Lecture 16
Concurrency Bugs
Concurrency Bugs

- Non-Deadlock Bugs
- Deadlock Bugs

<table>
<thead>
<tr>
<th>Application</th>
<th>What it does</th>
<th>Non-Deadlock</th>
<th>Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>Database Server</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>Apache</td>
<td>Web Server</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Web Browser</td>
<td>41</td>
<td>16</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>Office Suite</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>74</td>
<td>31</td>
</tr>
</tbody>
</table>

Figure 32.1: Bugs In Modern Applications

Non-Deadlock Bugs

- A large fraction (97%) of non-deadlock bugs are either atomicity or order violations.
- Atomicity-Violation Bugs
- Order-Violation Bugs
Concurrency Bugs

- Atomicity violation bugs
  - An example bug found in MySQL

```c
1  Thread 1::
2  if (thd->proc_info) {
3    ...
4    fputs(thd->proc_info, ...);
5    ...
6  }
7
8  Thread 2::
9  thd->proc_info = NULL;
```

- Discussion: why broken and how to fix it?
Concurrency Bugs

```c
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
4      pthread_mutex_lock(&proc_info_lock);
5    if (thd->proc_info) {
6        ...
7        fputs(thd->proc_info, ...);
8      ...
9  }
10     pthread_mutex_unlock(&proc_info_lock);

Thread 2::
13    pthread_mutex_lock(&proc_info_lock);
14      thd->proc_info = NULL;
15     pthread_mutex_unlock(&proc_info_lock);
```
Concurrency Bugs

- Order-Violation Bugs

```c
1  Thread 1::
2  void init() {
3      ...
4      mThread = PR_CreateThread(mMain, ...);
5      ...
6  }
7
8  Thread 2::
9  void mMain(...) {
10     ...
11     mState = mThread->State;
12     ...
13  }
```
Concurrency Bugs

```c
 pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
 pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
 int mtInit = 0;

 Thread 1::
 void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);
   // signal that the thread has been created...
   pthread_mutex_lock(&mtLock);
   mtInit = 1;
   pthread_cond_signal(&mtCond);
   pthread_mutex_unlock(&mtLock);
   ...
 }

 Thread 2::
 void mMain(...) {
   ...
   // wait for the thread to be initialized...
   pthread_mutex_lock(&mtLock);
   while (mtInit == 0)
     pthread_cond_wait(&mtCond, &mtLock);
   pthread_mutex_unlock(&mtLock);
   mState = mThread->State;
   ...
 }
```
Deadlock Bugs

- What is a deadlock?
- How to describe a deadlock?
- Why do deadlocks occur?
- How to ... (handle deadlocks)
  - Prevent deadlocks?
  - Avoid deadlocks?
  - Detect deadlocks?
  - Recover from deadlocks?
In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.
What is A Deadlock?

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  - **Request.** The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  - **Use.** The process can operate on the resource.
  - **Release.** The process releases the resource.
What is A Deadlock?

- Deadlock:
  - A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.
An Example of A Potential Deadlock

Thread 1:
lock(L1);
lock(L2);

Thread 2:
lock(L2);
lock(L1);
The Deadlock Dependency Graph

Thread 1

Holds

Lock L1

Wanted by

Lock L2

Wanted by

Thread 2

Holds

Holds
Contents

- What is a deadlock?
- How to describe a deadlock?
- Why do deadlocks occur?
- How to ...
  - Prevent deadlocks?
  - Avoid deadlocks?
  - Detect deadlocks?
  - Recover from deadlocks?
Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.

The set of vertices $V$ is partitioned into two different types of nodes: $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

The set of edges $E$ consists of request edges and assignment edges.
We represent each process $P_i$ as a circle and each resource type $R_j$ as a rectangle. Since resource type $R_j$ may have more than one instance, we represent each such instance as a dot within the rectangle.

When process $P_i$ requests an instance of resource type $R_j$, a request edge is inserted in the resource-allocation graph.

When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge.

When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.
Resource-Allocation
Graph

- Deadlock?
- No
Resource-Allocation Graph

- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
Resource-Allocation Graph

- Deadlock? 
- Yes
Deadlock?
No

Resource-Allocation Graph

- Deadlock?
- No
Contents

- What is a deadlock?
- How to describe a deadlock?
- Why do deadlocks occur?
- How to ...
  - Prevent deadlocks?
  - Avoid deadlocks?
  - Detect deadlocks?
  - Recover from deadlocks?
Why Do Deadlocks Occur? -1

- **Complex dependencies** arise between components in large code bases.
  - E.g., The virtual memory system might need to access the file system in order to page in a block from disk; the file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system.
Why Do Deadlocks Occur? -2

- The nature of **encapsulation**.
  - Modularity does not mesh well with locking.
  - Some seemingly innocuous interfaces almost invite you to deadlock. E.g., Java Vector class and the method `AddAll()`.

```java
Vector v1, v2;
v1.AddAll(v2);
```

“Deadlock Immunity: Enabling Systems To Defend Against Deadlocks”
Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
OSDI ’08, San Diego, CA, December 2008
Necessary Conditions

- Four conditions need to hold for a deadlock to occur
  - Mutual exclusion: Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
  - Hold-and-wait: Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
Necessary Conditions

- **No preemption**: Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.

- **Circular wait**: There exists a circular chain of threads such that each thread holds one more resources (e.g., locks) that are being requested by the next thread in the chain.

- If any of these four conditions are not met, deadlock cannot occur.
Contents

- What is a deadlock?
- How to describe a deadlock?
- Why do deadlocks occur?
- How to ...
  - Prevent deadlocks?
  - Avoid deadlocks?
  - Detect deadlocks?
  - Recover from deadlocks?
Prevention

- How to prevent deadlocks?
  - Prevent any of the above necessary conditions.
Preventing Circular Wait

The most practical prevention technique (and certainly one that is frequently employed) is to write your locking code such that you never induce a circular wait.

- total ordering
- partial ordering

E.g., *lock1* before *lock2* before *lock3*, etc.
Preventing Circular Wait

- Enforcing lock ordering by lock address

```c
do_something(mutex_t *m1, mutex_t *m2)
if (m1 > m2) { // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (it is not the same lock)
```

- Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering.
Preventing Hold-and-Wait

- The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically.
  - E.g.,
    1. `lock(prevention);`
    2. `lock(L1);`
    3. `lock(L2);`
    4. `...`
    5. `unlock(prevention);`
One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when it has none.

Note that the solution is problematic:
- Encapsulation works against us
- Decreased concurrency
If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.
Avoid the situation when waiting for one lock we are holding another

- E.g., Make use of the \texttt{trylock()} routine.

```c
1    top:
2      lock(L1);
3    if (trylock(L2) == -1) {
4      unlock(L1);
5      goto top;
6    }
```

- Problem:
  - Encapsulation works against us
  - Livelock
Livelock

- The livelock problem: It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks.

- Solution to the livelock problem: add a random delay before looping back and trying the entire thing over again.
Deadlock vs. Livelock

A livelock is like a ...

A deadlock is like a ...
Preventing Mutual Exclusion

- Wait-free concurrency

- Remember `CompareAndSwap()`?

```c
int CompareAndSwap(int *address, int expected, int new) {
    if (*address == expected) {
        *address = new;
        return 1; // success
    }
    return 0; // failure
}
```
Imagine we now wanted to atomically increment a value by a certain amount. We could do it as follows:

```c
void AtomicIncrement(int *value, int amount) {
    do {
        int old = *value;
    } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

No lock is acquired, and no deadlock can arise (though livelock is still a possibility).
Preventing Mutual Exclusion

- list insertion: inserts at the head of a list

```c
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    pthread_mutex_lock(listlock); // begin critical section
    n->next = head;
    head = n;
    pthread_mutex_unlock(listlock); // end critical section
}
```
Preventing Mutual Exclusion

- list insertion: inserts at the head of a list

```c
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```
Contents

- What is a deadlock?
- How to describe a deadlock?
- Why do deadlocks occur?
- How to ...
  - Prevent deadlocks?
  - Avoid deadlocks?
  - Detect deadlocks?
  - Recover from deadlocks?
Deadlock Avoidance via Scheduling

- Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur.
Deadlock Avoidance via Scheduling

An example:

Assume we have two processors and four threads which must be scheduled upon them. Assume further we know that Thread 1 (T1) grabs locks L1 and L2 (in some order, at some point during its execution), T2 grabs L1 and L2 as well, T3 grabs just L2, and T4 grabs no locks at all.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>L2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Deadlock Avoidance via Scheduling

- A smart scheduler could thus compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise.
Deadlock Avoidance via Scheduling

- Discussion:
  - What if T3 grabs L1 and L2?

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>L2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
We will detail two deadlock-avoidance algorithms.
A sequence of processes \(<P_1, P_2, ..., P_n>\) is a **safe sequence** for the current allocation state if, for each \(P_i\), the resource requests that \(P_i\) can still make can be satisfied by the currently available resources plus the resources held by all \(P_j\), with \(j<i\).

If no such sequence exists, then the system state is said to be unsafe.
Consider a system with 12 magnetic tape drives and three processes: P0, P1, and P2.

<table>
<thead>
<tr>
<th>Maximum Needs</th>
<th>Current Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>10</td>
</tr>
<tr>
<td>$P_1$</td>
<td>4</td>
</tr>
<tr>
<td>$P_2$</td>
<td>9</td>
</tr>
</tbody>
</table>

At time $t_0$, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies the safety condition.
Safe State

- A safe state is not a deadlocked state.
- An unsafe state may lead to a deadlock.

Discussion:
Show by an example that an unsafe state may not lead to a deadlock.
A system can go from a safe state to an unsafe state.

Discussion

Show by an example that a system can go from a safe state to an unsafe state.

Suppose that, at time $t_1$, process $P2$ requests and is allocated one more tape drive. The system is no longer in a safe state.
A claim edge $P_i \rightarrow R_j$ indicates that process $P_i$ may request resource $R_j$ at some time in the future.

- The resources must be claimed a priori in the system.

Discussion:

- Suppose $P_2$ requests $R_2$. Can it be allocated?
- No, since this action will create a cycle in the graph.
The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

The Banker’s algorithm is applicable to such a system but is less efficient than the resource-allocation graph scheme.
With multiple instances of each resource type.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. (This number may not exceed the total number of resources in the system.)

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.

If it will, the resources are allocated; Otherwise, the process must wait until some other process releases enough resources.
Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances.

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>3 3 2</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>

Is the system safe currently?
Example

- Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2).

- Discussion: Can the request be satisfied?

- Hint: suppose yes, the allocation state should be

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P_0</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P_1</td>
<td>3 0 2</td>
<td>0 2 0</td>
</tr>
<tr>
<td>P_2</td>
<td>3 0 2</td>
<td>6 0 0</td>
</tr>
<tr>
<td>P_3</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P_4</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

- Is it a safe state?
Banker’s Algorithm

- Discussion: Can you describe your algorithm?

- Most probably, what you are thinking about is the Banker’s algorithm.
Banker’s Algorithm

- data structures
  - Available[j]: the number of instances of resource type Rj which are available.
  - Max[i][j]: the number of instances of resource type Rj that process Pi may request at most.
  - Allocation[i][j]: the number of instances of resource type Rj that process Pi is currently allocated.
  - Need[i][j]: the number of instances of resource type Rj that process Pi may need to complete its task.
Safety Algorithm

1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n - 1$.

2. Find an index $i$ such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   If no such $i$ exists, go to step 4.

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.

4. If $Finish[i] == true$ for all $i$, then the system is in a safe state.
Resource-Request Algorithm

Let $Request_i$ be the request vector for process $P_i$. If $Request_i[j] == k$, then process $P_i$ wants $k$ instances of resource type $R_j$. When a request for resources is made by process $P_i$, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise, $P_i$ must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process $P_i$ by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process $P_i$ is allocated its resources. However, if the new state is unsafe, then $P_i$ must wait for $Request_i$, and the old resource-allocation state is restored.
What is a deadlock?
How to describe a deadlock?
Why do deadlocks occur?
How to ...  
  - Prevent deadlocks?
  - Avoid deadlocks?
  - Detect deadlocks?
  - Recover from deadlocks?
Detect and Recover

- Allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected.
If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph, by removing the resource nodes and collapsing the appropriate edges.
Detection

- With multiple instances of each resource type

1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively. Initialize $Work = Available$. For $i = 0, 1, \ldots, n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$. Otherwise, $Finish[i] = true$.

2. Find an index $i$ such that both
   - $Finish[i] = false$
   - $Request_i \leq Work$

   If no such $i$ exists, go to step 4.

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.

4. If $Finish[i] = false$ for some $i, 0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] = false$, then process $P_i$ is deadlocked.

Discussion:
Difference between detection algorithm and safety algorithm.
## Detection

- **Discussion: Deadlock?**

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>2 0 2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 3</td>
<td><strong>0 0 1</strong></td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>
Detection-Algorithm

Usage

- When should we invoke the detection algorithm? It depends on:
  - How often is a deadlock likely to occur?
  - How many processes will be affected by deadlock when it happens?

- If we invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately, we can identify not only the deadlocked set of processes but also the specific process that “caused” the deadlock.
Contents

- What is a deadlock?
- How to describe a deadlock?
- Why do deadlocks occur?
- How to ...
  - Prevent deadlocks?
  - Avoid deadlocks?
  - Detect deadlocks?
  - Recover from deadlocks?
Recovery

- Process Termination
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated.

- Resource Preemption (three issues need to be addressed)
  - Selecting a victim: Which resources and which processes are to be preempted?
  - Rollback: If we preempt a resource from a process, what should be done with that process?
  - Starvation: How can we guarantee that resources will not always be preempted from the same process?
Exercise 1

Tell the relations and differences between preventing, avoiding, detecting, and recovering from a deadlock.
7.11 考虑下面的一个系统在某一时刻的状态。

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C D</td>
<td>A B C D</td>
<td>A B C D</td>
</tr>
<tr>
<td>$P_0$</td>
<td>0 0 1 2</td>
<td>0 0 1 2</td>
<td>1 5 2 0</td>
</tr>
<tr>
<td>$P_1$</td>
<td>1 0 0 0</td>
<td>1 7 5 0</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>1 3 5 4</td>
<td>2 3 5 6</td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>0 6 3 2</td>
<td>0 6 5 2</td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 1 4</td>
<td>0 6 5 6</td>
<td></td>
</tr>
</tbody>
</table>

使用银行家算法回答下面问题：

a. *Need* 矩阵的内容是怎样的？

b. 系统是否处于安全状态？

c. 如果从进程 $P_1$ 发来一个请求 (0,4,2,0)，这个请求能否立刻被满足？