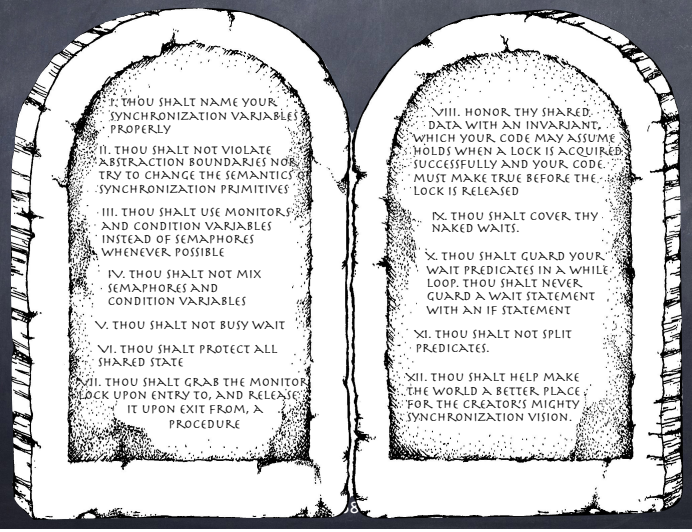
12 Commandments of Synchronization



Monitors in Python

```
class BK:
  def __init__(self):
    self.lock = Lock()
    self.hungrykid = Condition(self.lock)
    self.nBurgers= 0

 def kid_eat(self):
    with self.lock:
      while self.nBurgers == 0:
        self.hungrykid.wait()
      self.nBurgers = self.nBurgers - 1

 def make_burger(self):
   with self.lock:
      self.nBurgers = self.nBurgers + 1
          self.hungrykid.notify()
```

wait
- releases lock when called
- re-acquires lock when it returns

Monitors in "4410 Python": __init__

Python

```
class BK:
  def __init__(self):
      self.lock = Lock()
      self.hungrykid = Condition(self.lock)
      self.nBurgers= 0
```

4410 Python

```
from rvr import MP, MPthread

class BurgerKingMonitor(MP):
  def __init__(self):
    MP.__init__(self,None)
    self.lock = Lock("monitor lock")
    self.hungrykid = self.lock.Condition("hungry kid")
    self.nBurgers = self.Shared("num burgers", 0)
```


Monitors in "4410 Python": kid_eat

Python

```
def kid_eat(self):
  with self.lock:
    while self.nBurgers == 0:
      self.hungrykid.wait()
    self.nBurgers = self.nBurgers - 1
```

4410 Python

```
def kid_eat(self):
  with self.lock:
    while (self.nBurgers.read() == 0):
      self.hugryKid.wait()
    self.nBurgers.dec()
```

We do this for helpful feedback:
- from auto-grader
- from debugger

Look in the A2/doc directory for details and example code.

Readers/Writers

- Safety

$$(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge (\#r > 0) \Rightarrow (\#w = 0))$$

- What about fairness?
  - the last thread to live the critical section will give priority to writers
  - To implement this policy, one needs to keep track of waitingWriters, waitingReaders, activeWriters, and activeReaders

Readers/Writers

```
Monitor ReadersNWriters {

 int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
 Condition canRead, canWrite;

void BeginWrite()
  with monitor.lock:
    ++waitingWriters
    while (activeWriters >0 or activeReaders >0)
      canWrite.wait();
    --waitingWriters
    activeWriters = 1;


}
```


Readers/Writers

```
Monitor ReadersNWriters {

 int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
 Condition canRead, canWrite;

void BeginWrite()
  with monitor.lock:
    ++waitingWriters
    while (activeWriters >0 or activeReaders >0)
      canWrite.wait();
    --waitingWriters
    activeWriters = 1;

void EndWrite()
  with monitor.lock:
    activeWriters = 0
    if waitingWriters > 0
      canWrite.signal();
    else if waitingReaders > 0
      canRead.broadcast();
}
```


Readers/Writers

```
Monitor ReadersNWriters {

 int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
 Condition canRead, canWrite;

void BeginWrite()
  with monitor.lock:
    ++waitingWriters
    while (activeWriters >0 or activeReaders >0)
      canWrite.wait();
    --waitingWriters
    activeWriters = 1;

void EndWrite()
  with monitor.lock:
    activeWriters = 0
    if waitingWriters > 0
      canWrite.signal();
    else if waitingReaders > 0
      canRead.broadcast();
}
```

```
void BeginRead()
  with monitor.lock:
    ++waitingReaders
    while (activeWriters>0 or waitingWriters>0)
      canRead.wait();
    --waitingReaders
    ++activeReaders
```

Readers/Writers

```
Monitor ReadersNWriters {

 int waitingWriters=0, waitingReaders=0, activeReaders=0, activeWriters=0;
 Condition canRead, canWrite;

void BeginWrite()
  with monitor.lock:
    ++waitingWriters
    while (activeWriters >0 or activeReaders >0)
      canWrite.wait();
    --waitingWriters
    activeWriters = 1;

void EndWrite()
  with monitor.lock:
    activeWriters = 0
    if waitingWriters > 0
      canWrite.signal();
    else if waitingReaders > 0
      canRead.broadcast();
}
```

```
void BeginRead()
  with monitor.lock:
    ++waitingReaders
    while (activeWriters>0 or waitingWriters>0)
      canRead.wait();
    --waitingReaders
    ++activeReaders

void EndRead()
  with monitor.lock:
    --activeReaders;
    if (activeReaders==0 and waitingWriters>0)
      canWrite.signal();
```


Barrier Synchronization

- n threads divide work, run rounds of computation separated by barriers
  - Common paradigm in HPC
    - Create n threads and barrier
    - Each thread does round1()
    - barrier.chackin()
    - Each thread does round2()
    - barrier.checkin()

Checkin with one condition variable

```
self.allCheckedIn = Condition(self.lock)

def checkin():
  with self.lock:
    nArrived++
    if nArrived < nThreads:
      while nArrived < nThreads:
        allCheckedIn.wait()
    else:
      allCheckedIn.broadcast()
      nArrived = 0
```

What's wrong with this?

Deadlocks: Prevention, Avoidance, Detection, Recovery

System Model

- Exclusive (one-at-a-time) computer resources
  - CPU, printers, memory, locks, etc.
- Processes
  - Acquire resource
    - if resource is available, access is granted
    - if not, process is blocked
  - Use resource
  - Release resource

Deadlock

- A cycle of waiting among a set of threads
- A violation of liveness
  - T_1 acquire resource 1, waits for resource 2
  - T_2 acquires resource 2, waits for resource 1

Deadlock

- A cycle of waiting among a set of threads
- A violation of liveness
  - T_1 acquire resource 1, waits for resource 2
  - T_2 acquires resource 2, waits for resource 1

semaphore:
file_mutex = 1
printer_mutex = 1

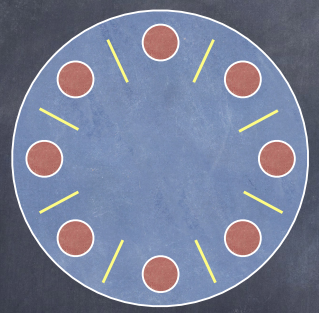
T_1

```
{
  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)
}
```

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])
```

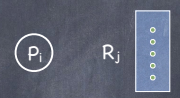
- N philosophers; N plates; N chopsticks
- If all philosophers grab right chopstick
  - deadlock!
- Need exclusive access to two chopsticks

Musings on Deadlock

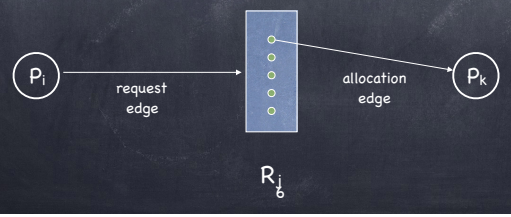
- Deadlock vs Starvation
  - Starvation: some thread's access to a resource is indefinitely postponed
  - Deadlock: circular waiting for resources
  - Deadlock implies Starvation, but not vice versa
- "Subject to deadlock" does not imply "Will deadlock"
  - Testing is not the solution
  - System must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

A Graph Theoretic Model of Deadlock

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



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

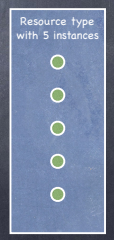


Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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

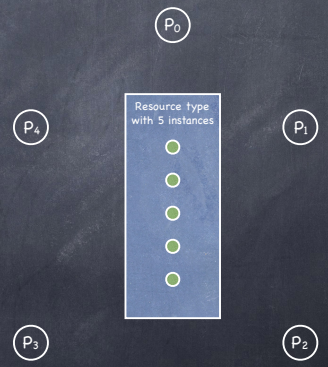
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



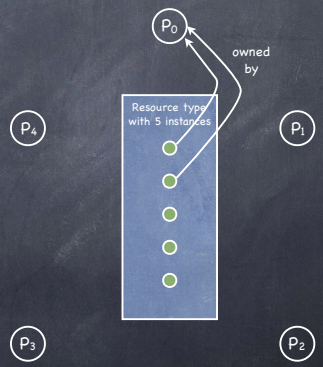
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



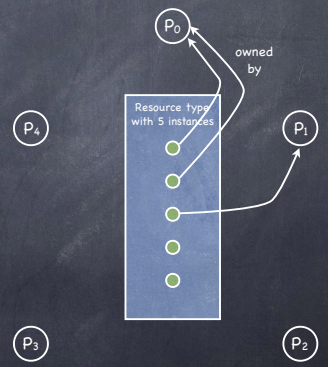
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



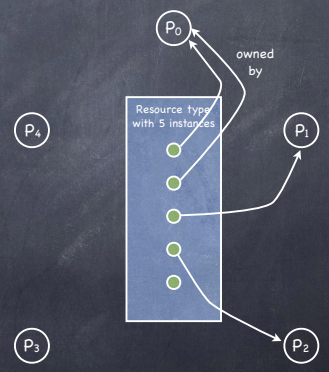
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



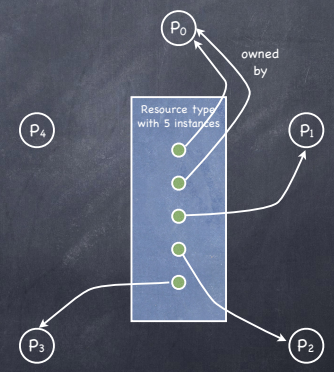
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



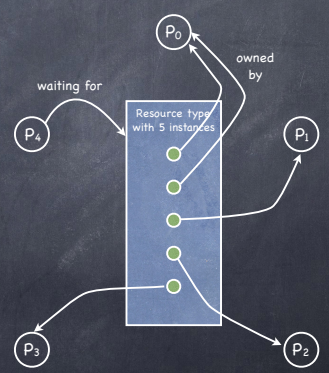
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



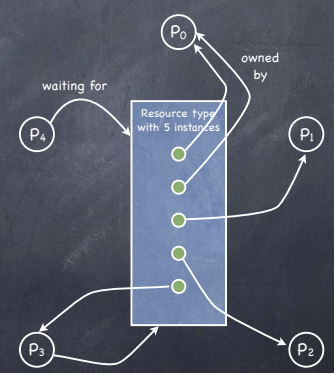
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



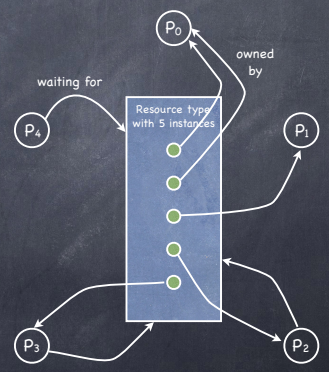
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



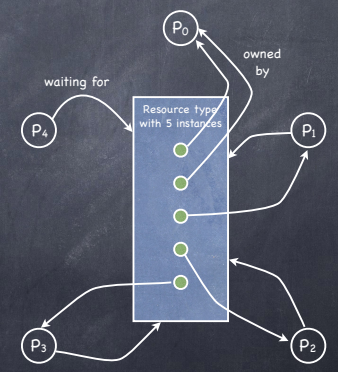
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



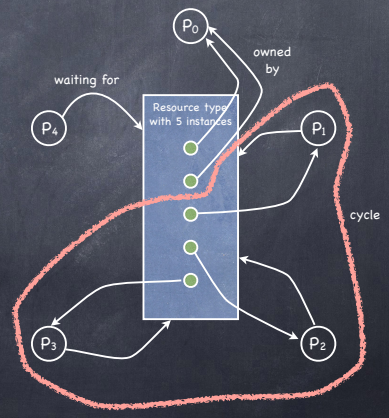
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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



Necessary Conditions for Deadlock

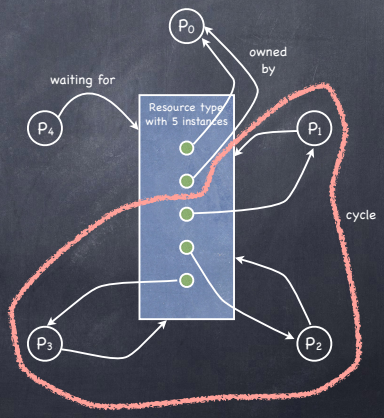
- Deadlock possible only if all four hold
  - Bounded resources
    - A finite number of threads can use a resource; resources are finite
  - No preemption
    - the resource is mine, MINE! (until I release it)
  - Hold & Wait
    - holds one resource while waiting for another
  - Circular waiting
 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



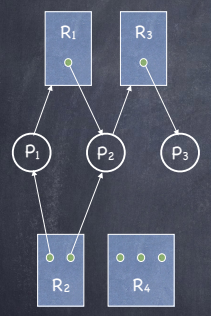
Necessary Conditions for Deadlock

- Deadlock possible only if all four hold
  - Bounded resources
    - 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

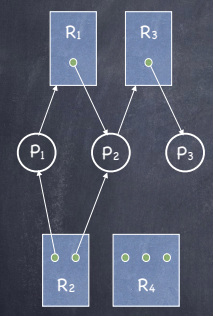


RAG Reduction



Deadlock?

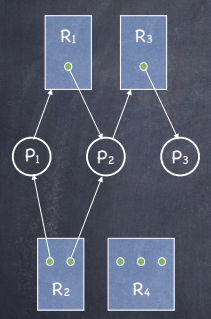
RAG Reduction



Deadlock?

NO! (no cycles)

RAG Reduction

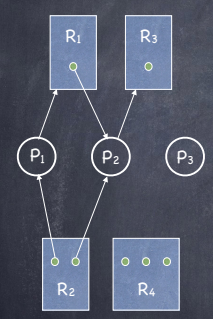


Deadlock?

NO! (no cycles)

Step 1: Satisfy P_3's requests

RAG Reduction

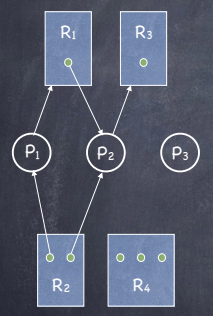


Deadlock?

NO! (no cycles)

Step 1: Satisfy P_3's requests

RAG Reduction

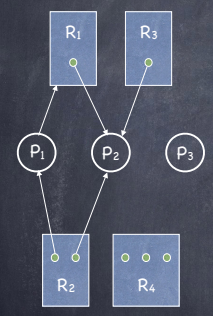


Deadlock?

NO! (no cycles)

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

RAG Reduction

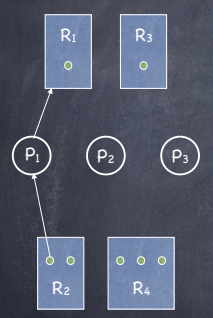


Deadlock?

NO! (no cycles)

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

RAG Reduction

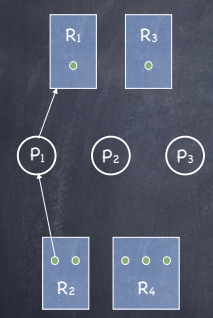


Deadlock?

NO! (no cycles)

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

RAG Reduction

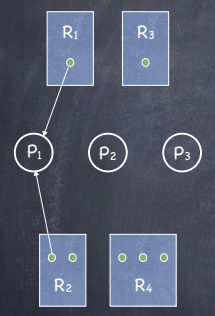


Deadlock?

NO! (no cycles)

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

RAG Reduction

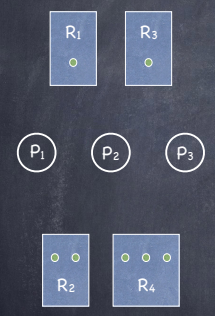


Deadlock?

NO! (no cycles)

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

RAG Reduction

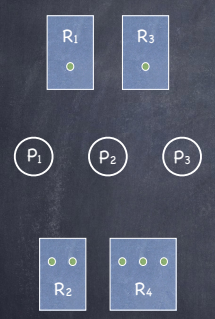


Deadlock?

NO! (no cycles)

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

RAG Reduction



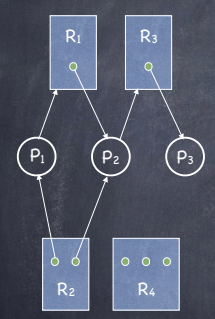
Deadlock?

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

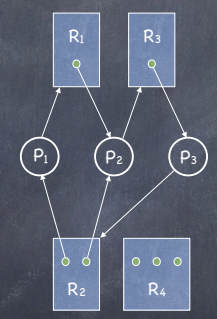


Deadlock?

NO! (no cycles)

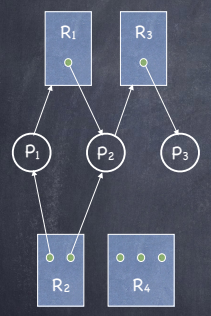
Step 1: Satisfy P_3's requests
Step 2: Satisfy P_2's requests
Step 3: Satisfy P_1's requests

Schedule $[P_3\ P_2\ P_1]$ completely eliminates edges!



Deadlock?

RAG Reduction

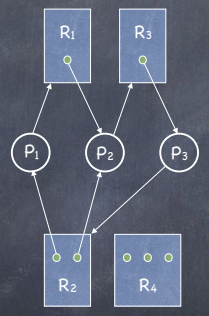


Deadlock?

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!

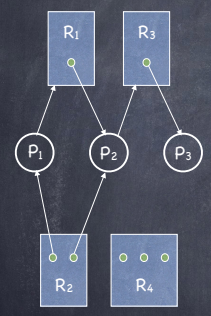


Deadlock?

RAG has a cycle

9

RAG Reduction

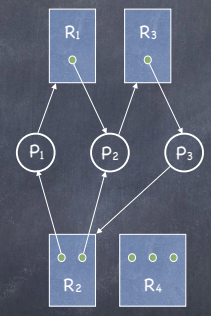


Deadlock?

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!



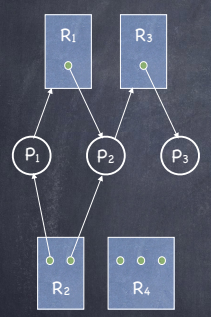
Deadlock?

RAG has a cycle

Cannot satisfy any of P_1, P_2, P_3 requests!

9

RAG Reduction

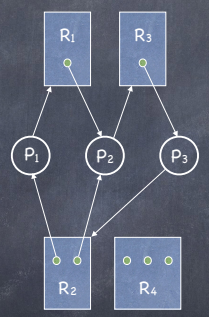


Deadlock?

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?

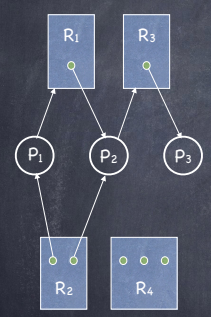
Yes!

RAG has a cycle

Cannot satisfy any of P_1, P_2, P_3 requests!

9

RAG Reduction

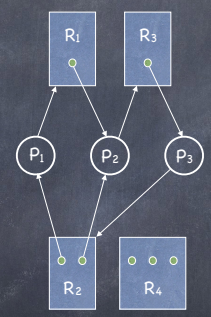


Deadlock?

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!



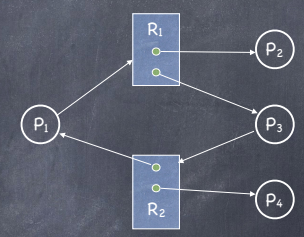
Deadlock?

Yes!

RAG has a cycle

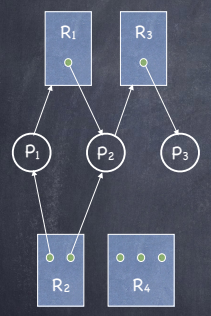
Cannot satisfy any of P_1, P_2, P_3 requests!

10



Deadlock?

RAG Reduction

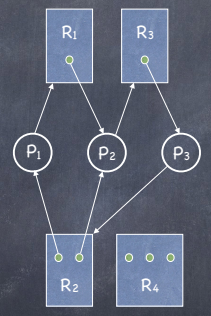


Deadlock?

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!

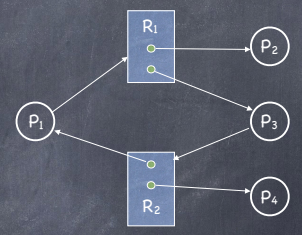


Deadlock?

Yes!

RAG has a cycle

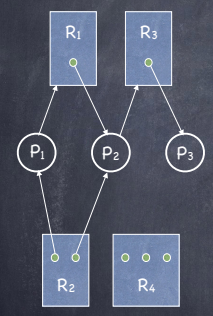
Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

RAG has a cycle

RAG Reduction

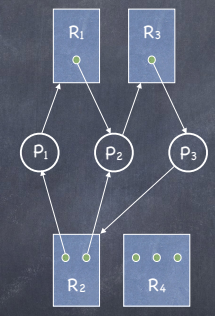


Deadlock?

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!

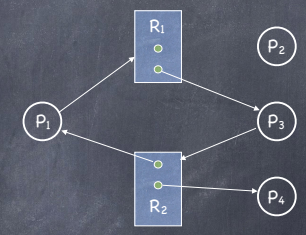


Deadlock?

Yes!

RAG has a cycle

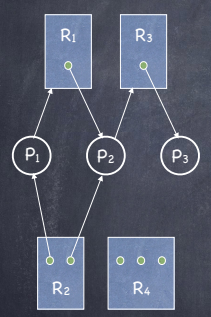
Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

RAG has a cycle

RAG Reduction

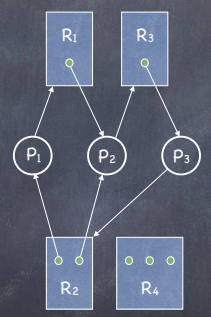


Deadlock?

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!

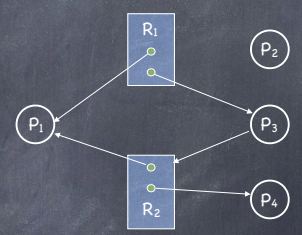


Deadlock?

Yes!

RAG has a cycle

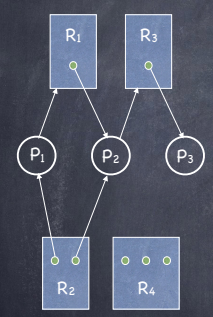
Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

RAG has a cycle

RAG Reduction

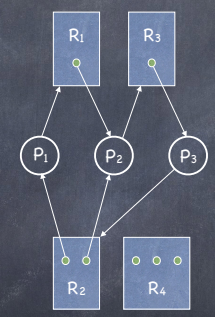


Deadlock?

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!

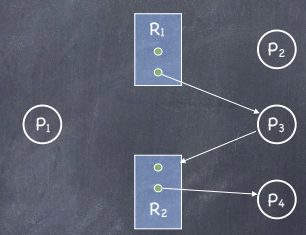


Deadlock?

Yes!

RAG has a cycle

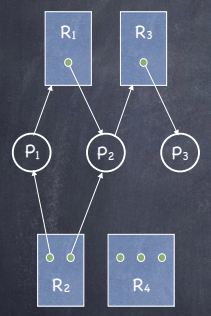
Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

RAG has a cycle

RAG Reduction

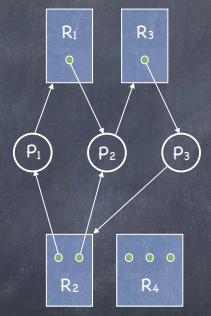


Deadlock?

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!

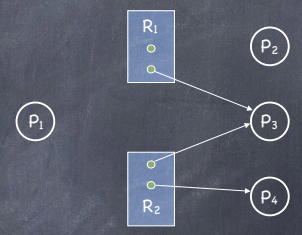


Deadlock?

Yes!

RAG has a cycle

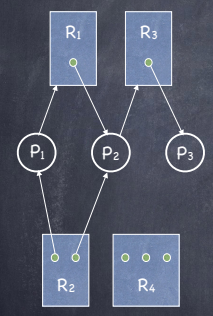
Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

RAG has a cycle

RAG Reduction

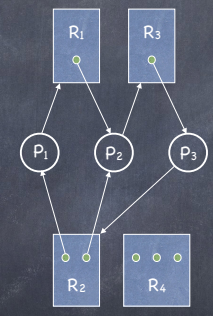


Deadlock?

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!

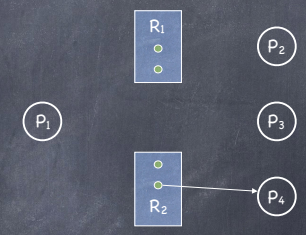


Deadlock?

Yes!

RAG has a cycle

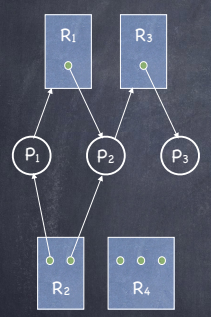
Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

RAG has a cycle

RAG Reduction

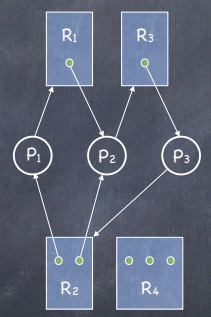


Deadlock?

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!

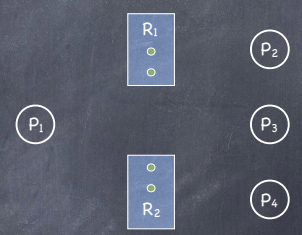


Deadlock?

Yes!

RAG has a cycle

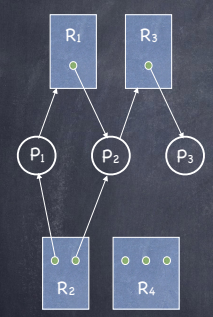
Cannot satisfy any of P_1, P_2, P_3 requests!



Deadlock?

RAG has a cycle

RAG Reduction

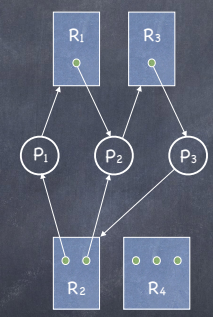


Deadlock?

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!

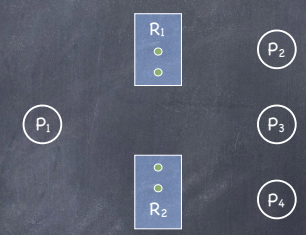


Deadlock?

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?

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

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?

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!

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?

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 "Mutual exclusion/Bounded Resources"
    - 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
      - A cycle needs edges to go from low to high, and then back to low (or to cycle on the same node)
      - Only a convention...

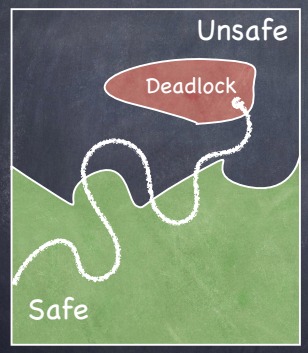
Preventing Philosopher's Deadlock

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

def __init__(mynum)
  self.id = mynum

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

- Bounded resources
- No preemption
- Hold & Wait
- Circular waiting

Can we prevent one of these conditions? Ideas?

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?

Why is George Bailey in trouble?

- George let his Building & Loan entered an unsafe state

Why is George Bailey in trouble?

- George let his Building & Loan entered an unsafe state
  - If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt

Why is George Bailey in trouble?

- George let his Building & Loan entered an unsafe state
  - If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt
  - If lenders reduced or delayed their requests, all would be well!

Why is George Bailey in trouble?

- George let his Building & Loan entered an unsafe state
  - If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt
  - If lenders reduced or delayed their requests, all would be well!
    - spoiler alert: this is exactly what happens…

Why is George Bailey in trouble?

- George let his Building & Loan entered an unsafe state
  - If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt
  - If lenders reduced or delayed their requests, all would be well!
    - spoiler alert: this is exactly what happens…
  - George relied on the bad execution in which "everyone wants everything now" not happening

Why is George Bailey in trouble?

- George let his Building & Loan entered an unsafe state
  - If all his customers ask at the same time to have back all the money they have lent, he is going bankrupt
  - If lenders reduced or delayed their requests, all would be well!
    - spoiler alert: this is exactly what happens…
  - George relied on the bad execution in which "everyone wants everything now" not happening
- Still begs the question:
  - Can resources be allocated so that the system always transitions among safe states?

Proactive Responses to Deadlock: Avoidance

The Banker's Algorithm

E.W. Dijkstra & N. Habermann

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

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)

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 $[P_1, P_2, \ldots, P_n]$ 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

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 $[P_1, P_2, \ldots, P_n]$ 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?

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 $[P_1, P_2, \ldots, P_n]$ 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

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 $[P_1, P_2, \ldots, P_n]$ 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

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 $[P_1, P_2, \ldots, P_n]$ 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

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 $[P_1, P_2, \ldots, P_n]$ 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

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 $[P_1, P_2, \ldots, P_n]$ 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_2's needs

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 $[P_1, P_2, \ldots, P_n]$ 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 $[P_1, P_2, \ldots, P_n]$ 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?

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 $[P_1, P_2, \ldots, P_n]$ 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 = 2

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

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 $[P_1, P_2, \ldots, P_n]$ 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 = 2

Process	Max	Holds	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

- 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 $P_1, P_2, \ldots, P_n$ 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

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

- Is this a safe state?

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

−

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

- Is this a safe state?

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	7	5	0
P_3	2	3	5	6
P_4	0	6	5	2
P_5	0	6	5	6

−

Holds

	R_1	R_2	R_3	R_4
P_1	0	0	1	2
P_2	1	0	0	0
P_3	1	3	5	3
P_4	0	6	3	2
P_5	0	0	1	4

Available

R_1	R_2	R_3	R_4
1	5	2	0

Needs

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

- Is this a safe state?

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

- Is this a safe state?
  - While safe permutation does not include all processes:
    - Is there a P_i such that $Needs_i \leq Avail$?
      - 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		R_1	R_2	R_3	R_4
P_1	0	0	1	2	P_1	0	0	1	2		1	5	2	0	P_1	0	0	0	0
P_2	1	7	5	0	P_2	1	0	0	0						P_2	0	7	5	0
P_3	2	3	5	6	P_3	1	3	5	3						P_3	1	0	0	3
P_4	0	6	5	2	P_4	0	6	3	2						P_4	0	0	2	0
P_5	0	6	5	6	P_5	0	0	1	4						P_5	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 $Needs_i \leq Avail$?
      - 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		R_1	R_2	R_3	R_4
P_1	0	0	1	2	P_1	0	0	1	2		1	5	2	0	P_1	0	0	0	0
P_2	1	7	5	0	P_2	1	0	0	0						P_2	0	7	5	0
P_3	2	3	5	6	P_3	1	3	5	3						P_3	1	0	0	3
P_4	0	6	5	2	P_4	0	6	3	2						P_4	0	0	2	0
P_5	0	6	5	6	P_5	0	0	1	4						P_5	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		R_1	R_2	R_3	R_4
P_1	0	0	1	2	P_1	0	0	1	2		1	5	2	0	P_1	0	0	0	0
P_2	1	7	5	0	P_2	1	0	0	0						P_2	0	7	5	0
P_3	2	3	5	6	P_3	1	3	5	3						P_3	1	0	0	3
P_4	0	6	5	2	P_4	0	6	3	2						P_4	0	0	2	0
P_5	0	6	5	6	P_5	0	0	1	4						P_5	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	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	1	2	2	1	0	0	0	0	0	0
P_2	1	7	5	0	0	4	2	0					1	3	3	0
P_3	2	3	5	6	1	3	5	3					1	0	0	3
P_4	0	6	5	2	0	6	3	2					0	0	2	0
P_5	0	6	5	6	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	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	1	2	2	1	0	0	0	0	0	0
P_2	1	7	5	0	0	4	2	0					1	3	3	0
P_3	2	3	5	6	1	3	5	3					1	0	0	3
P_4	0	6	5	2	0	6	3	2					0	0	2	0
P_5	0	6	5	6	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	R_1	R_2	R_3	R_4
P_1	0	0	1	2	0	0	1	2	2	1	0	0	0	0	0	0
P_2	1	7	5	0	0	4	2	0					1	3	3	0
P_3	2	3	5	6	1	3	5	3					1	0	0	3
P_4	0	6	5	2	0	6	3	2					0	0	2	0
P_5	0	6	5	6	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 know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	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 know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	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

Detecting Deadlock

- 5 processes, 3 resources. We no longer know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	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?

Detecting Deadlock

- 5 processes, 3 resources. We no longer know Max.

	Holds			Available			Pending		
	R_1	R_2	R_3	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 scree & 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