ANNOUNCEMENTS

- Recitation for this week will cover required material (Barrier Synchronization) assigned in the reading (C. 21 of the Harmony book.
- Recitation recording will be available!
- Released homework for CS5410 students
- Spring '24: CS5220: Applied High-Performance and Parallel Computing

  Overview of computer architecture and memory hierarchy, performance basics, parallel programming models, and survey of parallel machines. Parallel programming languages, vectorizing compilers, parallel libraries and toolboxes, overview of modern parallel algorithms.
- Spring '24: CS5414: Principles of Distributed Computing

Previously, on CS4410...

Necessary conditions for deadlock

Deadlock only if they all hold

1. Bounded resources
   - Acquire can block invoker
2. No preemption
   - the resource is mine, MINE! (until I release it)
3. Wait while holding
   - holds one resource while waiting for another
4. Circular waiting
   - P_i waits for P_{i+1} and holds a resource requested by P_{i-1}
   - sufficient if one instance of each resource

DAG Reduction

- Reduction Algorithm
  - Find a node with no outgoing edges
    - Erase any edges coming into it
    - Repeat until no such node
- Intuition: Node with no outgoing edges is not waiting on any resource
  - It will eventually finish and release its resources
  - Processes waiting for <u>those</u> resources will be able to acquire them and will no longer be waiting!

Erase all edges $\Longleftrightarrow$ Graph has no cycles

Edges remain $\Longleftrightarrow$ Deadlock

Deadlock Prevention: Negate ①

- Eliminate "Acquire can block invoker/bounded resources"
  - Make resources sharable without locks
    - Wait-free synchronization
    - The Harmony book (Chapter 24) has examples of non-blocking data structures
  - Have sufficient resources available, so acquire never delays (duh!)
    - E.g., use an unbounded queue, or make sure that queue is "large enough"

Deadlock Prevention: Negate ②

- Allow preemption
  - Requires mechanisms to save/restore resource state
    - multiplexing (registers, memory, etc). VS.
    - undo/redo (database transaction processing)
  - Allow OS to preempt resources of waiting processes
  - Allow OS to preempt resources of requesting processes

Deadlock Prevention: Negate ③

- Eliminate Hold & Wait
  - Don't hold resource while waiting for others
    - Rewrite code

```
def foo():
  acquire(?mutex);
  doSomeStuff();
  bar();
  doOtherStuff();
  release(?mutex);
```

code in some other module that may acquire more locks

→

```
def foo():
  acquire(?mutex);
  doSomeStuff();
  release(?mutex);
  bar();
  acquire(?mutex);
  doOtherStuff();
  release(?mutex);
```

Q: If bar() does not access shared variables and does not need a lock, are these the same?

Deadlock Prevention: Negate ③

- Eliminate Hold & Wait
  - Don't hold resource while waiting for others
    - Rewrite code

```
def foo():
  acquire(?mutex);
  doSomeStuff();
  bar();
  doOtherStuff();
  release(?mutex);
```

code in some other module that may acquire more locks

→

```
def foo():
  acquire(?mutex);
  doSomeStuff();
  release(?mutex);
  bar();
  acquire(?mutex);
  doOtherStuff();
  release(?mutex);
```

A: No! In the code on the right, the state that the mutex protects can change between doSomeStuff and doOtherStuff

Deadlock Prevention: Negate ③

- Eliminate Hold & Wait
  - Don't hold resource while waiting for others
    - Rewrite code
    - Request all resources before execution begins... but
      - Processes don't know what they need
      - Starvation (if waiting on popular resources)
      - Low utilization (if resources needed only briefly)
    - Release all resources before asking new ones
      - Still has the last two problems...

Deadlock Prevention: Negate ④

- Eliminate circular waiting
  - Single lock for the entire system?
  - Impose a total order on the sequence in which different types of resources can be acquired
    - Each resource type is assigned to a level
    - Makes cycles impossible, since cycles would have to go from low to high level resources, and then back to low
    - Can be relaxed to a strict partial order* if all resources "of the same level" are acquired together

*a binary relation < that is:

1. irreflexive: not $a < a$
2. asymmetric: if $a < b$, then not $b < a$
3. transitive: if $a < b$ and $b < c$, then $a < c$

Havender's Scheme (OS/360)

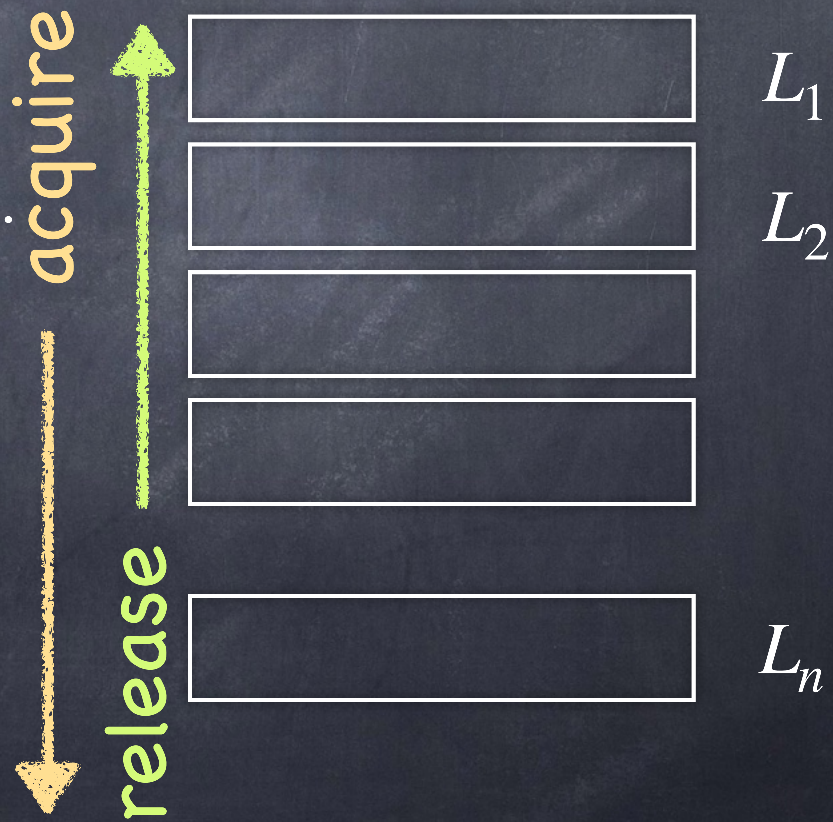
Hierarchical Resource Allocation

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 (and holding) 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 $j > i$.

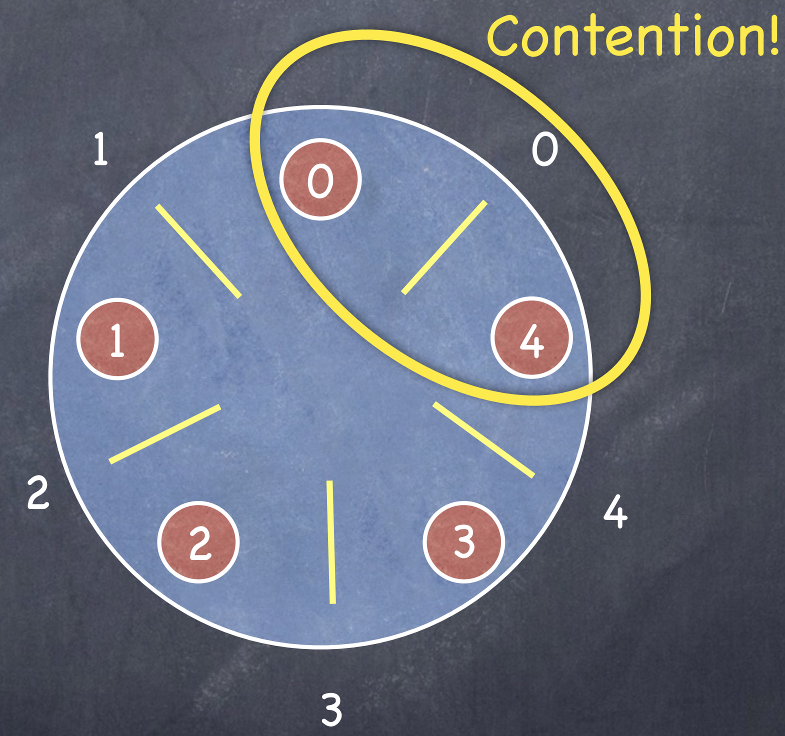


Example of allowed sequence:

1. acquire(W@L_1, X@L_1)
2. acquire(Y@L_3)
3. release(Y@L_3)
4. acquire(Z@L_2)

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)

Ordering Resources in Harmony

```
if left < right:
    synch.acquire(?forks[left])
    synch.acquire(?forks[right])
else:
    synch.acquire(?forks[right])
    synch.acquire(?forks[left])
```

or

```
synch.acquire(?forks[min(left, right)])
synch.acquire(?forks[max(left, right)])
```

Simultaneous Acquisition in Harmony

```
mutex = synch.Lock()
forks = [False,] * N
conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(?conds[left], ?mutex)
                if forks[right]:
                    synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

one mutex

initially, no forks are held

one boolean and one CV per fork

if left fork is used, wait until free

if right fork is used, wait until free

Wait for both forks and then grab them both!

Release both forks

Simultaneous Acquisition in Harmony

```
    mutex = synch.Lock()
    forks = [False,] * N
    conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left] or forks[right]:
                if forks[left]:
                    synch.wait(?conds[left], ?mutex)
                if forks[right]:
                    synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

Wait for both forks to be available

Simultaneous Acquisition in Harmony

```
    mutex = synch.Lock()
    forks = [False,] * N
    conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left]:
                synch.wait(?conds[left], ?mutex)
            while forks[right]:
                synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

Wait for left fork then wait for right fork

Wouldn't this be just as good?

Simultaneous Acquisition in Harmony

```
    mutex = synch.Lock()
    forks = [False,] * N
    conds = [synch.Condition(?mutex),] * N

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            synch.acquire(?mutex)
            while forks[left]:
                synch.wait(?conds[left], ?mutex)
            while forks[right]:
                synch.wait(?conds[right], ?mutex)
            assert not (forks[left] or forks[right])
            forks[left] = forks[right] = True
            synch.release(?mutex)
            # dine
            synch.acquire(?mutex)
            forks[left] = forks[right] = False
            synch.notify(?conds[left]);
            synch.notify(?conds[right])
            synch.release(?mutex)
            # think
```

Run it through Harmony!

Wait for left fork then wait for right fork

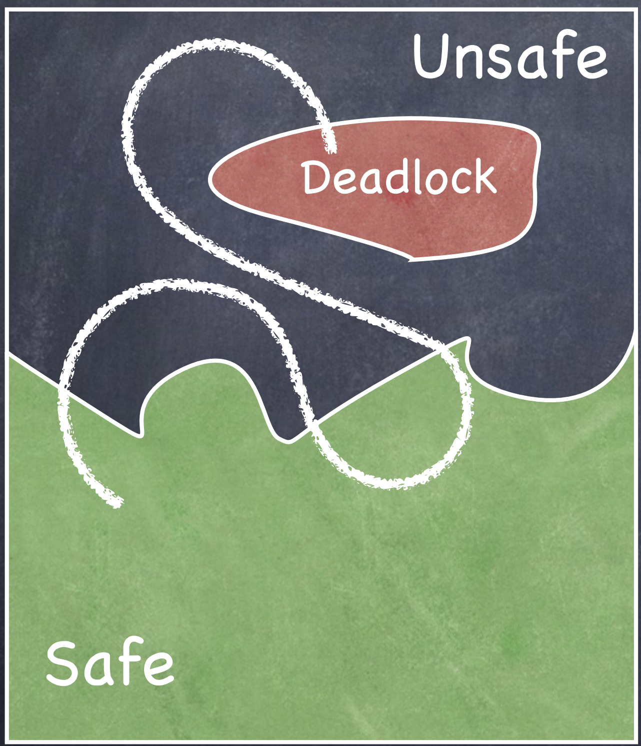
NO!

Avoiding Deadlock: The Banker's Algorithm

E.W. Dijkstra & N. Habermann

- Sum of max resources needs can exceed total available resources
- Acquiring all resources at once can be inefficient!
- Allow to parcel out resources incrementally as long as
  - there exists a schedule of loan fulfillments such that
    - all clients receive their maximal loan
    - build their house
    - pay back all the loan

Living dangerously: Safe, Unsafe, Deadlocked



A system's trajectory through its state space

- Safe: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests
  - no deadlock as long as system can enforce that safe schedule!
- Unsafe: There exists a set of (pending and future) resource requests that leads to a deadlock, independent of the schedule in which requests are processed
  - unlucky set of requests can force deadlock
- Deadlocked: The system has at least one deadlock

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 resulting 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

If granted, is the resulting state 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 — $MaxClaim_i[j] = Max_{ij}$
- $Holds_{ij}$ = current allocation of R_j held by P_i
  - $HasNow_i$ = Vector of size m — $HasNow_i[j] = Holds_{ij}$
- Available = Vector of size m — 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

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

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

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?

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

	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

- P_2 wants to change its holdings to 0 4 2 0

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	0	4	2	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
2	1	0	0

Needs

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

- P_2 wants to change its holdings to 0 4 2 0
- Safe? Reduce P_1

An Example

- 5 processes, 4 resources

Max

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
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	0	0
P_2	0	4	2	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
2	1	1	2

Needs

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

- P_2 wants to change its holdings to 0 4 2 0
- Safe? Reduce P_1; can't reduce any further

Unsafe!

If all processes were to ask together all the resources they may need, **deadlock!**

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.

Holds

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

Available

R_1	R_2	R_3
0	0	0

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

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

Available

R_1	R_2	R_3
0	0	0

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
P_1	0	1	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	0	3

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

	R_1	R_2	R_3
→ P_1	0	1	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	0	3

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

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

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

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

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

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

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

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

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

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

Available

R_1	R_2	R_3
7	2	4

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

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

Available

R_1	R_2	R_3
7	2	4

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

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

Available

R_1	R_2	R_3
7	2	6

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Yes, there is a safe schedule!

Detecting Deadlock

- 5 processes, 3 resources.

Holds

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

Available

R_1	R_2	R_3
0	0	0

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Yes, there is a safe schedule!

but it is not a safe state!

Detecting Deadlock

- 5 processes, 3 resources.

Holds

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

Available

R_1	R_2	R_3
0	0	0

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- But can determine if the state has a deadlock
  - Given the set of pending requests, is there a safe sequence? If no, deadlock

Detecting Deadlock

- 5 processes, 3 resources.

Holds

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

Available

R_1	R_2	R_3
0	0	0

Pending

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

- Cannot determine whether the state is safe
  - I need Max and Needs for that!
- Without Max, can we avoid deadlock by delaying granting requests?
  - NO! 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 (if application permits)
  - 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