Deadlocks: Detection & Avoidance

(Chapter 32)

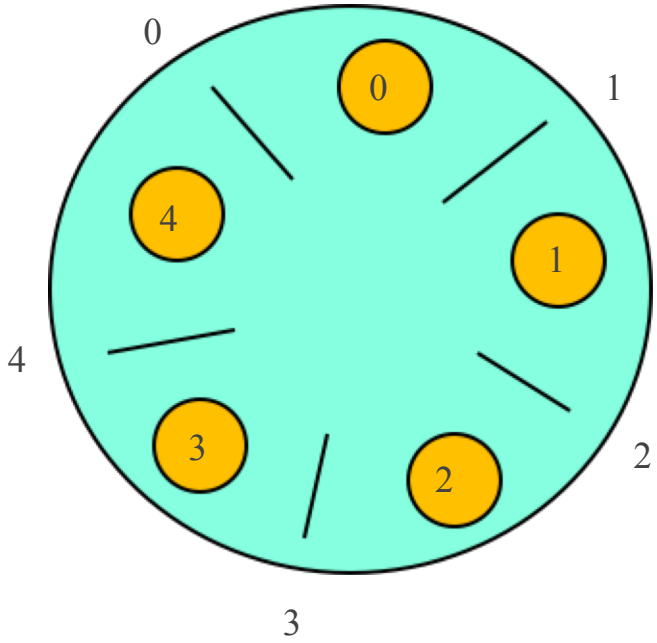
CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

The slides are the product of many rounds of teaching CS 4410 by Professors Agarwal, Bracy, George, Schneider, Sirer, and Van Renesse.

Dining Philosophers [Dijkstra 68]

```
Pi:  do forever
       acquire( left(i) );
       acquire( right(i) );
       eat
       release( left(i));
       release( right(i));
     end
```



left(i): i+1 mod 5
right(i): i

Problematic Emergent Properties

Starvation: Process waits forever.

Deadlock: A set of processes exists, where each is blocked and can become unblocked only by actions of another process in the set.

- Deadlock implies Starvation (but not *vice versa*).
- Starvation often tied to **fairness**: A process is not forever blocked awaiting a condition that (i) becomes continuously true or (ii) infinitely-often becomes true.

Testing for starvation or deadlock unlikely to work.

More Examples of Deadlock

Example (initially in1 = in2 = false):

in1 := true; **await** not in2; in1 := false
//
In2 := true; **await** not in1; in2 := false

Example (initially m1 = m2 = 1):

P(m1); P(m2); V(m1); V(m2)
//
P(m2); P(m1); V(m1); V(m2)

System Model

- Set of resources requiring “exclusive” access
  - Might be “k-exclusive access” if resource has capacity for k.
  - Examples: buffers, packets, I/O devices, processors, …

- Protocol to access a resource causes blocking:
  - If resource is free then access is granted; process proceeds.
  - If resource is in use then process blocks;
  - Use resource
  - Release resource.

When is deadlock possible?

Necessary Conditions for Deadlock

1. Acquire can block invoker.

2. Hold & wait. A process can be blocked while holding resources.

3. No preemption. Allocated resources cannot be reclaimed. Explicit release operation needed.

4. Circular waits are possible.

   Let p → q denote "p waits for q to release a resource". Then

   P1 → P2 → … → Pn → P1

Deadlock is Undesirable

- Deadlock prevention: Ensure that a necessary condition cannot hold.
- Deadlock avoidance: Monitor does not allocate resources that will lead to a deadlock.
- Deadlock detection: Allow system to deadlock; detect it; recover.

Deadlock Prevention: Negate 1

#1: Eliminate mutual exclusion / bounded resources:

- Make resources sharable without locks
- Have sufficient resources available, so acquire never delays.

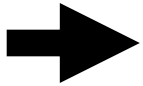
Deadlock Prevention: Negate 2

#2: Eliminate hold and wait

Don't hold some resources when waiting for others.

- Re-write code:

```
Module:: foo()  {
  lock.acquire();
  doSomeStuff();
  otherModule->bar();
  doOtherStuff();
  lock.release(); }
```



Have these fns acquire & release

```
Module:: foo()  {
  doSomeStuff();
  otherModule->bar();
  doOtherStuff();
}
```

- Request all resources before execution begins
  - Processes don't know what they need ahead of time.
  - Starvation (if waiting on many popular resources).
  - Low utilization (need resource only for a bit).

Deadlock Prevention: Negate 1 of 4

#3: Allow preemption

Requires mechanism to save / restore resource state:

multiplexing vs undo/redo

- Examples of multiplexing:
  - processor registers
  - Regions of memory (pages)
- Examples of undo/redo
  - Database transaction processing

Deadlock Prevention: Negate 1 of 4

#4: Eliminate circular waits.

Let R = {R1, R2, … Rn} be the set of resource types.
Let (R , <) be an non symmetric relation:
not r < r [irreflexive]
If r < s and s < t then r < t [transitive]
not r < s and s < r [non symmetric]
for every r and s (r ≠ s): r < s or s < r [total order]

Rule: Request resources in increasing order by <.
(All resources from type Ri must be requested together)

Rule: To request the resources of type Rj, first release all resources from type Ri where Ri < Rj.

Why < Rules Work

Thm: Total order resource allocation avoids circular waits.

Proof: By contradiction. Assume a circular wait exists.

P1 → P2 → P3 → … → Pn → P1.

P1 requesting R1 held by P2.

P2 requesting R2 held by P3. (So R1 < R2 holds)

…

Conclude: R1 < R2, R2 < R3, …, Rn < R1.

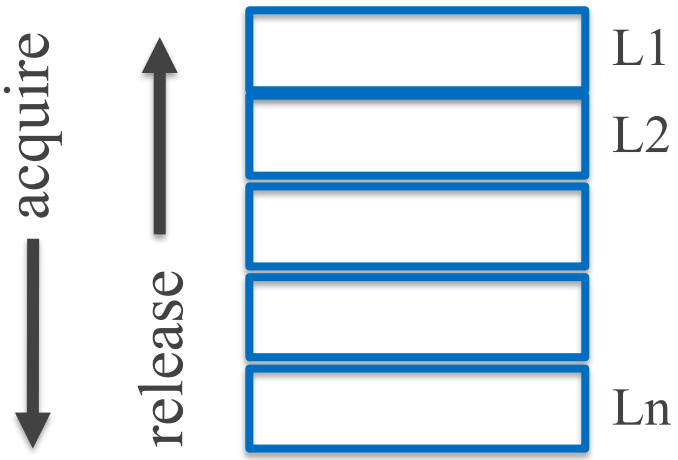
By transitivity: R1 < R1. A contradiction!

Havender's Scheme (OS/360)

Hierarchical Resource Allocation

Every resource is associated with a level.

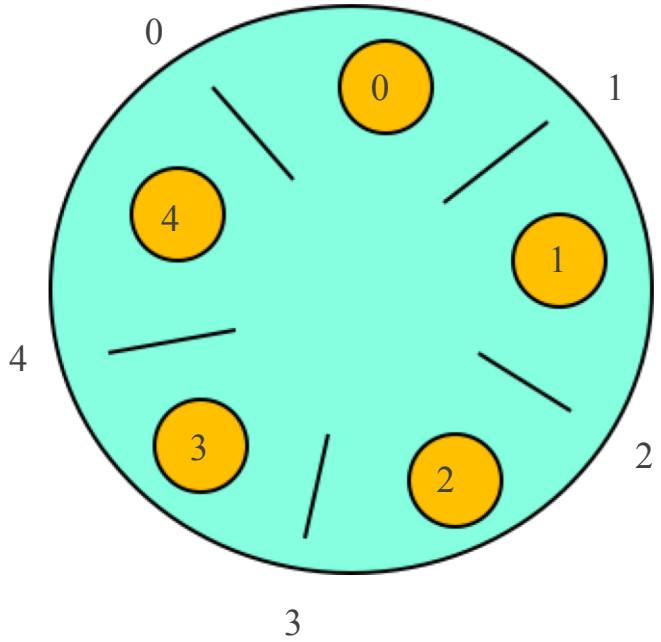
- **Rule H1**: All resources from a given level must be acquired using a single request.
- **Rule H2**: After acquiring from level Lj must not acquire from Li where i<j.
- **Rule H3**: May not release from Li unless already released from Lj where j>i.



Dining Philosophers (Again)

Pi: **do forever**

acquire(F(i));
acquire(G(i));
eat
release(F(i));
release(G(i));
end



F(i): min(i, i+1 mod 5)
G(i): max(i, i+1 mod 5)

Deadlock Avoidance

state: allocation to each process
safe state: a state from which some execution is possible that does not cause deadlock.

- Requires knowing max allocation for each process.
- Check that
  - Exists sequence P1 P2 … Pn of processes where:

    Forall i where $1 \leq i \leq n$:

    Pi can be satisfied by Avail + resources held by P1 … Pi-1.

Assumes no synchronization between processes, except for resource requests.

Safe State Example

Suppose: 12 tape drives and 3 processes: p0, p1, and p2

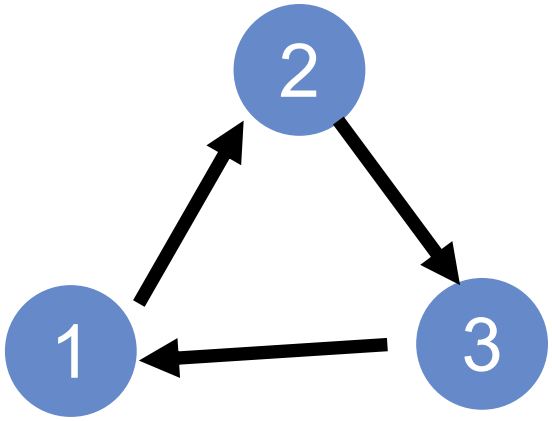
	max need	current usage	could still ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

Current state is *safe* because a safe sequence exists: [p1, p0, p2]

- p1 can complete with remaining resources
- p0 can complete with remaining+p1
- p2 can complete with remaining+p1+p0

What if p2 requests 1 drive? Grant or not?

Deadlock Detection



Create a Wait-For Graph

- 1 Node per Process
- 1 Edge per Waiting Process, P
  (from P to the process it's waiting for)

Note: graph holds for a single instant in time

Cycle in graph indicates deadlock

Testing for cycles (= deadlock)

Reduction Algorithm:
Find a node with no outgoing edges

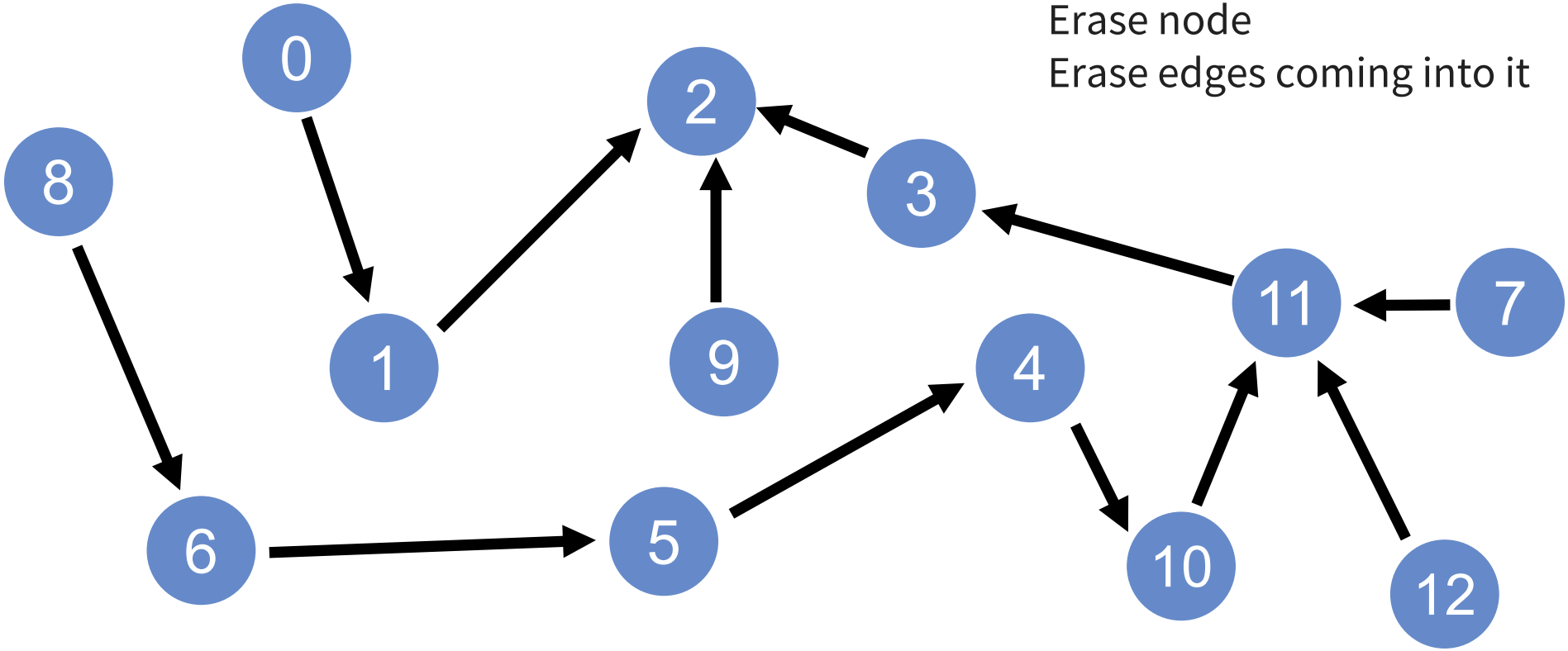
- Erase node
- Erase any edges coming into it

Proof: Deleted node is for process that is not waiting. It will eventually finish and release its resources, so any process waiting for those resources will longer be waiting.

Erase whole graph ↔ graph has no cycles
Graph remains ↔ deadlock

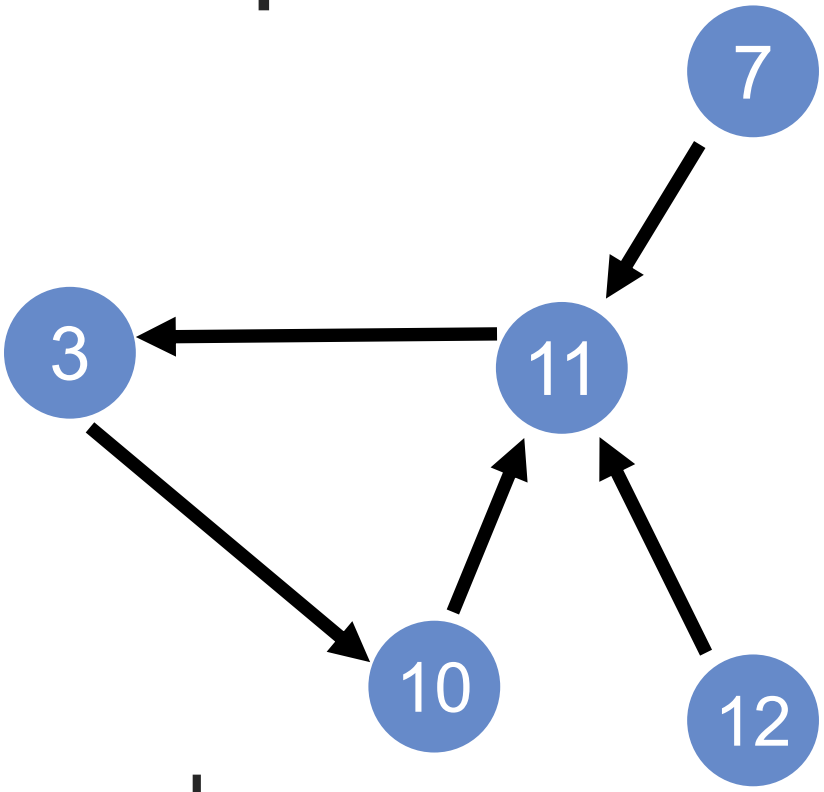
Graph Reduction: Example 1

Find node w/o outgoing edges
Erase node
Erase edges coming into it



Graph can be fully reduced, hence there was no deadlock at the time the graph was drawn. (Obviously, things could change later!)

Graph Reduction: Example 2



No node with no outgoing edges…
Irreducible graph, contains a cycle
(only some processes are in the cycle)
➛ deadlock

Banker's Algorithm

- from 10,000 feet:
  - Process declares its worst-case needs, asks for what it "really" needs, a little at a time
  - Algorithm decides when to grant requests
    - Build a graph assuming request granted
    - Reducible? yes: grant request, no: wait

Problems:

- Fixed number of processes
- Need worst-case needs ahead of time
- Expensive

Question:

Does choice of node for reduction matter?

Answer: No.
Explanation: an unchosen candidate at one step remains a candidate for later steps. Eventually—regardless of order—every node will be reduced.

Question:

Suppose no deadlock detected at time T.
Can we infer about a later time T+x?

Answer: Nothing.

Explanation: The very next step could be to run some process that will request a resource…

- … establishing a cyclic wait
- … and causing deadlock

Implementing Deadlock Detection

- Track resource allocation (who has what)
- Track pending requests (who's waiting for what)

Maintain a wait-for graph.

When to run graph reduction?

- Whenever a request is blocked?
- Periodically?
- Once CPU utilization drops below a threshold?

Deadlock Recovery

Blue screen & reboot?

Kill one/all deadlocked processes

- Pick a victim
- Terminate
- Repeat if needed

Preempt resource/processes till deadlock broken

- Pick a victim (# resources held, execution time)
- Rollback (partial or total, not always possible)
- Starve (prevent process from being executed)