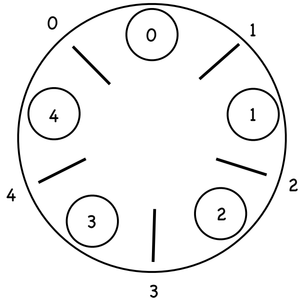
Deadlocks:

Prevention, Avoidance, Detection, Recovery

Dining Philosophers



```
class Philosopher:
chopsticks[N] = [Semaphore(1),...]

def __init__(mynum)
  self.id = mynum

def eat():
   right = self.id
   left = (self.id+1) % N
   while True:
     P(chopsticks[left])
     P(chopsticks[right])
      # om nom nom nom
     V(chopsticks[right])
     V(chopsticks[left])
```

- N philosophers; N plates; N chopsticks
- If all philosophers grab right chopstick
  - deadlock!
- Need exclusive access to two chopsticks

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.

```
semaphore:
file_mutex = 1
printer_mutex = 1
```

T_1

```
{
  P(file_mutex)
  P(printer_mutex)

  /* use resources */

  V(printer_mutex)
  V(file_mutex)
}
```

T_2

```
{
  P(printer_mutex)
  P(file_mutex)

  /* use resources */

  V(file_mutex)
  V(printer_mutex)
}
```

Musings on Deadlock & Starvation

- Deadlock vs Starvation
  - Starvation: some thread's access to a resource is indefinitely postponed
  - Deadlock: circular waiting for resources
  - Deadlock implies Starvation, but not vice versa
- "Subject to deadlock" does not imply "Will deadlock"
  - Testing is not the solution
  - System must be deadlock-free by design

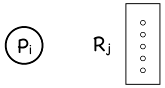
System Model

- Set of resources requiring "exclusive" access
  - might be "k-exclusive access" if resource has capacity for k
  - Examples: CPU, printers, memory, locks, etc.
- Acquiring a resource can cause blocking:
  - if resource is free, then access is granted; process proceeds
  - if resource is in use, then process blocks
  - process uses resource
  - process releases resource

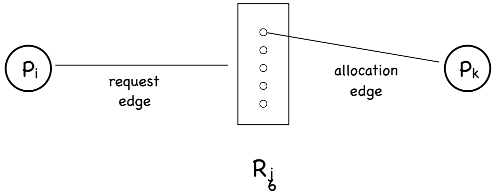
A Graph Theoretic Model of Deadlock

Resource Allocation Graph

- Computer system modeled as a RAG, a directed graph G(V, E)
  - $V = \{P_1,\dots,P_n\} \cup \{R_1,\dots,R_n\}$



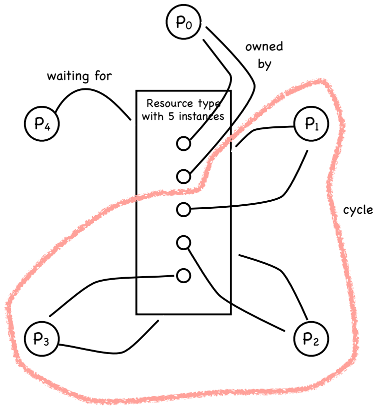
  - E = {edges from a resource to a process} ∪ {edges from a process to a resource}



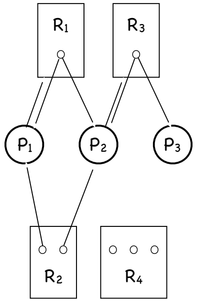
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources (Acquire can block invoker)
    - A finite number of threads can use a resource; resources are finite
  - No preemption
    - the resource is mine, MINE! (until I release it)
  - Hold & Wait
    - holds one resource while waiting for another
  - Circular waiting
    - T_i waits for T_{i+1} and holds a resource requested by T_{i-1}
    - sufficient only if one instance of each resource

Not sufficient in general



RAG Reduction



Deadlock?

NO! (no cycles)

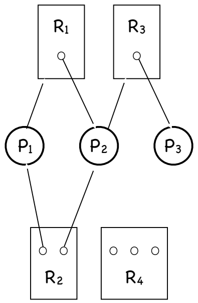
Step 1: Satisfy P_3's requests

Step 2: Satisfy P_2's requests

Step 3: Satisfy P_1's requests

Schedule [P_3 P_2 P_1] completely eliminates edges!

RAG Reduction

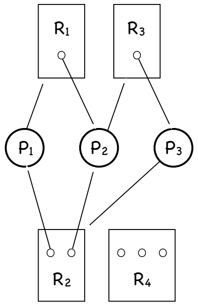


Deadlock?

NO! (no cycles)

Step 1: Satisfy P_3's requests
Step 2: Satisfy P_2's requests
Step 3: Satisfy P_1's requests

Schedule [P_3 P_2 P_1] completely eliminates edges!



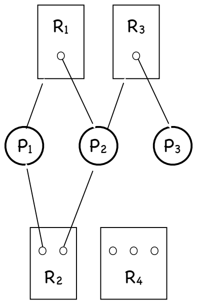
Deadlock?

Yes!

RAG has a cycle

Cannot satisfy any of P_1, P_2, P_3 requests!

RAG Reduction

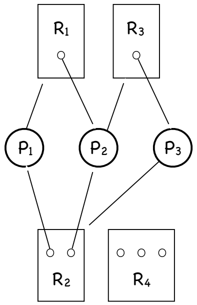


Deadlock?

NO! (no cycles)

Step 1: Satisfy P_3's requests
Step 2: Satisfy P_2's requests
Step 3: Satisfy P_1's requests

Schedule [P_3 P_2 P_1] completely eliminates edges!

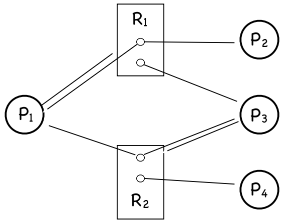


Deadlock?

Yes!

RAG has a cycle

Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

NO!

RAG has a cycle

Schedule [P_2 P_1 P_3 P_4] completely eliminates edges!

More Musings on Deadlock

- Does the order of RAG reduction matter?
  - No. If P_i and P_j can both be reduced, reducing P_i does not affect the reducibility of P_j
- Does a deadlock disappear on its own?
  - No. Unless a process is killed or forced to release a resource, we are stuck!
- If a system is not deadlock at time T, is it guaranteed to be deadlock-free at T+1?
  - No. Just by requesting a resource (never mind being granted one) a process can create a circular wait!

Proactive Responses to Deadlock: Prevention

- Negate one of deadlock's four necessary conditions
  - Remove "Acquire can block invoker"
    - Make resources sharable without locks
      - Wait-free synchronization
    - Make more resources available (duh!)
  - Remove "No preemption"
    - Allow OS to preempt resources of waiting processes
    - Allow OS to preempt resources of requesting process if not all available

Proactive Responses to Deadlock: Prevention

- Negate one of deadlock's four necessary conditions
  - Remove "Hold & Wait"
    - Request all resources before execution begins
      - Processes may not know what they will need
      - Starvation (if waiting for many popular resources)
      - Low utilization (if resource needed only for a bit)
    - Release all resources before asking anything new
      - Still has the last two problems...

Proactive Responses to Deadlock: Prevention

- Negate one of deadlock's four necessary conditions
  - Remove "Circular waiting"
    - Single lock for entire system?
    - Impose total/partial order on resources
      - Makes cycles impossible, since a cycle needs edges to go from low to high, and then back to low

---

Havender's Scheme (OS/360)

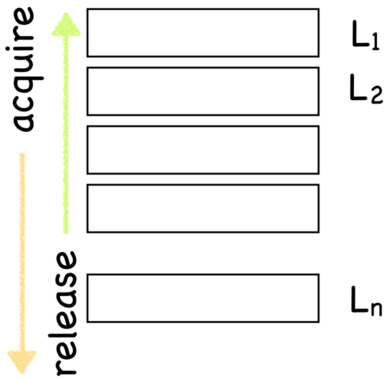
Hierarchical Resource Allocation

Every resource is associated with a level.

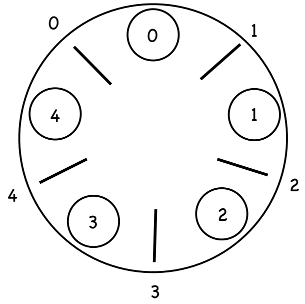
Rule H1: All resources from a given level must be acquired using a single request.

Rule H2: After acquiring from level L_j must not acquire from L_i where $i<j$.

Rule H3: May not release from L_i unless already released from L_j where $j>i$.



Dining Philosophers (Again)

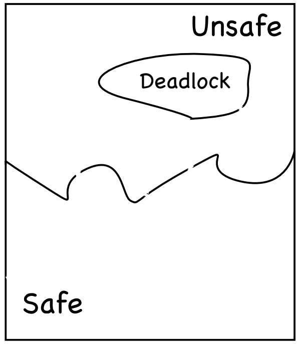


```
$P_i$: do forever
    acquire(min(i, i+1 mod 7)
    acquire(max(i, i+1 mod 7)
    eat
    release(min(i, i+1 mod 7)
    release(max(i, i+1 mod 7)
  end
```

- N philosophers; N plates; N chopsticks

Living dangerously: Safe, Unsafe, Deadlocked States

Living dangerously: Safe, Unsafe, Deadlocked States



A system's trajectory through its state space

- Safe state:
  - It is possible to avoid deadlock and eventually grant all resource by careful scheduling (a safe schedule)
  - Transitioning among safe states may delay a resource request even when resources are available
- Unsafe state:
  - Unlucky sequence of requests can force deadlock
- Deadlocked state:
  - System has at least one deadlock

Why is George Bailey in trouble?

- If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt
- But his bank is actually in a safe state!
  - If only lenders delayed their requests, all would be well!
    - spoiler alert: this is exactly what happens…
- It still begs the question:
  - How can the OS allocate resources so that the system always transitions among safe states?

Proactive Responses to Deadlock: Avoidance

The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Processes declare worst-case needs (big assumption!), but then ask for what they "really" need, a little at a time
  - Sum of maximum resource needs can exceed total available resources
- Algorithm decides whether to grant a request
  - Build a graph assuming request granted
  - Check whether state is safe (i.e., whether RAG is reducible)
    - A state is safe if there exists some permutation of [P_1, P_2,…,P_n] such that, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources currently held by all P_j, for P_j preceding P_i in the permutation

Available = 3

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Safe?

- ✓ Available resources can satisfy P_1's needs
- ✓ Once P_1 finishes, 5 available resources
- ✓ Now, available resources can satisfy P_0's needs
- ✓ Once P_0 finishes, 10 available resources
- ✓ Now, available resources can satisfy P_3's needs

Yes! Schedule: [P_1, P_0, P_3]

Proactive Responses to Deadlock: Avoidance

The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Processes declare worst-case needs (big assumption!), but then ask for what they "really" need, a little at a time
  - Sum of maximum resource needs can exceed total available resources
- Algorithm decides whether to grant a request
  - Build a graph assuming request granted
  - Check whether state is safe (i.e., whether RAG is reducible)
    - A state is safe if there exists some permutation of [P_1, P_2,…,P_n] such that, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources currently held by all P_j, for P_j preceding P_i in the permutation

Available = 3

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Suppose P_2 asks for 2 resources

Safe?

Proactive Responses to Deadlock: Avoidance

The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Processes declare worst-case needs (big assumption!), but then ask for what they "really" need, a little at a time
  - Sum of maximum resource needs can exceed total available resources
- Algorithm decides whether to grant a request
  - Build a graph assuming request granted
  - Check whether state is safe (i.e., whether RAG is reducible)
    - A state is safe if there exists some permutation of [P_1, P_2,…,P_n] such that, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources currently held by all P_j, for P_j preceding P_i in the permutation

Available = 3

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

Safe?

Available = 1

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	4	5

  - If so, request is granted; otherwise, requester must wait

---

The Banker's books

- Assume n processes, m resources
- Max_{ij} = max amount of units of resource R_j needed by P_i
  - $MaxClaim_i$: Vector of size m such that $MaxClaim_i[j] = Max_{ij}$
- $Holds_{ij}$ = current allocation of R_j held by P_i
  - $HasNow_i$ = Vector of size m such that $HasNow_i[j] = Holds_{ij}$
- Available = Vector of size m such that Available[j] = units of R_j available
- A request by P_k is safe if, assuming the request is granted, there is a permutation of P_1, P_2,…, P_n such that, for all P_i in the permutation

$$Needs_i = MaxClaim_i - HasNow_i \le Avail + \sum_{j=1}^{i-1} HasNow_j$$

An Example

- 5 processes, 4 resources

Max

0	0	1	2
1	7	5	0
2	3	5	6
0	6	5	2
0	6	5	6

Holds

0	0	1	2
1	0	0	0
1	3	5	3
0	6	3	2
0	0	1	4

Available

- Is this a safe state?

An Example

- 5 processes, 4 resources

Max					Holds				Available	Needs			
0	0	1	2	−	0	0	1	2		0	0	0	0
1	7	5	0		1	0	0	0		0	7	5	0
2	3	5	6		1	3	5	3		1	0	0	3
0	6	5	2		0	6	3	2		0	0	2	0
0	6	5	6		0	0	1	4		0	6	4	2

- Is this a safe state?

P_1, P_4, P_2, P_3, P_5

  - While safe permutation does not include all processes:
    - Is there a P_i such that $\text{Needs}_i \leq \text{Avail}$?
      - if no, exit with unsafe
      - if yes, add P_i to the sequence and set Avail = Avail + HasNow_i
  - Exit with safe

An Example

- 5 processes, 4 resources

	Max					Holds				Available				Needs			
	R_1	R_2	R_3	R_4		R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4				
P_1	0	0	1	2	P_1	0	0	1	2	1	5	2	0	0	0	0	0
P_2	1	7	5	0	P_2	1	0	0	0					0	7	5	0
P_3	2	3	5	6	P_3	1	3	5	3					1	0	0	3
P_4	0	6	5	2	P_4	0	6	3	2					0	0	2	0
P_5	0	6	5	6	P_5	0	0	1	4					0	6	4	2

- P2 want to change its holdings to 0 4 2 0

An Example

- 5 processes, 4 resources

	Max					Holds				Available				Needs			
	R_1	R_2	R_3	R_4		R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4				
P_1	0	0	1	2	P_1	0	0	1	2	2	1	0	0	0	0	0	0
P_2	1	7	5	0	P_2	0	4	2	0					1	3	3	0
P_3	2	3	5	6	P_3	1	3	5	3					1	0	0	3
P_4	0	6	5	2	P_4	0	6	3	2					0	0	2	0
P_5	0	6	5	6	P_5	0	0	1	4					0	6	4	2

- P2 want to change its holdings to 0 4 2 0
- Safe?

Reactive Responses to Deadlock

- Deadlock Detection
  - Track resource allocation (who has what)
  - Track pending requests (who's waiting for what)
- When should it run?
  - For each request?
  - After each unsatisfiable request?
  - Every hour?
  - Once CPU utilization drops below a threshold?

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	1	0	0	0	0	0	0	0
P_2	2	0	0				2	0	2
P_3	3	0	3				0	0	0
P_4	2	1	1				1	0	2
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	1	0	0	0	0	0	0	0
P_2	2	0	0				2	0	2
P_3	3	0	3				0	0	0
P_4	2	1	1				1	0	2
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	1	0	3	0	3	0	0	0
P_2	2	0	0				2	0	2
P_3	0	0	0				0	0	0
P_4	2	1	1				1	0	2
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	1	0	3	0	3	0	0	0
P_2	2	0	0				2	0	2
P_3	0	0	0				0	0	0
P_4	2	1	1				1	0	2
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
3	1	3

Pending

0	0	0
2	0	2
0	0	0
1	0	2
0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	2	1	1
P_5	0	0	2

Available

R_1	R_2	R_3
3	1	3

Pending

0	0	0
2	0	2
0	0	0
1	0	2
0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

Available

R_1	R_2	R_3
5	2	4

Pending

0	0	0
2	0	2
0	0	0
0	0	0
0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

Holds

	R_1	R_2	R_3
P_1	0	0	0
P_2	2	0	0
P_3	0	0	0
P_4	0	0	0
P_5	0	0	2

Available

R_1	R_2	R_3
5	2	4

Pending

0	0	0
2	0	2
0	0	0
0	0	0
0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	0	0	7	2	4	0	0	0
P_2	0	0	0				0	0	0
P_3	0	0	0				0	0	0
P_4	0	0	0				0	0	0
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	0	0	7	2	4	0	0	0
P_2	0	0	0				0	0	0
P_3	0	0	0				0	0	0
P_4	0	0	0				0	0	0
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	0	0	7	2	6	0	0	0
P_2	0	0	0				0	0	0
P_3	0	0	0				0	0	0
P_4	0	0	0				0	0	0
P_5	0	0	0				0	0	0

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Yes, there is
a safe sequence!

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	1	0	0	0	0	0	0	0
P_2	2	0	0				2	0	2
P_3	3	0	3				0	0	0
P_4	2	1	1				1	0	2
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

Yes, there is
a safe sequence!

Detecting Deadlock

- 5 processes, 3 resources. We no longer (need to) know Max

	Holds			Available			Pending		
	R_1	R_2	R_3	R_1	R_2	R_3			
P_1	0	1	0	0	0	0	0	0	0
P_2	2	0	0				2	0	2
P_3	3	0	3				0	0	1
P_4	2	1	1				1	0	2
P_5	0	0	2				0	0	2

- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock
- Can we avoid deadlock by delaying granting requests?
  - Deadlock triggered when request formulated, not granted!

Deadlock Recovery

- Blue screen & reboot
- Kill one/all deadlocked processes
  - Pick a victim (how?); Terminate; Repeat as needed
    - Can leave system in inconsistent state
- Proceed without the resource
  - Example: timeout on inventory check at Amazon
- Use transactions
  - Rollback & Restart
  - Need to pick a victim...

Summary

- Prevent
  - Negate one of the four necessary conditions
- Avoid
  - Schedule processes carefully
- Detect
  - Has a deadlock occurred?
- Recover
  - Kill or Rollback