# CS310 Operating Systems

## Lecture 31: Deadlock Prevention

Ravi Mittal
IIT Goa

# Acknowledgements !


- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, University of California, Berkeley
  - Book: Modern Operating System, Andrew Tanenbaum and Herbert Bos – Chapter 6
  - Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II
    - Chapter 5.6

# Reading

- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II
- Book: Operating System Concepts, 10th Edition, by Silberschatz, Galvin, and Gagne
- Book: Modern Operating System, Andrew Tanenbaum and Herbert Bos – Chapter 6

# Previous Classes

# Necessary Conditions for Deadlock

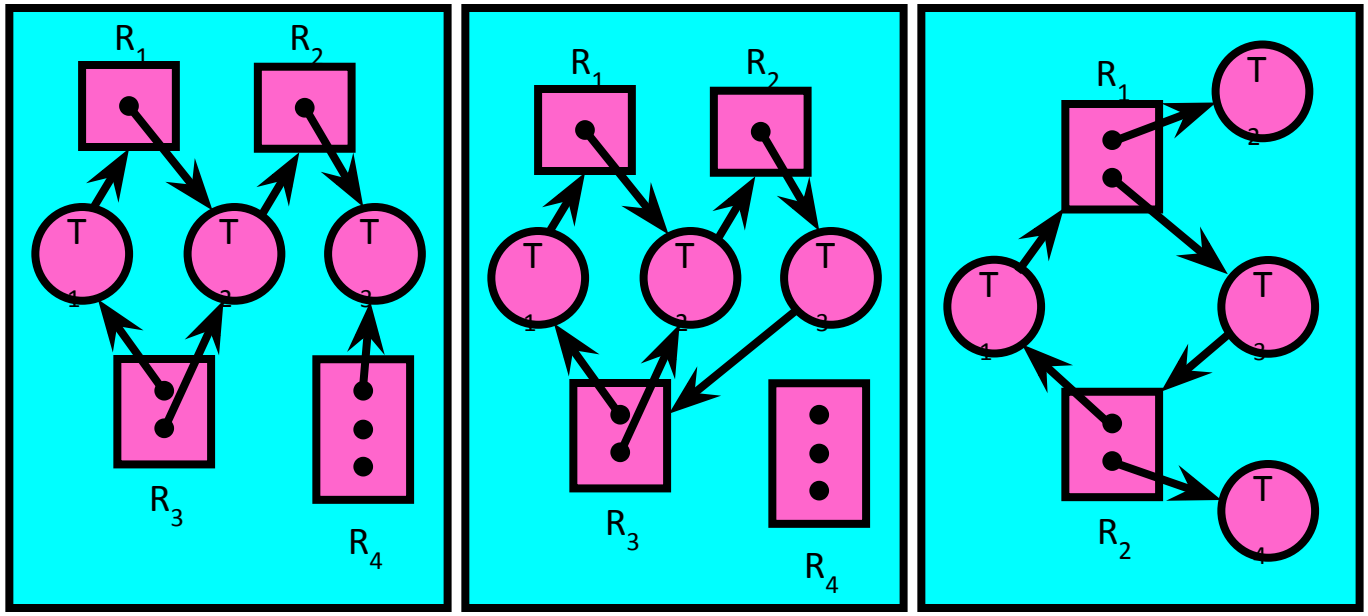
- There are 4 necessary conditions for deadlock to occur
- If you can prevent any one of these conditions, you can eliminate the possibility of deadlock

1. Bounded Resources (or Mutual Exclusion)
   - There are a finite number of threads that can simultaneously use a resource. Only one thread at a time can use a resource
2. No preemption
   - Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it
3. Wait while Holding
   - A thread holds one resource while waiting for another
     - Also called multiple independent requests
4. Circular Waiting
   - A set of waiting threads such that each thread is waiting for a resource held by another

# Resource-Allocation Graph (RAG) Examples

Model: Directed Graph

- request edge
  - $T_i \rightarrow R_j$
- assignment edge
  - $R_j \rightarrow T_i$




Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock

**Instead of thread T, we can also represent a process with a circle**

# Banker's Algorithm

- Suppose we know the worst case resource needs of processes in advance
  - A bit like knowing the credit limit on your credit cards. (This is why they call it the Banker's Algorithm)
- Observation: Suppose we just give some process ALL the resources it could need…
  - Then it will execute to completion.
  - After which it will give back the resources.
- Like a bank: If Visa just hands you all the money your credit lines permit, at the end of the month, you'll pay your entire bill

# Banker's Algorithm

- So…
  - A process pre-declares its worst-case needs
  - Then it asks for what it "really" needs, a little at a time
  - The algorithm decides when to grant requests
- It delays a request unless:
  - It can find a sequence of processes…
  - such that it could grant their outstanding need…
  - so they would terminate…
  - letting it collect their resources…
  - and in this way it can execute everything to completion!

# Today we will study

- Deadlock Prevention Approaches
- Deadlock Recovery Approaches

# How Should a System Deal With Deadlock?

Three different approaches:

1. Deadlock avoidance
2. Deadlock prevention
3. Deadlock recovery

# Deadlock Prevention

# Issues with Deadlock Avoidance

- Deadlock avoidance is very hard
  - It requires information about future resource requests
- Future Resource requests are not easily known
- Then, how to prevent deadlocks?
- Relook at 4 conditions that can cause deadlock

# Conditions for Deadlock

- If you can prevent any one of these conditions, you can eliminate the possibility of deadlock

1. Mutual Exclusion (Bounded Resources)
   - Only one thread at a time can use a resource. Only one thread at a time can use a resource
2. No preemption
   - Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it
3. Wait while Holding
   - A thread holds one resource while waiting for another
     - Also called multiple independent requests
4. Circular Waiting
   - A set of waiting threads such that each thread is waiting for a resource held by another

# Attack Mutual Exclusion Condition

# Deadlock Prevention (remove Mutual Exclusion)

- Remove “Mutual Exclusion”
  - Infinite resources
    - Example: Virtual Memory

# (Virtually) Infinite Resources

| Thread A | Thread B |
| --- | --- |
| AllocateOrWait(1 MB) | AllocateOrWait(1 MB) |
| AllocateOrWait(1 MB) | AllocateOrWait(1 MB) |
| Free(1 MB) | Free(1 MB) |
| Free(1 MB) | Free(1 MB) |

With virtual memory we have infinite space so everything will just succeed.

# Attack Mutual Exclusion – Example

- Don't assign a resource exclusively to a single process
- How about writing to a printer?
  - Printer is dedicated to a process at a time
- Solution: Use printer spooler
  - Now, several processes can generate output at a time
  - The only process that actually requests the physical printer is the printer daemon
  - Daemons are programmed to to print only after the complete file is available

# Attack Mutual Exclusion – Example

- However, NOT all devices can be spooled
- Principle:
  - Avoid assigning a resource unless it is absolutely necessary and try to make sure that as few processes as possible may actually claim the processor

# Attack Hold and Wait Condition

# Attacking the Hold and Wait Condition

- Require all processes request resources before starting
  - If everything is available, the process will be allocated whatever it needs and can run to completion
- Problem with the above approach:
  - Many processes don't know how many resources they need until they start running
  - Resources are not used optimally – resources get tied up until process completes (may take hours)
- Another Solution: back-off and retry
  - A process requesting a resource to first temporarily release all the resources it currently holds
  - Then it tries to get everything it needs all at once

# Request Resource Atomically

```
Thread A
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

```
Thread B
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

**Consider instead:**

```
Thread A
Acquire_both(x, y);
…
y.Release();
x.Release();
```

```
Thread B
Acquire_both(y, x);
…
x.Release();
y.Release();
```

**Or consider this:**

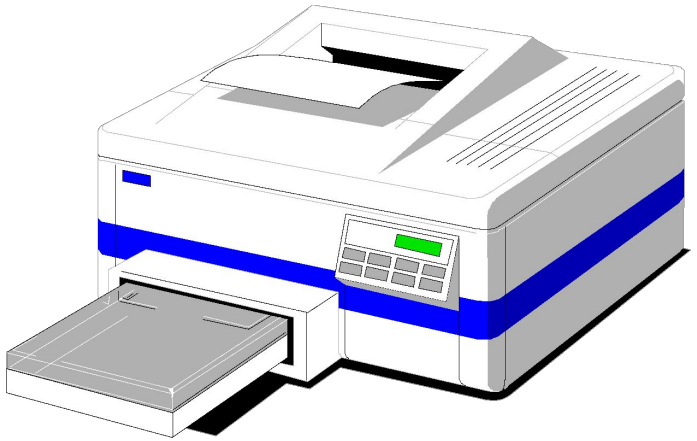
```
Thread A
z.Acquire();
x.Acquire();
y.Acquire();
z.Release();
…
y.Release();
x.Release();
```

```
Thread B
z.Acquire();
y.Acquire();
x.Acquire();
z.Release();
…
x.Release();
y.Release();
```

# Attacking – No preemption Condition

# Attacking the No Preemption Condition

- This is not a viable option
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



# Attacking – Circular Wait Condition

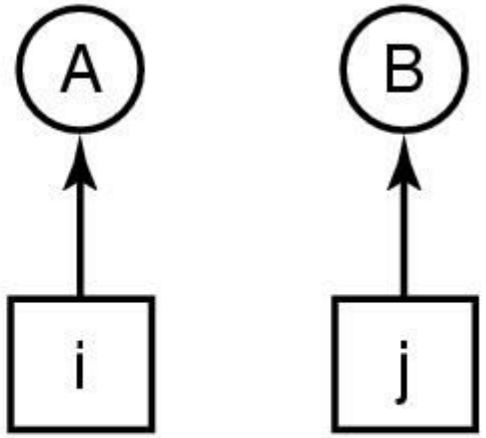
# Attacking Circular Wait Condition

- There are several ways to eliminate circular wait

- Process is entitled to a single resource at any moment
  - If it needs the second one, it has to release the first one
  - Not possible in many use cases – eg copy a huge file from a tape to a printer
- Provide a global numbering of all resources
- Rule:
  - Processes can request resources whenever they want to
  - All requests must be made in numerical order
  - Example: A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer (see diagram in the next slide)

# Attacking the Circular Wait Condition

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive




(a) (b)

- Resource allocation graph will never have cycles
- Two processes A and B
- Deadlock can happen when when A requests j and B requests I
  - If i < j, B is not allowed to request I, and vice versa
- Many variations to this approach exist

# Summary of all deadlock prevention approaches

| Condition | Approach |
|---|---|
| Mutual exclusion | Spool everything |
| Hold and wait | Request all resources initially |
| No preemption | Take resources away |
| Circular wait | Order resources numerically |

# Deadlock Recovery

# How Should a System Deal With Deadlock?

Three different approaches:

1. Deadlock avoidance
2. Deadlock prevention
3. Deadlock recovery

# Deadlock Recovery

- Let deadlock happen, and then figure out how to deal with it

# How to Deal with Deadlock?

- Terminate thread, force it to give up resources
  - Dining Philosophers Example: Remove a dining philosopher
  - In `AllocateOrWait` example, OS kills a process to free up some memory
  - Not always possible—killing a thread holding a lock leaves world inconsistent
- Roll back actions of deadlocked threads
  - Common techniques in databases (transactions)
  - Of course, if you restart in exactly the same way, may enter deadlock again
- Preempt resources without killing off thread
  - Temporarily take resources away from a thread
  - Doesn't always fit with semantics of computation

# Lecture Summary

- Resource deadlock can be avoided by keeping track of which states are safe and which are unsafe.
- Resource deadlock can be structurally prevented by building the system in such a way that it can never occur by design
  - Allowing processes to hold one resource at a time
  - Resource deadlock can also be prevented by numbering all the resources and making processes request them in strictly increasing order
- Once deadlock happens, system must recover from it by
  - Killing a thread or process
  - Rolling back actions of deadlocked thread or process
  - Preempt resources without killing thread or process