Deadlocks: Part I Prevention and Avoidance

Yi Shi

Fall 2018

Xi'an Jiaotong University

Review: Motivation for Monitors and Condition Variables

- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - Both mutual exclusion and scheduling constraints
    - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language
- The lock provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free

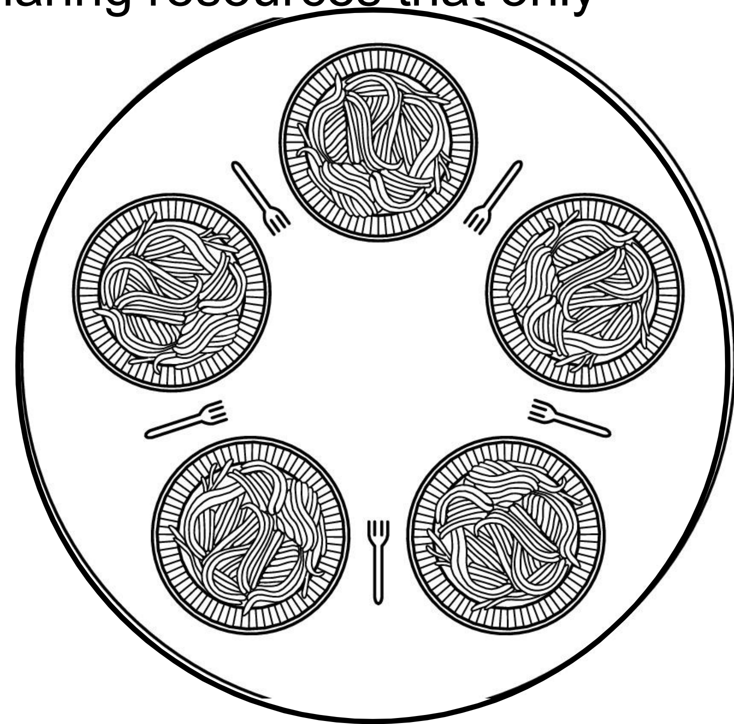
Review: Condition Variables

- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- Operations:
  - `Wait()`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!

Dining Philosophers and the Deadlock Concept

Dining Philosopher’s

- Dijkstra
  - A problem that was invented to illustrate a different aspect of communication
  - Our focus here is on the notion of sharing resources that only one user at a time can own



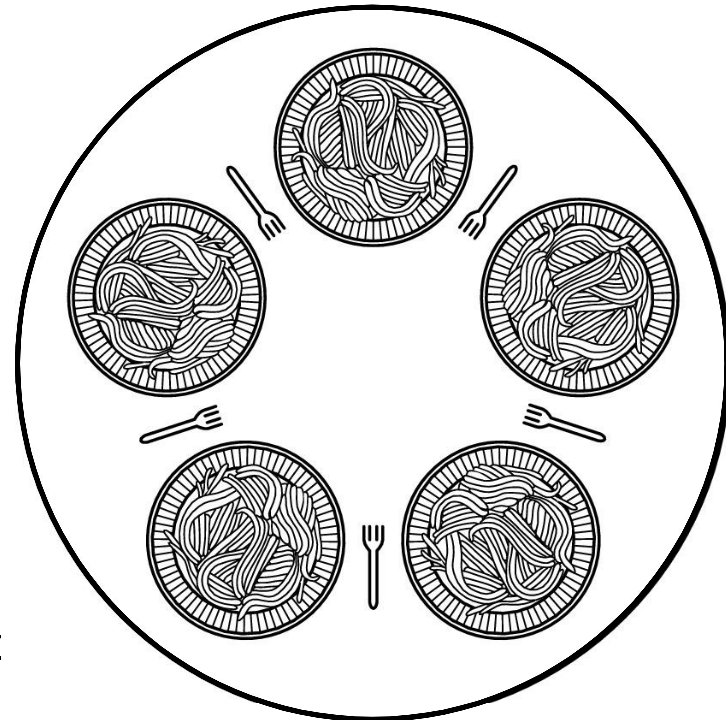
- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time

Idea is to capture the concept of multiple processes competing for limited resources

Coding our flawed solution?

```
Shared: semaphore fork[5];
Init: fork[i] = 1 for all i=0 .. 4

Philosopher i

do {
    P(fork[i]);
    P(fork[i+1]);

    /* eat */

    V(fork[i]);
    V(fork[i+1]);

    /* think */
} while(true);
```

Oops! Subject to deadlock if they all pick up their “left” fork simultaneously!



Goals for Today

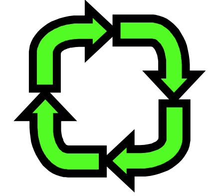
- Discussion of Deadlocks
- Conditions for its occurrence

System Model

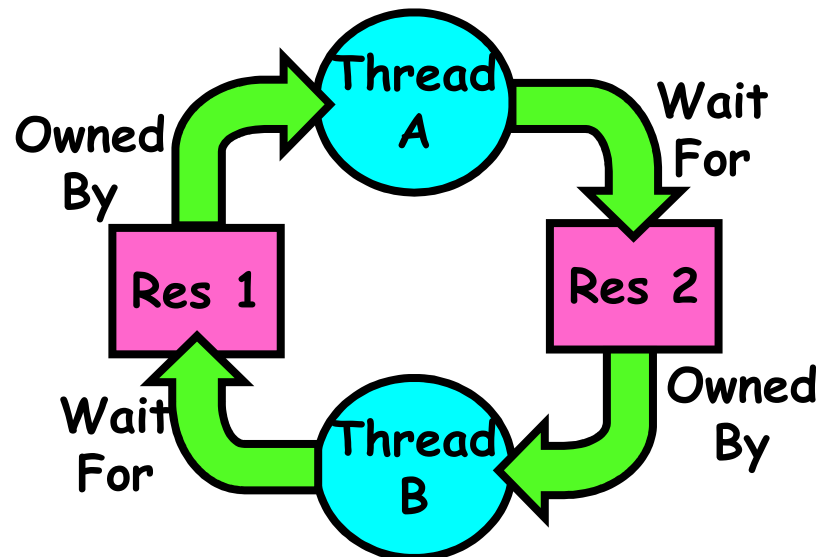
- There are non-shared computer resources
  - Maybe more than one instance
  - Printers, Semaphores, Tape drives, CPU
- Processes need access to these resources
  - Acquire resource
    - If resource is available, access is granted
    - If not available, the process is blocked
  - Use resource
  - Release resource
- Undesirable scenario:
  - Process A acquires resource 1, and is waiting for resource 2
  - Process B acquires resource 2, and is waiting for resource 1

  $\Rightarrow$ Deadlock!

Starvation vs Deadlock



- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock $\Rightarrow$ Starvation but not vice versa
    - Starvation can end (but doesn't have to)
    - Deadlock can't end without external intervention

For example: Semaphores

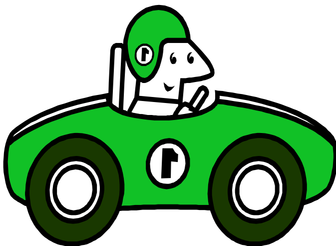
```
semaphore:    mutex1 = 1    /* protects resource 1 */
              mutex2 = 1    /* protects resource 2 */
```

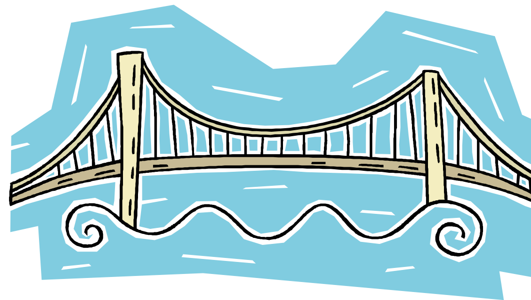
```
Process A code:
 {
    /* initial compute */
   P(mutex1)
   P(mutex2)

  /* use both resources */

   V(mutex2)
   V(mutex1)
}
```

```
Process B code:
 {
    /* initial compute */
   P(mutex2)
   P(mutex1)

  /* use both resources */

   V(mutex2)
   V(mutex1)
}
```

Deadlocks

- Definition: Deadlock exists among a set of processes if
  - Every process is waiting for an event
  - This event can be caused only by another process in the set
    - Event is the acquire of release of another resource





One-lane bridge

Four Conditions for Deadlock

- Coffman et. al. 1971
- Necessary conditions for deadlock to exist:
  - **Mutual Exclusion**
    - At least one resource must be held is in non-sharable mode
  - **Hold and wait**
    - There exists a process holding a resource, and waiting for another
  - **No preemption**
    - Resources cannot be preempted
  - **Circular wait**
    - There exists a set of processes $\{P_1, P_2, \ldots P_N\}$, such that
      - P_1 is waiting for P_2, P_2 for P_3, …. and P_N for P_1

All four conditions must hold for deadlock to occur

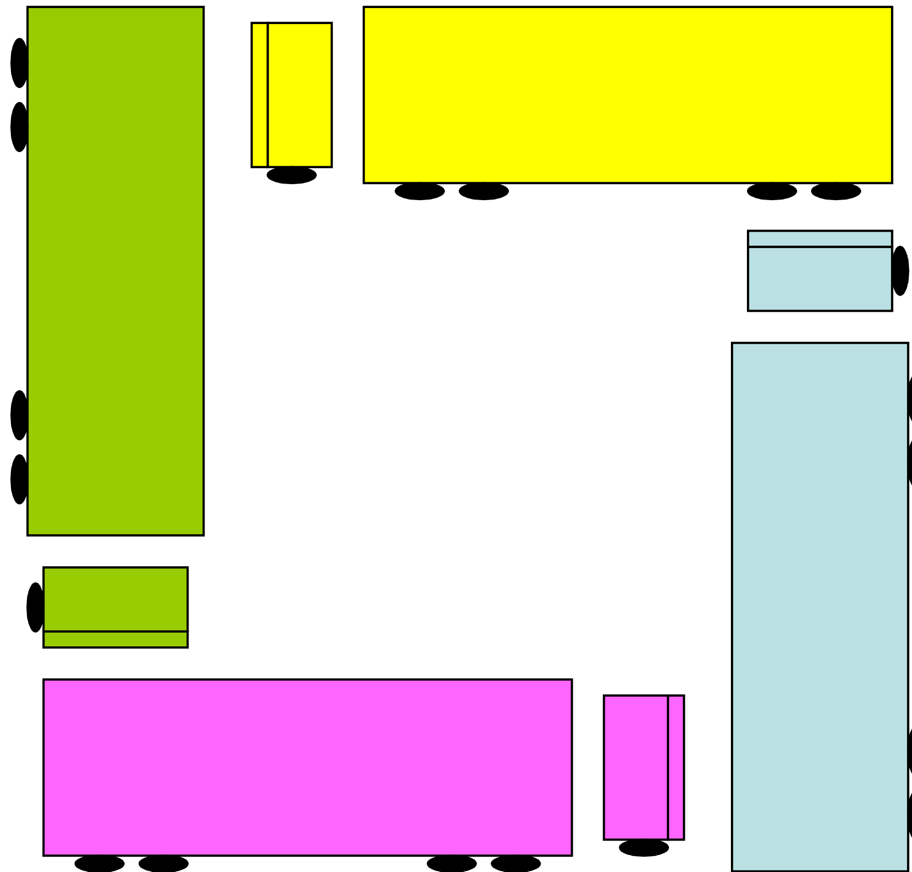
Real World Deadlocks?

- Truck A has to wait for truck B to move

- Not deadlocked

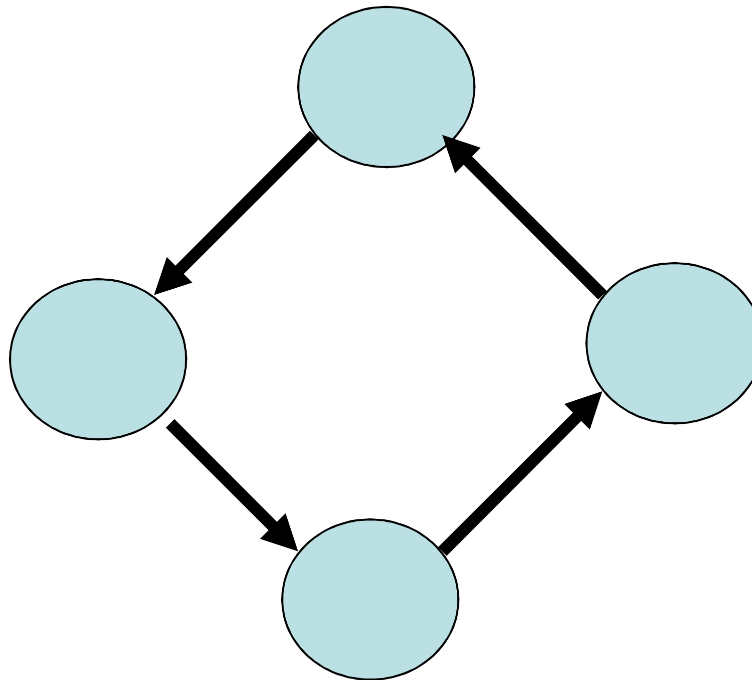
Real World Deadlocks?

- Gridlock



Real World Deadlocks?

- Gridlock



Testing for deadlock

- Steps
  - Collect “process state” and use it to build a graph
    - Ask each process “are you waiting for anything”?
    - Put an edge in the graph if so
  - We need to do this in a single instant of time, not while things might be changing
- Now need a way to test for cycles in our graph

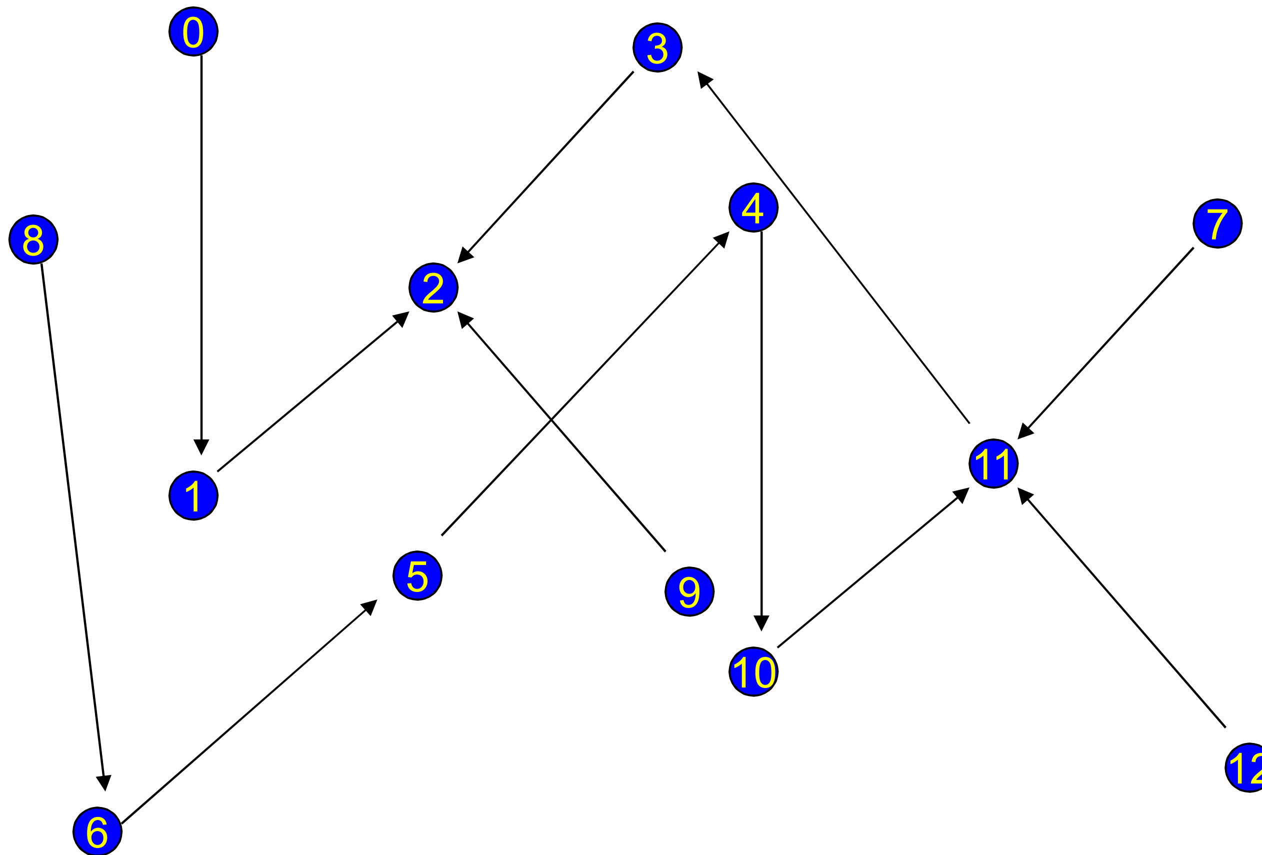
Testing for deadlock

- One way to find cycles
  - Look for a node with no outgoing edges
  - Erase this node, and also erase any edges coming into it
    - Idea: This was a process people might have been waiting for, but it wasn't waiting for anything else
  - If (and only if) the graph has no cycles, we'll eventually be able to erase the whole graph!
- This is called a graph reduction algorithm

Graph reduction example

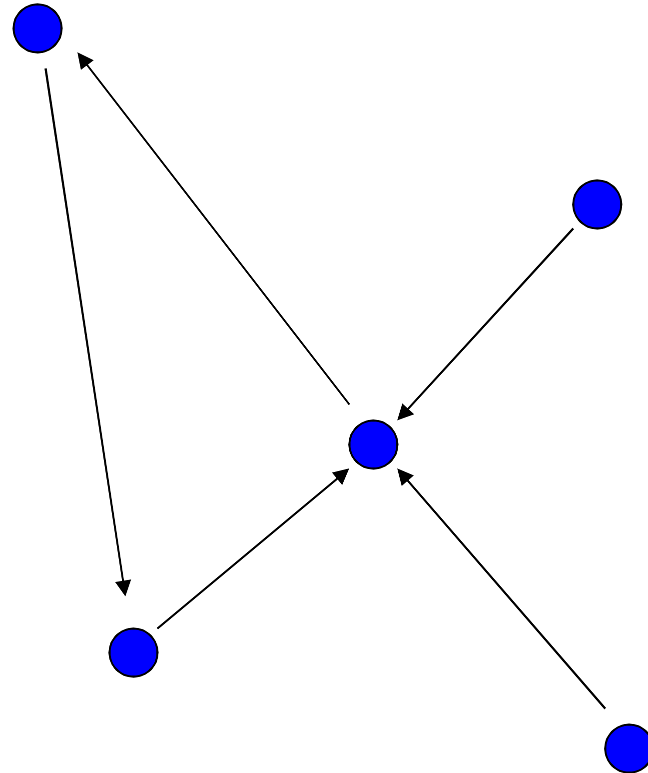
This graph can be “fully reduced”, hence there was no deadlock at the time the graph was drawn.

Obviously, things could change later!



Graph reduction example

- This is an example of an “irreducible” graph
- It contains a cycle and represents a deadlock, although only some processes are in the cycle

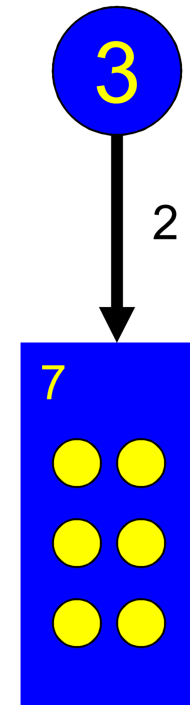


What about “resource” waits?

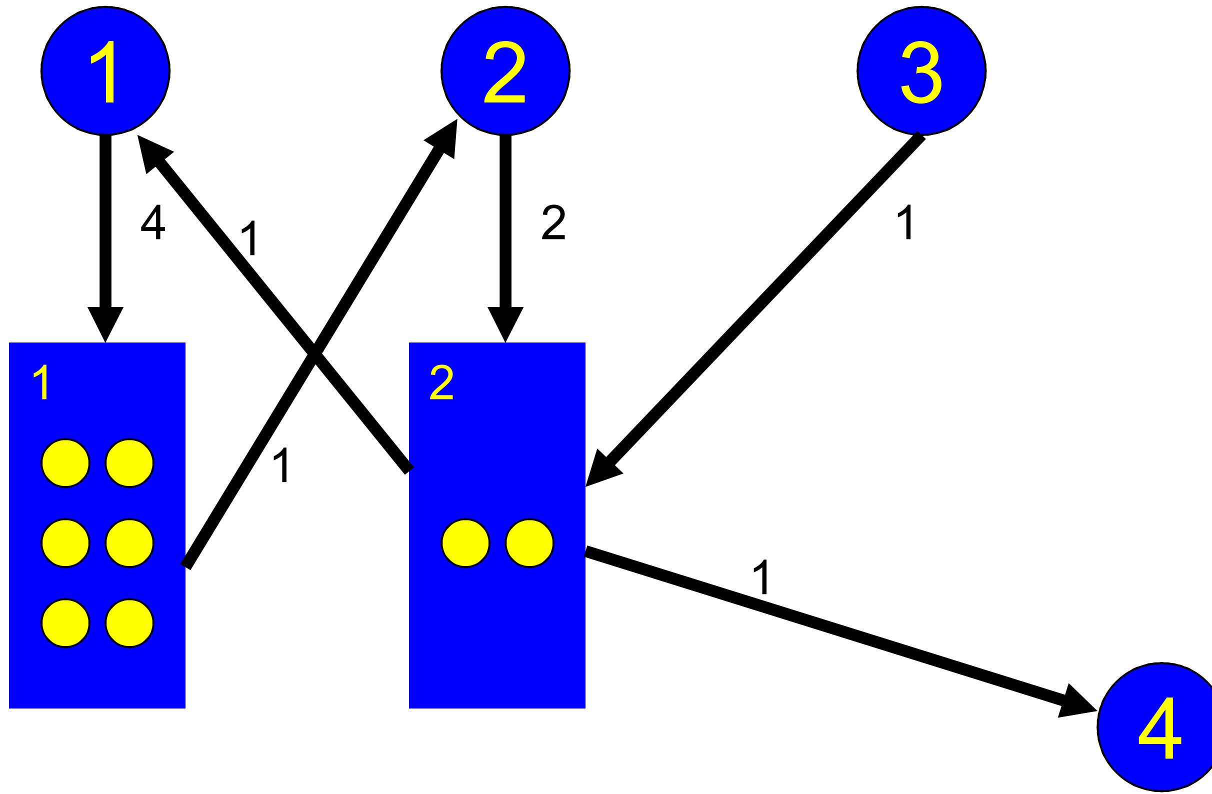
- When dining philosophers wait for one-another, they don’t do so directly
  - Erasmus doesn’t “wait” for Ptolemy
- Instead, they wait for resources
  - Erasmus waits for a fork… which Ptolemy exclusively holds
- Can we extend our graphs to represent resource wait?

Resource-wait graphs

- We'll use two kinds of nodes
- A process: P_3 will be represented as circle:
- A resource: R_7 will be represented as rectangle:
  - A resource often has multiple identical units, such as "blocks of memory"
  - Represent these as circles in the box
- Arrow from a process to a resource: "I want *k* units of this resource." Arrow to a process: this process holds *k* units of the resource
  - P_3 wants 2 units of R_7



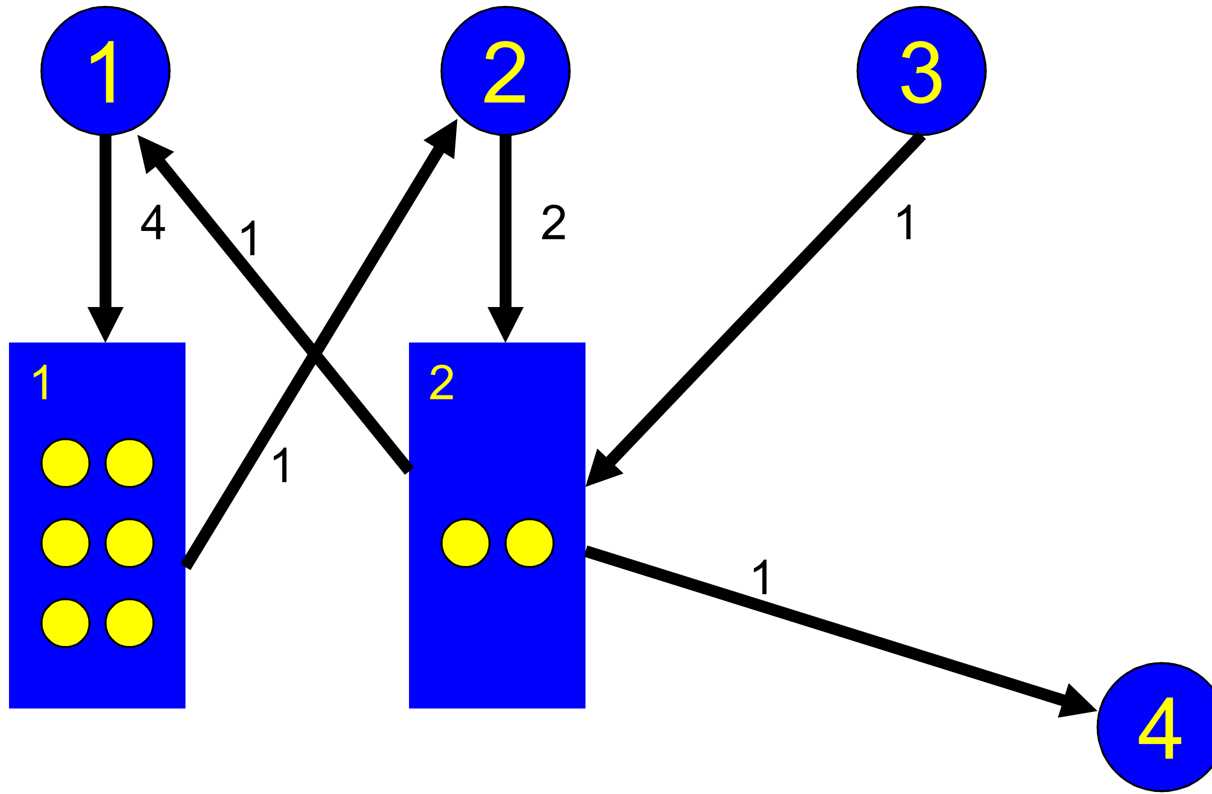
Resource-wait graphs



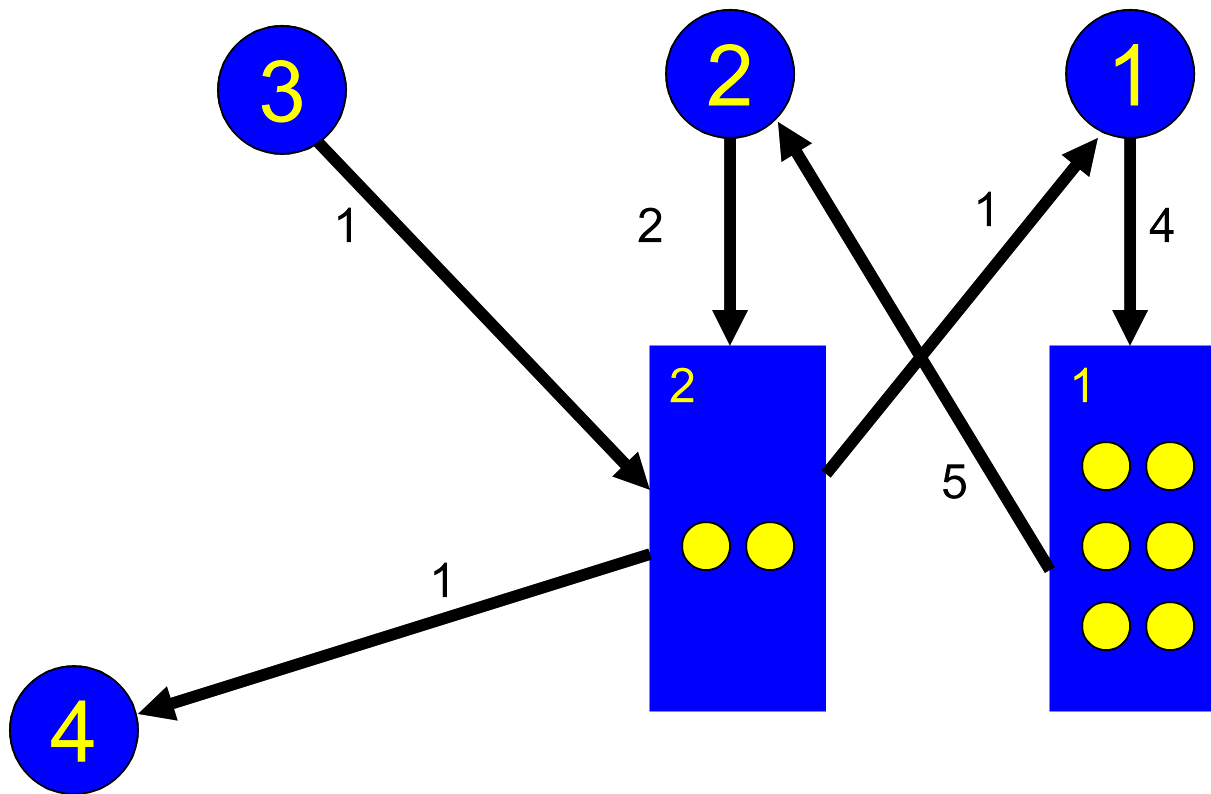
Reduction rules?

- Find a process that can have all its current requests satisfied (e.g. the “available amount” of any resource it wants is at least enough to satisfy the request)
- Erase that process (in effect: grant the request, let it run, and eventually it will release the resource)
- Continue until we either erase all the process nodes or have an irreducible component. In the latter case we’ve identified a deadlock

This graph is reducible: The system is not deadlocked



This graph is not reducible: The system is deadlocked



Comments

- It isn’t common for systems to actually implement this kind of test
- However, we’ll later use a version of the resource reduction graph as part of an algorithm called the “Banker’s Algorithm”
- Idea is to schedule the granting of resources so as to avoid potentially deadlock states

Some questions you might ask

- Does the order in which we do the reduction matter?
  - Answer: No. The reason is that if a node is a candidate for reduction at step i, and we don't pick it, it remains a candidate for reduction at step i+1
  - Thus eventually, no matter what order we do it in, we'll reduce by every node where reduction is feasible

Some questions you might ask

- If a system is deadlocked, could this go away?
  - No, unless someone kills one of the threads or something causes a process to release a resource
  - Many real systems put time limits on “waiting” precisely for this reason. When a process gets a timeout exception, it gives up waiting and this also can eliminate the deadlock
  - But that process may be forced to terminate itself because often, if a process can’t get what it needs, there are no other options available!

Some questions you might ask

- Suppose a system isn't deadlocked at time T.
- Can we assume it will still be free of deadlock at time T+1?
  - No, because the very next thing it might do is to run some process that will request a resource…

    … establishing a cyclic wait

    … and causing deadlock

Dealing with Deadlocks

1. Reactive Approaches:
   - Periodically check for evidence of deadlock
     - For example, using a graph reduction algorithm
   - Then need a way to recover
     - Could blue screen and reboot the computer
     - Could pick a “victim” and terminate that thread
       - But this is only possible in certain kinds of applications
       - Basically, thread needs a way to clean up if it gets terminated and has to exit in a hurry!

Dealing with Deadlocks

2. Proactive Approaches:
   - Deadlock Prevention
     - Prevent one of the 4 necessary conditions from arising
     - …. This will prevent deadlock from occurring
   - Deadlock Avoidance
     - Carefully allocate resources based on future knowledge
     - Deadlocks are prevented
3. Ignore the problem
   - Pretend deadlocks will never occur
   - Ostrich approach… but surprisingly common!

Deadlock Prevention

Deadlock Prevention

- Can the OS prevent deadlocks?
- Prevention: Negate one of necessary conditions
  - Mutual exclusion:
    - Make resources sharable
    - Not always possible (spooling?)
  - Hold and wait
    - Do not hold resources when waiting for another

    ⇒ Request all resources before beginning execution

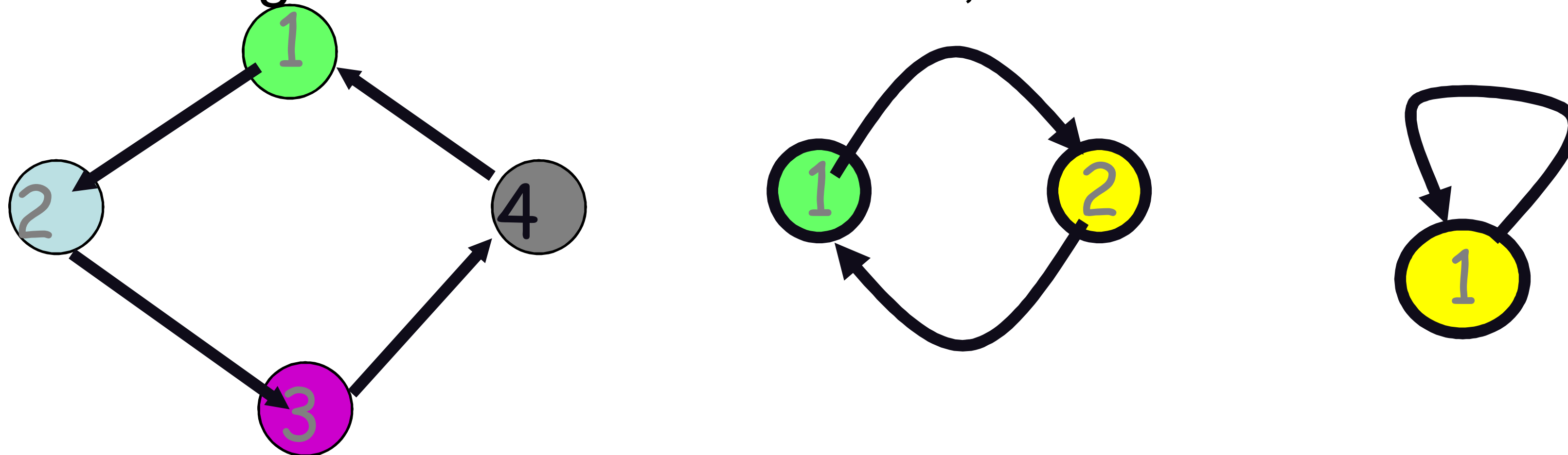
    - Processes do not know what all they will need
    - Starvation (if waiting on many popular resources)
    - Low utilization (Need resource only for a bit)
    - Alternative: Release all resources before requesting anything new
      - Still has the last two problems

Deadlock Prevention

- Prevention: Negate one of necessary conditions
  - No preemption:
    - Make resources preemptable (2 approaches)
    - Preempt requesting processes' resources if all not available
    - Preempt resources of waiting processes to satisfy request
    - Good when easy to save and restore state of resource
      - CPU registers, memory virtualization
    - Bad if in middle of critical section and resource is a lock
  - Circular wait:
    - Impose partial ordering on resources, request them in order

Breaking Circular Wait

- Order resources (lock1, lock2, …)
- Acquire resources in strictly increasing/decreasing order
- When requests to multiple resources of same order:
  - Make the request a single operation
- Intuition: Cycle requires an edge from low to high, and from high to low numbered node, or to same node



- Ordering not always possible, low resource utilization

Deadlocks: Part II
Avoidance, Detection and Recovery

Yi Shi

Fall 2018

Xi'an Jiaotong University

Review

- What is deadlocks
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Review (2)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will *never* enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system
- Deadlock prevention
  - Prevent one of four necessary conditions for deadlock

Goals for today

- Deadlock avoidance
  - Assesses, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this
- Deadlock detection and recover
  - Attempts to assess whether waiting graph can ever make progress
  - Recover it not

Deadlock Avoidance

Deadlock Avoidance

- If we have future information
  - Max resource requirement of each process before they execute
- Can we guarantee that deadlocks will never occur?
- Avoidance Approach:
  - Before granting resource, check if state is **safe**
  - If the state is safe $\Rightarrow$ no deadlock!

Safe State

- A state is said to be **safe**, if it has a process sequence
  $\{P_1, P_2, \ldots, P_n\}$, 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 held by all P_j, where $j < i$

- State is safe because OS can definitely avoid deadlock
  – by blocking any new requests until safe order is executed

- This avoids circular wait condition
  – Process waits until safe state is guaranteed

Safe State Example

- Suppose there are 12 tape drives

	max need	current usage	could ask for
P0	10	5	5
P1	4	2	2
P2	9	2	7

3 drives remain

- current state is safe because a safe sequence exists: <p1,p0,p2>
  - p1 can complete with current resources
  - p0 can complete with current+p1
  - p2 can complete with current +p1+p0
- What if p2 requests 1 drive now?

Safe State Example

- Suppose p2 gets 1 drive

	max need	current usage	could ask for
P0	10	5	5
P1	4	2	2
P2	9	3	6
		2 drives remain	

- no safe sequence exists:
  - p1 can complete with current resources
  - p0 and p2 can not complete with current+p1=2+2=4
- so p2's request is denied
  - then it must wait to avoid unsafe state.

Safe State Example

(One resource class only)

process	holding	max claims	need
A	4	6	2
B	4	11	7
C	2	7	5

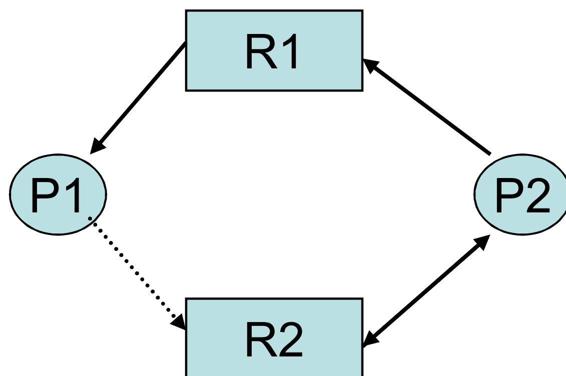
unallocated: 2

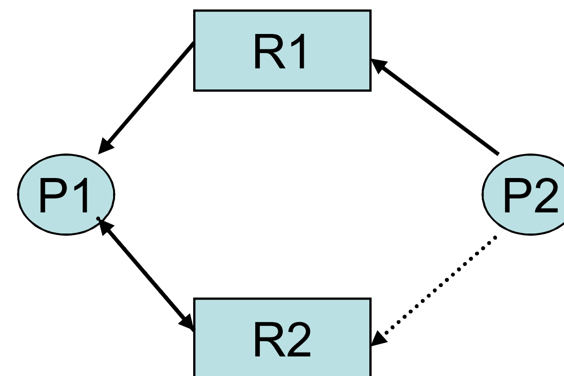
safe sequence: A,C,B

If C should have a claim of 9 instead of 7,
there is no safe sequence.

Res. Alloc. Graph Algorithm

- Deadlock can be described using a *resource allocation graph, RAG*
- Works if only **one** instance of each resource type
- Algorithm:
  - Add a **claim edge**, $P_i \rightarrow R_j$ if P_i can request R_j in the future
    - Represented by a dashed line in graph
  - A request $P_i \rightarrow R_j$ can be granted only if:
    - Adding an **assignment edge** $R_j \rightarrow P_i$ does not introduce cycles
      - Since cycles imply unsafe state





Res. Alloc. Graph issues:

- Works if only **one** instance of each resource type
- A little complex to implement
  - Would need to make it part of the system
  - E.g. build a “resource management” library

Banker's Algorithm

- Suppose we know the "worst case" resource needs of processes in advance
  - A bit like knowing the credit limit on your credit cards. (This is why they call it the Banker's Algorithm)
- Observation: Suppose we just give some process ALL the resources it could need…
  - Then it will execute to completion.
  - After which it will give back the resources.
- Like a bank: If Visa just hands you all the money your credit lines permit, at the end of the month, you'll pay your entire bill, right?

Banker's Algorithm

- So…
  - A process pre-declares its worst-case needs
  - Then it asks for what it "really" needs, a little at a time
  - The algorithm decides when to grant requests
- It delays a request unless:
  - It can find a sequence of processes…
  - …. such that it could grant their outstanding need…
  - … so they would terminate…
  - … letting it collect their resources…
  - … and in this way it can execute everything to completion!

Banker’s Algorithm

- How will it really do this?
  - The algorithm will just implement the graph reduction method for resource graphs
  - Graph reduction is “like” finding a sequence of processes that can be executed to completion
- So: given a request
  - Build a resource graph
  - See if it is reducible, only grant request if so
  - Else must delay the request until someone releases some resources, at which point can test again

Banker's Algorithm

- Decides whether to grant a resource request.
- Data structures:

n: integer	# of processes
m: integer	# of resources
available[1..m]	available[i] is # of avail resources of type i
max[1..n,1..m]	max demand of each Pi for each Ri
allocation[1..n,1..m]	current allocation of resource Rj to Pi
need[1..n,1..m]	max # resource Rj that Pi may still request
	$need_i = max_i - allocation_i$

let request[i] be vector of # of resource Rj Process Pi wants

Basic Algorithm

1. If request[i] > need[i] then

   error (asked for too much)

2. If request[i] > available[i] then

   wait (can't supply it now)

3. Resources are available to satisfy the request

   Let's assume that we satisfy the request. Then we would have:

   available = available - request[i]

   allocation[i] = allocation [i] + request[i]

   need[i] = need [i] - request [i]

   Now, check if this would leave us in a safe state:

   if yes, grant the request,

   if no, then leave the state as is and cause process to wait.

Safety Check

```
work[1..m] = available          /* how many resources are available */
finish[1..n] = false (for all i)  /* none finished yet */

Step 1: Find an i such that finish[i]=false and need[i] <= work
          /* find a proc that can complete its request now */
         if no such i exists, go to step 3   /* we're done */

Step 2: Found an i:
         finish [i] = true  /* done with this process */
         work= work + allocation [i]
        /* assume this process were to finish, and its allocation  back to the
        available list */
         go to step 1

Step 3: If finish[i] = true for all i, the system is safe. Else Not
```

Banker's Algorithm: Example

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

this is a safe state: safe sequence <P1, P3, P4, P2, P0>

Suppose that P1 requests (1,0,2)

- (1,0,2)<(3,2,2) and (1,0,2)<(1,2,2)
- add it to P1's allocation and subtract it from Available

Banker's Algorithm: Example

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	0	7	4	3
P1	3	0	2	3	2	2				0	2	0
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

This is still safe: safe seq <P1, P3, P4, P0, P2>, so request of p1 can be granted

In this new state,

P4 requests (3,3,0)

not enough available resources , p4's request will be denied

P0 requests (0,2,0)

let's check resulting state

Banker's Algorithm: Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	2	1	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

This is unsafe state (why?)

So P0's request will be denied

Problems with Banker's Algorithm?

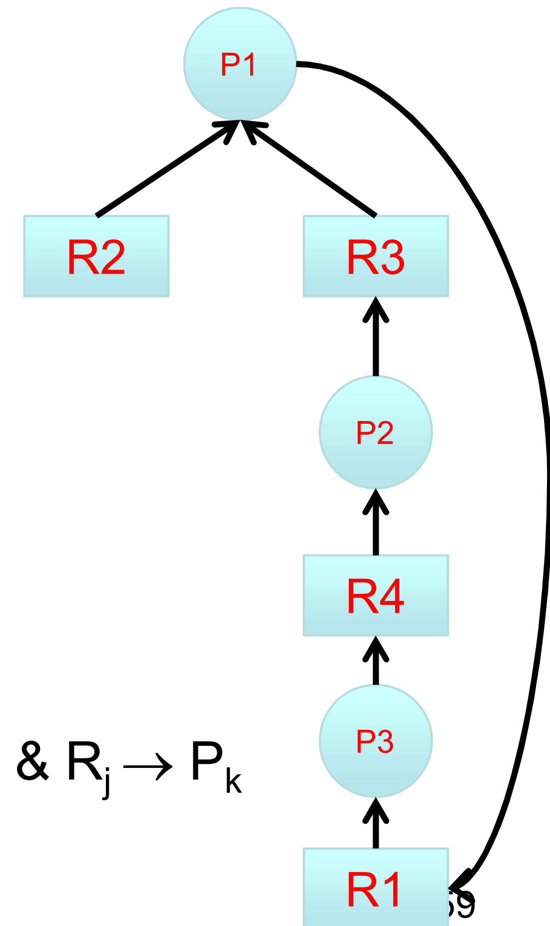
Deadlock Detection & Recovery

Deadlock Detection & Recovery

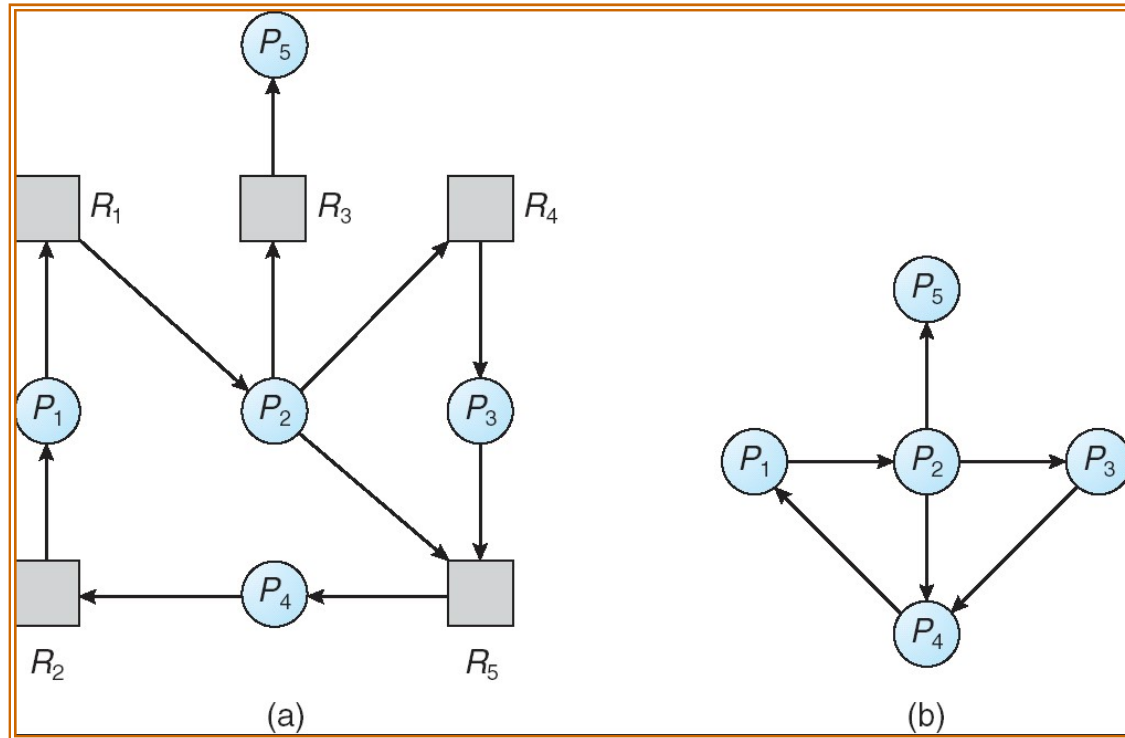
- If neither avoidance or prevention is implemented, deadlocks can (and will) occur.
- Coping with this requires:
  - Detection: finding out if deadlock has occurred
    - Keep track of resource allocation (who has what)
    - Keep track of pending requests (who is waiting for what)
  - Recovery: untangle the mess.
- Expensive to detect, as well as recover

Using the RAG Algorithm to detect deadlocks

- Suppose there is only one instance of each resource
- Example 1: Is this a deadlock?
  - P1 has R2 and R3, and is requesting R1
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Example 2: Is this a deadlock?
  - P1 has R2, and is requesting R1 and R3
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Use a **wait-for graph**:
  - Collapse resources
  - An edge $P_i \rightarrow P_k$ exists only if RAG has $P_i \rightarrow R_j$ & $R_j \rightarrow P_k$
  - Cycle in wait-for graph $\Rightarrow$ deadlock!



Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

2nd Detection Algorithm

- What if there are multiple resource instances?
- Data structures:

```
n: integer              # of processes
m: integer              # of resources
available[1..m]         available[i] is # of avail resources of type i
request[1..n,1..m]      max demand of each Pi for each Ri
allocation[1..n,1..m]   current allocation of resource Rj to Pi
finish[1..n]            true if Pi's request can be satisfied

let request[i] be vector of # instances of each resource Pi wants
```

2nd Detection Algorithm

1. work[]=available[]

   for all $i < n$, if allocation[i] $\neq 0$

   then finish[i]=false else finish[i]=true

2. find an index i such that:

   finish[i]=false;

   request[i]$\leq$=work

   if no such i exists, go to 4.

3. work=work+allocation[i]

   finish[i] = true, go to 2

4. if finish[i] = false for some i,

   then system is deadlocked with Pi in deadlock

Example

Finished = {F, F, F, F};

Work = Available = (0, 0, 1);

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

Allocation

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

Request

Example

Finished = {F, F, T, F};

Work = (1, 1, 1);

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

Allocation

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

Request

Example

Finished = {F, F, T, T};

Work = (2, 2, 2);

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

Allocation

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

Request

Example

Finished = {F, T, T, T};

Work = (4, 3, 4);

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

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2			
P_3			
P_4			

Request

Example

Finished = {T, T, T, T};

Work = (5, 4, 5);

	R_1	R_2	R_3
P_1			
P_2			
P_3			
P_4			

Allocation

	R_1	R_2	R_3
P_1			
P_2			
P_3			
P_4			

Request

When to run Detection Algorithm?

- For every request that cannot be immediately satisfied?
- For every resource request?
- Once every hour?
- When CPU utilization drops below 40%?

Deadlock Recovery

- Killing one/all deadlocked processes
  - Crude, but effective
  - Keep killing processes, until deadlock broken
  - Repeat the entire computation
- Preempt resource/processes until deadlock broken
  - Selecting a victim (# resources held, how long executed)
  - Rollback (partial or total)
  - Starvation (prevent a process from being executed)

Summary

- Dining Philosophers Problem
  - Highlights need to multiplex resources
  - Context to discuss starvation, deadlock, livelock

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Summary (2)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will *never* enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system
- Deadlock prevention
  - Prevent one of four necessary conditions for deadlock
- Deadlock avoidance
  - Assesses, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this
- Deadlock detection and recover
  - Attempts to assess whether waiting graph can ever make progress
  - Recover it not

Exercises:

7.7

7.11