CS341: Operating System

**Deadlock**

Lect 25: 7th Oct 2014  
Dr. A. Sahu  
Dept of Comp. Sc. & Engg.  
Indian Institute of Technology Guwahati

---

**Outline**

- **Deadlock Conditions**  
  - Mutex, Hold & Wait, No-Preemption and Circular wait
- **Deadlock Prevention Avoidance**  
  - RAG, Cycle
- **Deadlock Detection and Recovery**  
  - WFG

---

**Deadlock**

- Mutual exclusion  
- Hold and wait  
- No preemption  
- Circular wait

---

**Deadlock**

--

**Deadlock**

[Diagram of two people representing deadlock]

---

**Deadlock**

[Diagram showing a process and a lock, with arrows indicating wait conditions]

---

**Deadlock**

[Diagram with two processes and a lock, showing wait conditions]

---

**Deadlock**

[Diagram of a bird and a fish, possibly illustrating a metaphor or concept related to deadlock]
**System Model**

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type $R_i$ has $W_i$ instances.
- Each process utilizes a resource as follows:
  - Request // Similar to Lock
  - Use // Similar to CS
  - Release // Similar to Unlock

**Deadlock Characterization**

- Deadlock can arise if four conditions hold simultaneously.
- Mutual exclusion
  - Only one process at a time can use a resource
- Hold and wait
  - A process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption
- Circular wait

**Deadlock Characterization**

- Mutual exclusion, Hold and wait
- No preemption
  - A resource can be released only voluntarily by the process holding it, after that process has completed its task
- Circular wait
  - There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that $P_i$ is waiting for a resource that is held by $P_{i+1}$, and $P_0$ is waiting for a resource that is held by $P_1$

**Resource Allocation Graph (RAG)**

RAG to Characterize Deadlock

- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

**Deadlock with Mutex Locks**

- Deadlocks can occur via system calls, locking, etc.

**Resource-Allocation Graph**

A set of vertices $V$ and a set of edges $E$.

- Wait for Lock ($Y$)
- Wait for Lock ($X$)
**Resource Allocation Graph (RAG)**

RAG to Characterize Deadlock

**Resource-Allocation Graph (Cont.)**

- Process
- Resource Type with 4 instances
- \( P_i \) requests instance of \( R_j \)
- \( P_i \) is holding an instance of \( R_j \)

**Example of a Resource Allocation Graph with a Deadlock**

\( P_1 \) requests instance of \( R_2 \)

\( P_2 \) requests instance of \( R_1 \)

\( P_3 \) requests instance of \( R_4 \)

\( P_4 \) requests instance of \( R_3 \)

\( P_4 \) holds instance of \( R_3 \)

\( P_2 \) holds instance of \( R_1 \)

\( P_3 \) holds instance of \( R_4 \)

\( P_3 \) is holding an instance of \( R_4 \)

\( P_4 \) requests instance of \( R_3 \)

**Example of a Resource Allocation Graph with a Deadlock**

\( P_1 \) requests instance of \( R_2 \)

\( P_2 \) requests instance of \( R_1 \)

\( P_3 \) requests instance of \( R_4 \)

\( P_4 \) requests instance of \( R_3 \)

\( P_4 \) holds instance of \( R_3 \)

\( P_2 \) holds instance of \( R_1 \)

\( P_3 \) holds instance of \( R_4 \)

\( P_3 \) is holding an instance of \( R_4 \)

\( P_4 \) requests instance of \( R_3 \)
Graph with a Cycle but No Deadlock

P1 → R2 → P2
   ↓     ↑
     ↓     ↓
   R2 → P3 → P4

P4 and P2 are not in loop they will release

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- **Ostrich Approach**: 😞 😞 Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

Basic Facts

- If graph contains no cycles → no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, **then deadlock**
  - if several instances per resource type, **possibility of deadlock**

Prevention, Avoidance and Detection

- **Prevention**: Cold wave in December
- **Avoidance**: don’t go outside: it is too restrictive
- **Recovery**: Got cold: Take medicine

Prevention, Avoidance and Detection

- **Diabetes and Sugar**
  - **Prevention**: Don’t take sugar, fruits, rice, patato, Cake, Rasogola, Laddu
  - With out having any symptoms of diabetes: it is too restrictive
  - **Avoidance**: Take all food but care fully, if you are symptom is boundary case
  - **Recovery**: Got diabetes: Take medicine

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion**: not required for sharable resources (e.g., read-only files);
  - must hold for non-sharable resources
- **Hold and Wait**: Must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution
  - Or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible
Deadlock Prevention (Cont.)

- No Preemption –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- Circular Wait
  - Impose a **total ordering** of all resource types
  - And require that each process requests resources in an increasing order of enumeration

---

Deadlock Example

```c
#define pthread_mutex PM

/* thread one runs in this function */
void *do_work_one(void *param) {
    PM_lock(&Mutex1); PM_lock(&Mutex2);
    /* * Do some work */
    PM_unlock(&Mutex2); PM_unlock(&Mutex1);
}

/* thread two runs in this function */
void *do_work_two(void *param) {
    PM_lock(&Mutex1); PM_lock(&Mutex2);
    /* * Do some work */
    PM_unlock(&Mutex1); PM_unlock(&Mutex2);
}
```

---

Thread Lock are not in Orders

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Lock1 → Lock2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 2</td>
<td>Lock2 → Lock1</td>
</tr>
</tbody>
</table>

At this point Both may hold one lock and wait for other

---

Deadlock Example with Lock Ordering

```c
void transaction(Account from, Account to, double amount) {
    mutex lock1, lock2;
    lock1 = get_lock(from); lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount); deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers Rs 25,000 from account A to account B, and Transaction 2 transfers Rs 50,000 from account B to account A

---

Thread Locks are in Orders: but Still Deadlock...?

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>LockS → LockD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction 2</td>
<td>LockS → LockD</td>
</tr>
</tbody>
</table>

Let source of T1 is A and source of T2 is B
And Dest of T1 is B and dest T2 is A

At this point Both may hold one lock and wait for other

---

Deadlock Avoidance

- Requires that the system has some additional **a priori** information available
- Simplest and most useful model
- It requires
  - Each process declare the **maximum number** of resources of each type that it may need
**Deadlock Avoidance**

- Dynamically examines the resource-allocation state
  - To ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by
  - The number of available
  - The number of allocated resources
  - And the maximum demands of the processes

**Safe, Unsafe, Deadlock State**

- When a process requests an available resource
  - System must decide if immediate allocation leaves the system in a safe state
- **Safe state** if there exists a sequence $<P_1, P_2, ..., P_n>$ of all the processes in the systems
  - Such that for each $P_i$, the resources that
    - $P$ can still request can be satisfied by currently available resources + resources held by all the $P_j$ with $j < i$
  - That is:
    - If $P_i$, resource needs are not immediately available, then $P_i$ can wait until all $P_j$ have finished
    - When $P_i$ is finished, $P_i$ can obtain needed resources, execute, return allocated resources, and terminate
    - When $P_i$ terminates, $P_{i+1}$ can obtain its needed resources, and so on

**Basic Facts**

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state

**Safe State**

- If there exists a sequence $<P_1, P_2, ..., P_n>$ of all the processes in the systems
  - Such that for each $P_i$, the resources that
    - $P_i$ can still request can be satisfied by currently available resources + resources held by all the $P_j$ with $j < i$
  - That is:
    - If $P_i$, resource needs are not immediately available, then $P_i$ can wait until all $P_j$ have finished
    - When $P_i$ is finished, $P_i$ can obtain needed resources, execute, return allocated resources, and terminate
    - When $P_i$ terminates, $P_{i+1}$ can obtain its needed resources, and so on

**Avoidance Algorithms**

- Single instance of a resource type
  - Use a resource-allocation graph (RAG)
- Multiple instances of a resource type
  - Use the Banker’s algorithm
- Credit card issued by bank: If you use X amount, you need to pay X at the end of the month.
**Resource-Allocation Graph Scheme**

- **Claim edge** $P_i \rightarrow R_j$ indicated that process $P_i$ may request resource $R_j$ represented by a dashed line.
- **Claim edge** converts to request edge when a process requests a resource.

**Resource-Allocation Graph**

[Diagram showing a graph with nodes P1, P2, R1, and R2 with edges and claim edge explained.]

*Corrected: There was a problem in this example (in class)*

**Resource-Allocation Graph Scheme**

- Request edge converted to an assignment edge when the resource is allocated to the process.
- Red Edge
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

**Resource-Allocation Graph**

[Diagram showing a graph with nodes P1, P2, R1, and R2 with assignment edge explained.]

*Corrected: There was a problem in this example (in class)*

**Resource-Allocation Graph (Unsafe State)**

[Diagram showing a graph with nodes P1, P2, R1, and R2 with unsafe state explained.]

*Corrected: There was a problem in this example (in class)*

**Resource-Allocation Graph Algorithm**

- Suppose that process $P_i$ requests a resource $R_j$.
- The request can be granted only if:
  - Converting the request edge to an assignment edge does not form a cycle.
Multiple Instance of Resources: Banker’s Algorithm

- Proposed by Dijkstra, 1965
  - Works for Multiple Instance of Resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait

Banker algorithm

- For each request
  - The system checks if granting this request may lead to a deadlock in the worse case
    - If yes, do not grant the request
    - If no, grant the request
- When a process gets all its resources it must return them in a finite amount of time

Relation to Bank

- Relation to Bank
  - Each customer tells banker the maximum number of resources it needs
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Customer eventually pays back loan
  - Banker only lends resources if the system will be in a safe state after the loan
- Safe state: there is a lending sequence such that all customers can take out a loan
- Unsafe state: there is a possibility of deadlock

Safe State and Unsafe State

- Safe State
  - there is some scheduling order in which every process can run to completion even if all of them request their maximum number of resources immediately
  - From safe state, the system can guarantee that all processes will finish
- Unsafe state: no such guarantee
  - Not deadlocked state
  - Some process may be able to complete

An Example of Deadlock Avoidance

- 5 Processes: P0, P1, P2, P3 and P4
- Three Resource type A, B and C
- Many instances of resources
  - A: 10, B: 5, and C:5
- Allocation: Already allocated
- Max: Maximum need
- Need/request for: Current need

Is Allocation (1 0 2) to P1 Safe?

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td>1 2 2</td>
<td>3 3 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

If P1 requests max resources, can complete
Run Safety Test

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

If P1 requests max resources, can complete

Allocate to P1, Then

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>0 0 0</td>
<td>2 1 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Now P2 can acquire max resources and release

Release - P1 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>0 0 0</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Now P3 can acquire max resources and release

Release - P3 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>0 0 0</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0 0 0</td>
<td>2 2 2</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 0</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Now P4 can acquire max resources and release

Release - P4 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>0 0 0</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>10 4 5</td>
</tr>
<tr>
<td>P2</td>
<td>0 0 0</td>
<td>9 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0 0 0</td>
<td>2 2 2</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 0</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Now P0 can acquire max resources and release

Release - P2 Finishes
**So P1 Allocation (1 0 2) Is Safe**

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

**Is allocation (0 2 0) to P0 Safe?**

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Try to Allocate 2 B to P0

**Run Safety Test**

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0 3 0</td>
<td>7 5 3</td>
<td>7 2 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 1 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

No Processes may get max resources and release

**So Unsafe State- Do Not Enter**

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0 3 0</td>
<td>7 5 3</td>
<td>7 2 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 1 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Return to Safe State and do not allocate resource

**P0 Suspended Pending Request**

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 2</td>
<td>3 2 2</td>
<td>0 2 0</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

When enough resources become available, P0 can awake

**Data Structures for the Banker’s Algorithm**

- \( N \)=# process, \( m \) = # resource type
- \( \text{int AVL}[M] \)
- \( \text{int Max}[N][M], \) \( \text{int Alloc}[N][M], \) \( \text{int Need}[N][M] \)
- \( \text{AVL}[j] \) instances of resource type \( R_j \) available
- Process \( P_i \) may request at most \( \text{Max}[i][j] \) instances of resource type \( R_j \)
- \( P_i \) is currently allocated \( \text{Alloc}[i][j] \) instances of \( R_j \)
- \( P_i \) may need \( \text{Need}[i][j] \) more instances of \( R_j \) to complete its task

\[
\text{Need}[i][j] = \text{Max}[i][j] - \text{Alloc}[i][j]
\]
Safety Algorithm
Let Work[M] and Finish[N]; Found = true;
for (i = 0; i < N; i++)   { // Initialize
    Work[1:m] = Available[1:m]; Finish[i] = false;
}
while (Found == true) {
    Find an i such that both
    if (Found == false) Break;
}
if (Finish[i] == true for all i)
    The system is in a safe state

Resource-Request Algorithm for Process Pi
Request[1:M] = Req vector for process Pi
if { Request[1:M] > Need[1:M] }
    Raise error (process has exceeded its max claim)
if { Request[1:M] > Available[1:M] }
    Pi must wait, since resources are not available
    Pretend to allocate requested resources to Pi by modifying
    the state as follows:
    Available = Available − Request;
    Allocation = Allocation + Request;
    Need = Need − Request;
    if (safe_allocation())
        The requested resources are allocated to Pi,
    else
        Pi must wait, and the old resource-allocation state is restored

Deadlock Detection
• Allow system to enter deadlock state
• Detection algorithm
• Recovery scheme

Single Instance of Each Resource Type
• Maintain wait-for graph
  – Nodes are processes
  – Pi → Pj if Pj is waiting for Pi
• Periodically searches for a cycle in WFG
  – If there is a cycle, there exists a deadlock
• Detect a cycle in a graph
  – \( O(n^2) \) operations, where \( n \) = number of vertices

Several Instances of a Resource Type
• Similar to Safety Algorithm of Deadlock Avoidance
  • Available[1:M]: Currently Available
  • Allocation[N:M]: currently allocated
  • Request[N:M]: current request
**Deadlock Detection Graphical Approach**

- Resource Allocation Graph
- Reduction (Erase)
  - If a resource has only arrow away from it
    - No request pending to resource
  - If a process has only arrow pointing towards it
    - All the request granted
  - If a process has arrows pointing away from it but each such request arrow there is an available dot in resource: Erase all the process arrow

**Example**

If a process has only arrow pointing towards it
All the request granted

If a process has arrows pointing away from it but each such request arrow there is an available dot in resource: Erase all the process arrow
Example

If a process has arrows pointing away from it but each such request arrow there is an available dot in resource: Erase all the process arrow.

No Deadlock

Another Example

If a resource has only arrow away from it. No request pending to resource.

No-further reduction: There must be cycle.