Deadlock

Detection and recovery
Deadlock Humor

Two drivers were approaching a one-lane bridge. They reached the middle of the bridge at the same time, forcing a confrontation. They got out of their cars and went nose to nose. One driver said, “I don’t back up for idiots.” The other driver calmly got back in his car and started backing up. He said, “No problem --- I do.”

(Moral of the story): Processes are neither stubborn nor cooperative, so deadlock is a painful experience at best. Usually, one or more processes will have to be “backed out” of the deadlock, losing some or all of their work.
Deadlock

*Permanent blocking* of a set of processes that either
- compete for system resources or
- communicate with each other

- Involve conflicting needs for resources by two or more processes
- No efficient solution
Necessary conditions for deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process holding one resource is waiting to acquire additional resources held by other processes
- No preemption
  - No resource can be forcibly removed form a process holding it
Deadlocked condition

- Circular wait
  - There exists a permanent, circular sequence of processes such that each process is waiting for a resource held by the preceding process.
Resource-Allocation Graph (RAG)

- A set of Vertices $V$ and a set of Edges $E$
- $V$ is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.

- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$
RAG example with a deadlock
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.
Approaches to deadlock handling

- Ignore Deadlock (Ostrich approach)
  - If infrequent enough and result is not serious
  - Used by most operating systems, including UNIX
- Deadlock Prevention
  - Prevent one of the necessary/sufficient conditions
- Deadlock Avoidance
  - Allow the 3 necessary conditions
  - Dynamically make choices to avoid deadlock
    - decide based on knowledge of future requests
    - i.e., find a *safe path*
Approaches to deadlock handling

- Deadlock Detection and Recovery
  - Periodically run algorithm to detect circular waiting
  - After detecting deadlock,
    - run a *recovery algorithm* to remove deadlock
Deadlock detection and recovery

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme
Deadlock detection algorithm

- Determine if all requests can be satisfied eventually
- Simulate
  - granting of resources to processes and
  - returning of those resources when processes complete
- *Mark processes* that can complete
  - These are NOT deadlocked
- A deadlock exists if and only if there are *unmarked processes* at the end of the algorithm
  - These ARE deadlocked processes
Comments

• Algorithm does not guarantee that deadlock will not occur
  • depends on order in which requests are granted

• Just determines if deadlock currently exists
Deadlock detection algorithm

- Process \( i \) requests \( \text{req}[i, \ast] \) additional resources

\[
\begin{align*}
\text{Avail}_\text{tmp}[\ast] &= \text{Avail}[\ast]; \\
\text{Rest} &= \ldots \text{ Set of all processes with } \text{Alloc}[i,\ast] \neq 0 \ldots \\
\text{safePath} &= \text{True}; \\
\text{while (safePath)} \{ \\
    &\quad \text{// find a process whose request can be satisfied} \\
    &\quad \text{if (there exists a process } i \text{ such that } \text{req}[i, \ast] \leq \text{Avail}_\text{tmp}[\ast]) \{ \\
    &\quad\quad \text{// simulate process } i \text{ completing} \\
    &\quad\quad \text{Avail}_\text{tmp}[\ast] = \text{Avail}_\text{tmp}[\ast] + \text{Alloc}[i,\ast]; \\
    &\quad\quad \text{Rest} = \text{Rest} - \{ P_i \}; \\
    &\quad\} \text{ else} \\
    &\quad \text{safePath} = \text{False}; \\
\} \\
\ldots &\text{ Each process in Rest is deadlocked } \ldots
\end{align*}
\]
Example of detection algorithm

- Five processes $P_0$ through $P_4$
- three resource types $A$ (7), $B$ (2), and $C$ (6).
- Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th></th>
<th>Allocation $A$</th>
<th>Allocation $B$</th>
<th>Allocation $C$</th>
<th>Request $A$</th>
<th>Request $B$</th>
<th>Request $C$</th>
<th>Available $A$</th>
<th>Available $B$</th>
<th>Available $C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td>0 0 0</td>
<td></td>
<td></td>
<td>0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td></td>
<td></td>
<td>2 0 2</td>
<td></td>
<td></td>
<td>0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 3</td>
<td></td>
<td></td>
<td>0 0 0</td>
<td></td>
<td></td>
<td>0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td></td>
<td></td>
<td>1 0 0</td>
<td></td>
<td></td>
<td>0 0 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td></td>
<td></td>
<td>0 0 2</td>
<td></td>
<td></td>
<td>0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example continues ...

- $P_2$ requests an additional instance of type $C$ after $P_0$ completes

<table>
<thead>
<tr>
<th>Allocation Available</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>B</strong></td>
</tr>
<tr>
<td>$P_0$</td>
<td>0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0</td>
</tr>
</tbody>
</table>

- State of the system?
Recovery from deadlock

- Kill deadlocked processes
- Preempt resources from deadlocked processes
- Roll back each deadlocked process to some safe state and restart them
  - Widely used in current database systems.
  - Danger of the same situation occurring again.
Recovery: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until no deadlock
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?
Recovery: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process from that state
- **Starvation** – same process may always be picked as victim
  - include number of rollback in cost factor
Combined Approach

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

using the optimal approach for each resources in the system

- Partition resources into ordered classes

- Use most appropriate technique for handling deadlocks within each class
Deadlock should not be ignored

- A real-time control system monitoring a gasoline refinery
  - Need to ensure the safe and proper operation of the refinery
  - An avoidance strategy or prevention strategy must be employed
- A computerized pacemaker
  - Can’t afford to miss a single beat
  - Deadlock avoidance or prevention must be employed
Deadlock should not be ignored

- Embedded systems – cell phones, PDAs.
  - These are SoCs or system on chips i.e. everything is on one chip and SoCs are characterized by a small set of resources and real-time processing constraints.
  - Can’t have a system administrator come in and decide which process to kill after deadlock has been detected.
  - Either a prevention or avoidance strategy must be employed.
Dining Philosopher Problem

- Five philosophers who alternately think and eat
  - Share a fork with each neighbor
  - Assume each philosopher picks up left fork, then right fork, then eats
  - Deadlock if all are hungry at the same time
Solutions: Dining Philosophers Problem

- **Buy more Forks**
  - Equivalent to increasing resources

- **Put fork down if 2nd fork busy**
  - Can produce “livelock” if philosophers stay synchronized

- **Room Attendant**
  - Only let 4 of the philosophers into the room at once
  - May have 4 philosophers in room, but only 1 might eat

- **Mix order of grabbing forks**
  - Some grab forks in the other order (right fork, then left fork)
  - Any mix will avoid deadlock
Correct solution with semaphores

```c
semaphore fork[5] = {1, 1, 1, 1, 1};
semaphore room[4] = {1, 1, 1, 1};
int i;
process philosopher(i) {
  while(true) {
    think();
    wait(room);
    wait(fork[i]);
    wait(fork[(i+1) mod 5]);
    eat();
    signal(fork[(i+1) mod 5]);
    signal(fork[i]);
    signal(room);
  }
}

void main() {
  for (int i =0; i < 5; i++) {
    philosopher(i);
  }
} 
```
Incorrect solution with semaphores

```c
semaphore fork[5] = {1,1,1,1,1};
process philosopher(int i){
    while(true){
        think();
        wait(fork[i]);
        wait(fork[(i+1)mod5]);
        eat();
        signal(fork[(i+1)mod5]);
        signal(fork[i]);
    }
}

void main() {
    for (int i =0; i < 5; i++) {
        philosopher(i);
    }
}
```
Another incorrect solution

binary semaphore x = 1;
process philosopher(int i){
    while(true){
        think();
        wait(x);
        pick up fork[i];
        pick up fork[(i+1)mod5];
        signal(x);
        eat();
        wait(x);
        return fork[i] ;
        return fork[(i+1)mod5] ;
        signal(x);
    }
}

void main() {
    for (int i =0; i < 5; i++) {
        philosopher(i);
    }
}