Common Concurrency Problems (When Good Threads go Bad)
Common Concurrency Problems

• We’ve briefly talked about deadlock
  • Lots of early research focused on this
  • We’ll dive in a bit more deeply today

• What other concurrency problems exist?
  • Look at some example concurrency problems found in real code bases
What Types Of Bugs Exist?

• Focus on four major open-source applications
  • MySQL, Apache, Mozilla, OpenOffice

<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>Open Office</td>
<td>Office Suite</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>Total</strong></td>
<td><strong>74</strong></td>
<td><strong>31</strong></td>
</tr>
</tbody>
</table>

Bugs In Modern Applications

• Non-deadlock bugs make up a majority of concurrency bugs
• Two major types of non deadlock bugs
  • Atomicity violation
  • Order violation
Atomicity-Violation Bugs

- The desired **serializability** among multiple memory accesses is **violated**
- Simple Example found in MySQL:
  - Two different threads access the field `proc_info` in the struct `thd`

```
1   Thread1::
2      if(thd->proc_info){
3          ...
4          fputs(thd->proc_info, ...);
5          ...
6      }
7
8   Thread2::
9      thd->proc_info = NULL;
```
Atomicity-Violation Bugs (Cont.)

• **Solution**: Simply add locks around the shared-variable references

```c
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1::
    pthread_mutex_lock(&lock);
    if(thd->proc_info){
        ... 
        fputs(thd->proc_info, ...);
        ... 
    }
    pthread_mutex_unlock(&lock);

Thread2::
    pthread_mutex_lock(&lock);
    thd->proc_info = NULL;
    pthread_mutex_unlock(&lock);
```
Order-Violation Bugs

• The desired order between two memory accesses is flipped
  • I.e., \textbf{A} should always be executed before \textbf{B}, but the order is not enforced during execution
  • \textbf{Example:}
    • The code in Thread2 seems to assume that the variable \texttt{mThread} has already been \textit{initialized} (and is not \texttt{NULL})

```c
Thread1::
void init(){
    mThread = PR_CreateThread(mMain, ...);
}

Thread2::
void mMain(...){
    mState = mThread->State
}
```
Order-Violation Bugs Solution

- **Solution**: enforce ordering using condition variables

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

4   Thread 1::
5   void init(){
6       ...
7       mThread = PR_CreateThread(mMain,...);
8   }
9   // signal that the thread has been created.
10  pthread_mutex_lock(&mtLock);
11  mtInit = 1;
12  pthread_cond_signal(&mtCond);
13  pthread_mutex_unlock(&mtLock);
14  ...
15 }
16
17   Thread2::
18   void mMain(...){
19       // wait for the thread to be initialized ...
20       pthread_mutex_lock(&mtLock);
21       while(mtInit == 0)
22       pthread_cond_wait(&mtCond, &mtLock);
23       pthread_mutex_unlock(&mtLock);
24       mState = mThread->State;
25       ...
26 }
```
Deadlock Bugs

• The presence of a **cycle** in a resource-allocation graph
  - Thread1 is holding a lock \( L_1 \) and waiting for another one, \( L_2 \).
  - Thread2 that holds lock \( L_2 \) is waiting for \( L_1 \) to be release.

\[
\begin{align*}
\text{Thread 1:} & \\
& \text{lock(L1);} \\
& \text{lock(L2);} \\
\text{Thread 2:} & \\
& \text{lock(L2);} \\
& \text{lock(L1);} \\
\end{align*}
\]
Why Do Deadlocks Occur?

• Reason 1:
  • In large code bases, complex dependencies arise between components

• Reason 2:
  • Due to the nature of encapsulation
    • Hide details of implementations and make software easier to build in a modular way
    • Such modularity does not mesh well with locking
Why Do Deadlocks Occur? (Cont.)

• Example: Java Vector class and the method `AddAll()`

```
1    Vector v1,v2;
2    v1.AddAll(v2);
```

• **Locks** for both the vector being added to (v1) and the parameter (v2) need to be acquired
  
  • The routine acquires said locks in some order (e.g., v1 then v2)
  • If some other thread calls `v2.AddAll(v1)` at nearly the same time → We have the potential for deadlock
Conditional for Deadlock

• **Four conditions** need to hold for a deadlock to occur

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual Exclusion</td>
<td>Threads claim exclusive control of resources that they require</td>
</tr>
<tr>
<td>Hold-and-wait</td>
<td>Threads hold resources allocated to them while waiting for additional resources</td>
</tr>
<tr>
<td>No preemption</td>
<td>Resources cannot be forcibly removed from threads that are holding them</td>
</tr>
<tr>
<td>Circular wait</td>
<td>There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain</td>
</tr>
</tbody>
</table>

• If any of these four conditions are not met, **deadlock cannot occur**
Deadlock vs. Starvation

• Deadlock: A circular waiting for resources
• Starvation: A thread never makes progress because other threads are using resources it needs
• Starvation ≠ Deadlock
  • Deadlock can be seen as a special case of starvation
Methods for Handling Deadlocks

• Ensure that the system will **never** enter a deadlock state:
  • Deadlock prevention – deadlock is not possible in the system
  • Deadlock avoidance – prevent a particular instance of deadlock from happening

• Allow the system to enter a deadlock state and then recover

• Ignore the problem and pretend that deadlocks never occur in the system
  • Used by most operating systems, including UNIX
Deadlock Prevention

• Restrain the ways requests can be made to make at least one of the four deadlock conditions does not hold

• **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 Avoidance

• Requires that the system has some additional *a priori* information available
• Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
• The deadlock-avoidance algorithm 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 and allocated resources, and the maximum demands of the processes
Safe State

• When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

• System is in safe state if there exists a safe sequence \(<P_1, P_2, \ldots, P_n>\) 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 \(P_j\) 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.
Avoidance Algorithms

• Single instance of a resource type
  • Use a resource-allocation graph
  • Check for cycles

• Multiple instances of a resource type
  • Use the banker’s algorithm
Resource-Allocation Graph

- We have 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 $R_j \rightarrow P_i$
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

\[ \begin{align*}
\text{P}_1 & \rightarrow \text{R}_1 \\
\text{P}_1 & \rightarrow \text{R}_2 \\
\text{P}_2 & \rightarrow \text{R}_1 \\
\text{P}_2 & \rightarrow \text{R}_3 \\
\text{P}_3 & \rightarrow \text{R}_3 \\
\text{R}_2 & \rightarrow \text{R}_4
\end{align*} \]
Resource Allocation Graph With A Deadlock
Graph With A Cycle But No 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
Avoidance Algorithms

• Single instance of a resource type
  • Use a resource-allocation graph
  • Check for cycles

• Multiple instances of a resource type
  • Use the banker’s algorithm
Banker’s Algorithm

• Have multiple instances of resources

• Each process must **a priori** claim maximum resource use (not to exceed total resources in the system)

• When a process requests a resource, it may have to wait

• When a process gets all its resources, it must return them in a finite amount of time
Deadlock Detection

• Allow system to enter deadlock state
• Detect that deadlock has occurred
  • Detection algorithm
• Recover from deadlock
  • Recovery scheme
Maintain **wait-for** graph

- Nodes are processes
- \( P_i \rightarrow P_j \) if \( P_i \) is waiting for \( P_j \)

Periodically invoke an algorithm that searches for a cycle in the graph

- If there is a cycle, deadlock exists

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
Resource-Allocation Graph and Wait-for Graph

Resource-Allocation Graph

Corresponding wait-for graph
Recovery from Deadlock: Process Termination

• Abort all deadlocked processes

• Abort one process at a time until the deadlock cycle is eliminated

• 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 from Deadlock: Resource Preemption

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

• Non-deadlock bugs are common
  • Atomicity violations
  • Order violations
  • Often easy to fix (once discovered)

• Deadlock
  • Why it occurs
  • What can be done about it
    • Can schedule to avoid deadlock (Banker’s Algorithm)
      • Assumes we know maximum resources used a priori
    • Most practical: develop a lock acquisition order which will prevent deadlock from occurring