Lecture - XI
Deadlocks - II

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

1. **Mutual exclusion**: nonshared resources; only one process at a time can use a specific resource.
2. **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Deadlock Characterization (cont.)

Deadlock can arise if four conditions hold simultaneously.

4. **Circular wait**: there exists a set \{P_0, P_1, ..., P_n\} of waiting processes such that \(P_0\) is waiting for a resource that is held by \(P_1\), \(P_1\) is waiting for a resource that is held by \(P_2\), ..., \(P_{n-1}\) is waiting for a resource that is held by \(P_n\), and \(P_n\) is waiting for a resource that is held by \(P_0\).

Resource-Allocation Graph

- Used to describe deadlocks.
- Consists of 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.
- \(P \) requests \(R\): directed edge \(P_i \rightarrow R_j\)
- \(R\) is assigned to \(P\): directed edge \(R_j \rightarrow P_i\)

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

Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒ there may be a deadlock
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Resource Allocation Graph – Example 1

⇒ No Cycle, no Deadlock

Resource Allocation Graph – Example 2

⇒ Cycle, but no Deadlock

Resource Allocation Graph – example 3

⇒ Deadlock
Which Processes deadlocked?
⇒ P1 & P2 & P3

Exercise

In the code below, three processes are competing for six resources labeled A to F.

Using a resource allocation graph (Searschutz, pp. 249-251) show the possibility of a deadlock in this implementation.
Rule of Thumb

- A cycle in the resource allocation graph
  - is a necessary condition for a deadlock
  - but not a sufficient condition

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state.
  - deadlock prevention or avoidance
- Allow the system to enter a deadlock state and then recover.
  - deadlock detection
- Ignore the problem and pretend that deadlocks never occur in the system
  - Programmers should handle deadlocks (UNIX, Windows)

Deadlock Prevention

- Ensure one of the deadlock conditions cannot hold
- Restrain the ways request can be made.

- Mutual Exclusion - not required for sharable resources; must hold for non-sharable resources.
  - Eg. read-only files
- Hold and Wait - must guarantee that whenever a process requests a resource, it does not hold any other resources.
  1. Require process to request and be allocated all its resources before it begins execution
  2. or allow process to request resources only when the process has none.
  - Example: Read from DVD to memory, then print.

  1. holds printer unnecessarily for the entire execution
  2. may never get the printer later

- Low resource utilization
- starvation possible

Programmers should handle deadlocks (UNIX, Windows)

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.

Exercise

In the code below, these processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp. 249-251) show the possibility of a deadlock in this implementation.

b. Modify the order of some of the "get" requests to prevent the possibility of any deadlock.
   You cannot move requests across procedures, only change the order inside each procedure.
   Use a resource allocation graph to justify your answer.

```c
void v0()
{
  while (true) {
    get(A);
    get(B);
    // critical region:
    // use A, B, C
    release(A);
    release(B);
    release(B);
  }
}

void v1()
{
  while (true) {
    get(C);
    get(D);
    // critical region:
    // use C, D
    release(C);
    release(D);
  }
}

void v2()
{
  while (true) {
    get(E);
    get(F);
    // critical region:
    // use E, F
    release(E);
    release(F);
  }
}
```
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.
  - \( P_i \rightarrow P_j \) if \( P_i \) is waiting for \( P_j \).

```
void FC();
{
    while (true) {
        get(A);
        get(B);
        get(C);
        // critical region:
        // use A, B, C
        release(A);
        release(B);
        release(C);
    }
}
```

```
void FC();
{
    while (true) {
        get(D);
        get(E);
        get(F);
        // critical region:
        // use D, E, F
        release(D);
        release(E);
        release(F);
    }
}
```

```
void FC();
{
    while (true) {
        get(G);
        get(H);
        get(I);
        // critical region:
        // use G, H, I
        release(G);
        release(H);
        release(I);
    }
}
```

Single Instance of Each Resource Type

- 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.
- Only good for single-instance resource allocation systems.

Detection Algorithm

1. Let \( \text{Work} \) and \( \text{Finish} \) be vectors of length \( m \) and \( n \), respectively. Initialize:
   (a) \( \text{Work} = \text{Available} \)
   (b) For \( i = 0, 2, \ldots, n-1 \), if \( \text{Allocation} \neq 0 \), then \( \text{Finish}[i] = \text{false} \); otherwise, \( \text{Finish}[i] = \text{true} \).
2. Find an index \( i \) such that both:
   (a) \( \text{Finish}[i] = \text{false} \)
   (b) \( \text{Request}_i \leq \text{Work}_i \)

   If no such \( i \) exists, go to step 4.

Several Instances of a Resource Type

- **Available:** A vector of length \( m \) indicates the number of available resources of each type.
- **Allocation:** An \( n \times m \) matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An \( n \times m \) matrix indicates the current request of each process. If \( \text{Request}[i]_j = k \), then process \( P_i \) is requesting \( k \) more instances of resource type \( R_j \).
Detection Algorithm (Cont.)

3. \( \text{Work} = \text{Work} + \text{Allocation} \)
\( \text{Finish}[i] = \text{true} \)
go to step 2.

4. If \( \text{Finish}[i] == \text{false} \), for some \( i \), \( 0 \leq i \leq n-1 \), then the system is in deadlock state. Moreover, if \( \text{Finish}[i] == \text{false} \), then \( P_i \) is deadlocked.

Algorithm requires an order of \( O(m \times n^2) \) operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

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

  \[
  \begin{array}{ccc}
  & A & B & C \\
  \text{Allocation} & 0 & 1 & 0 \\
  \text{Request} & 0 & 0 & 0 \\
  \text{Available} & 0 & 0 & 0 \\
  \text{P_0} & 2 & 0 & 2 \\
  \text{P_1} & 2 & 0 & 0 \\
  \text{P_2} & 3 & 0 & 3 \\
  \text{P_3} & 2 & 1 & 1 \\
  \text{P_4} & 0 & 0 & 2 \\
  \end{array}
  \]

- Sequence \( \langle P_0, P_2, P_3, P_1, P_4 \rangle \) will result in \( \text{Finish}[i] = \text{true} \) for all \( i \).

Example (Cont.)

- \( P_2 \) requests an additional instance of type \( C \).

  \[
  \begin{array}{ccc}
  & A & B & C \\
  \text{Request} & 0 & 0 & 0 \\
  \text{P_0} & 2 & 0 & 2 \\
  \text{P_1} & 2 & 0 & 1 \\
  \text{P_2} & 2 & 0 & 1 \\
  \text{P_3} & 1 & 0 & 0 \\
  \text{P_4} & 0 & 0 & 2 \\
  \end{array}
  \]

- State of system?
  - Can reclaim resources held by process \( P_0 \), but insufficient resources to fulfill other processes’ requests.
  - Deadlock exists, consisting of processes \( P_1 \), \( P_2 \), \( P_3 \), and \( P_4 \).

Summary

- Deadlocks
  - Resource Allocation Graphs
  - Deadlock Prevention
  - Deadlock Detection

- \textbf{Next Lecture: Deadlocks - III & Main Memory}
- HW-2 due next Tuesday!

Acknowledgements

- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR