CS 370: Operating Systems

[Deadlocks (again!)]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: Shrideep Pallickara

Topics covered in this lecture

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

BANKER'S ALGORITHM

Banker's Algorithm

- Designed by Dijkstra in 1965
- 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 he/she has **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

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

- $\texttt{Allocation}_{\texttt{i}}$
  - Resource instances allocated for process $\texttt{P}_{\texttt{i}}$

- $\texttt{Need}_{\texttt{i}}$
  - Additional resource instances process $\texttt{P}_{\texttt{i}}$ may still request

Banker's algorithm: Notations

- $\mathbf{X}$ and $\mathbf{Y}$ are vectors of length m
- $\mathbf{X} \leq \mathbf{Y}$ if-and-only-if

  $\mathbf{X}[i] \leq \mathbf{Y}[i]$ *for all* $i=1,2,...,m$

- $\mathbf{X} = \{1,7,3,2\}$ and $\mathbf{Y} = \{0,3,2,1\}$

  So, $\mathbf{Y} \leq \mathbf{X}$

  Also $\mathbf{Y} < \mathbf{X}$ if $\mathbf{Y} \leq \mathbf{X}$ and $\mathbf{Y} \neq \mathbf{X}$

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

Bankers Algorithm: Resource-request

$\text{Request}_i \leq \text{Need}_i$ — **NO** → **Error Exceeded claim**

Yes ↓

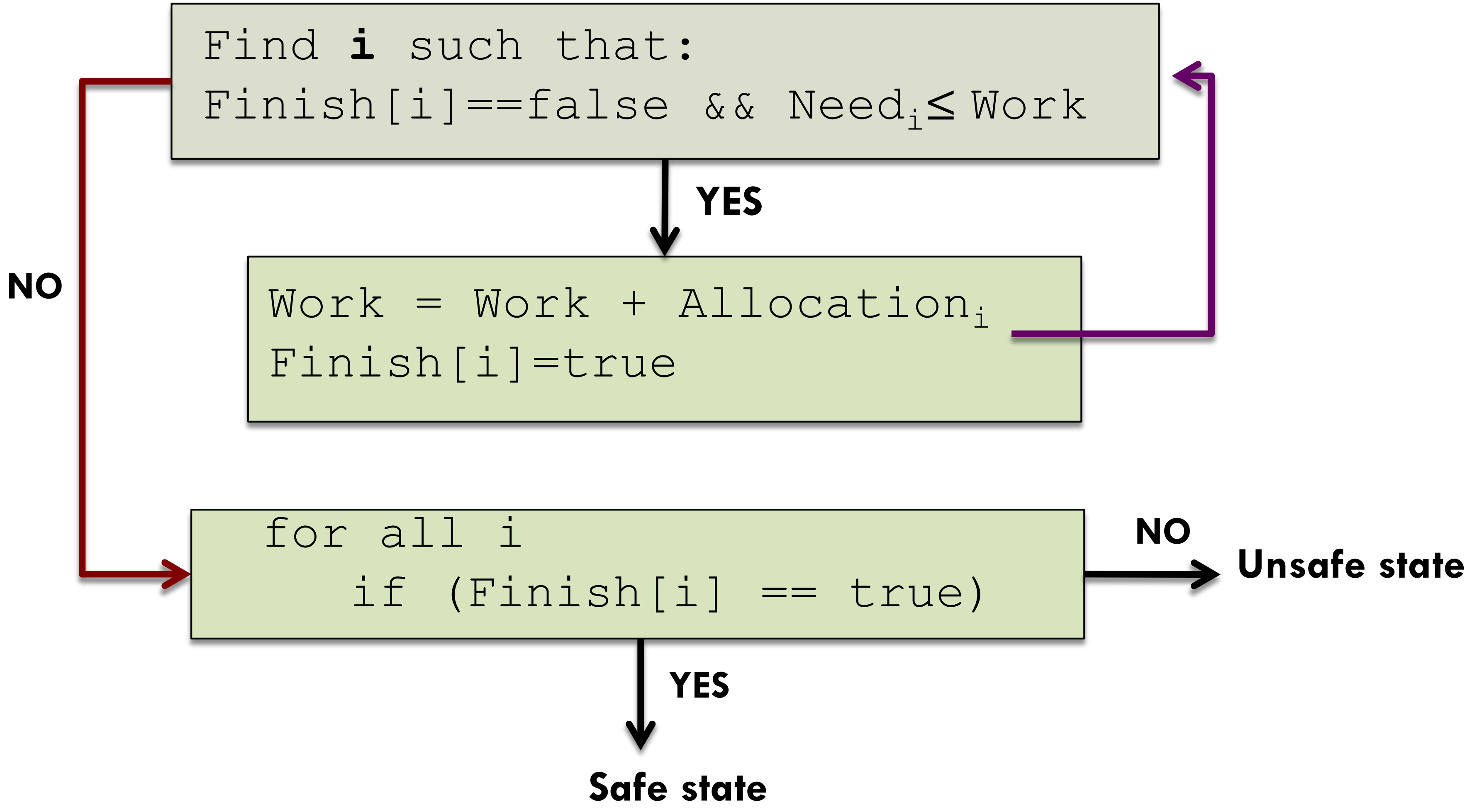
$\text{Request}_i \leq \text{Available}$ — **NO** → **Wait for availability**

Yes ↓

$$\text{Available} = \text{Available} - \text{Request}_i$$
$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$$
$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$

Bankers Algorithm: Safety

Initialize `Work = Available`



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: $\mathtt{Request}_1$ = (1,0,2)

$\mathtt{Request}_1 \leq$ Available

Pretend request was fulfilled

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

$\mathrm{Request}_4$ = (3,3,0) from process **P4** cannot be granted: resources unavailable

$\mathrm{Request}_0$ = (0,2,0) from process **P0** cannot be granted: unsafe state

Bankers Algorithm: Limited practical value

- Processes *rarely know in advance* about their maximum resource needs
- Number of processes is not fixed
  - Varies dynamically
- Resources thought to be available can vanish
- Few systems use this for avoiding deadlocks

DEADLOCK DETECTION

Single instance of EACH resource type

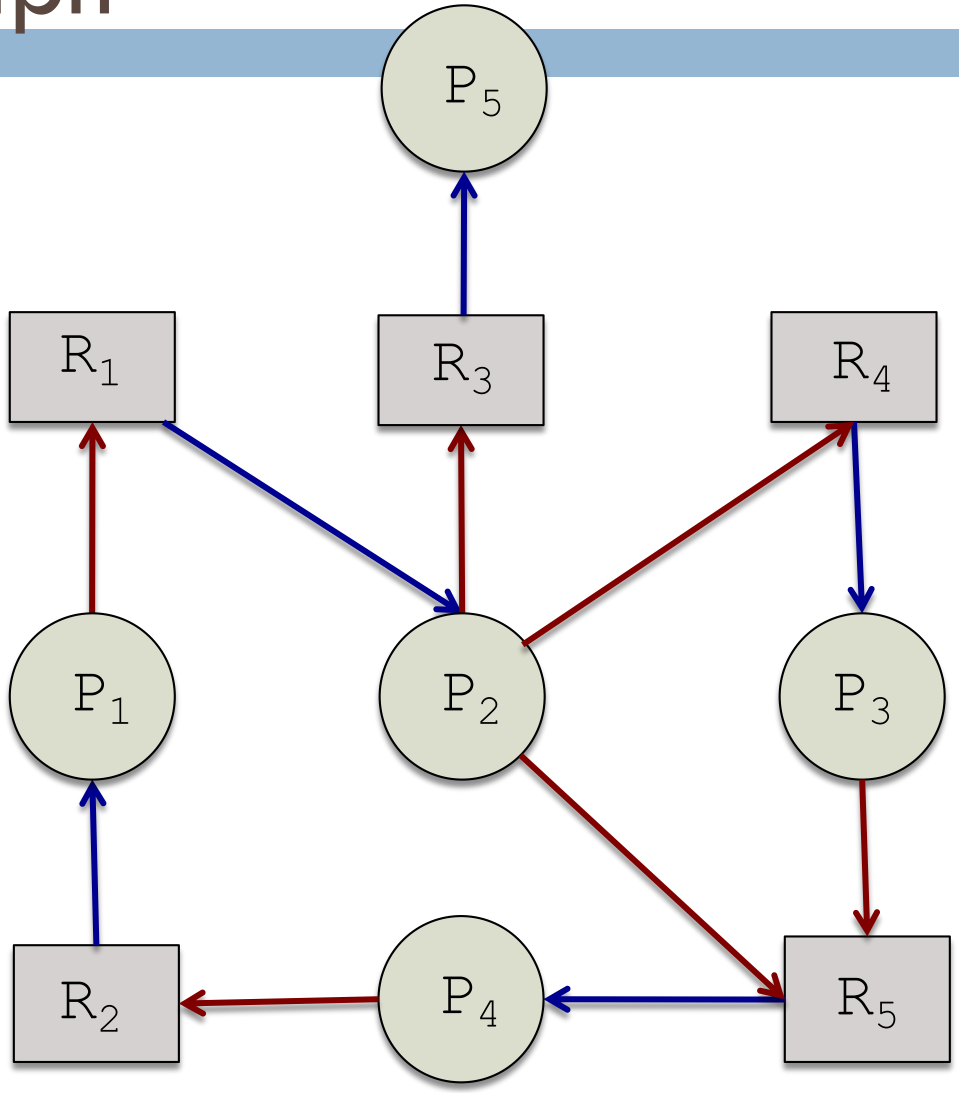
- Use **wait-for** graph
  - *Variant* of the resource allocation graph
- Deadlock exists if there is a **cycle** in the graph
- Transformation
  1. **Remove** resource nodes
  2. **Collapse** appropriate edges

What the edges in the wait-for graph imply

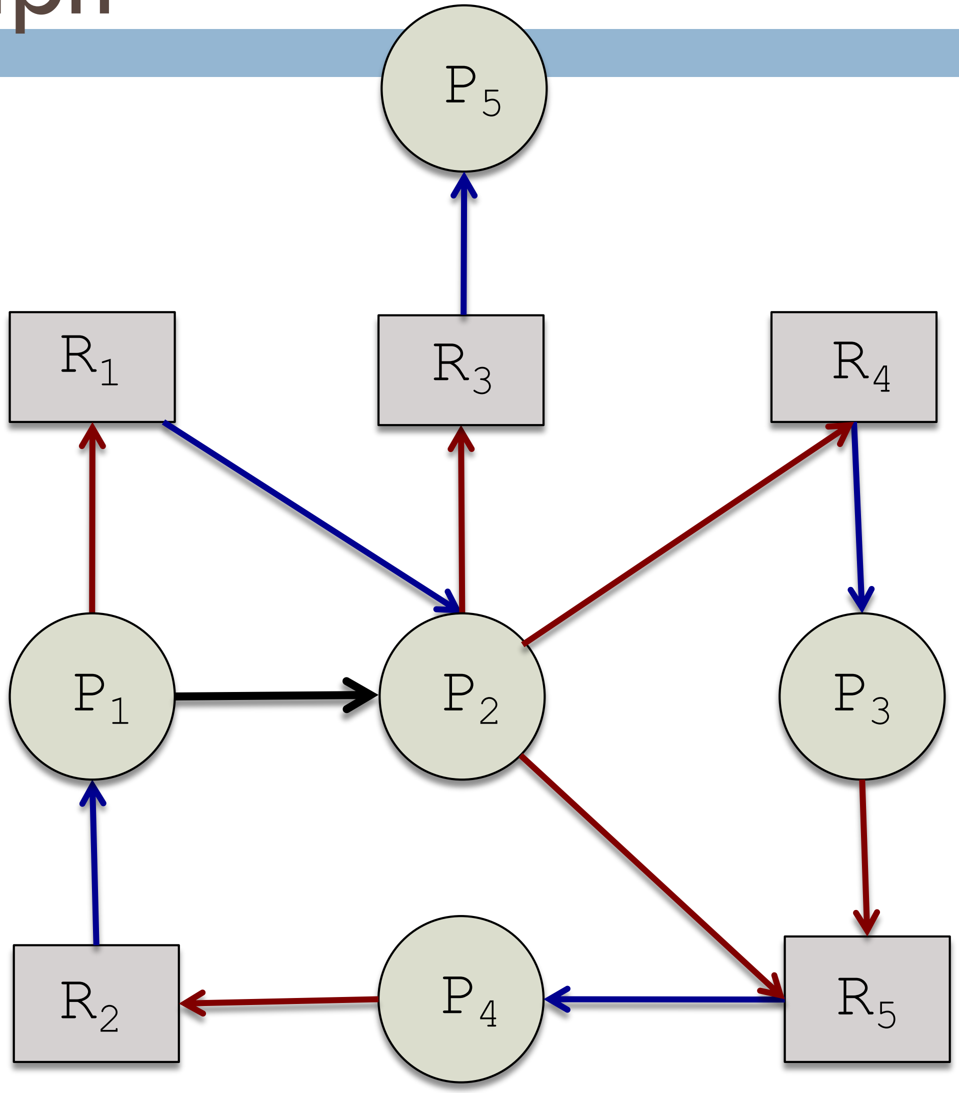
- $P_i \rightarrow P_j$
  - Process P_i is waiting for a resource held by P_j

- $P_i \rightarrow P_j$ only if resource allocation graph has
  1. $P_i \rightarrow R_q$ and
  2. $R_q \rightarrow P_j$ for some resource R_q

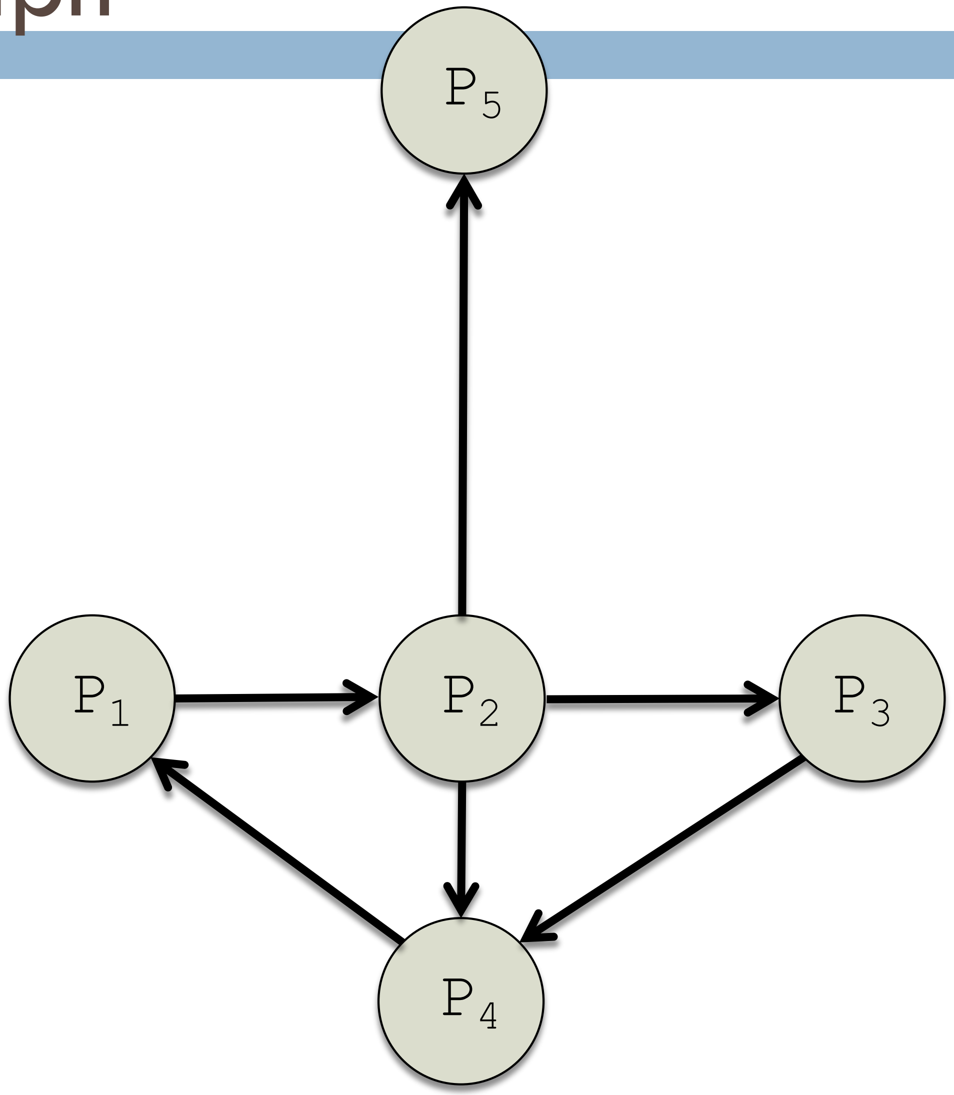
Transforming a resource allocation graph into a wait-for graph



Transforming a resource allocation graph into a wait-for graph



Transforming a resource allocation graph into a wait-for graph



Deadlock Detection

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

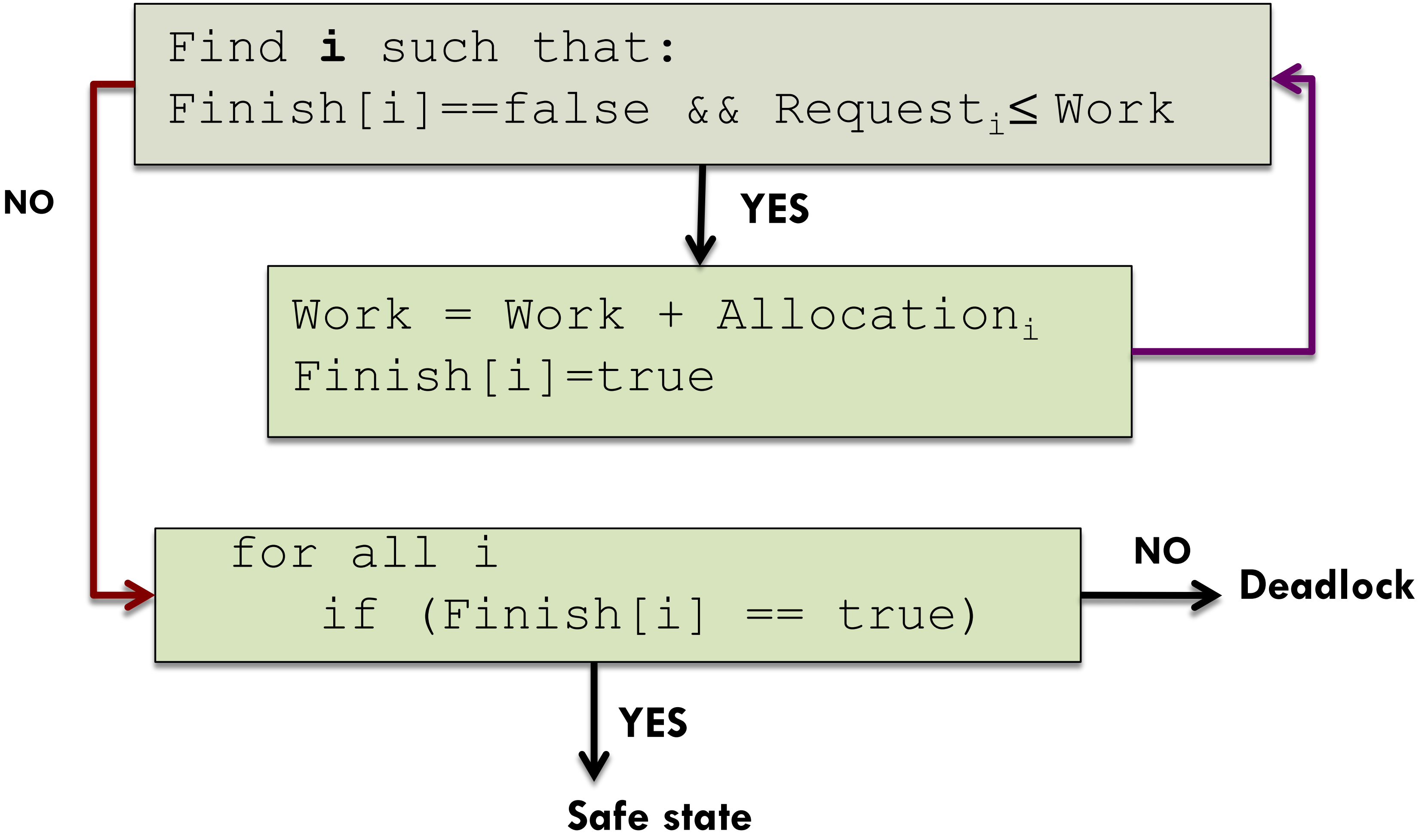
- `Request`: `n x m` matrix
  - Current request for each process
  - `Request[i,j]=k`
    - Process P_i **requests** `k` **more** instances of R_j

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*

Recovery from Deadlock

Recovery from deadlock

- Automated or manual

- OPTIONS
  - Break the circular wait: **Abort** processes
  - **Preempt** resources from deadlocked process(es)

Breaking circular wait: Process termination

- Abort **all** deadlocked processes

- Abort processes **one at a time**
  - After each termination, check if deadlock *persists*

- <u>Reclaim all resources</u> allocated to terminated process

Terminating a Process

- Process may be in the midst of something
  - Updating files, printing data etc

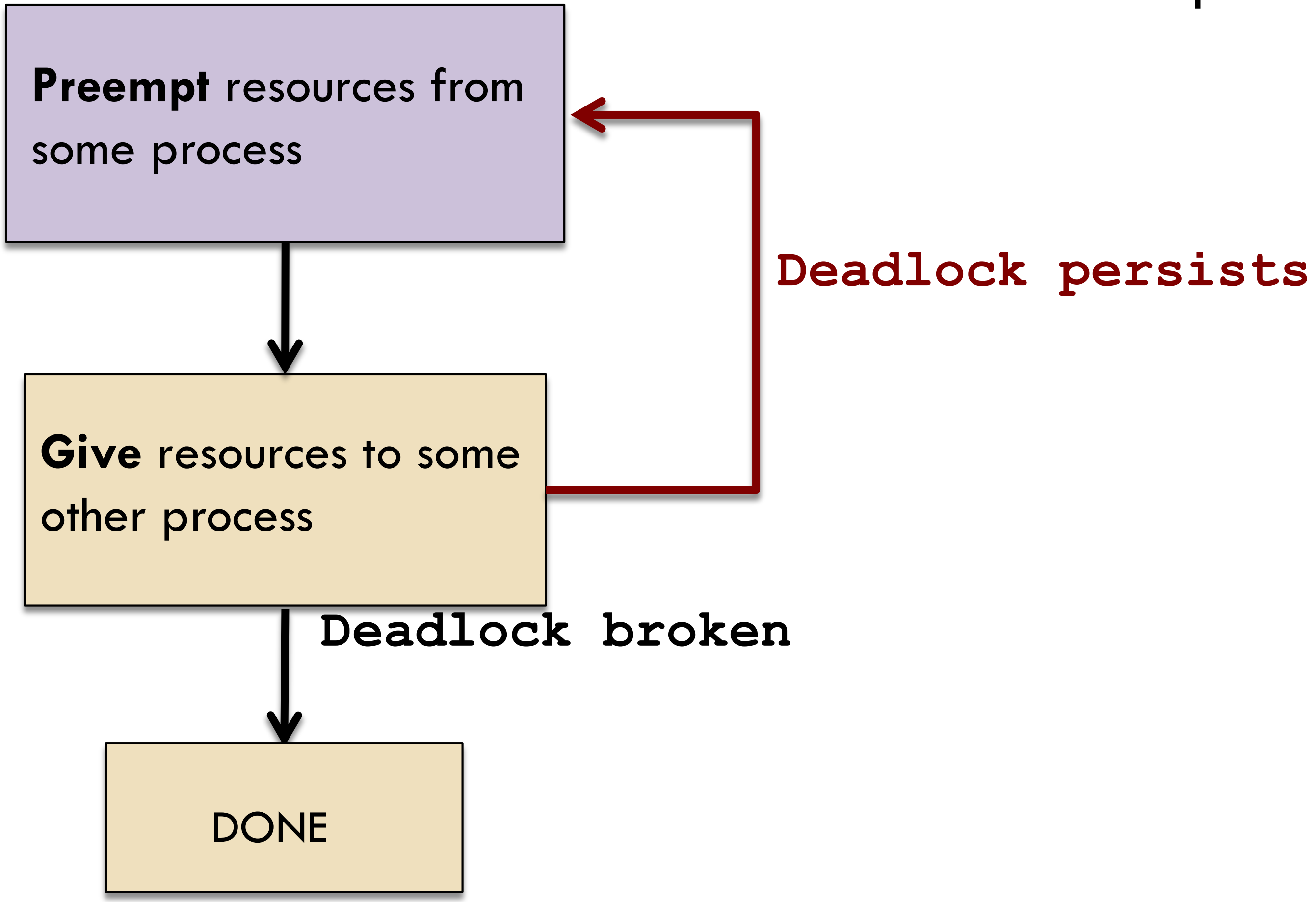
- Abort process whose termination will incur **minimum** costs
  - Policy decision similar to scheduling decisions

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

For a set of deadlocked processes



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

Starvation

- In dynamic systems, some policy is needed to make decision about who gets resource when
  - Some processes never get service even though they are not deadlocked
- E.g.: Give printer to process with the smallest file to print
  - If there is constant stream of small jobs, process with large file will starve
  - Can be avoided with first-come-first-served policy

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]*