Lecture - X
Deadlocks - I

Tevfik Koşar

Louisiana State University
September 24th, 2009

Roadmap

- Deadlocks
  - Deadlock Characterization
  - Deadlock Detection
    - Resource Allocation Graphs
- Classic Problems of Synchronization
  - Bounded Buffer
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - $P_1$ and $P_2$ each hold one disk drive and each needs another one.
- Example
  - semaphores $A$ and $B$, initialized to 1

\[
\begin{align*}
P_0 & \quad P_1 \\
wait(A); & \quad wait(B) \\
wait(B); & \quad wait(A)
\end{align*}
\]

Bridge Crossing Example

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
Deadlock vs Starvation

• **Deadlock** - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

![Diagram showing possible deadlock](image)

• **Starvation** - indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

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 \( \{P_0, P_1, \ldots, P_0\} \) 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 \( P_2 \), \ldots, \( P_{n-1} \) 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:
  - \( P = \{P_1, P_2, \ldots, P_n\} \), 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.
- \( P \) requests \( R \) - directed edge \( P_1 \rightarrow R_j \)
- \( R \) 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 $\Rightarrow$ no deadlock.

- If graph contains a cycle $\Rightarrow$ 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
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.

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

```c
void P0()
{
    while (true) {
        get(A);
        get(B);
        get(C);
        // critical region:
        // use A, B, C
        release(A);
        release(B);
        release(C);
    }
}

void P1()
{
    while (true) {
        get(D);
        get(E);
        get(B);
        // critical region:
        // use D, E, B
        release(D);
        release(E);
        release(B);
    }
}

void P2()
{
    while (true) {
        get(C);
        get(F);
        get(D);
        // critical region:
        // use C, F, D
        release(C);
        release(F);
        release(D);
    }
}
```

---

Rule of Thumb

- A cycle in the resource allocation graph is a necessary condition for a deadlock
- But not a sufficient condition
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- Sleeping Barber Problem

Bounded-Buffer Problem

- Shared buffer with N slots to store at most N items
- Producer processes data items and puts into the buffer
- Consumer gets the data items from the buffer
- Variable empty keeps number of empty slots in the buffer
- Variable full keeps number of full items in the buffer
Bounded Buffer - 1 Semaphore Soln

• The structure of the **producer process**

```c
int empty=N, full=0;
do {
  // produce an item
  wait (mutex);
  if (empty> 0){
    // add the item to the buffer
    empty --; full++;
  }
signal (mutex);
}
} while (true);
```

Bounded Buffer - 1 Semaphore Soln

• The structure of the **consumer process**

```c
do {
  wait (mutex);
  if (full>0){
    // remove an item from buffer
    full--; empty++;
  }
signal (mutex);
  // consume the removed item
}
} while (true);
```
Summary

- Deadlocks
  - Deadlock Characterization
  - Resource Allocation Graphs
- Classic Problems of Synchronization
  - Bounded Buffer

- **Next Lecture: Deadlocks - II**
- **Reading Assignment: Chapter 7 from Silberschatz.**

Acknowledgements

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