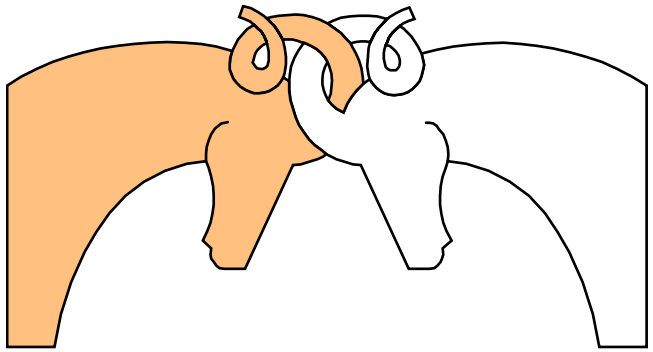
CIS 520, *Operating Systems Concepts*

Lecture 5

Deadlocks (Part 2 of 2)



The Four Strategies for Dealing with Deadlocks

1. The *Ostrich Algorithm* (Stick you head in the sand and pret[illegible]oblem at all)
2. *Prevention* by [illegible] the four necessary conditions
3. *Detection* and *recovery*
4. Dynamic *avoidance* by specially designed resource allocation

Done

Deadlock Detection and Recovery

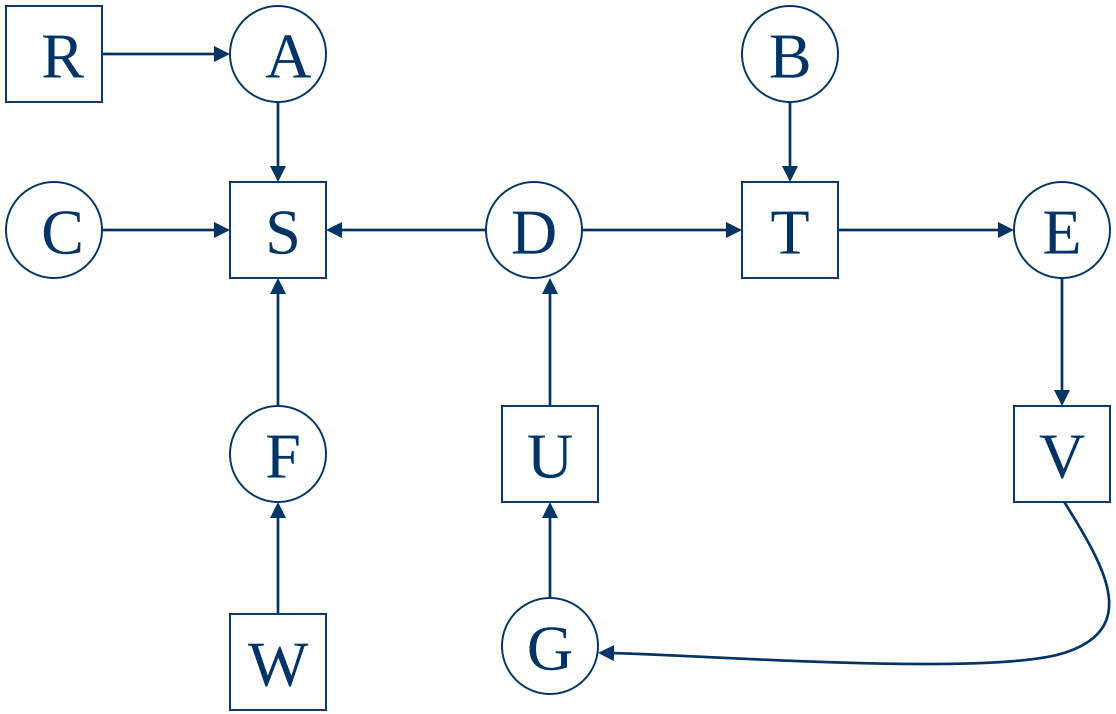
- With this technique, the system does not attempt to *prevent* deadlocks. It lets them occur, finds out what happened, and attempts to recover after the fact.
- In what follows, we will concentrate on
  1. deadlock detection in the case of single-instance resources, and then proceed with
  2. deadlock detection for multiple-instance resources case

Deadlock Detection in the Case of Single-Instance Resources (An example)

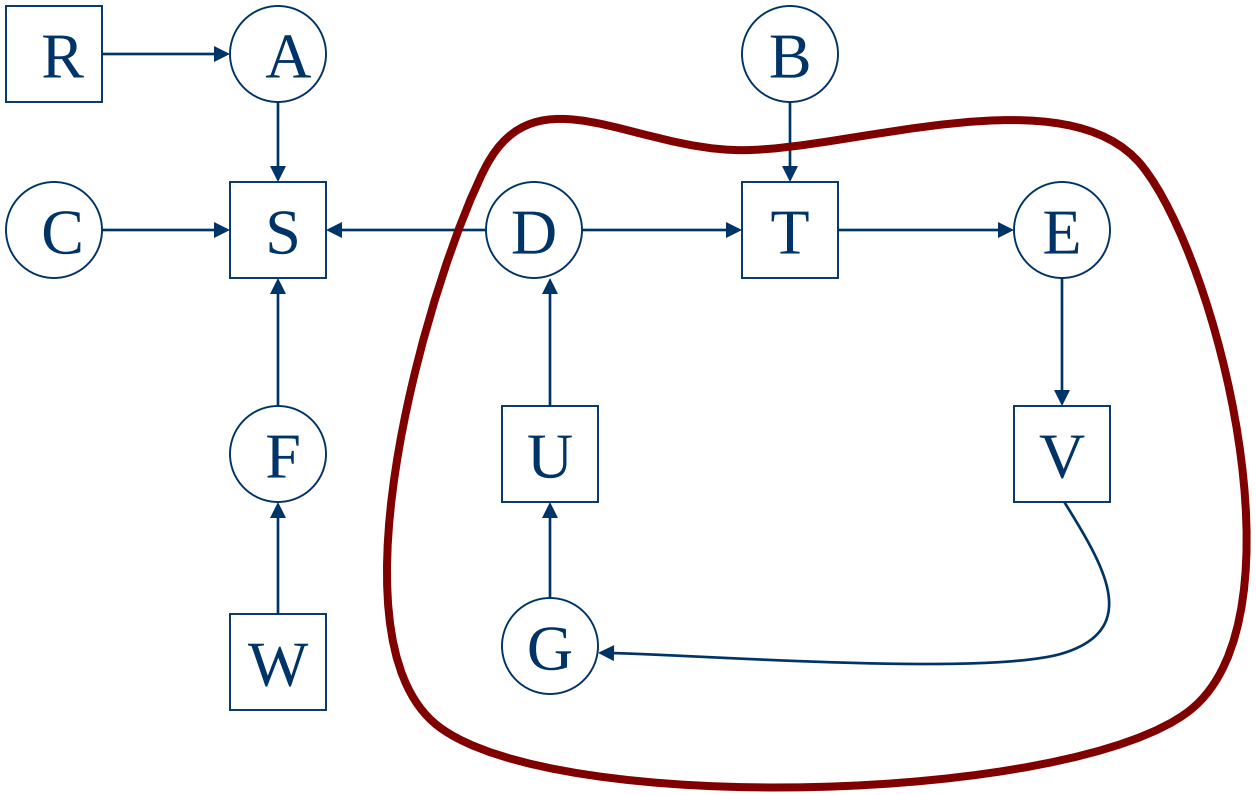
Consider a system with seven processes, *A* through *G*, six resources, *R* through *W*, and the following history:

1. Process *A* holds *R* and requests S
2. Process *B* holds nothing and requests *T*
3. Process *C* holds nothing and requests *S*
4. Process *D* holds *U* and requests *S* and *T*
5. Process *E* holds *T* and requests *V*
6. Process *F* holds *W* and requests *S*
7. Process *G* holds *V* and requests *U*

Deadlock Detection in the Case of Single-Instance Resources (An example system graph with a deadlock)



Deadlock Detection in the Case of Single-Instance Resources (An example system graph with a deadlock)



1. *S* can be allocated to *A*, *C*, and *F*, which will then complete.

2. *(B* is in trouble waiting for T, but it is not part of the deadlock—no one is blocked by it!)

3. *D*, *E*, and *G* are truly deadlocked!

Deadlock Detection in the Case of Single-Instance Resources (cont.)

- An algorithm for detecting the deadlock **in the case of single instance resources** reduces to the algorithm for finding a cycle in a (directed) graph
- In general, we can use any algorithm (such as *depth-first search*) for finding a cycle in a directed graph
- The *worst-case* complexity of finding a cycle in a graph is $O(n^2)$, where n is the number of vertices

A step aside: Theory of Complexity

We say that

$$x_n = O(y_n),$$

if there are constants C and N such that

$$x_n < Cy_n, \text{ for all } n > N.$$

When *n* defines the problem size, the asymptotic behavior of the execution time is essential for determining if the program can be used at all for the problem. Look at the table on the following slide (from *The Design and Analysis of Computer Algorithms* by Aho, Hopcroft, and Ullman—I highly recommend this book, by the way!) and decide what sizes of problems can an operating system deal with...

Limits on Problem Size

(A. Aho, J. E. Hopcroft, and J. D. Ullman; *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974, p.3)

Algorithm	Time complexity	Maximum problem size		
		1 sec	1 min.	1 hour
A_1	n	1000	$6 \bullet 10^4$	$3.6 \bullet 10^6$
A_2	$n \log n$	140	4893	$6 \bullet 10^4$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Deadlock Detection in the Case of Multiple-Instance Resources

Structures and Definitions

Let

- *n* be the number of processes in the system
- *m* the number of resource types
- ***E*** **=** $(e_1, e_2, ..., e_m)$ the vector of total *(existing)* numbers of resources in each resource instance
- ***A*** **=** $(a_1, a_2, ..., a_m)$ the vector of the numbers of *available* resources (after some have been allocated)

$$C = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}, \qquad R = \begin{bmatrix} R_{11} & R_{12} & \dots & R_{1m} \\ R_{21} & R_{22} & \dots & R_{2m} \\ \dots & \dots & \dots & \dots \\ R_{n1} & R_{n2} & \dots & R_{nm} \end{bmatrix}$$

respectively, the matrices of current *allocation* of resources to processes and **present** *requests* of processes

Deadlock Detection in the Case of Multiple-Instance Resources

Structures and Definitions (cont.)

- We say that vector $\boldsymbol{A} = (a_1, a_2, ..., a_k)$ is less than vector $\boldsymbol{B} = (b_1, b_2, ..., b_k)$

$$\boldsymbol{A} < \boldsymbol{B}$$

if and only if

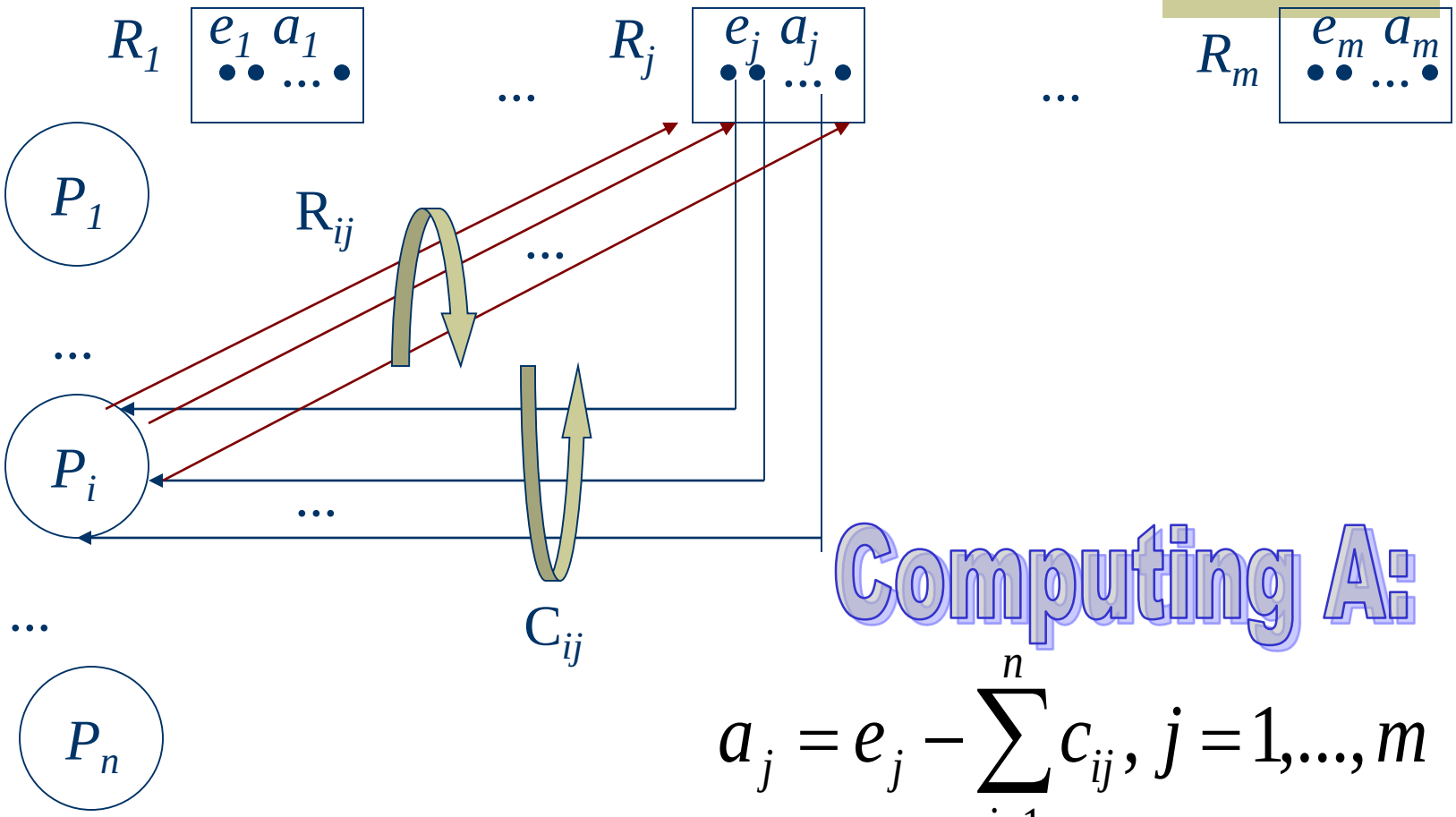
$$a_i < b_i \text{ for } i = 1, 2, ..., k.$$

(All other inequality relations extend similarly from scalar to vector types.)

- An i-th row of matrix $C = \begin{bmatrix} C_{11} & C_{12} & ... & C_{1m} \\ C_{21} & C_{22} & ... & C_{2m} \\ ... & ... & ... & ... \\ C_{n1} & C_{n2} & ... & C_{nm} \end{bmatrix}$ is sometimes written as $\mathbf{C}_i$:

$$\mathbf{C}_i = (c_{i1}, c_{i2}, ... c_{im})$$

Deadlock Detection in the Case of Multiple-Instance Resources
Structures and Definitions (cont.)



Computing A:

$$a_j = e_j - \sum_{i=1}^{n} c_{ij},\ j = 1,\ldots,m$$

Deadlock Detection in the Case of Multiple-Instance Resources
Detection Algorithm Strategy

- Look for a process that can be run to completion (by giving it all the resources it has requested)
- Consider it finished and take away its resources
- Repeat the above steps until no processes that can run to completion can be found
- All processes that are left (if any) are deadlocked; if none are left, there is no deadlock

Deadlock Detection in the Case of Multiple-Instance Resources
Detection Algorithm

```
j=0;
Initialize all processes to unmarked;
loop = TRUE;
do
  {
   look for an unmarked process Pi: Ri ≤ A;
          if found
             {
                  j++;
                  mark Pi ;
                  A = A + Ci ;
             }
           else
              loop = FALSE;
   } while loop && (j<n);
     /* The unmarked processes, if any, are deadlocked*/
```

Deadlock Detection in the Case of Multiple-Instance Resources
Detection Algorithm Use Example

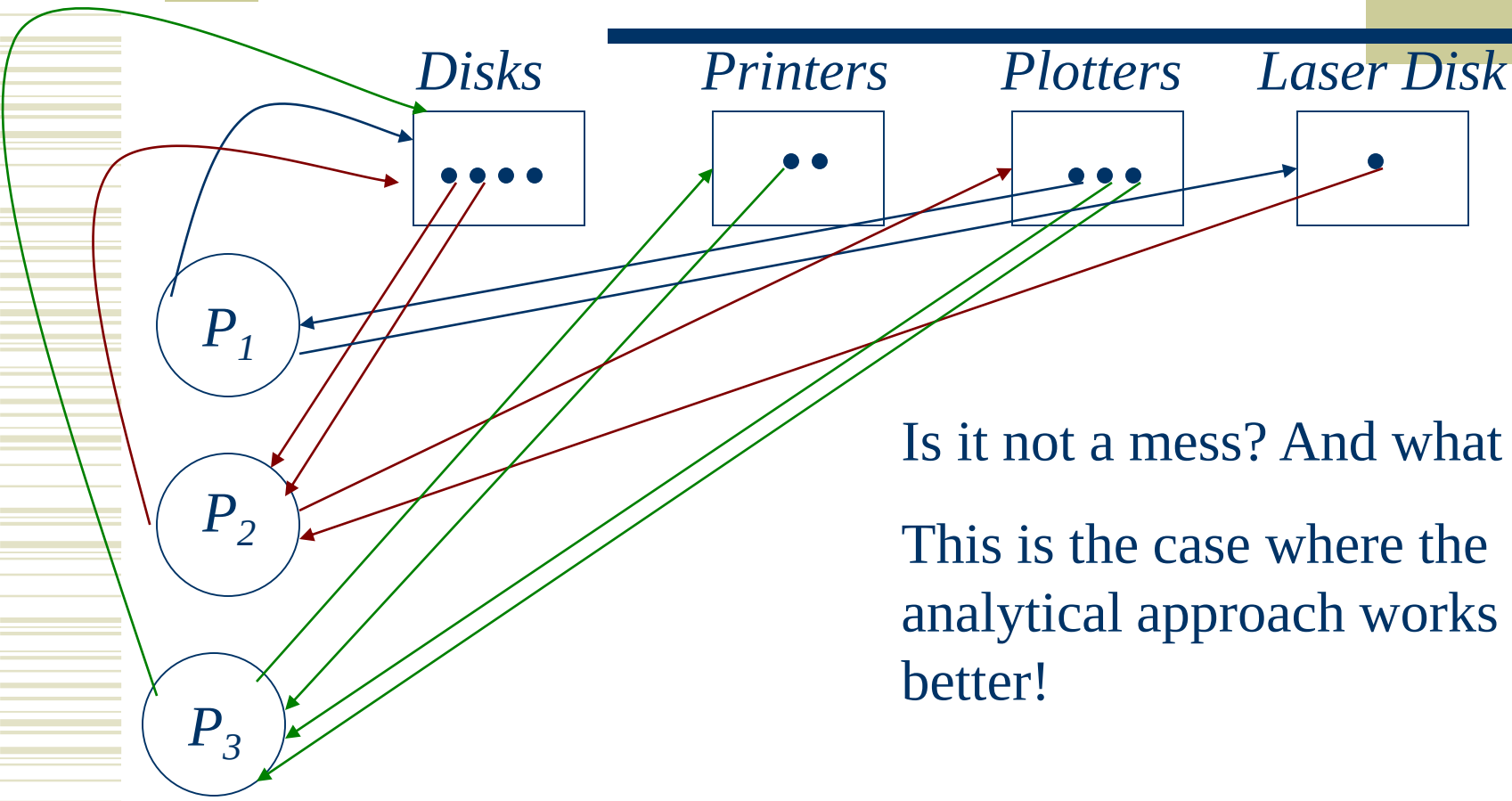
A system has four disks, two printers, three plotters, and one laser disk. [Thus **E** = (4, 2, 3, 1).]

Three processes are running, and the state of the resource use and requests is as follows:

A = (2, 1, 0, 0)

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

Deadlock Detection in the Case of Multiple-Instance Resources
Detection Algorithm Use Example (cont.)



Is it not a mess? And what a mess!

This is the case where the analytical approach works much better!

Deadlock Detection in the Case of Multiple-Instance Resources
Detection Algorithm Use Example (cont.)

1) $i = 3$: $\boldsymbol{R_3} <= \boldsymbol{A}; \boldsymbol{A} = \boldsymbol{A} + \boldsymbol{C_3} =$
$= (2, 1, 0, 0) + (0, 1, 2, 0) =$
$= (2, 2, 2, 0);$

2) $i = 2$: $\boldsymbol{R_2} < \boldsymbol{A}; \boldsymbol{A} = \boldsymbol{A} + \boldsymbol{C_2} =$
$= (2, 2, 2, 0) + (2, 0, 0, 1) =$
$= (4, 2, 2, 1);$

2) $i = 1$: $\boldsymbol{R_1} < \boldsymbol{A}; \boldsymbol{A} = \boldsymbol{A} + \boldsymbol{C_1} =$
$= (4, 2, 2, 1) + (0, 0, 1, 0) =$
$= (4, 2, 3, 1) \Rightarrow$ **No deadlock!**

$\boldsymbol{A} = (2, 1, 0, 0)$

$\boldsymbol{E} = (4, 2, 3, 1)$

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

Deadlock Detection in the Case of Multiple-Instance Resources
Detection Algorithm Use Example (cont.)

Now, let us see what would happen if P_3 needed a laser disk...

A = *(2, 1, 0, 0)*

E **=** *(4, 2,3,1)*

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

1

Deadlock Detection in the Case of Multiple-Instance Resources *Detection Algorithm* Complexity

```
Initialize all processes to unmarked;
j=0; loop = TRUE;
do
  {
   look for an unmarked process Pi: Ri ≤ A ;
         if (found)
            {
                 mark Pi ; j++;
                 A = A + Ci ;
            }
          else
             loop = FALSE;
       } while loop && (j < n;
   } /* The unmarked processes, if any, are deadlocked*/
```

In the *worst case* the good process is always at the end of the loop, so we will repeat *n* searches *O(n)* times, each time adding *O(m)* operations ending with total $O(n^2m)$ operations.

When Do We Look for Deadlocks (on a Mainframe)?

- When users complain that they get no response, but the system is running
- When CPU utilization becomes and stays very low (10%-20%)

Deadlock Recovery
We detected it—what's next?

- **Preemption:** In some cases, resources can be taken away from a process—preemption, as an action of an OS. (Imagine though what would need to be done with a preempted printout?) In general, even manual preemption is difficult or impossible
- **Murder:** If it is essential to recover from a deadlock, one of the deadlocked (and relatively unimportant) processes can be simply killed, and its resources taken back. *Maybe* then the system will get out of deadlock, but maybe not, so another process will need to be killed, and so on...
- **Rollback:** In the systems where deadlocks are likely, it is essential to implement *checkpointing*, where the state information of a process (including its resource use) is saved so that the process can restart completely from any checkpoint. In such cases, the resources can be taken from a process to get out of deadlock, and the process is *rolled back* to an earlier checkpoint, where it used fewer resources. Even if the process is killed, it could be restarted from the latest checkpoint later.

The Four Strategies for Dealing with Deadlocks

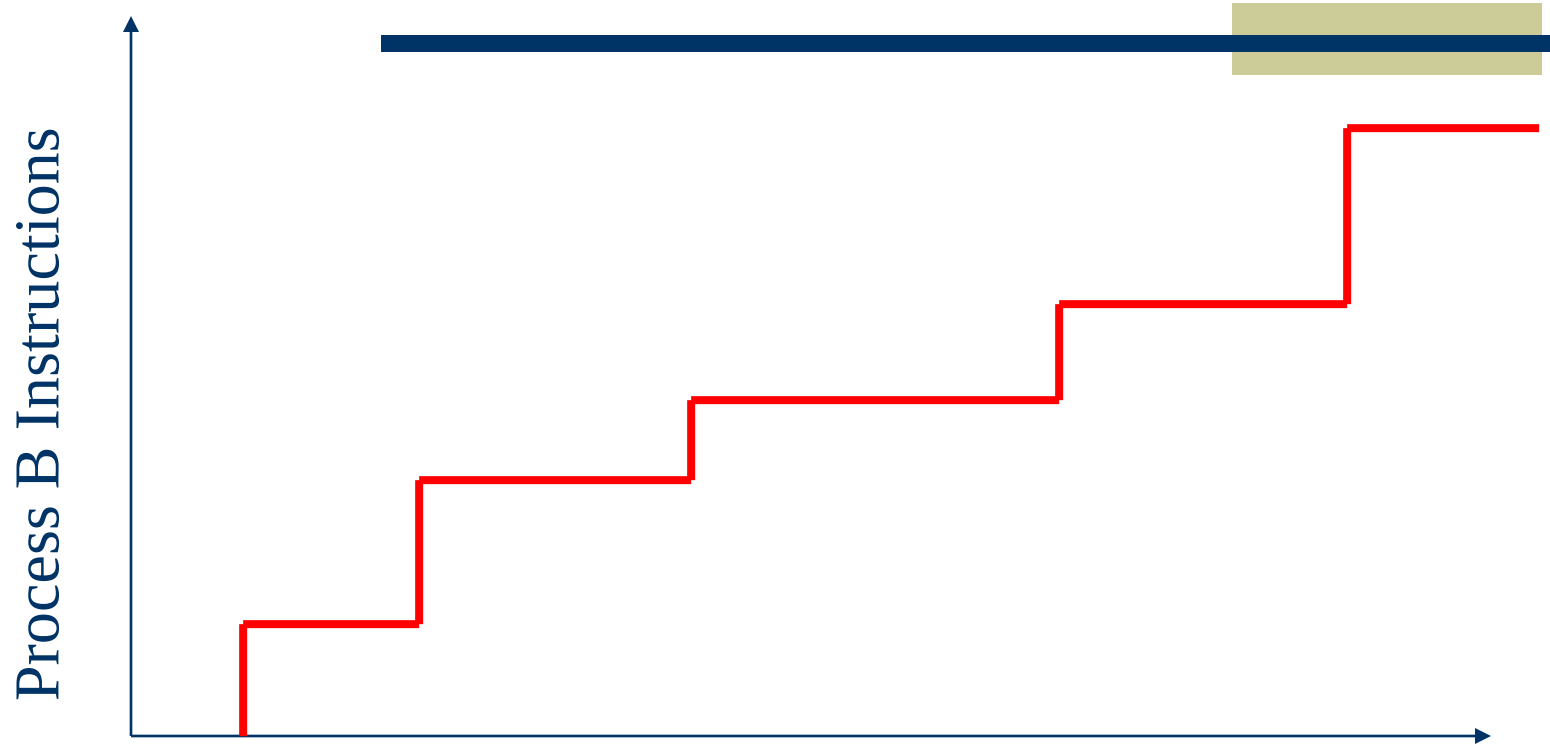
1. The *Ostrich Algorithm* (Stick you head in the sand and pretend there is no problem at all)
2. *Prevention* (make sure that at least four necessary conditions is not satisfied)
3. *Detection* and *recovery*
4. Dynamic *avoidance* by specially designed resource allocation

Done

Dynamic Avoidance

1. Here, the system neither statically prevents deadlocks from happening (e.g., by imposing a protocol), nor deals with the deadlocks after the fact
2. Instead, we watch the dramatic development of granting the resources and interfere the moment the situation becomes dangerous (i.e., may lead to the deadlock)
3. As in the case of the *Bicycles and Cars* problem, the idea of the *state space* is the key to the solution

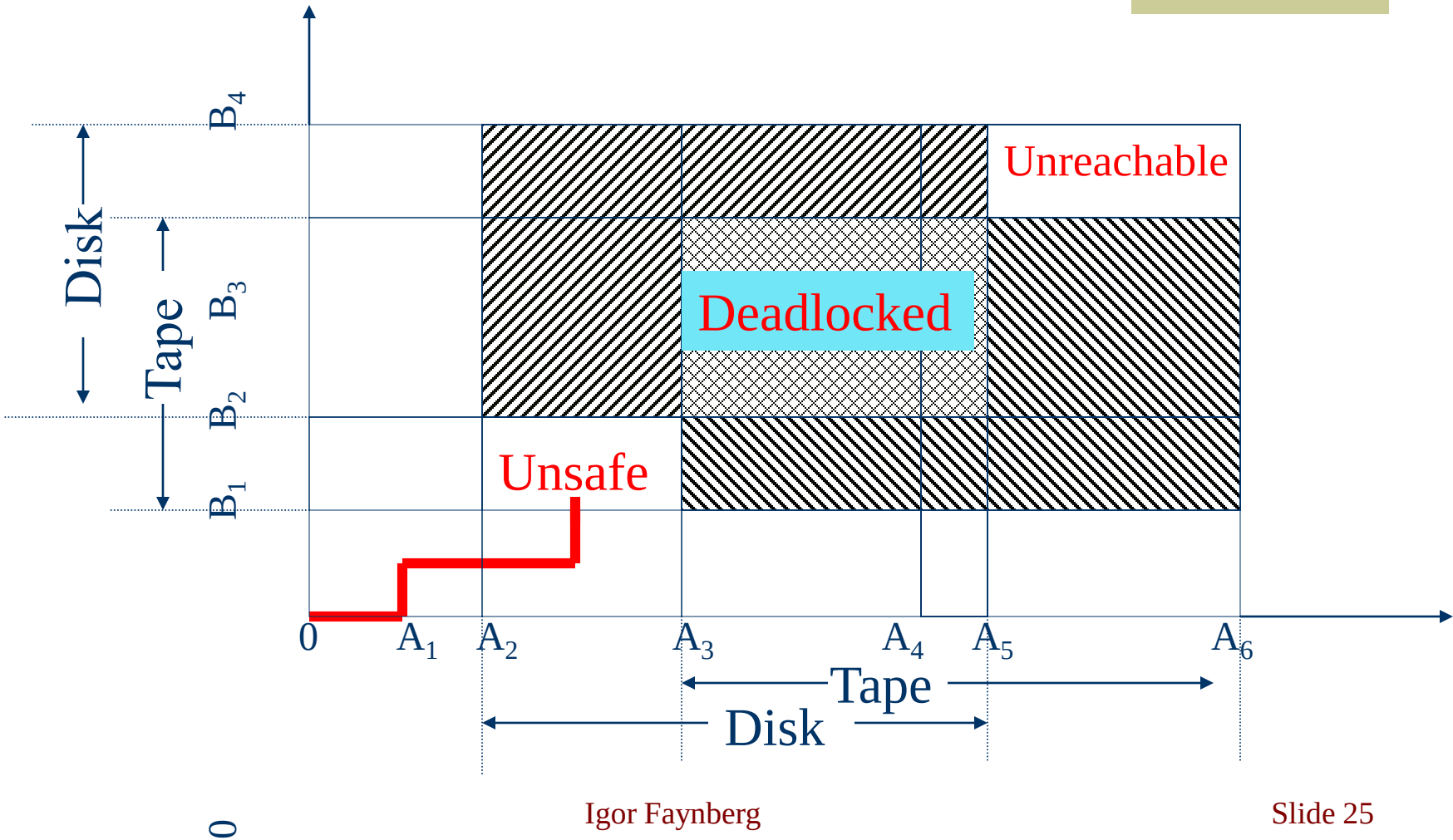
Resource Trajectories (Two Processes)



- The Scheduler Determines the Trajectory
- With one CPU, it may move *only* north or east

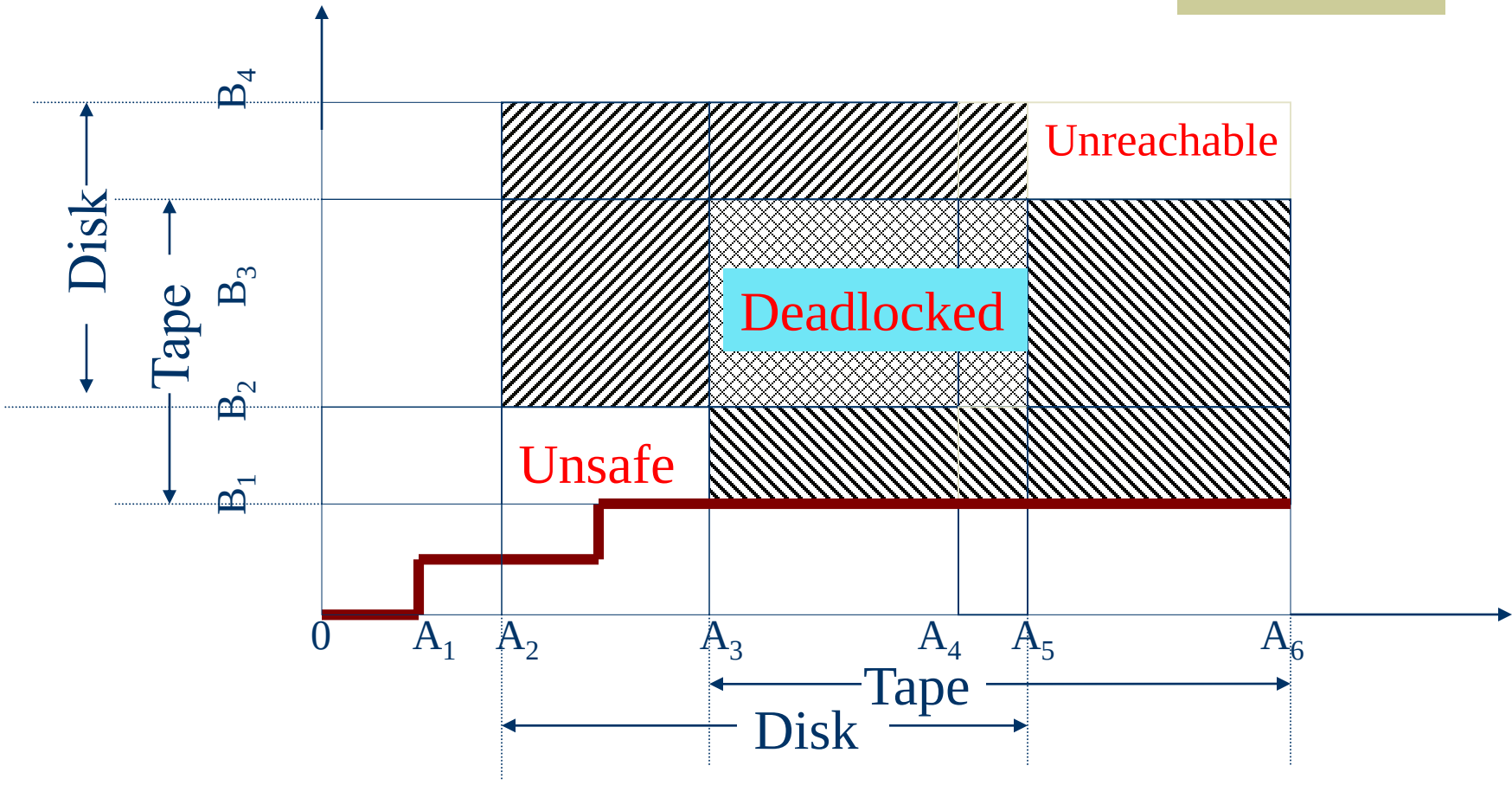
The Resource Trajectory of a System with Two Processes, a Tape, and a Disk

Entering *unsafe* state



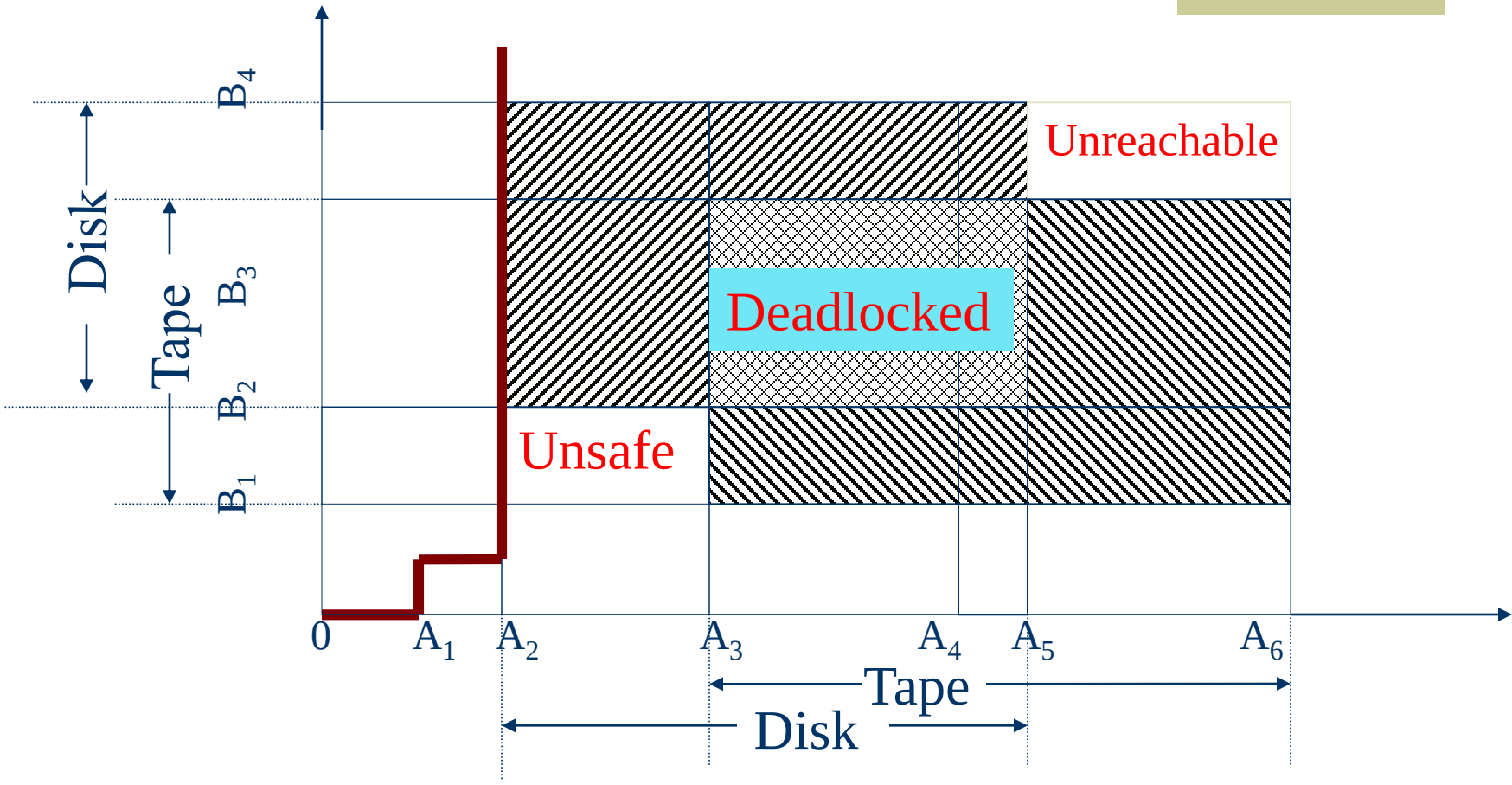
The Resource Trajectory of a System with Two Processes, a Tape, and a Disk

Avoiding *unsafe* state by letting A through



The Resource Trajectory of a System with Two Processes, a Tape, and a Disk

Avoiding *unsafe* state by letting B through



Deadlock Avoidance in the Case of Multiple-Instance Resources

Structures and Definitions

Compare with the structures for detection

Let

- n be the number of processes in the system
- m the number of resource types
- $\boldsymbol{E} = (e_1, e_2, ..., e_m)$ the vector of total (*existing*) numbers of resources in each resource instance—**a system constant**
- $\boldsymbol{A} = (a_1, a_2, ..., a_m)$ the vector of the *available* numbers of resources (after some have been allocated)—**a state variable**

$$C = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \dots & \dots & \dots & \dots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}, \qquad N = \begin{bmatrix} N_{11} & N_{12} & \dots & N_{1m} \\ N_{21} & N_{22} & \dots & N_{2m} \\ \dots & \dots & \dots & \dots \\ N_{n1} & N_{n2} & \dots & N_{nm} \end{bmatrix}$$

respectively, the matrices of current *allocation* of resources to processes and **the present *needs*** of processes for resources

Yet Another System Constant

$$M = \begin{bmatrix} M_{11} & M_{12} & \dots & M_{1m} \\ M_{21} & M_{22} & \dots & M_{2m} \\ \dots & \dots & \dots & \dots \\ M_{n1} & M_{n2} & \dots & M_{nm} \end{bmatrix}$$

M is a matrix of maximum resources of each type a process can hold at any given moment

Thus, $\boldsymbol{N = M - C}$.

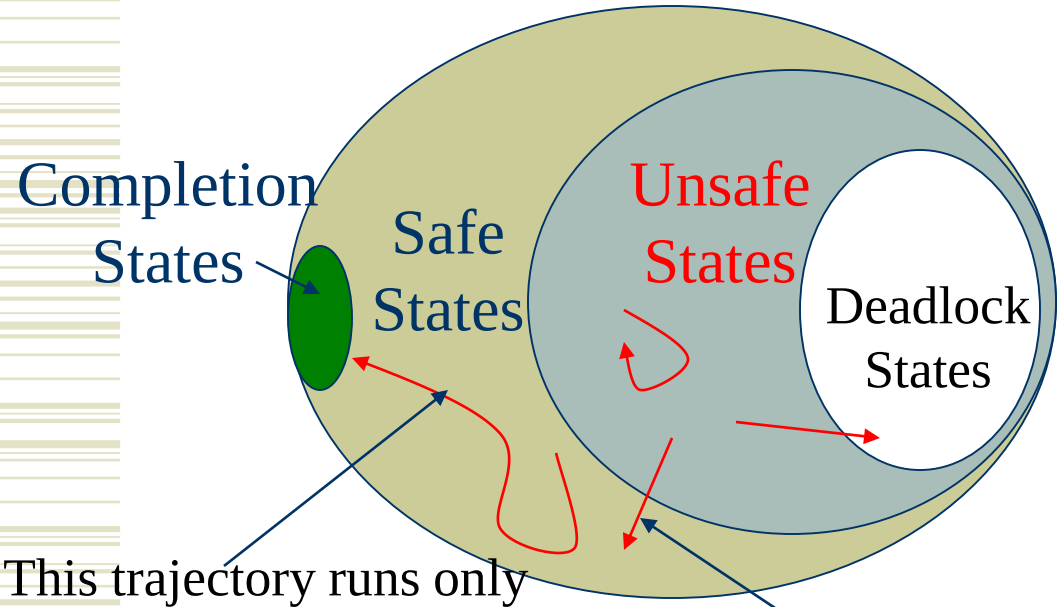
And, remember, $A = E - \sum_{i=1}^{n} C_i,$

so ***C*** determines the system state.

A *Safe* and *Unsafe States*: Definition

- A state (***C***) is called *safe* if there is a way to satisfy all pending requests by running the processes in some order (without releasing any resources already granted). In other words, there is a trajectory in the n-dimensional space of the processes, which bypasses the *k*-dimensional deadlock parallelepipeds
- A state is *unsafe* if it is not safe (i.e., there is no sequence of resource assignments in which all processes can complete—without releasing the resources already granted)

Safe, Unsafe, Deadlock



- An unsafe state does not *necessarily* lead to a deadlock, but a deadlock state *is* unsafe

This trajectory runs only through the states where processes *never* release their resources before completion

This trajectory depends on processes releasing their resources before asking for others

The *Banker's Algorithm*

- The scheduling algorithm for avoiding deadlocks was invented (or *discovered*?) by Dijkstra in 1965
- In its single-resource version, the problem is similar to that of a banker's who has *k* dollars to start with
- This amount is borrowed by customers, whose total demand is greater than *k*, but as they return the money (no interest is considered in this model), others can borrow. The demands of customers are known in advance
- At each step, as the request for loan arrives, the banker considers it, but grants it immediately *only* if doing so will lead to a safe state. Otherwise, the loan is postponed

The Banker's Algorithm: Example

	C	*M*
Jack	0	6
Jill	0	5
Peter	0	4
Anne	0	7
A	*10*	

→ … →

	C	*M*
Jack	1	6
Jill	1	5
Peter	2	4
Anne	4	7
A	*2*	

Jill, 1?

	C	*M*
Jack	1	6
Jill	2	5
Peter	2	4
Anne	4	7
A	*1*	

This state is safe.

Example Schedule:
{Peter, Jill, Jack, Anne}

This state is unsafe! So, Jill's request is postponed until Peter's request is serviced

The Banker's Algorithm for the Case of Multiple Resources

- In a system of *n* processes and *m* resources, defined by constants ***E*** and ***M*,** and the *presently safe* state $S = (C)$
- When a process P_i issues a resource request $R = (r_1, r_2, \ldots r_m)$, check if the request is valid $(R + C_i \leq M_i)$ and if so whether state $S_{test} = C_i + R$ is safe
- If it is safe, grant the request and update $C_i = C_i + R$, $A = E - C_i$, and $N_i = N_i - R$; otherwise, postpone granting the request

Banker's Algorithm (cont.): Checking if a State is Safe

```
BOOLEAN safe (value: matrix C, vector A)
Initialize all processes to unmarked;
loop = TRUE;
j=0;
do
     {
          look for an unmarked process Pi such that Ni ≤ A
          if (found)
               {
                    mark Pi; /* Consider it finished */
                    A = A + Ci; Ci = 0; j++;/*Take its resources*/
               }
            else
               loop = FALSE;
     } while loop && (j<n);
if (all marked) return TRUE; else return FALSE;
```

An Example

The system: five processes,

Six *tape drives*, three *plotters*, four *printers*, and two *CD ROM drives*

E = *(6, 3, 4 , 2)*

Computed ***A*** = *(1, 0, 2, 0)*

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad < \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont.)

P_4 is marked, and its resources taken:

$\boldsymbol{E} = (6, 3, 4, 2)$

Computed $\boldsymbol{A} = (2, 1, 2, 1)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \hline 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont.)

P_1 is marked, and its resources taken:

$\boldsymbol{E} = (6, 3, 4, 2)$ Computed $\boldsymbol{A} = (5, 1, 3, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont.)

P_2 is marked, and its resources taken:

$\boldsymbol{E} = (6, 3, 4, 2)$ Computed $\boldsymbol{A} = (5, 2, 3, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad < \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont.)

P_3 is marked, and its resource taken:

$\boldsymbol{E} = (6, 3, 4, 2)$ Computed $\boldsymbol{A} = (6, 3, 4, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

<

An Example (cont.)

P_5 is marked; the state is safe!

$\boldsymbol{E} = (6, 3, 4, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont)

Now, P_2 *asks for a printer:* $\boldsymbol{R} = (0, 0, 1, 0)$. Let us check if the new state is safe.

$\boldsymbol{E} = (6, 3, 4, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

Computed $\boldsymbol{A} = (1, 0, 1, 0)$

$$C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont.)

P_4 is marked, and its resources taken:

$\boldsymbol{E} = (6, 3, 4, 2)$

Computed $\boldsymbol{A} = (2, 1, 1, 1)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ \hline 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

<

An Example (cont.)

P_1 is marked, and its resources taken:

$\boldsymbol{E} = (6, 3, 4, 2)$ Computed $\boldsymbol{A} = (5, 1, 2, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont.)

P_2 is marked, and its resources taken:

E = *(6, 3, 4 , 2)*

Computed ***A*** = *(5, 2, 3, 2)*

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \qquad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

<

An Example (cont.)

P_3 is marked, and its resource taken:

$\boldsymbol{E} = (6, 3, 4, 2)$ Computed $\boldsymbol{A} = (6, 3, 4, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont.)

P_5 is marked; the state is safe!

$\boldsymbol{E} = (6, 3, 4, 2)$

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

An Example (cont)

So, we can grant P_2 a printer. The next request: P_4 wants the last tape drive; ***R*** = *(0, 0, 1, 0)*. But the new state is unsafe; P_4 will have to wait...

E = *(6, 3, 4 , 2)*

$$M = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 0 & 2 & 1 & 2 \\ 4 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Computed ***A*** = *(1, 0, 0, 0)*

$$N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

A Few Notes on Deadlock Avoidance

- The worst-case complexity of the Banker's Algorithm is $O(n^2m)$. You have to prove it as part of the homework assignment
- To use it in a system, it must be run every time a resource is requested—an expensive proposition, in general
- In addition, processes rarely know (and are not required to know) the maximum use of their resource in advance—this renders the Banker's algorithm useless for general operating systems (but applications can still use it!)

Summary

- Deadlock is a potential problem in any system. It occurs when each process in a group cannot proceed without an action from some other process in a group.
- Deadlocks can be prevented (at a cost of a quadratic complexity Banker's Algorithm) if maximum resource requests of the processes are known in advance. This is done by avoiding *unsafe* states and thus plotting *safe* process *trajectories*. The state is *safe* if there exist a schedule for all processes to finish their jobs. Deadlocks can be also prevented by eliminating one of the four necessary conditions for their occurrence.
- Deadlocks can be detected (also by employing a quadratic complexity algorithm)

Final Notes

- In the past two lectures, we have dealt with the resource-related deadlocks, but the processes can deadlock not *only* waiting for resources, but also waiting for each other *to get something* (like receiveing a message)
- A problem closely related to deadlocks is *starvation*. Once the system gets an algorithm to ensure that granting a resource to a process would not lead to a deadlock, who should get it if *more than one* process needs it? (As we learned when dealing with the allocation of the CPU, the resources can be granted using the FCFS policy or multi-level queuing to avoid starvation)