Sistemas Operativos: Concurrency

Bugs

Pedro F. Souto (`pfs@fe.up.pt`)

April 1, 2020

Roadmap

Concurrency Bugs

Deadlocks

Introduction

- Concurrent programming is hard, and concurrent bugs are frequent:

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

src: Lu et al.

Non-deadlocks Of which 97% are of one one of two types

- Atomicity violation
- Order violation

Deadlocks Represent about one 3rd of all concurrency bugs fixed

Atomicity Violation Bugs

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

What is wrong here?

Atomicity Violation Bugs

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

What is wrong here?

- There is an assumption that a code segment is executed atomically
- But the scheduler has the last word

How can we fix, this?

Atomicity Violation Bugs

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

What is wrong here?

- There is an assumption that a code segment is executed atomically
- But the scheduler has the last word

How can we fix, this?

- Use synchronization

Atomicity Violation Bugs Fix

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}
pthread_mutex_unlock(&proc_info_lock);

Thread 2::
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

Order Violation Bugs

```
Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);
   ...
}

Thread 2::
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

What is wrong here?

Order Violation Bugs

```
Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);
   ...
}

Thread 2::
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

What is wrong here?

- There is an assumption that code segment are executed in a given order
- But, again, the scheduler has the last word

How can we fix, this?

Order Violation Bugs

```
Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);
   ...
}

Thread 2::
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

What is wrong here?

- There is an assumption that code segment are executed in a given order
- But, again, the scheduler has the last word

How can we fix, this?

- Use synchronization

Order Violation Bugs Fix

```
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;
    ...
```

Roadmap

Concurrency Bugs

Deadlocks

Deadlocks

Deadlock is a classic problem

- We have already mentioned it a couple of times
- Atomicity-violation and order-violation problems are "easy" to avoid
- Deadlocks are hard to avoid, and have been the subject of extensive research

Example

Thread 1:	Thread 2:
`lock(L1);`	`lock(L2);`
`lock(L2);`	`lock(L1);`

What can go wrong here?

Deadlocks

Deadlock is a classic problem

- We have already mentioned it a couple of times
- Atomicity-violation and order-violation problems are "easy" to avoid
- Deadlocks are hard to avoid, and have been the subject of extensive research

Example

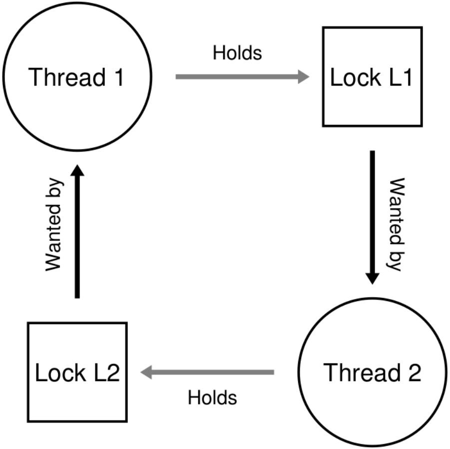
```
Thread 1:                Thread 2:
lock(L1);                lock(L2);
lock(L2);                lock(L1);
```

What can go wrong here?

- Think about the wrong order of scheduling decision

Deadlock: Resource allocation graph

```
Thread 1:    Thread 2:
lock(L1)
             lock(L2)
lock(L2)
             lock(L1)
```



- A cycle in a resource allocation graph (when there is one instance of each resource) is indicative of deadlock

Deadlocks: Are hard to avoid

Complex dependencies in large code bases. For example in an OS:

- The VM system may use the file system to page in a block from disk;
- The file system may request a page from the VM to read the block into

Abstraction more precisely encapsulation.

- Hiding implementation details helps code modularity and reuse
- But implementation details may be critical for avoiding deadlocks

Deadlocks: Encapsulation

Example Consider the Java `Vector` class and its `AddAll()` method for merging two vectors

```
Vector v1, v2;

v1.addAll(v2);
```

What can go wrong?

Deadlocks: Encapsulation

Example Consider the Java `Vector` class and its `AddAll()` method for merging two vectors

```
Vector v1, v2;

v1.addAll(v2);
```

What can go wrong?

- For this method to be thread-safe, we need to acquire locks on both vectors
  - The `java.util.Vector` class is not thread-safe
- Let's assume we take a lock on `v1` first and then on `v2`

And now, is it clear what can go wrong?

Deadlocks: Encapsulation

Example Consider the Java `Vector` class and its `AddAll()` method for merging two vectors

```
Vector v1, v2;

v1.addAll(v2);
```

What can go wrong?

- For this method to be thread-safe, we need to acquire locks on both vectors
  - The `java.util.Vector` class is not thread-safe
- Let's assume we take a lock on `v1` first and then on `v2`

And now, is it clear what can go wrong?

```
    Thread 1                Thread 2

v1.addAll(v2)           v2.addAll(v1)
```

Deadlocks: Necessary conditions

Mutual exclusion Threads claim exclusive control of resources they require

- E.g. a thread grabs a lock

Hold and wait Threads hold resources allocated to them (e.g. locks) while waiting for additional resources (e.g. other locks)

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

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

Handling Deadlocks: Approaches

Prevention Ensure that at least one of the 4 required conditions do not hold

Avoidance At run-time take resource allocation decisions (e.g. granting locks) that cannot lead to deadlocks, i.e. play it safe

- Prevention (impedir) is a structural approach
- Avoidance (evitar) is a run-time approach

Detect and Recover

Ignore Tanembaum calls it the ostrich algorithm

Prevention: Circular Wait

Obs. This is the approach most often used in practice

How to implement it? Define a total order for acquiring resources (locks)

- E.g. if there are only 2 locks, `L1` and `L2`
- If a thread needs both locks, always acquire `L1` before `L2`

Optimization Use a partial order rather than a total order

- This makes it more manageable, not more performant

Example Linux memory mapping code

- Defines 10 partial orders

Issues The partial orders must be defined very carefully

- And followed strictly by the programmers

Check Remzi's nice tip on the use of lock address for defining a total order on pg. 8

Prevention: Hold-and-Wait

Idea Acquire all locks (resources) at once, atomically

```
lock(prevention);
lock(L1);
lock(L2);
...
unlock(prevention);
```

What is the `prevention` mutex used for?

Issues

- Must know which locks are necessary ahead of time
- Likely to reduce concurrency

Prevention: No Preemption

Obs. Many resources must be held until explicitly released

- E.g., this is critical to prevent atomicity violations with locks

Idea If forced to wait, release all the locks already acquired

- Actually, this can be seen more as a way to prevent *hold-and-wait*

Example Using `try_lock()` rather than `lock()`

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

- Hard to use with encapsulation
- Nevertheless, might work with the `addAll()` method
  - Although it might lead to a livelock (very unlikely)

Livelock a situation in which two or more threads are not blocked, but nevertheless do not make progress, e.g. keep jumping to `top`, never acquiring both locks

Prevention: Mutual Exclusion

Obs. Many resources must be held in mutual exclusion

- E.g., this is critical to prevent atomicity violations with locks

Idea Use wait-free data structures

- I.e. use powerful read-modify-write HW instructions to build common data-structures without explicit locking (check some examples, on pp. 10 and 11)

Issues

Livelock if there is contention and a thread looses, it needs to retry;

Complexity it is not trivial to design wait-free data structures, but slowly they are being added to some kernels, e.g. in Linux

Deadlock Avoidance

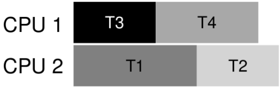
Assumes knowledge about which locks/resources various thread may require

Idea dynamically allocate resources so as to avoid the occurrence of deadlocks

Simple Example For illustration purposes, from the book:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- Deadlocks can be avoided by not running threads `T1` and `T2` simultaneously



Deadlock Avoidance

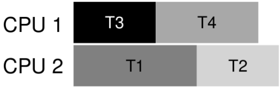
Assumes knowledge about which locks/resources various thread may require

Idea dynamically allocate resources so as to avoid the occurrence of deadlocks

Simple Example For illustration purposes, from the book:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- Deadlocks can be avoided by not running threads `T1` and `T2` simultaneously



- What if `T1` or `T2` are preempted?

Deadlock Avoidance

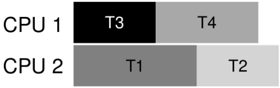
Assumes knowledge about which locks/resources various thread may require

Idea dynamically allocate resources so as to avoid the occurrence of deadlocks

Simple Example For illustration purposes, from the book:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- Deadlocks can be avoided by not running threads `T1` and `T2` simultaneously



- What if `T1` or `T2` are preempted?

Issue Assumption

- May be acceptable for critical systems
  - These use more sophisticated algorithms, e.g. priority ceiling

Detection and Recovery

Idea Let deadlocks occur, but when they occur detect them and recover.

- May be acceptable, depending on the probability of deadlock and on the cost of recovery

Detection algorithm Basically, checks if there is some order for threads that currently have some lock/resource to terminate

When should it be run?

Detection and Recovery

Idea Let deadlocks occur, but when they occur detect them and recover.

- May be acceptable, depending on the probability of deadlock and on the cost of recovery

Detection algorithm Basically, checks if there is some order for threads that currently have some lock/resource to terminate

When should it be run?

- Every time a resource request is made
- Periodically
- When the CPU utilization drops significantly

Recovery Just break one of the necessary conditions

Preempt may require human intervention;

Rollback requiries checkpointing the state periodically (used in databases)

Terminate one or more threads, breaking the cycle

Which thread should be the victim?

Conclusion

- Concurrent programming is hard, and concurrent bugs are frequent:

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Deadlocks Represent about one 3rd of all concurrency bugs fixed (except for database systems)

Non-deadlocks Of which 97% are of one one of two types

Atomicity violation

Order violation

What does this mean?

- Deadlocks are not that common?
- Deadlocks are rare and elusive (and users just reboot the system when they happen)?