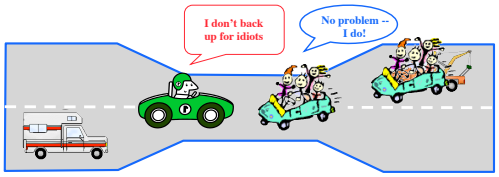
CSCI [4|6]730 Operating Systems

Deadlock

Chapter 8: Deadlock Questions?

- What is a deadlock?
- What causes a deadlock?
- How do you deal with (potential) deadlocks?

Deadlock: What is a deadlock?



Deitel & Deitel anecdote

- All are waiting for a resource that is held by another waiting entity. Since all are waiting, none can provide any of the things being waited for.
- Example: narrow bridge (resource) --
  - » if a deadlock occurs, resolved if one car back up (preempts resource and rollback).

Example: Two Threads?

- Two threads access two shared variables, A and B
  - » Variable A is protected by lock a
  - » Variable B by lock b
- How to add lock and unlock statements?

Thread Maria

```
lock(a);
A += 10;
lock(b);
B += 20;

A += B;
unlock(b)
A += 30
unlock(a)
```

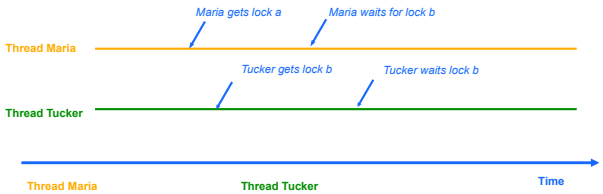
Does this work?

Thread Tucker

```
lock(b)
B += 10;
lock(a);
A += 20;

A += B;
unlock(a);
B += 30
unlock(b);
```


Example: Maria & Tucker



Thread Maria

```
lock(a);
A += 10;
lock(b);
B += 20;

A += B;
unlock(b)
A += 30
unlock(a)
```

Thread Tucker

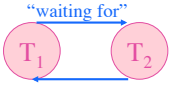
```
lock(b)
B += 10;
lock(a);
A += 20;

A += B;
unlock(a);
B += 30
unlock(b);
```

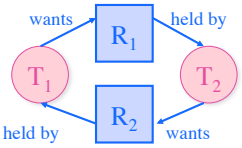

Representing Deadlock

- Two common ways of representing deadlock:
  - » Vertices (circles or rectangles)
    - – threads (or processes) in system
    - – resources [types] (e.g., locks, semaphores, printers)
  - » Edges: indicates (`waiting for' or `wants') or `held by' direction of the edge determines the type.

Wait-For Graph



Resource Allocation Graph



Conditions for Deadlock

- **Mutual exclusion**:
  - » Resource cannot be shared
  - » Requests are delayed until resource is released
- **Hold and wait**:
  - » Thread holds one resource while waits for another
- **No preemption**:
  - » previously granted resources cannot forcibly be taken away
- **Circular wait**:
  - » Circular dependencies exist in "waits-for" or "resource-allocation" graphs
  - » Each is waiting for a resource held by next member of the chain.

All for conditions must hold simultaneously

Handling Deadlock

Ostrich algorithm

1. **Ignore**
   - » Easiest and most common approach (e.g., UNIX).
2. **Deadlock prevention**
   - » Ensure deadlock does not happen
   - » Ensure at least one of 4 conditions does not occur
     1. Hold&Wait, No Preemption, Circularity, Mutual Exclusion
     2. System build so deadlock cannot happen
3. **Deadlock detection and recovery**
   - » Allow deadlocks, but detect when occur
   - » Recover and continue
4. **Deadlock avoidance**
   - » Ensure deadlock does not happen
   - » Use information about resource requests to dynamically avoid unsafe situations

Deadlock Prevention

Mutual exclusion
Hold and wait
No preemption
Circular wait

- **Approach**
  - » Ensure 1 of 4 conditions cannot occur
  - » Negate each of the 4 conditions
- **No single approach is appropriate (or possible) for all circumstances**

Deadlock Prevention: *Mutual Exclusion*

Mutual exclusion
Hold and wait
No preemption
Circular wait

- **No mutual exclusion --> Make resource sharable ; examples:**
  - » Read-only files
  - » Printer daemon needs exclusive access to the printer, there is only one printer daemon -- uses spooling.

Deadlock Prevention *Hold and Wait*

Mutual exclusion
Hold and wait
No preemption
Circular wait

- **Two Approaches:**
  1. **Only request resources when it does not hold other resources**
     - – release resources before requesting new ones

Thread Maria

```
lock(a);
A += 10;
unlock(a)
lock(b);
B += 20;
unlock(b)
lock(a)
A += 30
unlock(a)
```

Thread Tucker

```
lock(b)
B += 10;
Unlock(b);
lock(a);
A += 20;
unlock(a);
lock(b)
B += 30
unlock(b);
```


Deadlock Prevention *Hold and Wait*

Mutual exclusion
Hold and wait
No preemption
Circular wait

- **Two Approaches:**
  2. **Atomically acquire all resources at once**
     - » Example: Single lock to protect all (other variations - e.g., release access to one variable earlier)

Thread Maria

```
lock(AB);
A += 10;
B += 20;
A += 30
unlock(AB)
```

Thread Tucker

```
lock(AB)
B += 10;
A += 20;
B += 30
unlock(AB);
```

Deadlock Prevention
Hold and Wait

Mutual exclusion
Hold and wait
No preemption
Circular wait

- **Summary the Two Approaches:**
  1. Only request resources when it does not hold other resources
  2. Atomically acquire all resources at once
- **Problems:**
  - » Low resource utilization: ties up resources other processes could be using
  - » May not know required resources before run
  - » **Starvation**: A thread that need popular resources may wait forever

Deadlock Prevention
No Preemption

Mutual exclusion
Hold and wait
No preemption
Circular wait

- **Two Approaches:**
  1. Preempt **requestors** resource
     - **Example**: B is holding some resources and then requests additional resources that are held by other threads, then B releases all its resources (and start over)
  2. Preempt **holders** resource
     - **Example**: A waiting for something held by B, then take resource away from B and give them to A (B starts over).
- **Not possible if resource cannot be saved and restored**
  - » Can't take away a lock without causing problems
- **Only works for some resources (e.g., CPU and memory)**

Deadlock Prevention
Circular Wait Condition

Mutual exclusion
Hold and wait
No preemption
Circular wait

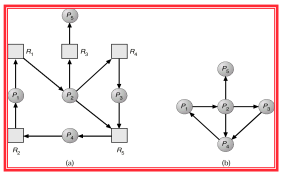
- **Impose ordering on resources**
  - » Give all resources a ranking; must acquire highest ranked resource first

Deadlock Detection & Recovery

1. **Allow system to enter deadlock state**
2. **Detection algorithm**
3. **Recovery scheme**

Deadlock Detection
Single Instance of Each Resource Type

- **Maintain *wait-for* graph**
  - » Nodes are processes.
  - » removes resource nodes and collapsing edges
  - » $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- **Periodically invoke an algorithm that searches for a cycle in the graph.**
- **An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.**



Depth first search (example)

For each node in the graph:

```
L = {empty list} and Nodes = {unvisited};
current node = initial node ;
while( current node is not the initial node twice )
     L.enqueue(current node); // add to node to end of L
     if( current node is in L twice )
        there is a cycle ⇒ cycle and return
     if( there is an unmarked arc )
        mark the arc as visited and use destination as new
          current node
     else
        go back to previous node
Back to initial node there is no cycle
```

Deadlock detection (1 resource of each)

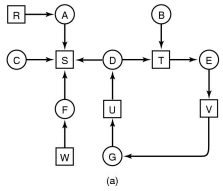
- Do a depth-first-search on the resource allocation graph

D, E, G ?

are deadlocked

A, C, F ?

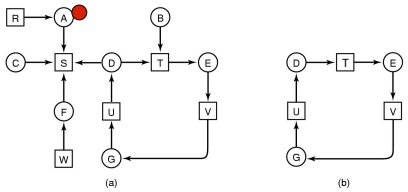
are not deadlocked because S can be allocated to either and then the other two can take turn to complete



Example: Deadlock Detection

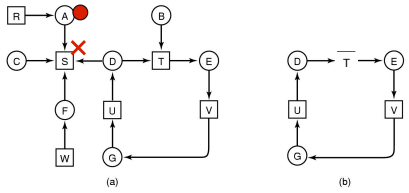
- Do a depth-first-search on the resource allocation graph

Initialize a list to the empty list, designate arcs as 'unvisited'



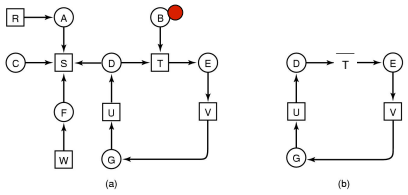
Example: Deadlock Detection

- Do a depth-first-search on the resource allocation graph



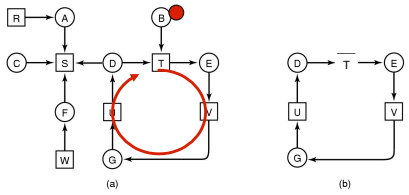
Example: Deadlock Detection

- Do a depth-first-search on the resource allocation graph



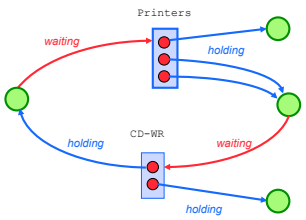
Example: Deadlock Detection

- Do a depth-first-search on the resource allocation graph

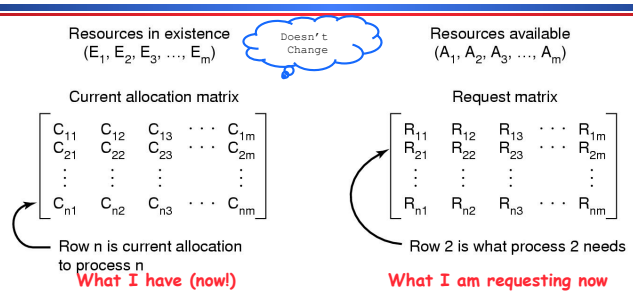


Deadlock Detection with Multiple Resources

- **Theorem**: ***If a graph does not contain a cycle then no processes are deadlocked***
  - » **A cycle in a RAG is a necessary condition for deadlock**
  - » **Is it a sufficient condition?**



Deadlock Detection Algorithm: Multiple Resource Instances



- ***Available*: Indicates the number of available resources of each type (m)**
- ***Allocation*: Number of resources of each type currently allocated (nxm)**
- ***Request*: current requests of each thread (nxm)**
  - » **If *Request* $[i_j] = k$, then process P_i is requesting k more instances of type. R_j.**

Example

- **Is there a possible allocation sequence of resources so that each process can complete?**

Tape drives, Plotters, Scanners, CD Roms

E = (4 2 3 1)

A = (2 1 0 0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Detection algorithm

A marked process means it can run to completion

1. **Look for an unmarked process Pi, for which the *i*th row of R (need) is less than or equal to A**
2. **If such a process is found, add the i-th row of C to A, mark the process and go back to step 1**
3. **If no such process exists the algorithm terminates**

If all marked, no deadlock

Detection algorithm

Tape drives, Plotters, Scanners, CD Roms

E = (4 2 3 1)

A = (2 1 0 0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Detection algorithm

Tape drives, Plotters, Scanners, CD Roms

E = (4 2 3 1)

A = (2 1 0 0)

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Detection algorithm

Tape drives, Plotters, Scanners, CD Roms

E = (4 2 3 1)

A = (2 1 0 0)

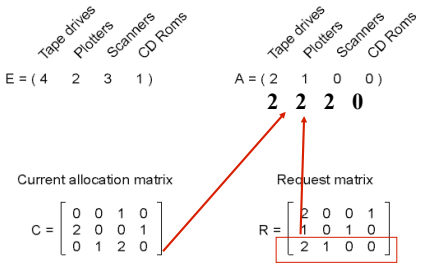
Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

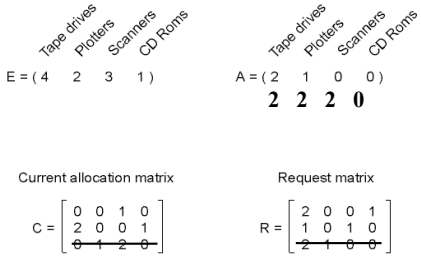
Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

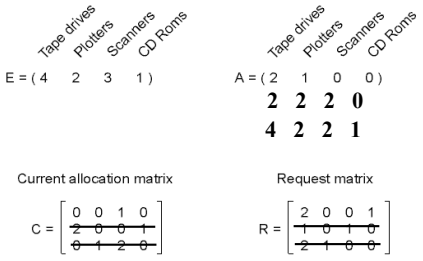
Detection algorithm



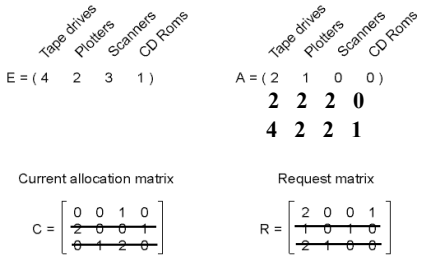
Detection algorithm



Detection algorithm



Detection algorithm



No deadlock!

Deadlock detection issues

- How often should the algorithm run?
  - After every resource request?
  - Periodically?
  - When CPU utilization is low?
  - When we suspect deadlock because some thread has been asleep for a long period of time?

Recovery from deadlock

- What should be done to recover?
  - Abort deadlocked processes and reclaim resources
  - Temporarily reclaim resource, if possible
  - Abort one process at a time until deadlock cycle is eliminated
- Where to start?
  - Low priority process
  - How long process has been executing
  - How many resources a process holds
  - Batch or interactive
  - Number of processes that must be terminated

Other deadlock recovery techniques

- Recovery through rollback
  - » Save state periodically
    - – take a checkpoint
    - – start computation again from checkpoint
  - » Done for large computation systems

Review: Handling Deadlock

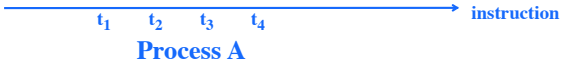
Ostrich algorithm

- Ignore
  - » Easiest and most common approach (e.g., UNIX).
- Deadlock prevention
  - » Ensure deadlock does not happen
  - » Ensure at least one of 4 conditions does not occur
- Deadlock detection and recovery
  - » Allow deadlocks, but detect when occur
  - » Recover and continue
- Deadlock avoidance
  - » Ensure deadlock does not happen
  - » Use information about resource requests to dynamically avoid unsafe situations

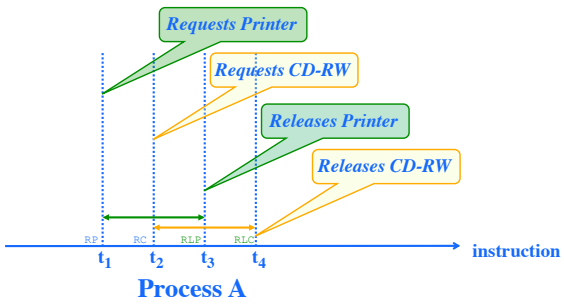
Deadlock avoidance

- Detection vs. avoidance…
  - » Detection – "optimistic" approach
    - – Allocate resources
    - – "Break" system to fix it
  - » Avoidance – "pessimistic" (conservative) approach
    - – Don't allocate resource if it may lead to deadlock
    - – If a process requests a resource...
      ... make it wait until you are sure it's OK
  - » Which one to use depends upon the application

Process-resource trajectories

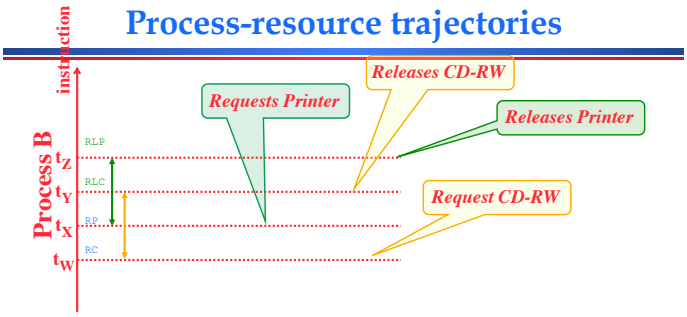


Process-resource trajectories

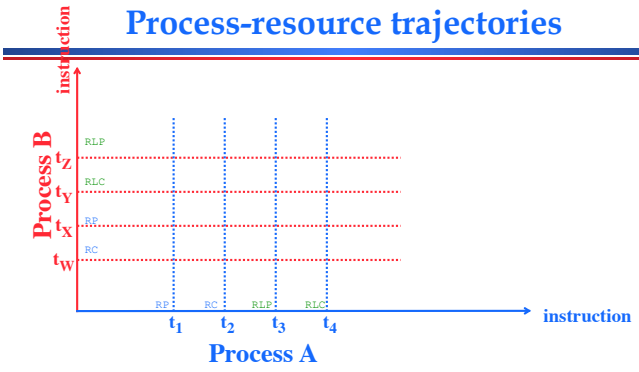


Process-resource trajectories

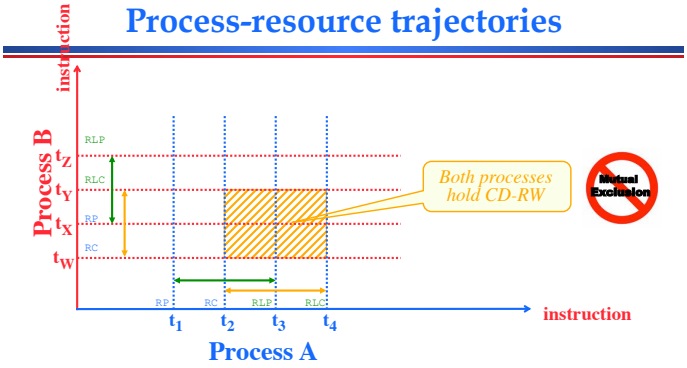
Process-resource trajectories



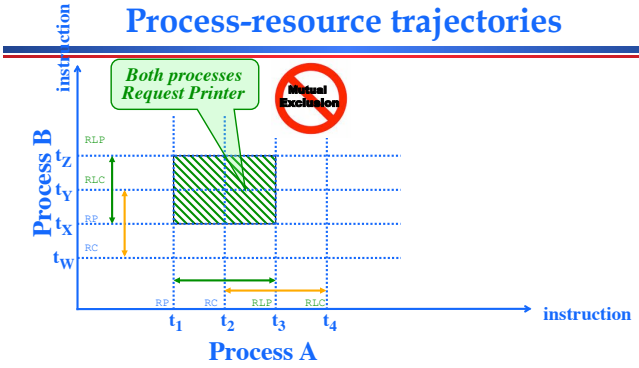
Process-resource trajectories



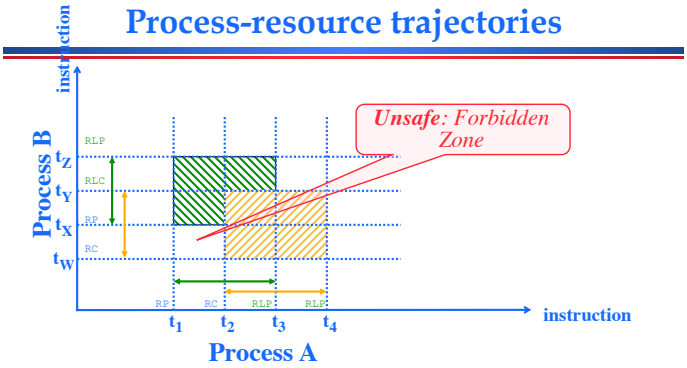
Process-resource trajectories



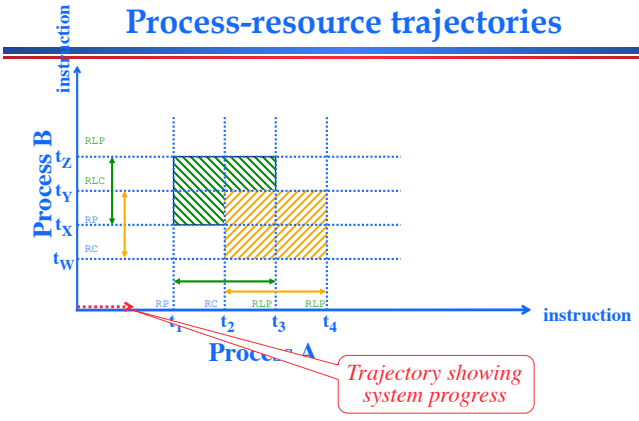
Process-resource trajectories



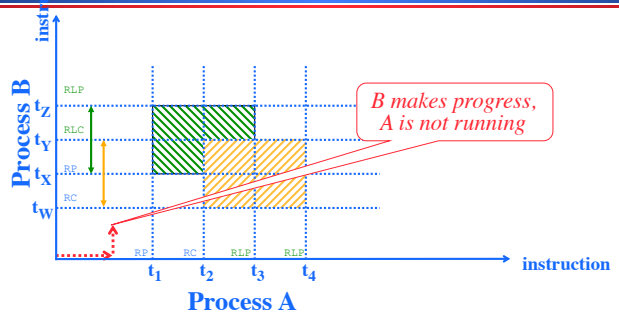
Process-resource trajectories



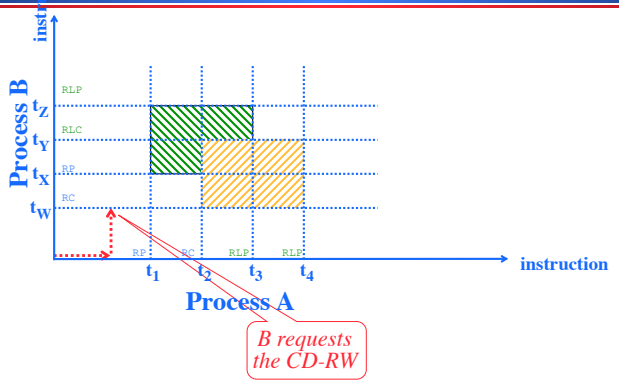
Process-resource trajectories



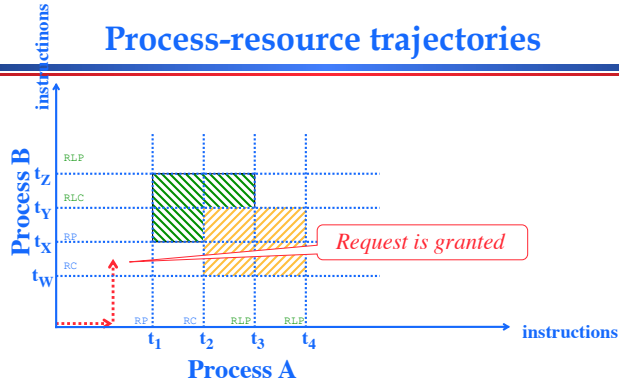
Process-resource trajectories



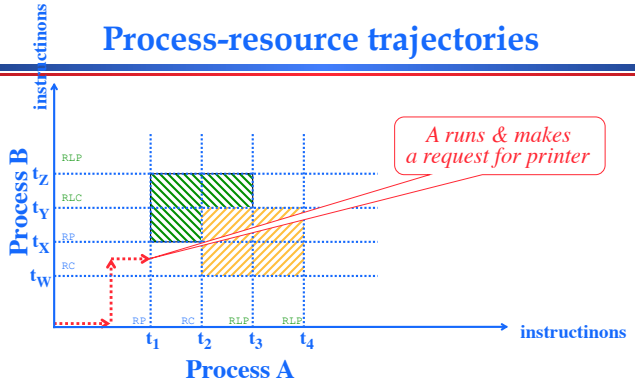
Process-resource trajectories



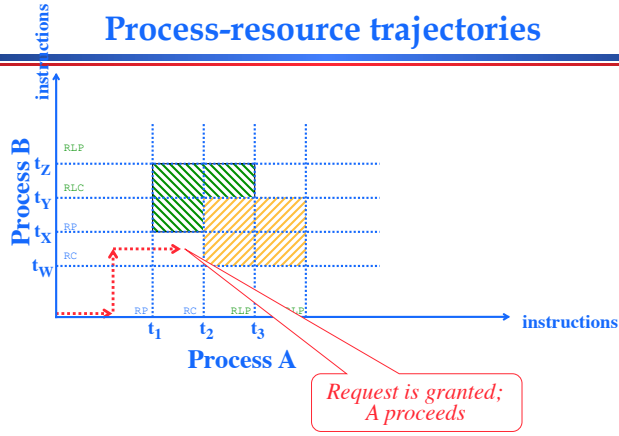
Process-resource trajectories



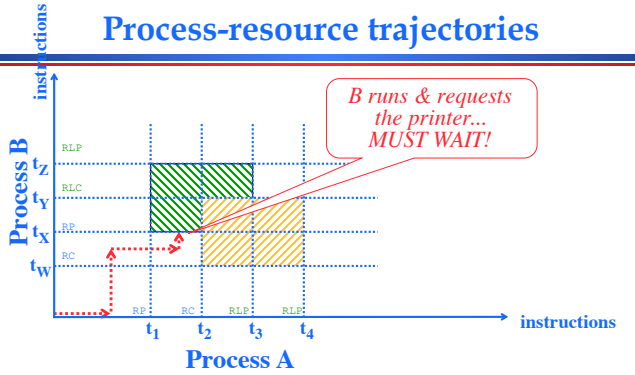
Process-resource trajectories



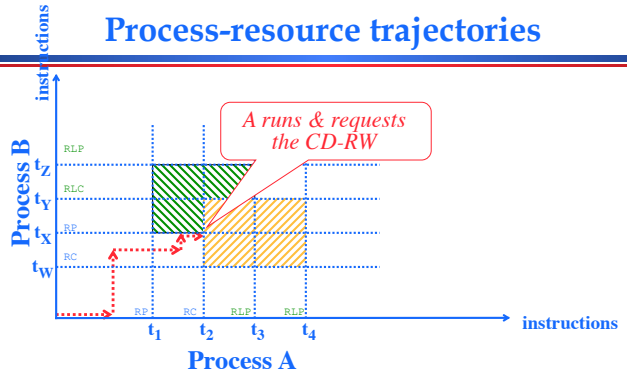
Process-resource trajectories



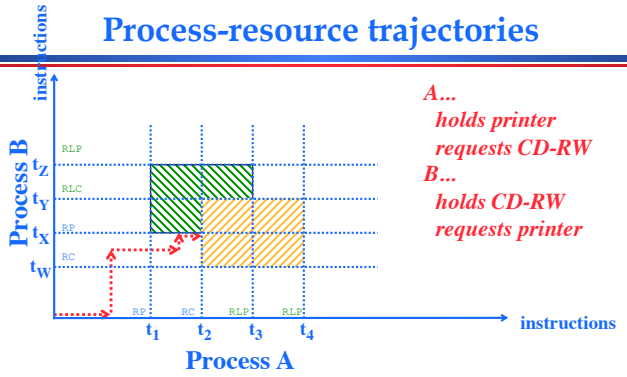
Process-resource trajectories



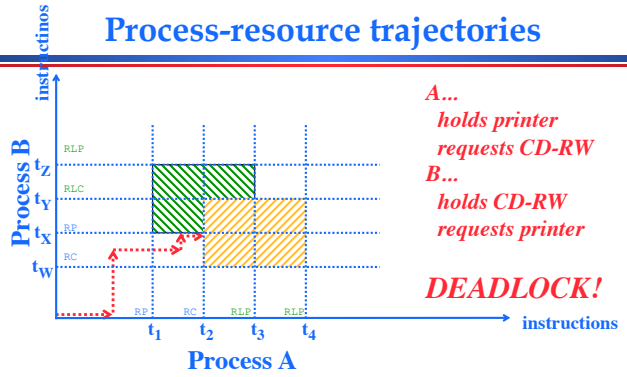
Process-resource trajectories



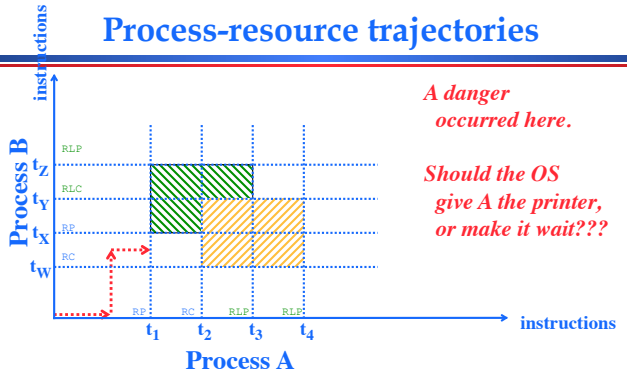
Process-resource trajectories



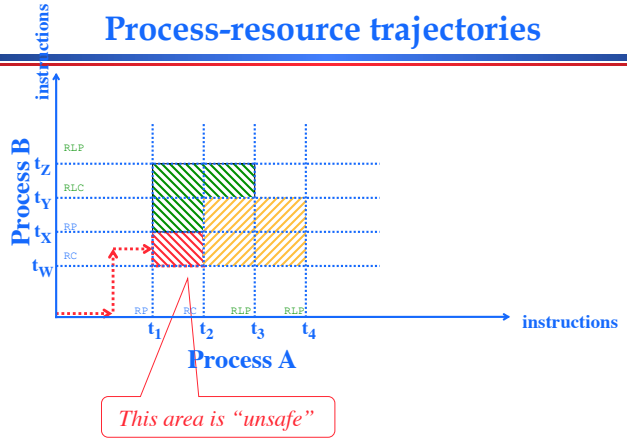
Process-resource trajectories



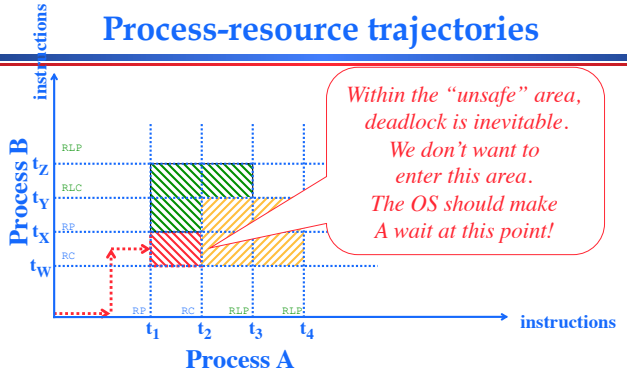
Process-resource trajectories



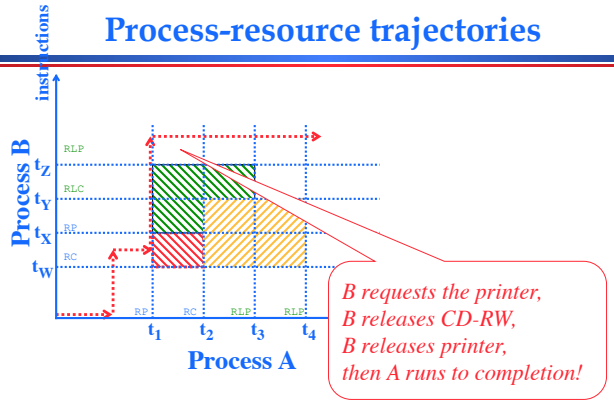
Process-resource trajectories



Process-resource trajectories



Process-resource trajectories



Safe states

- The current state:
  - "which processes hold which resources"
- A "safe" state:
  - » No deadlock, *and*
  - » There is some scheduling order in which every process can run to completion even if all of them request their maximum number of units immediately
- The Banker's Algorithm:
  - » *Goal:* Avoid unsafe states!!!
  - » *Question: When a process requests more units, should the system grant the request or make it wait?*

Deadlock Avoidance

- Dijkstra's Banker's Algorithm
- Idea: Avoid unsafe states of processes holding resources
  - » Unsafe states might lead to deadlock if processes make certain future requests
  - » When process requests resource, only give if doesn't cause unsafe state
  - » Problem: Requires processes to specify all possible future resource demands

The Banker's Algorithm

- Assumptions:
  - » Only one type of resource, with multiple units.
  - » Processes declare their maximum potential resource needs ahead of time (total sum is 22 units of credit but only has 10)
- *When a process requests more units should the system make it wait to ensure safety?*

Example: One resource type with 10 units

	Has	Max	
A	3	9	6
B	2	4	2
C	2	7	5

Free: 3

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

Safe states

- Safe state – "when system is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resource immediately"

10 total

	Has	Max	
A	3	9	6
B	2	4	2
C	2	7	5

Free: 3

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

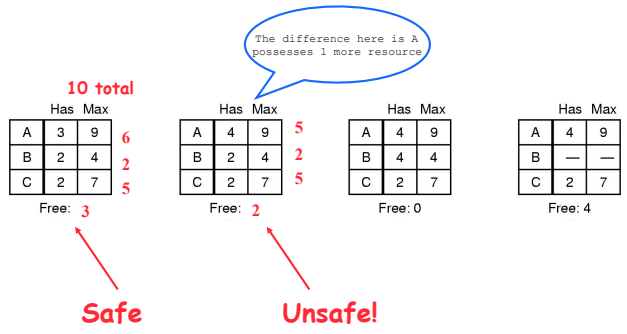
	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

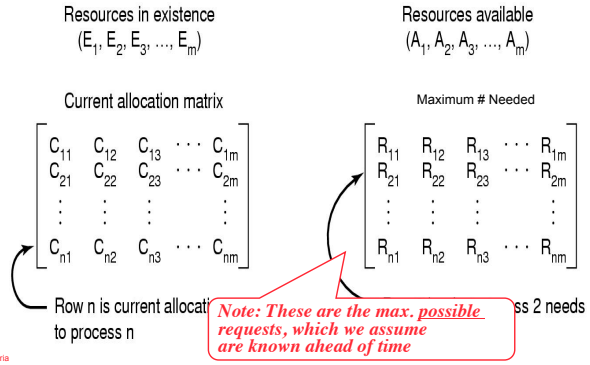
	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

Unsafe/Safe state?



Avoidance with multiple resource types

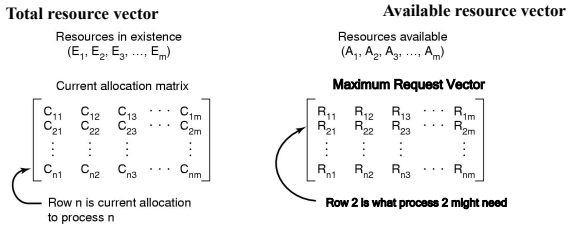


Note: These are the max. possible requests, which we assume are known ahead of time

Banker's algorithm for multiple resources

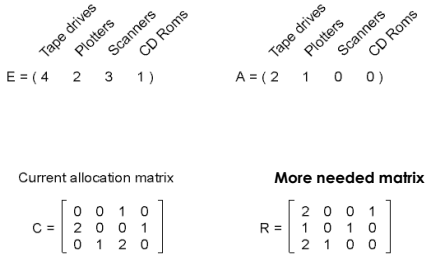
- **Look for a row, *R*, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion**
- **Assume the process of the row chosen requests all the resources that it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to A vector**
- **Repeat steps 1 and 2, until either all process are marked terminated, in which case the initial state was safe, or until deadlock occurs, in which case it was not**

Avoidance modeling

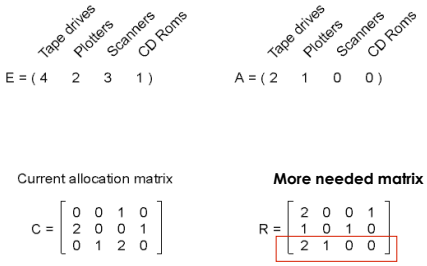


RUN ALGORITHM ON EVERY RESOURCE REQUEST

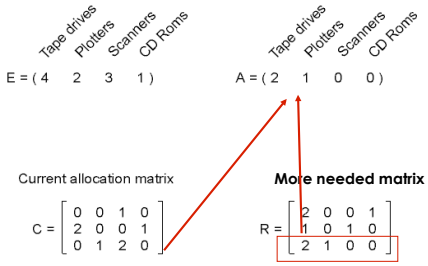
Avoidance algorithm



Avoidance algorithm



Avoidance algorithm



Avoidance algorithm

Tape drives Plotters Scanners CD Roms

E = (4 2 3 1)

Tape drives Plotters Scanners CD Roms

A = (2 1 0 0)

2 2 2 0

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

More needed matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Avoidance algorithm

Tape drives Plotters Scanners CD Roms

E = (4 2 3 1)

Tape drives Plotters Scanners CD Roms

A = (2 1 0 0)

2 2 2 0

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ \sout{0} & \sout{1} & \sout{2} & \sout{0} \end{bmatrix}$$

More needed matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ \sout{2} & \sout{1} & \sout{0} & \sout{0} \end{bmatrix}$$

Avoidance algorithm

Tape drives Plotters Scanners CD Roms

E = (4 2 3 1)

Tape drives Plotters Scanners CD Roms

A = (2 1 0 0)

2 2 2 0

4 2 2 1

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ \sout{2} & \sout{0} & \sout{0} & \sout{1} \\ \sout{0} & \sout{1} & \sout{2} & \sout{0} \end{bmatrix}$$

More needed matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ \sout{1} & \sout{0} & \sout{1} & \sout{0} \\ \sout{2} & \sout{1} & \sout{0} & \sout{0} \end{bmatrix}$$

Deadlock avoidance

- **Deadlock avoidance is usually impossible**
  - » **because you don't know in advance what resources a process will need!**