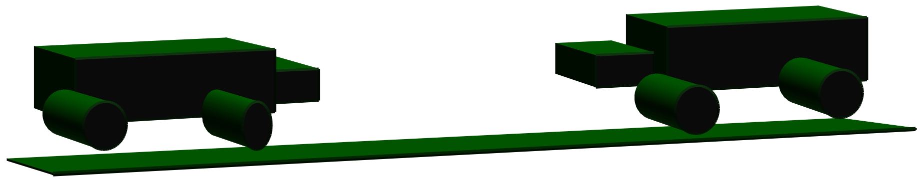
# Deadlocks

CS 447

Monday 3:30-5:00

Tuesday 2:00-3:30

# A deadlock situation

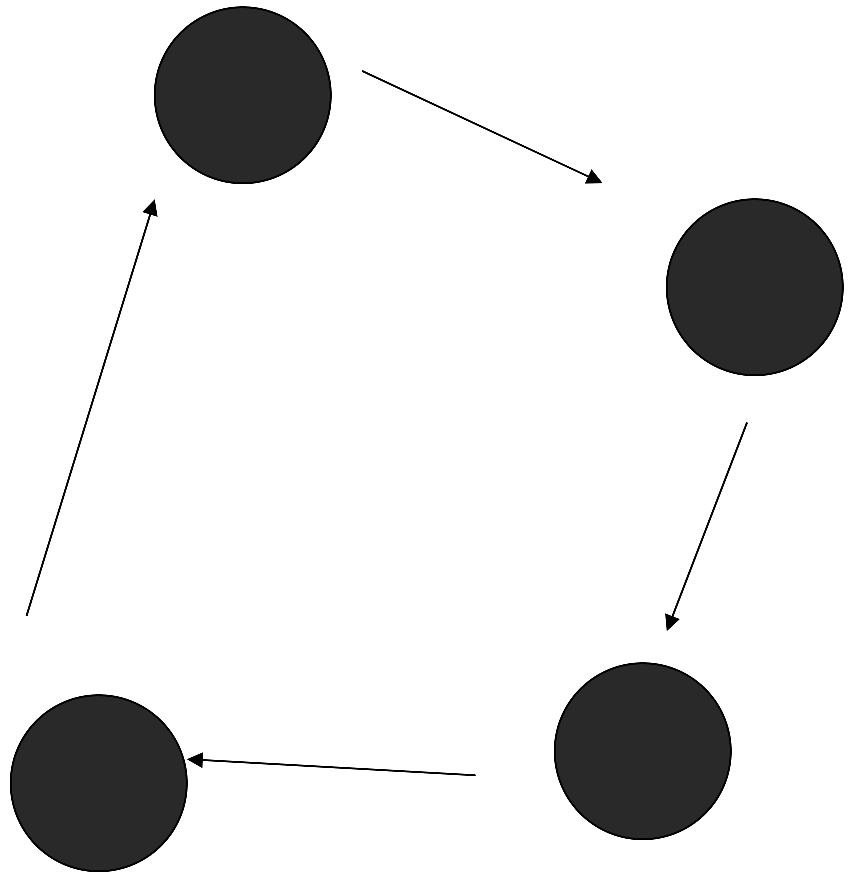


Only one vehicle can use the narrow road!

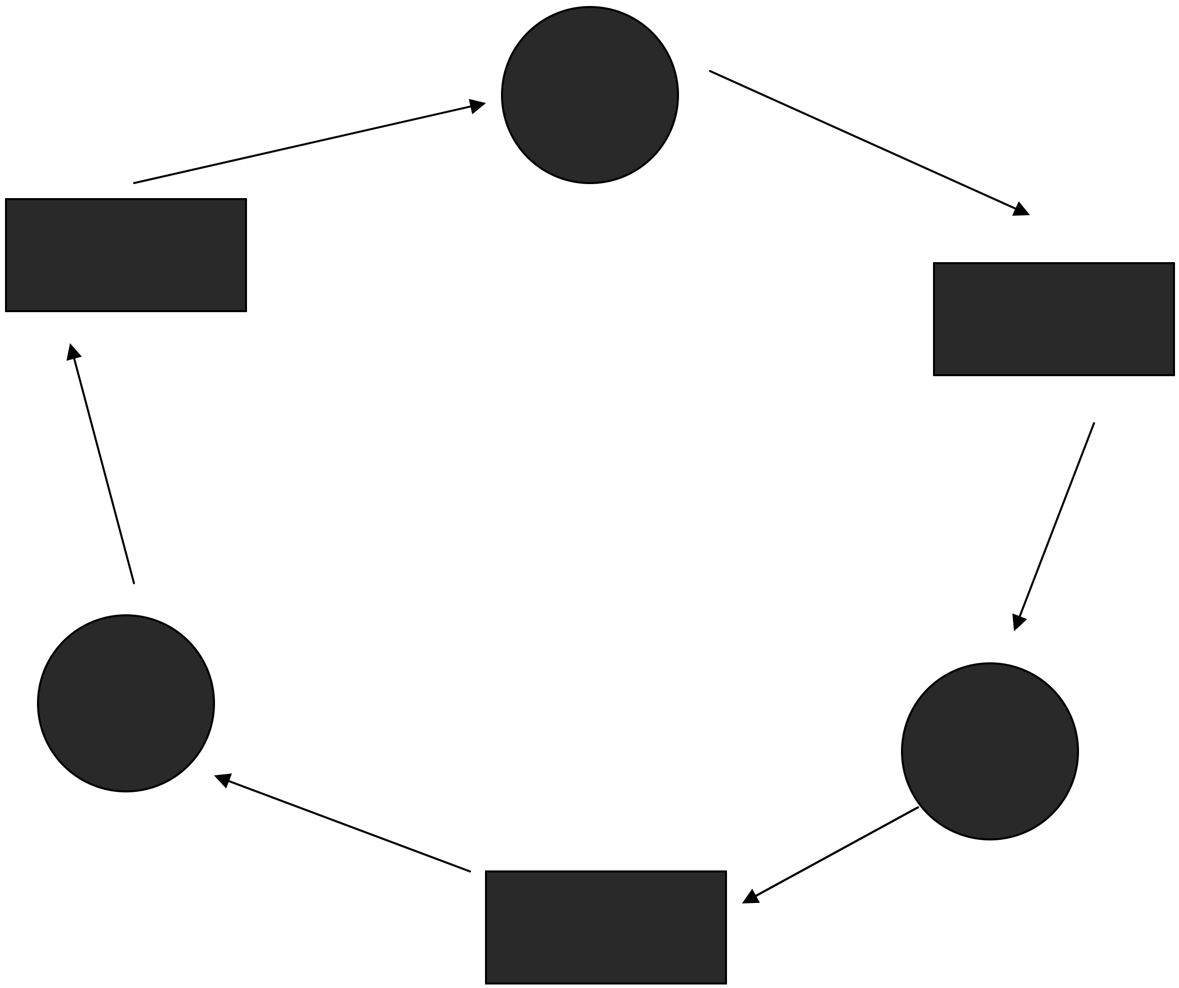
# Approaches to handling deadlocks

- Prevention better than cure
- Cure is possible after detection
- Avoid just when you think there is a possibility
- Ignore!

# Process wait for graphs



# Resource request-allocation graphs



# Necessary conditions for deadlocks to occur

- Hold and wait
- Cyclic wait
- No preemption
- Mutual exclusion

# Deadlock prevention

- Count on necessary conditions!

- A → B

# Prevent cyclic wait

- Impose a total order on resources
- Do not allow waiting on  a low ranked resource than the one already held

- E.g. Ricart and Agrawala distributed mutual exclusion

# Prevent mutual exclusion

- Allow unrestricted access
  - E.g. basic file system support

- File system semantics in presence of concurrency:
  - Unix semantics: the latest is reflected

- No deadlocks on basic file system calls:
  - E.g. as in

| | |
|---|---|
| Fopen (f1) | fopen (f2) |
| fopen (f2) | fopen (f1) |

# Prevent no-preemption

- On-demand preemption
  - Upon request, preempt a resource
- Periodic preemption
  - Strict round robin CPU allocator
- Risk of leaving preempted resource in an inconsistent state must be handled

# Prevent Hold and Wait

- Example:
  - Customer: delivery first, payment later
  - Dealer: pay first, deliver later
- To break the deadlock::
  - Do not hold payment while asking for delivery
    - Or
  - Do not hold delivery while asking for payment

# Deadlocks with multiple instance resources

- Example
- Consider each instance separately:
  - You will get an OR edge
  - All OR edges in a deadlock cycles

# Multiple blocked requests

- AND edges

# Process in a deadlocked set

- = Processes in a deadlock + all processes dependent on processes in a deadlock set

- example

# Deadlock Detection

- Data structures:
- M: no. of processes
- N: no. of resources
- bool Req[M][N] (which resources are requested?)
- int Allocated[M][N] (how many instances are allocated?)
- Boolean Completed [M] (temporary)
- int Free[N] (temporary)

# Deadlock Detection: Step 1

- Find in Req[][], all such processes that have not requested a resource
- If found, mark them completed in Completed[]
- Find all resources allocated to them from Allocated[]
- Mark these resources as free in Free[]

# Deadlock Detection: Step 2

- Find in Req[][], a a process for which all requested resources are marked free in Free[]
- If found, mark the process as completed in Completed[]
- Find all resources allocated to the process from Allocated[]
- Mark these resources as free in Free[]
- Repeat step 2 till no such process is found

# Deadlock Detection: Step 3

- If array Completed[] indicates true for all processes, there is no deadlock
- Else the processes which are not marked as completed in Completed[] are part of the deadlock set.

- Example Trace

# When to invoke deadlock detection?

- Major deadlock:
  - No of processes is high
  - But CPU utilization is low

# Deadlock Avoidance: Banker's algorithm

- <u>Data structures:</u>
- M: no of processes
- N: no of resources
- Int Need [M] [N] (indicates maximum need in future)
- Boolean Allocated [M] [N] (how many instances allocated?)
- Int Available [N] (how many instances are available)

# Upon a request Ri[N] by a process Pi::
# Banker's algorithm: step 1

- If Ri[0..N-1] <= Need[i][0.N-1]
  - continue with step 2
- Else invalid request error

# Upon a request Ri[N] by a process Pi:: Banker's algorithm: step 2

- Check from Available [0..N-1] whether the number of requested resources are available

- If not, the request cannot be considered at this time, return

- Else continue with step 3

# Upon a request Ri[N] by a process Pi:: Banker's algorithm: step 3

- Find out if a worst requesting situation that may follow can be taken care of

(i.e. all process asking for their maximum needs – after current Request from Pi is satisfied)

- i.e in such a case, can you find a safe sequence of allocations such that deadlock will not occur?
- If such a safe sequence exists, go ahead with the request
  - Else reject the request

# Example

| Processes | Allocated | Need |
|---|---|---|
| 0 | 2 1 2 | 0 0 1 |
| 1 | 0 1 1 | 7 3 3 |
| 2 | 2 2 1 | 4 0 1 |
| 3 | 1 2 0 | 1 1 1 |
| 4 | 3 1 1 | 0 1 4 |

Available: 2 2 4

# Apply banker's algo for the above example

- Is it safe to allow
  - Request2 [2 0 1]?    request from P2
  - Request1 [2 2 1]?    request from P1
  - Request4 [0 1 4]?    request from P4