - Resource allocation graph
  - Several Instances of a Resource Type
    - Matrix Approach

Dealing with Deadlocks

Detection and Recovery: Single Instance of Each Resource Type

- Let's begin with simplest case. There is only **Single resource** exists of each type.
  - E.g Such a system might have one scanner, one Blu-ray recorder, one plotter and tape driver.
- A graph approach can be used for such a system.
- If the graph contains one or more cycles, a **deadlock** exists. Any process that is part of a cycle is deadlocked.
- If no cycles exist, the system is not deadlocked.

Dealing with Deadlocks

Detection and Recovery: Resource allocation graph

- Consider the example below with seven processes (A – G) and six resources (R – W):
  1. Process A holds R and wants S.
  2. Process B holds nothing but wants T.
  3. Process C holds nothing but wants S.
  4. Process D holds U and wants S and T.
  5. Process E holds T and wants V.
  6. Process F holds W and wants S.
  7. Process G holds V and wants U.

The question is: "Is this system deadlocked, and if so, which processes are involved?

- Let's Construct a resource graph

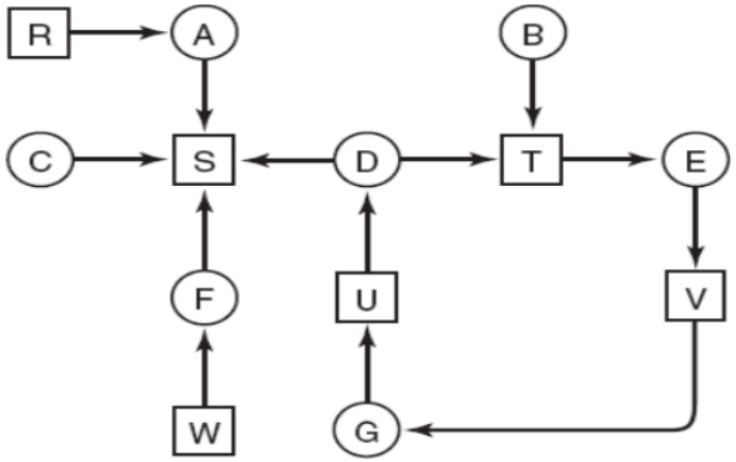


Figure: Detecting deadlocks (Tanenbaum)

- Process D, E, and G are all deadlock. Process A, C, and F are not deadlocked because S can be allocated to any one of them, which then finishes and returns it.

Dealing with Deadlocks

Graph Approach (cycle detection algorithm)

- Although it is relatively simple to pick out the deadlocked processes by visual inspection from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks.

- The algorithm is not optimal!

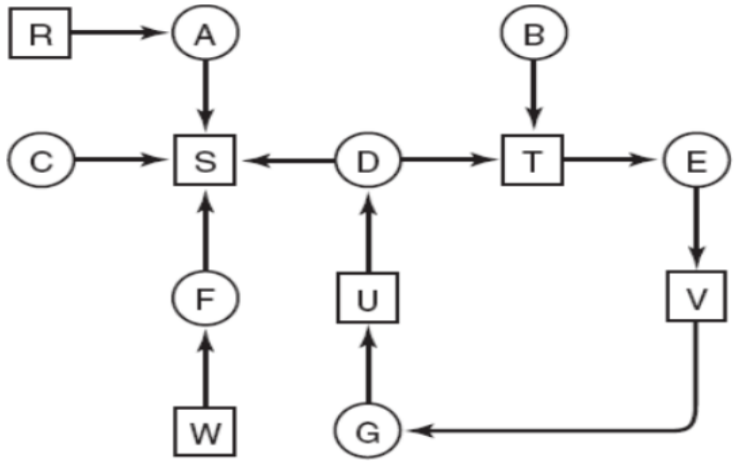


Figure: Detecting deadlocks (Tanenbaum)

For each node, N, in the graph, perform the following five steps with N as the starting node.

1. Initialize L to the empty list, and designate all the arcs as unmarked.
2. 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.
3. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 4; if not, go to step 5.
4. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 2.
5. 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. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 2.

Dealing with Deadlocks

Graph Approach (cycle detection algorithm)

- **Starting at Node R**:
  - Initialize L=[]
  - Add R to L, so L=[R]
  - Move to **node A** (if we follow the arc from R), and add it to L, making L=[R,A].
  - Move to **node S**, so L=[R,A,S]
  - Since S has no unmarked outgoing arcs, we reach a **dead end**.
  - Backtrack to A, remove S from L, making L=[R,A].
  - Since A also has no unmarked outgoing arcs, backtrack again to R.
  - R has no other outgoing arcs to explore, so we end the traversal for R without finding a cycle.

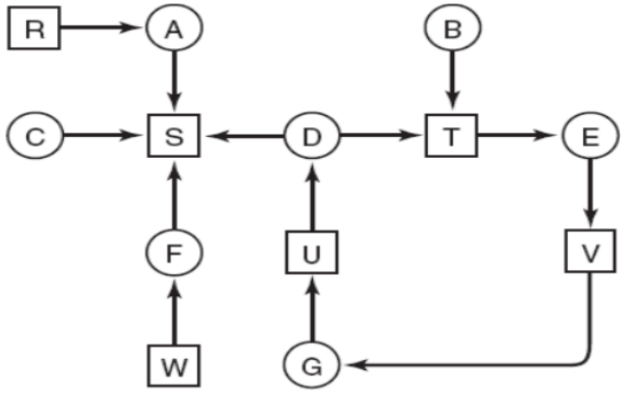


Figure: Detecting deadlocks (Tanenbaum)

Dealing with Deadlocks

Graph Approach (cycle detection algorithm)

- **Starting at A**
  - Initialize L=[].
  - This search also quickly stops, as A was already fully explored in the previous step with no cycle detected.

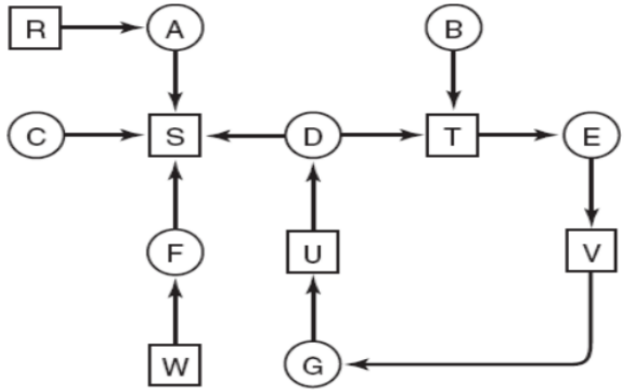


Figure: Detecting deadlocks (Tanenbaum)

Dealing with Deadlocks

Graph Approach (cycle detection algorithm)

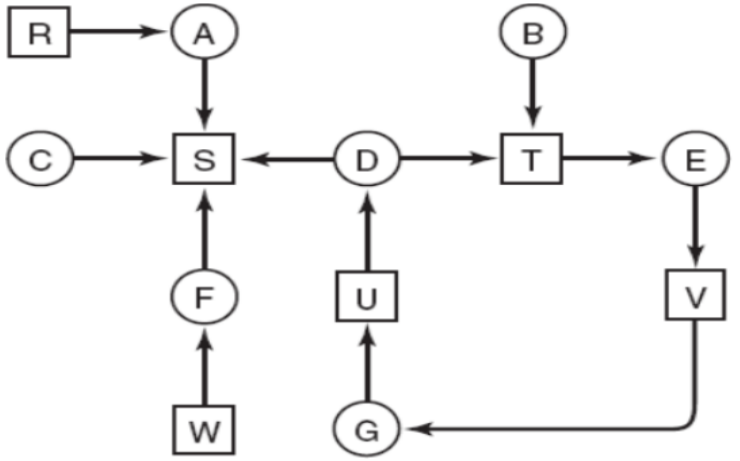


Figure: Detecting deadlocks (Tanenbaum)

- **Starting at Node B**:
  - Initialize L=[] and start with B.
  - L=[B], follow to T and add to L, so L=[B,T]
  - Move to E, making L=[B,T,E]
  - Move to V, making L=[B,T,E,V]
  - Move to G, making L=[B,T,E,V,G]
  - Move to U, making L=[B,T,E,V,G,U]
  - Move to D, making L=[B,T,E,V,G,U,D]
  - At D, there are multiple outgoing arcs, so we make a choice:
    - Choose S, follow to S, and add to L, so L=[B,T,E,V,G,U,D,S]
    - S is a dead end, so backtrack to D, removing S from L.
    - Back at D, select the arc to T.
    - L=[B,T,E,V,G,U,D,T] Since T is already in L, this indicates a **cycle**.

Dealing with Deadlocks

Summary: Graph Approach (cycle detection algorithm)

- What this algorithm does is take each node, in turn, as the root of what it hopes will be a tree, and do a depth-first search on it.
  - If it ever comes back to a node it has already encountered, then it has found a cycle.
  - If it exhausts all the arcs from any given node, it backtracks to the previous node.
  - If it backtracks to the root and cannot go further, the subgraph reachable from the current node does not contain any cycles.
  - If this property holds for all nodes, the entire graph is cycle free, so the system is not deadlocked.

Dealing with Deadlocks
Detection and Recovery: Several Instances of a Resource Type

- When resources have multiple instances, the graph-based approach is insufficient, and we use a matrix-based approach.
- A matrix based algorithm is used when multiple “copies” of the same resource exist.
- Processes ($\boldsymbol{P1\ to\ Pn}$): The system has n processes that require resources to execute.
- Resource Classes ($\boldsymbol{E1\ to\ Em}$): There are m types of resources in the system.
  - For example, class 1 could be tape drives, and class 2 could be printers.

Dealing with Deadlocks

Detection and Recovery: Matrix approach (detection)

- **Existing Resource Vector (E):**
  - E represents the total number of resources of each class available in the system.
  - E_i is the number of resources of type i available in the system (e.g., if $E_1 = 2$, it means there are two tape drives).

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

- **Available Resource Vector (A):**
  - A denotes the number of currently available (unassigned) resources of each type.
  - A_i is the number of available resources of type ii. For example, if both tape drives are in use, then $A_1 = 0$.

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

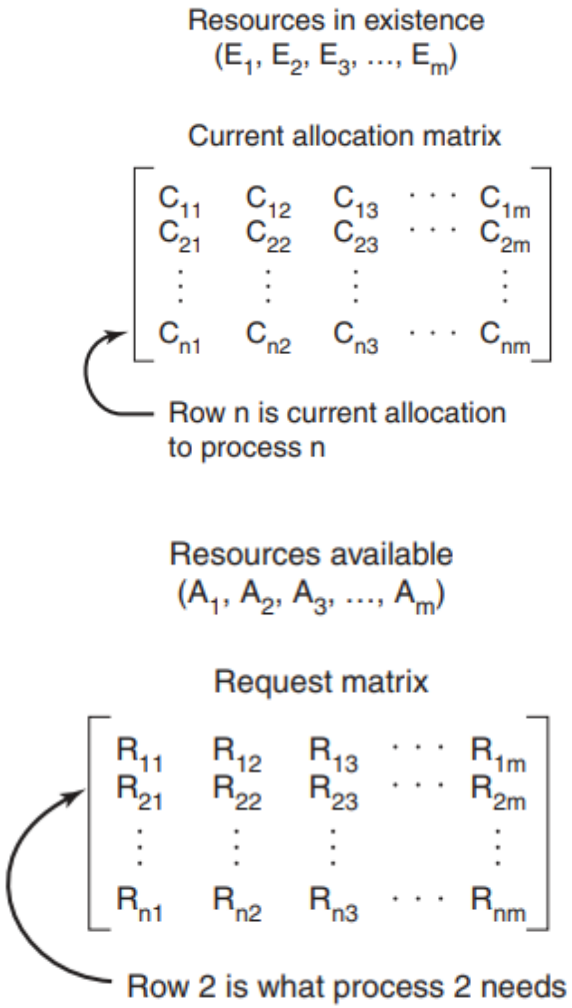
Dealing with Deadlocks

Detection and Recovery: Matrix approach (detection)

- Current Allocation Matrix (C):
  - This matrix indicates how many resources of each type are currently allocated to each process.
  - Each entry C_{ij} represents the number of instances of resource j allocated to process i.
  - Row i in this matrix corresponds to the resources allocated to process Pi.

- Request Matrix (R):
  - This matrix specifies the additional resources each process currently needs to complete.
  - Each entry Rij represents the number of instances of resource j that process i is requesting.
  - Row i in this matrix corresponds to the additional resources that process Pi needs to proceed.

$$E_j = \sum_{i=0}^{n} C_{ij} + Aj$$



Dealing with Deadlocks
Matrix Algorithm

1. Initialization:
   - Set up the Available vector A based on the current resource usage.
   - Initialize the Finish array for each process, setting it to False initially (indicating that each process has not yet completed).
2. Find a Process that Can Proceed:
   - Find an index i such that:
     - Finish[i] == False (process Pi is not yet finished),
     - and R_{ij} <= A_j for all j (all of the resources it requests can be satisfied with the currently available resources).
   - If such a process Pi is found:
     - Mark Finish[i] = True.
     - Release resources by adding them back to Available:
       - $A_j = Aj + Cij \ \ for\ each\ resource\ j$
   - Repeat this step until either:
     - All processes are marked as finished, meaning no deadlock.
     - Or no such process i can be found, meaning a deadlock may exist.
3. Check for Deadlock:
   - After the loop, if there are any processes Pi such that $Finish[i] == False$, then these processes are in a deadlock state (i.e., they are part of a cycle where they're waiting for resources held by each other).

Dealing with Deadlocks

Matrix approach (detection)

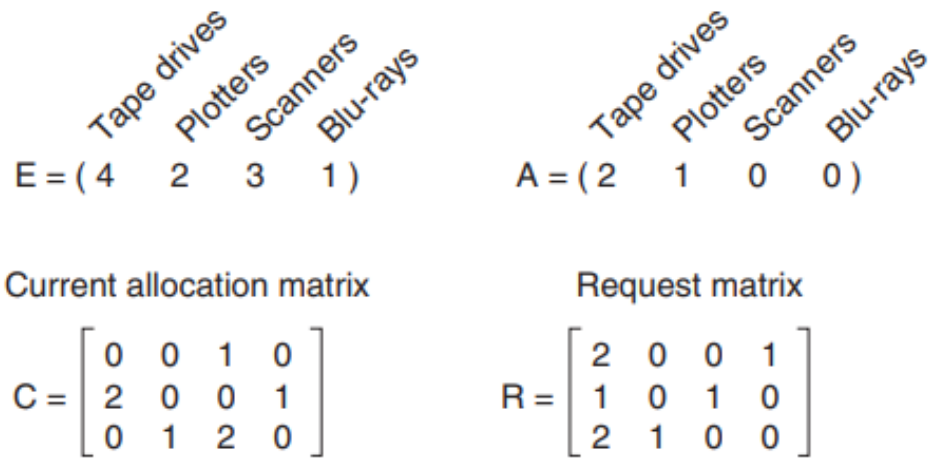


Fig: An example for the deadlock detection algorithm (Tanenbaum)

Is the system deadlocked? **NO!**

1. Run process 3: *A* = (2,2,2,0);
2. Run process 2: *A* = (4,2,2,1);
3. Run process 1: *A* = (4,2,3,1);

Available resources

P3	(2, 1, 0, 0) + (0, 1, 2, 0) = (2, 2, 2, 0)
P2	(2, 2, 2, 0) + (2, 0, 0, 1) = (4, 2, 2, 1)
P1	(4, 2, 2, 1) + (0, 0, 1, 0) = (4, 2, 3, 1)

Deadlocks
Recover from Deadlocks

- Recovery techniques
  - Recovery through Resource Preemption
  - Recovery through Process Termination
  - Recovery through Rollback

Deadlocks
Recover from Deadlocks: Recovery through Preemption

- This involves taking a resource from one process and temporarily assigning it to another.
  - **Example**: A laser printer is taken from one process, and after saving the printed sheets, the printer is assigned to a different process.

- **Challenges**:
  - Preemption is not always feasible, especially for resources that can't easily be reclaimed (e.g., memory or some types of hardware).
  - Some resources may have non-trivial states that are hard to pause or transfer (e.g., open files, network connections).
    - Simply taking them away from a process and giving them to another may lead to severe issues such as data corruption, loss of progress, or system instability.

  - It may require manual intervention, especially in legacy systems or batch processing environments.

Deadlocks
Recover from Deadlocks: Recovery through terminating Processes

- Killing one or more processes involved in the deadlock to free resources and break the cycle.
  - Example:
    - Process A holds Resource X and needs Resource Y to proceed.
    - Process B holds Resource Y and needs Resource X to proceed
    - Kill process B would allow process A to finish it's execution.
- Challenges:
  - Killing a process in the deadlock cycle may resolve the deadlock, but it could leave behind unfinished tasks or unsaved state, requiring rework or leading to loss of data.
  - It's crucial to choose which process to kill. The process should ideally be one that can be safely restarted or re-executed. For instance, a simple task like compilation can be rerun, but database transactions may need to be rolled back carefully to avoid data corruption.
  - If a killed process was part of a critical transaction, restarting it could create inconsistencies (e.g., database anomalies).

Deadlocks
Recover from Deadlocks: Recovery through Rollback

- In systems where processes are periodically checkpointed, deadlock recovery can be done by rolling back a process to an earlier checkpoint where it didn't hold critical resources.
  - Example: A process holding a printer might be rolled back to a checkpoint before it acquired the printer, allowing that resource to be reassigned to another process.

- Challenges:
  - Checkpointing overhead: Maintaining checkpoints consumes storage and may impact system performance.
  - Some work or outputs done after the checkpoint will be lost, including any side effects (e.g., files, printed output).
  - There can be complications with rolling back processes that have made irreversible changes (e.g., database updates).

University of Nottingham
UK | CHINA | MALAYSIA

Quiz!

Recap
Take-Home Message

- What is deadlock and how occurs?
- Graph approach to detect single-source resources
- Matrix approach to detect multi-source resources
- Deadlock recovery: preemption, rollback and/or kill

Recommended readings

- Modern Operating Systems (Tanenbaum): **Chapter 6(6.2-4)**
- Operating System Concepts (Silberschatz): **Chapter 7(7.2, 7.3, 7.6)**
- Operating Systems: Internals and Design Principles (Starlings): **Chapter 6(6.4)**