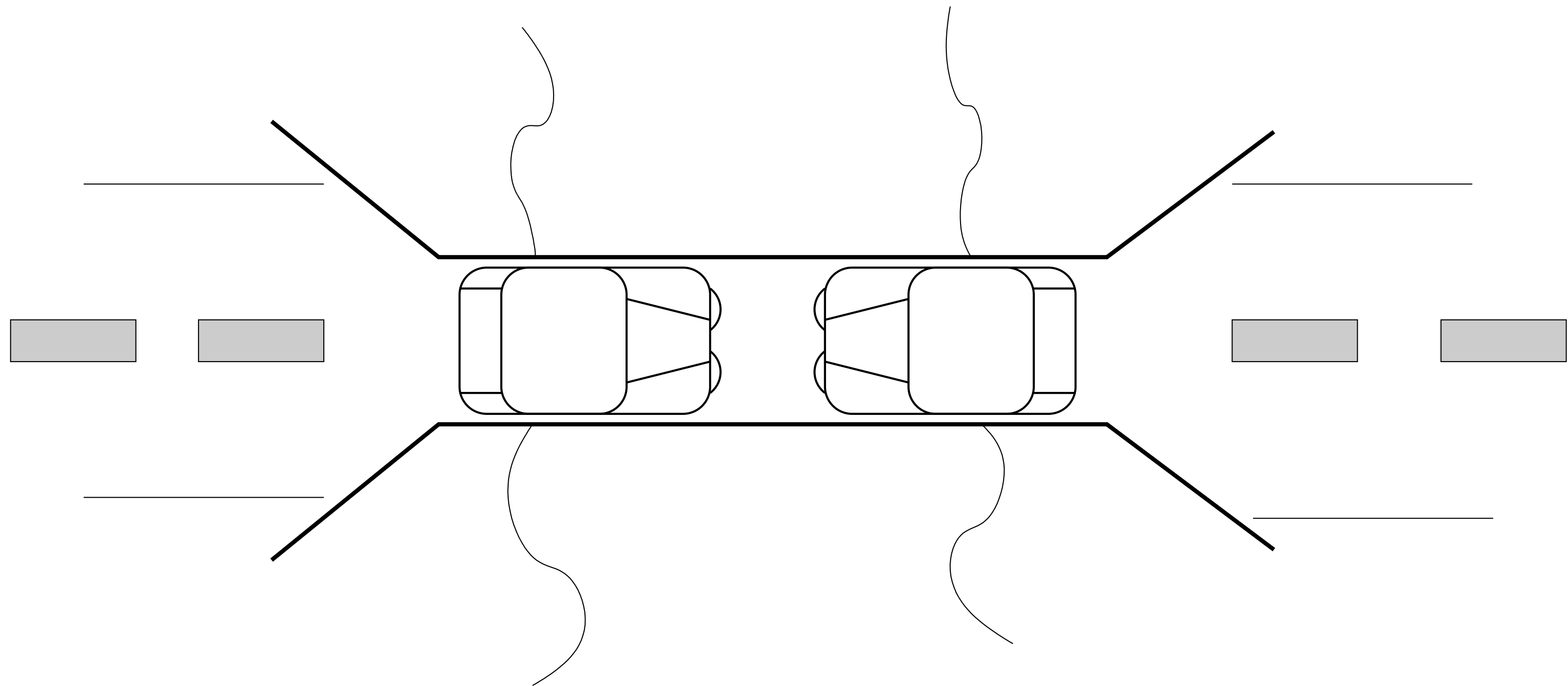
Deadlock
Ch. 3

Deadlock

When two or more processes are each waiting on resources held by another in such a way that none can progress until another one does.



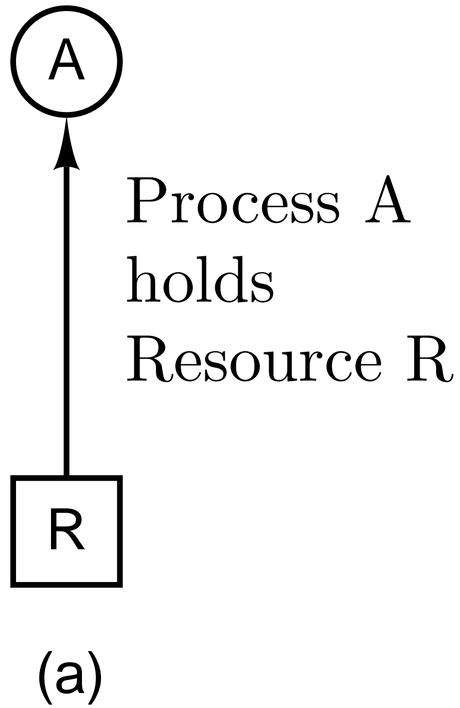
Shared and Preemptable Resources

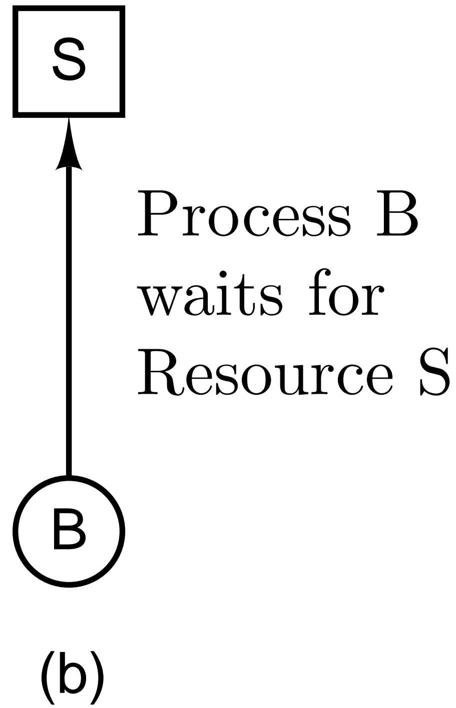
A few resources may be sharable.
Anyone can write the system log.

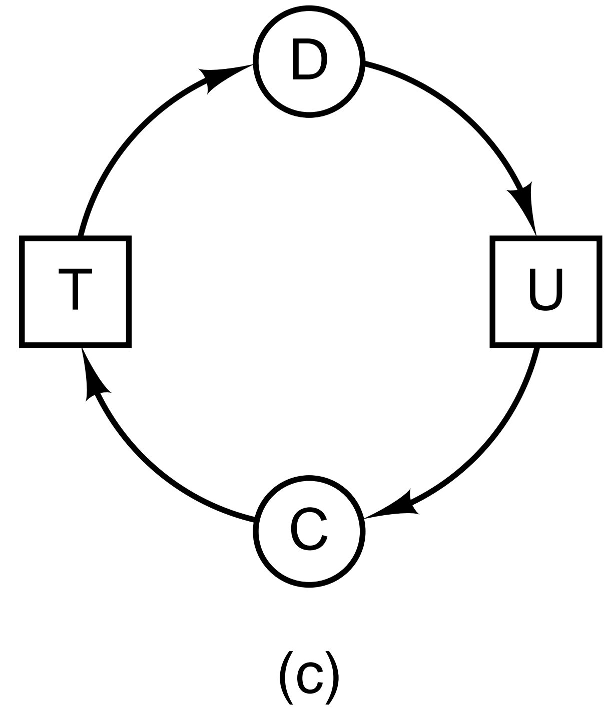
Some resources are preemptable.
They can be safely taken away.
Memory may be preempted by swapping a job out to disk.

Deadlock generally involves exclusive, non-preemptable resources.

Deadlock Requires Circular Wait







Two Processes, Two Resources

Process C

```
...
Get U
...
Get T
...
Release U
...
Release T
...
```

Process D

```
...
Get T
...
Get U
...
Release T
...
Release U
...
```

Order Matters

This Order Is Okay

Process C	Process D
Get U	•
•	•
Get T	•
•	Wait for T
Release U	-
•	-
Release T	Get T
•	•
•	Get U
•	•
•	Release T
•	•
•	Release U

This Order Deadlocks

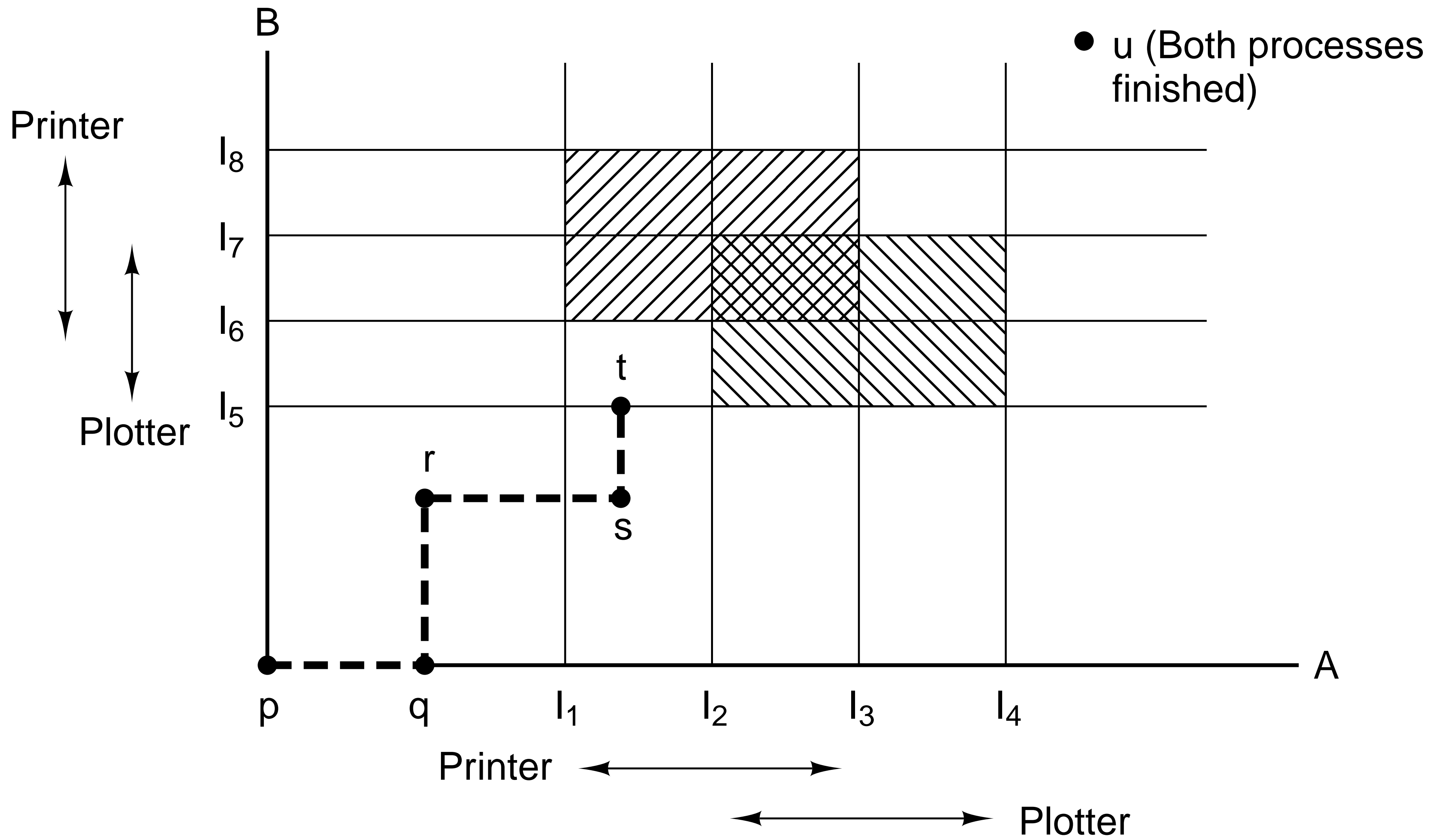
Process C

- •
- Get U
- •
- Wait for T
- –
- –
- –
- –
- –
- ...

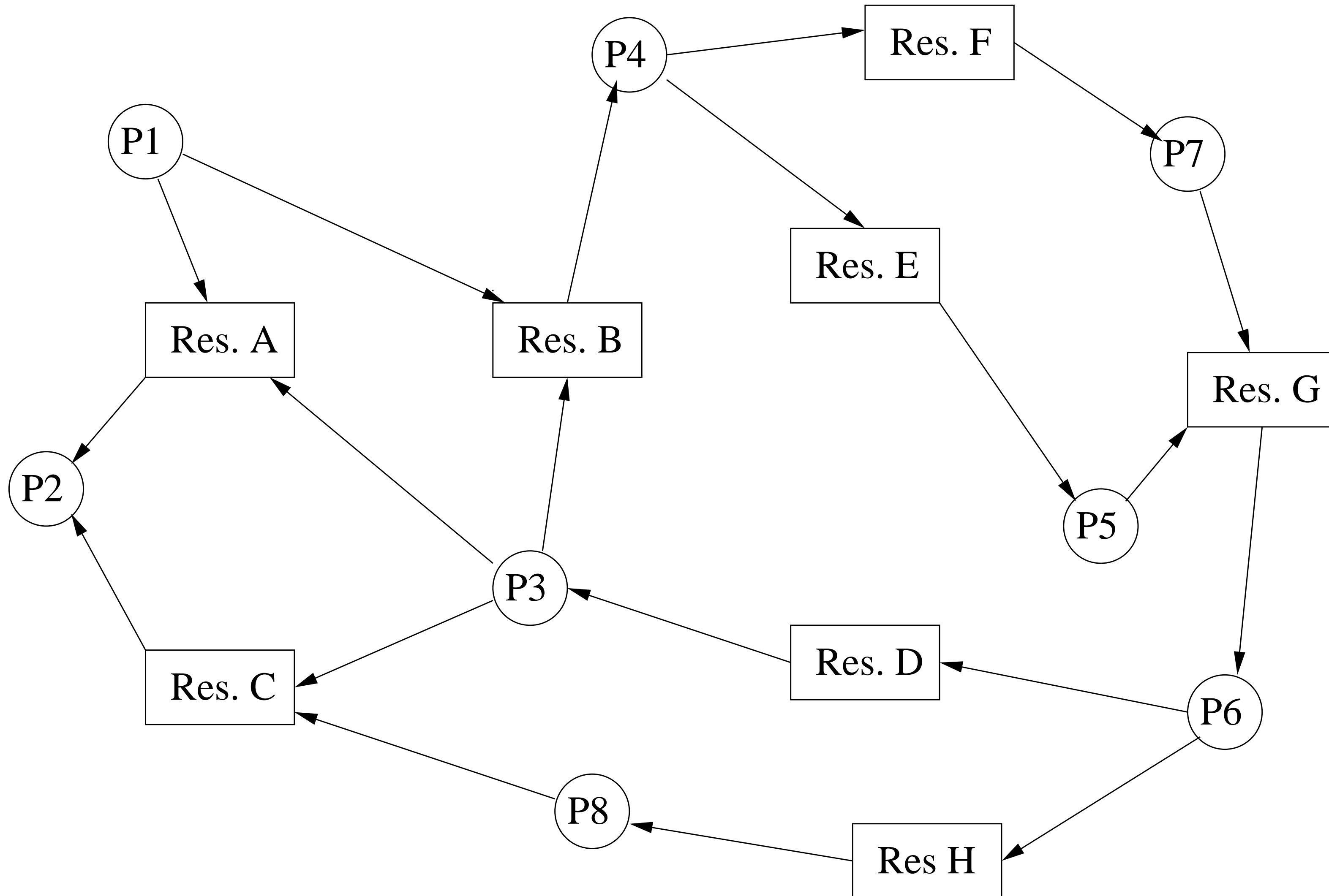
Process D

- •
- •
- Get T
- •
- Wait For U
- –
- –
- –
- –
- ...

Possible Execution Orders



More Complex Deadlock



Conditions for Deadlock

All four conditions must hold for a deadlock to exist:

Mutual Exclusion

Hold and wait

No preemption

Circular wait

Dealing with Deadlock

Ignorance
Hope it doesn't happen.

Detection and Recovery
Detect and eliminate deadlocks as they occur.

Dynamic avoidance
Refuse risky allocations.

Prevention
Eliminate one of the four conditions.

Ignorance

What if you just ignore the problem?

Deadlocked processes will eventually be killed by whoever ran them.

Deadlock is rare.

If the frequency of deadlock is less than the frequency of power failure, or O/S hangs, it may not be worth the trouble to avoid.

This is actually the most common approach.

Deadlock Detection

Detect deadlock and eliminate.

Graph cycles work when there is only one resource of each type.

We use a model of system resources and allocations.

Deadlock Detection

System state described by four matrices:

Existing resources: $(E_1, E_2, \ldots, E_m)$

Available (free) resources: $(A_1, A_2, \ldots, A_m)$

Current Allocation: $$\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$$

Current Request: $$\begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1m} \\ R_{21} & R_{22} & \cdots & R_{2m} \\ \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & \cdots & R_{nm} \end{bmatrix}$$

Invariant

Every resource is allocated or available.

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j, \text{ for each } j,\ 1 \leq j \leq m$$

Deadlock Detection Algorithm

For vectors, $X \leq Y$ means $X_i \leq Y_i$ for each i, $1 \leq i \leq m$.

$usable \leftarrow (A_1, A_2, \ldots, A_m)$;
$remaining \leftarrow \{1, 2, \ldots, n\}$ // All process ids
while $remaining \neq \emptyset$ **repeat**
  find some $p \in remaining$, such that $R_p \leq usable$
  if such a p exists
    $usable \leftarrow usable + C_p$;
    $remaining \leftarrow remaining - \{p\}$
  else
    return *remaining* deadlocked;
end;
return no deadlock;

Deadlock Detection Algorithm

There is some order of the processes such that each can:

- fill its outstanding request from the free pool
- finish and return its all its resources for use by the next

Formally, there exists some order of the processes $p_1, p_2, \ldots, p_n$, such that

$$R_{p_i} \leq A + \sum_{j=1}^{i-1} C_{p_j}, \quad 1 \leq i \leq n$$

Text Example

No Deadlock

$$E = (4, 2, 3, 1) \qquad A = (2, 1, 0, 0)$$

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

Has Deadlock When

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

When to Run Detection

After ever allocation.

Periodically.

When CPU is idle too often.

Recovery

Preempt resources.
Not usually possible.

Rollback a process.
Requires checkpoints.

Kill deadlocked processes until the deadlock vanishes.

Prefer ones that can be restarted.

Compiles are usually okay.

Database updates usually are not.

Avoidance

Avoid situations where there is a risk of deadlock.

System states are classified as safe or unsafe.

Safe states cannot deadlock.
Unsafe states might deadlock.

Refuse any allocation which enters an unsafe state.

System State for Avoidance

Maximum which might be wanted instead of current request.

Existing resources: $(E_1, E_2, \ldots, E_m)$

Available (free) resources: $(A_1, A_2, \ldots, A_m)$

$$\text{Current Allocation: } \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix}$$

$$\text{Max Resources: } \begin{bmatrix} M_{11} & M_{12} & \cdots & M_{1m} \\ M_{21} & M_{22} & \cdots & M_{2m} \\ \vdots & \vdots & & \vdots \\ M_{n1} & M_{n2} & \cdots & M_{nm} \end{bmatrix}$$

Safe State Determination

$usable \leftarrow (A_1, A_2, \ldots, A_m);$
$remaining \leftarrow \{1, 2, \ldots, n\}$ // All process ids
while $remaining \neq \emptyset$ **repeat**
  find some $p \in remaining$, such that $M_p - C_p \leq usable$
  if such a p exists
    $usable \leftarrow usable + C_p;$
    $remaining \leftarrow remaining - \{p\}$
  else
    return unsafe;
end;
return safe;

Safe State

Unsafe states are those which would deadlock if all processes asked for their maximum resources.

Safe when there is some order of the processes such that each can:

- reach its maximum its allocation from the free pool
- finish and return its resources for use by the next

Formally, there exists some order of the processes $p_1, p_2, \ldots, p_n$, such that

$$M_{p_i} \leq A + \sum_{j=1}^{i} C_{p_j}, \quad 1 \leq i \leq n$$

Allocation Procedure

Action taken upon a resource request:

If the new total would exceed the maximum claim,
abort the process.

If the requested resources are not free, suspend the process.

Otherwise:

- Pretend to grant, and find the new state of the system.
- If that state is safe, grant.
- If not, suspend.

Banker's Algorithm

A Safe State

$$E = (5, 11, 3, 2) \qquad A = (1, 2, 1, 1)$$

$$C = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 2 & 2 & 0 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 3 & 1 & 0 \end{bmatrix} \qquad M = \begin{bmatrix} 4 & 8 & 2 & 1 \\ 2 & 2 & 2 & 2 \\ 1 & 9 & 0 & 0 \\ 2 & 3 & 1 & 1 \end{bmatrix}$$

Some Requests To Process

Process 2 requests $(0, 0, 0, 1)$

Process 3 requests $(0, 2, 0, 0)$

Process 1 requests $(0, 0, 1, 0)$

Problems With Avoidance

Requires processes to declare their maximum resources.

- A bother.
- Often a guess; often very wrong.
- Usually exaggerated to avoid termination on over-allocation.

Conservative: May deny allocations which do not actually lead to deadlock.

Expensive: Must run the algorithm for every resource request.

No one actually uses this algorithm.

Elegant *Famous* *Too Expensive*

Prevention

Elimination of one condition of deadlock makes it impossible.

Mutual Exclusion.
Spooling does this; not generally applicable.

Hold and wait.
Allocate all resources at start.
Release and reallocate
Two-phase locking is a form of this.

No preemption.
Must be able to roll back processes.

Circular wait.
Allocation must be performed in a fixed order

Problems with Prevention

Can force a process to allocate sooner than needed.

Low utilization of resources

Preemption requires rollback.

Rollback requires check-pointing.

An Integrated Approach

Group resources into a number of classes.

Allocate resources from these classes in a fixed order.

Handle deadlock as appropriate within each class.

- Swap space
- Process resources: Devices and files.
- Main memory
- Internal resource: I/O channels.

Policies

- Swap space: Allocate all at start. (*Prevention*)
- Process resources: Avoidance.
- Main memory: Preemption by swapping. (*Prevention*)
- Internal resource: Ordering. (*Prevention*)

Sources

Tanenbaum, *Modern Operating Systems*
(Course textbook.)

The combined approach is due to J. Howard, CACM July 1973, cited by Stallings in his *Operating Systems*.