CS 370: OPERATING SYSTEMS
[DEADLOCKS]

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- Are particular deadlock prevention strategies better than others? Or is it dependent on the circumstances?
- Could there be a priority for resource preemption?
- Resource allocation graph: What if process needs to have a specific resource instance?
- Safe sequences
  - In deadlock avoidance, are you always exploring all possible sequences?
  - How does safe sequencing interact with the scheduler?

2

Topics covered in this lecture

- Deadlock Avoidance
  - Banker's Algorithm
- Deadlock Detection
  - And … recovery
- Other issues relating to deadlocks

A deadlock-prone system can be in one of three states: safe, unsafe, and deadlocked

- Safe state: For *any possible sequence of resource requests*, there is **at least one safe sequence** for processing them
  - That sequence eventually succeeds in **granting** all pending and future requests
- Unsafe state: *At least one sequence* of future resource requests leads to deadlock
  - Deadlock is possible, but **not inevitable**
- In a deadlocked state, the system has at least one deadlock
  - At least one set of processes is waiting on each other

A system in a safe state **controls its own destiny**

- For any workload, it can avoid deadlock by **delaying** the processing of some requests
  - Once the system enters an unsafe state, it may not be able to avoid deadlock
- In particular, the Banker's Algorithm (that we will look at next) **delays** any request that takes it from a safe to an unsafe state.

BANKER'S ALGORITHM



Banker's Algorithm

- Designed by Dijkstra
- Modeled on a small-town banker
  - Customers have been extended lines of credit
  - Not ALL customers will need their maximum credit immediately
- Customers make loan requests from time to time

Crux of the Banker's Algorithm

- Consider each request as it occurs
  - See if granting it is safe
- If safe: grant it; If unsafe: postpone
- For safety: banker checks if there is **enough** to satisfy some customer
  - If so, that customer's loans are assumed to be repaid
  - Customer closest to limit is checked next
  - **If all loans can be repaid; state is safe: loan approved**

Banker's Algorithm: Managing the customers. Banker has only reserved 10 units instead of 22

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

SAFE

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

Delay all requests except C

SAFE

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

UNSAFE

A customer may not need the entire credit line.
But the banker cannot count on this behavior.

There is ONLY ONE resource – Credit

Banker's algorithm: Crux

- Declare **maximum** number of resource instances needed
  - Cannot exceed resource thresholds
- Determine if resource allocations leave system in a safe state

You are about to see several matrices

- There is no matrix multiplication, no inversion, and no eigenvalues
- These are simply tables
  - Each row represents a process
  - Each column represents a resource type, and
  - Each entry is a count
- We will only subtract, compare, and add.

Bankers Algorithm: Data Structures [Overview]

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

A, **B**, and **C** are different *types* of resources

Data Structures: **n** is the number of processes and **m** is the number of resource types

- `Available`: Vector of length `m`
  - Number of resources for each type
    - `Available[i] = k`
- `Max`: `n` x `m` matrix
  - Maximum *demand* for each process (in each row)
  - `Max[i,j]= k`
    - Process P_i may request at most `k` instances of R_j

Data Structures: **n** is the number of processes and **m** is the number of resource types

- `Allocation`: `n` x `m` matrix
  - Resource instances allocated for each process (each row)
  - `Allocation[i,j]=k`
    - Process P_i currently **allocated** `k` instances of R_j
- `Need`: `n` x `m` matrix
  - Resource instances needed for each process (each row)
  - `Need[i,j]=k`
    - Process P_i *may need* `k` **more** instances of R_j

Vectors identifying a process' resource requirements: Rows in the matrices

- $Allocation_i$
  - Resource instances allocated for process P_i
- $Need_i$
  - Additional resource instances that process P_i may still request

Banker's Algorithm: Notations

- **X** and **Y** are vectors of length m
- **X** ≤ **Y** if-and-only-if

  $\mathbf{X}[i] \leq \mathbf{Y}[i]$ *for all* $i=1,2,\ldots,m$
- **X** = {1,7,3,2} and **Y** = {0,3,2,1}

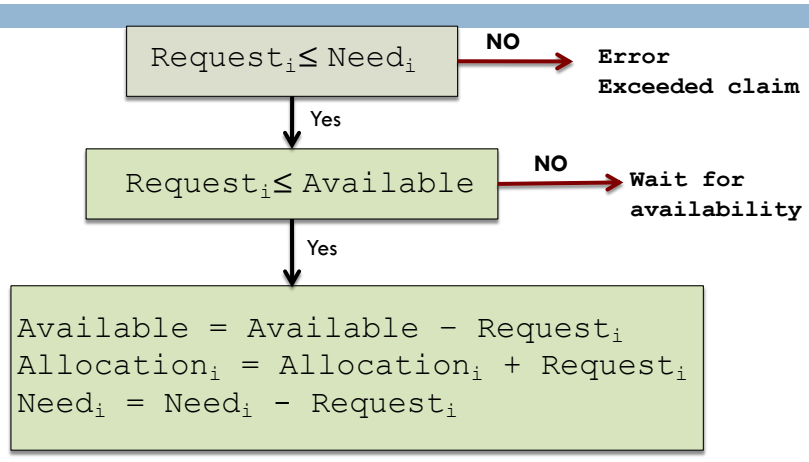
  So, **Y** ≤ **X**

  Also, **Y** < **X** if **Y** ≤ **X** and **Y** ≠ **X**

Banker's Algorithm: Resource-request

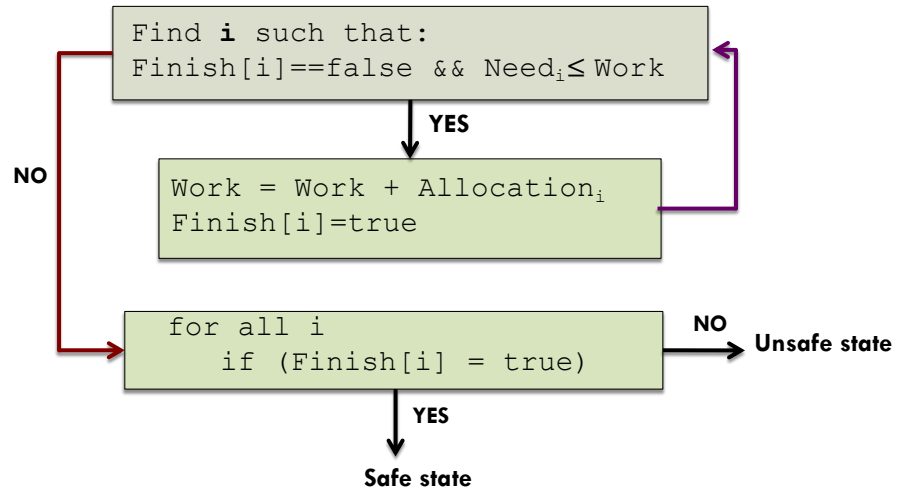
- $Request_i$: Request vector for process P_i
  - $Request_i[j]=k$
    - Process P_i wants k instances of R_j

Banker's Algorithm: Resource-request



Bankers Algorithm: Safety

Initialize `Work = Available`



19

Bankers Algorithm: Example

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

<P1, P3, P4, P2, P0> satisfies safety criteria

Suppose process **P1** requests 1 **A**, and 2 **C**s: $Request_1$ = (1,0,2)

$Request_1 \leq$ Available

Pretend request was fulfilled

20

Bankers Algorithm: Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	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			

<P1, P3, P4, P0, P2> satisfies safety criteria

$Request_4$ = (3,3,0) from process **P4** cannot be granted: resources unavailable

$Request_0$ = (0,2,0) from process **P0** cannot be granted: unsafe state

21

Bankers 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			

None of the processes can now satisfy their max resource needs.

$Request_0$ = (0,2,0) from process **P0** cannot be granted: unsafe state

22

Bankers Algorithm: Practical implications [1/2]

- Understanding the Banker's Algorithm can help in designing simple solutions for specific problems
- Banker's Algorithm to devise a rule for thread safe acquisition of a pair of locks, **A** and **B**, with **mutually recursive** locking?
  - Suppose a thread needs to acquire locks **A** and **B**, in that order, while another thread needs to acquire lock **B** first, then **A**
  - RULE: A thread is always allowed to acquire its second lock
    - Acquire first lock provided the other thread does not already hold its first lock

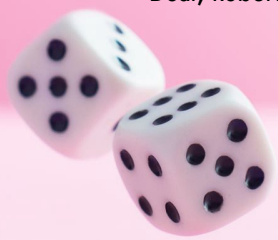
Bankers Algorithm: Practical implications [2/2]

- Processes *rarely know in advance* about their maximum resource needs
- Number of processes managed by the kernel is not fixed
  - Varies dynamically
- Resources thought to be available can vanish

Since it cost a lot to win
And even more to lose
You and me bound to spend some time
Wondering what to choose

It goes to show you don't ever know
Watch each card you play and play it slow
Wait until that deal come 'round
Don't you let that deal go down, no, no

Deal; Robert Hunter/Jerry Garcia; The Grateful Dead



DEADLOCK DETECTION

25

Single instance of EACH resource type

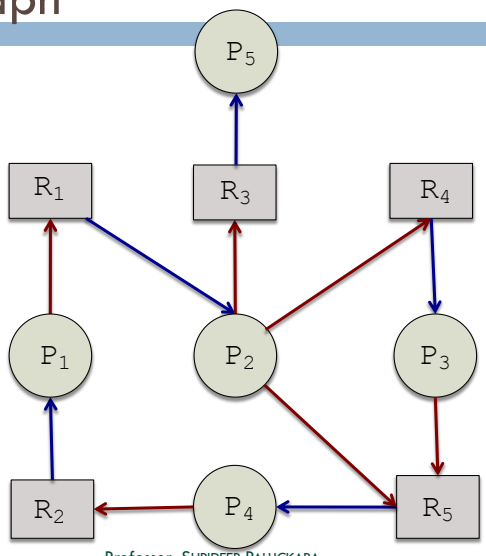
- Use **wait-for** graph
  - *Variant* of the resource allocation graph
- Deadlock exists if there is a **cycle** in the graph
- Transformation
  1. **Remove** resource nodes
  2. **Collapse** appropriate edges

26

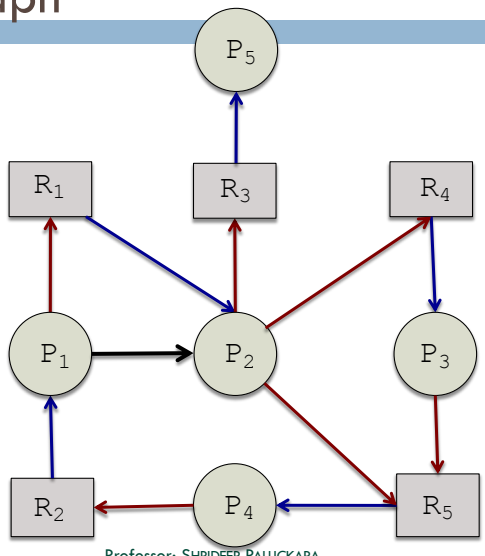
What the edges in the wait-for graph imply

- $P_i \rightarrow P_j$
  - Process P_i is waiting for a resource held by P_j
- $P_i \rightarrow P_j$ only if resource allocation graph has
  1. $P_i \rightarrow R_q$ and
  2. $R_q \rightarrow P_j$ for some resource R_q

Transforming a resource allocation graph into a wait-for graph

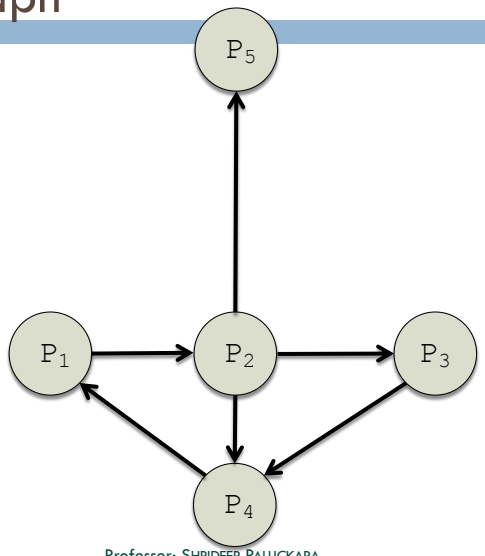


Transforming a resource allocation graph into a wait-for graph



29

Transforming a resource allocation graph into a wait-for graph



30

Deadlock detection for multiple instances of a resource type

- Wait-for graph is not applicable
- Approach uses data structures similar to Banker's algorithm

Data Structures: `n` is number of processes `m` is number of resource types

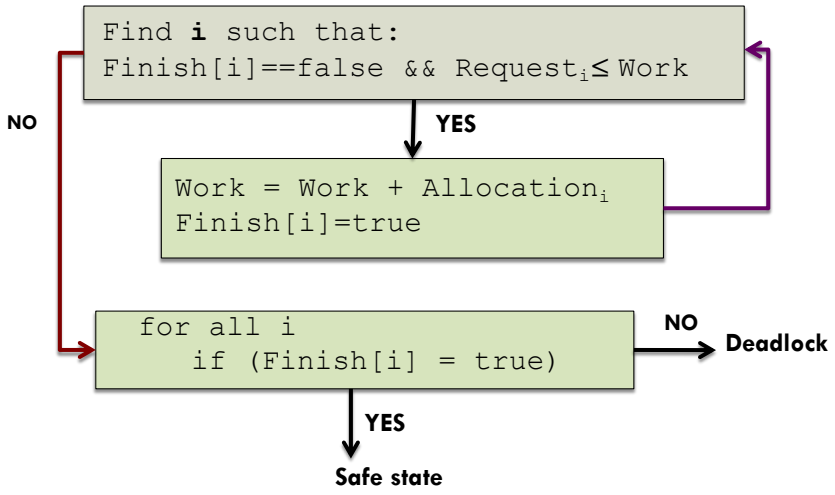
- `Available`: Vector of length `m`
  - Number of resources for each type
- `Allocation`: `n` x `m` matrix
  - Resource instances allocated for each process
  - `Allocation[i,j]=k`
    - Process P_i currently **allocated** `k` instances of R_j
- `Request`: `n` x `m` matrix
  - Current request for each process
  - `Request[i,j]=k`
    - Process P_i **requests** `k` **more** instances of R_j

Deadlock detection: Initialization

`Work` and `Finish` are vectors of length `m` & `n`

```
Work = Available
if (Allocation_i ≠ 0) {
  Finish[i] = false;
} else {
  Finish[i] = true;
}
```


Deadlock detection

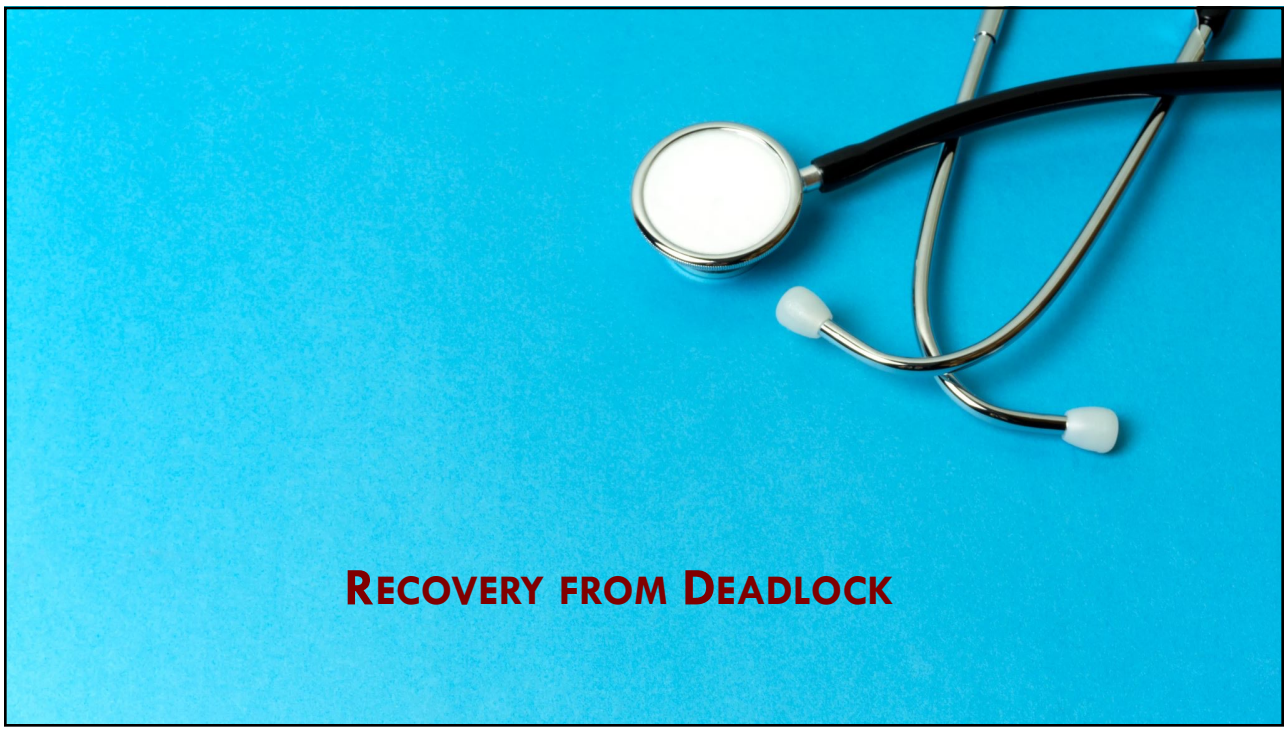


Deadlock detection: Usage

- How **often** will the deadlock occur?
- How **many** processes will be affected when it happens?

Frequency of invoking deadlock detection

- Resources allocated to deadlocked process **idle**
  - Until the deadlock can be broken
- Deadlocks occur *only* when process makes a request
  - Significant overheads to run detection per request
- Middle ground: Run at *regular intervals*



37

Recovery from deadlock

- Automated or manual
- OPTIONS
  - Break the circular wait: **Terminate processes**
  - **Preempt** resources from deadlocked process(es)

38

Breaking circular wait: Process termination

- Terminate **all** deadlocked processes
- Terminate processes **one at a time**
  - After each termination, check if deadlock *persists*
- Reclaim all resources allocated to terminated process

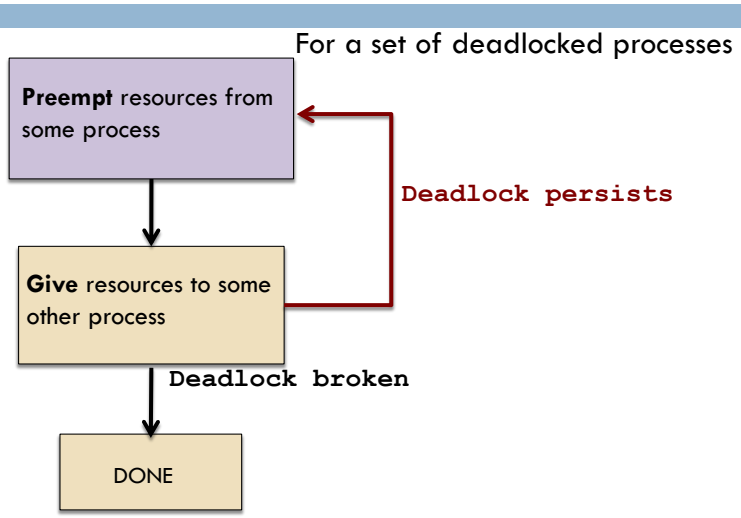
Terminating a Process

- Process may be in the midst of something
  - Updating files, printing data, etc.
- Terminate process whose termination will incur **minimum** costs
  - Policy decision similar to scheduling decisions

Factors determining process termination

- Priority
- How long has the process been running?
  - How much longer?
- Number and types of resources used
  - How many more needed?
- Interactive or batch

Deadlock recovery: Resource preemption



Resource preemption: Issues

- Selecting a victim
  - Which resource and process
  - Order of preemption to minimize cost
- Starvation
  - Process can be selected for preemption *finite* number of times

Deadlock recovery through rollbacks

- **Checkpoint** process periodically
  - Contains memory image and resource state
- Deadlock detection tells us *which* resources are needed
- Process owning a needed resource
  - **Rolled back** to before it acquired needed resource
    - Work done since rolled back checkpoint discarded
  - **Assign** resource to deadlocked process

OTHER ISSUES

Two-phase locking

- Used in database systems
- Operation involves requesting locks on several records and updating all the locked records
- When multiple processes are running?
  - Possibility of deadlocks

Two-Phase Locking

- First phase
  - Process tries to acquire all the locks it needs, one at time
  - If successful: start second-phase
  - If some record is already locked?
    - Release all locks and start the first phase all over
- Second-phase
  - Perform updates and release the locks

Communication Deadlocks

- Process **A** sends a request message to process **B**
  - Blocks until **B** sends a reply back
- Suppose, that the request was lost
  - **A** is blocked waiting for a reply
  - **B** is blocked waiting for a request to do something
  - **Communication deadlock**

Communication deadlocks

- Cannot be prevented by ordering resources (there are none)
  - Or avoided by careful scheduling (no moments when a request can be postponed)
- Solution to breaking communication deadlocks?
  - **Timeouts**
    - Start a timer when you send a message to which a reply is expected.

Livelocks

- Polling (busy waits) used to enter critical section or access a resource
  - Typically used for a short time when overhead for suspension is considered greater
- In a livelock two processes need each other's resource
  - Both run and make no progress, but neither process blocks
  - **Use CPU quantum over and over without making progress**

Livelocks do occur

- If fork fails because process table is full
  - Wait for some time and try again
- But there could be a collection of processes each trying to do the same thing

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 7]
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 6]*
- *Thomas Anderson and Michael Dahlin. Operating Systems Principles and Practice. 2nd Edition. ISBN: 978-0985673529.* [Chapter 6]