Synchronization: Review

- Need for Synchronization
- Typical concurrency problems
  - Atomicity Violation
  - Ordering Violation
- Introduction to Critical Sections

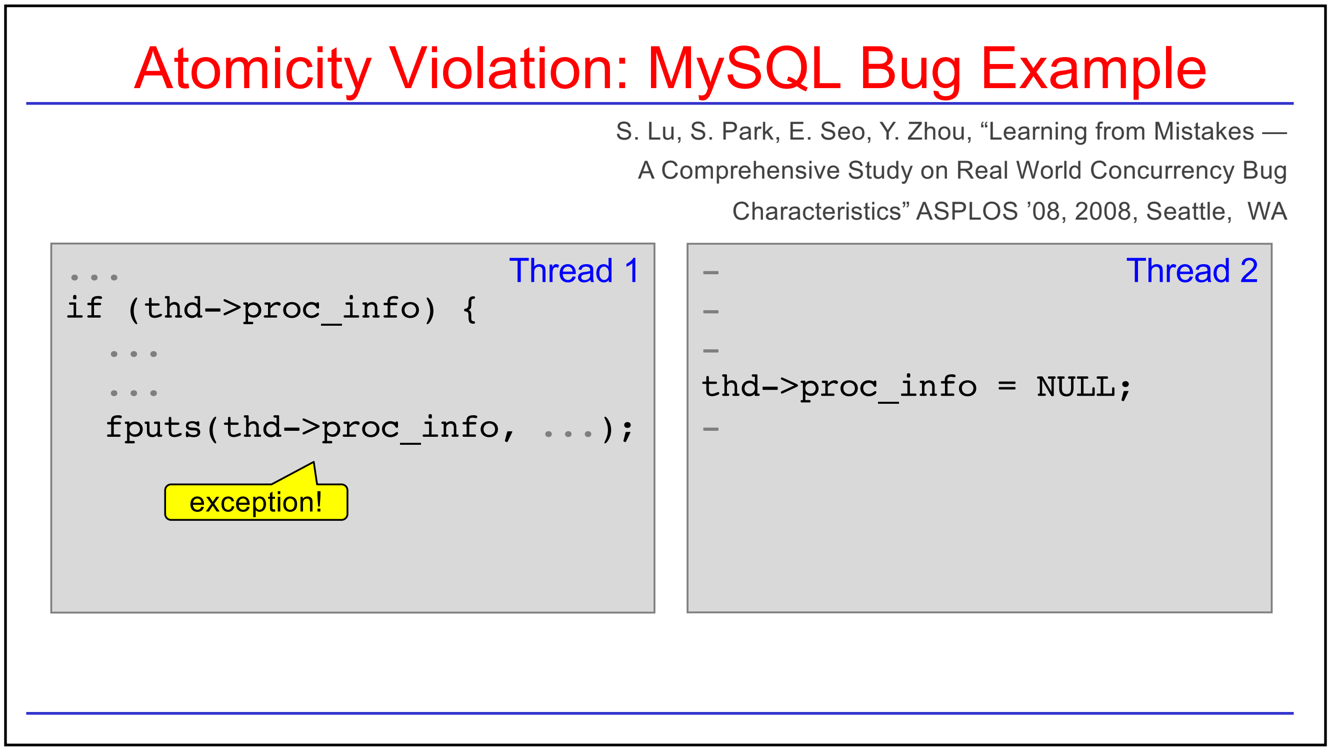
Atomicity Violation: MySQL Bug Example

S. Lu, S. Park, E. Seo, Y. Zhou, "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics" ASPLOS '08, 2008, Seattle, WA

```
thd->proc_info = NULL;
```

assumption:
`thd->proc_info != NULL`

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



Atomicity Violation: MySQL Bug Example

S. Lu, S. Park, E. Seo, Y. Zhou, “Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics” ASPLOS ’08, 2008, Seattle, WA

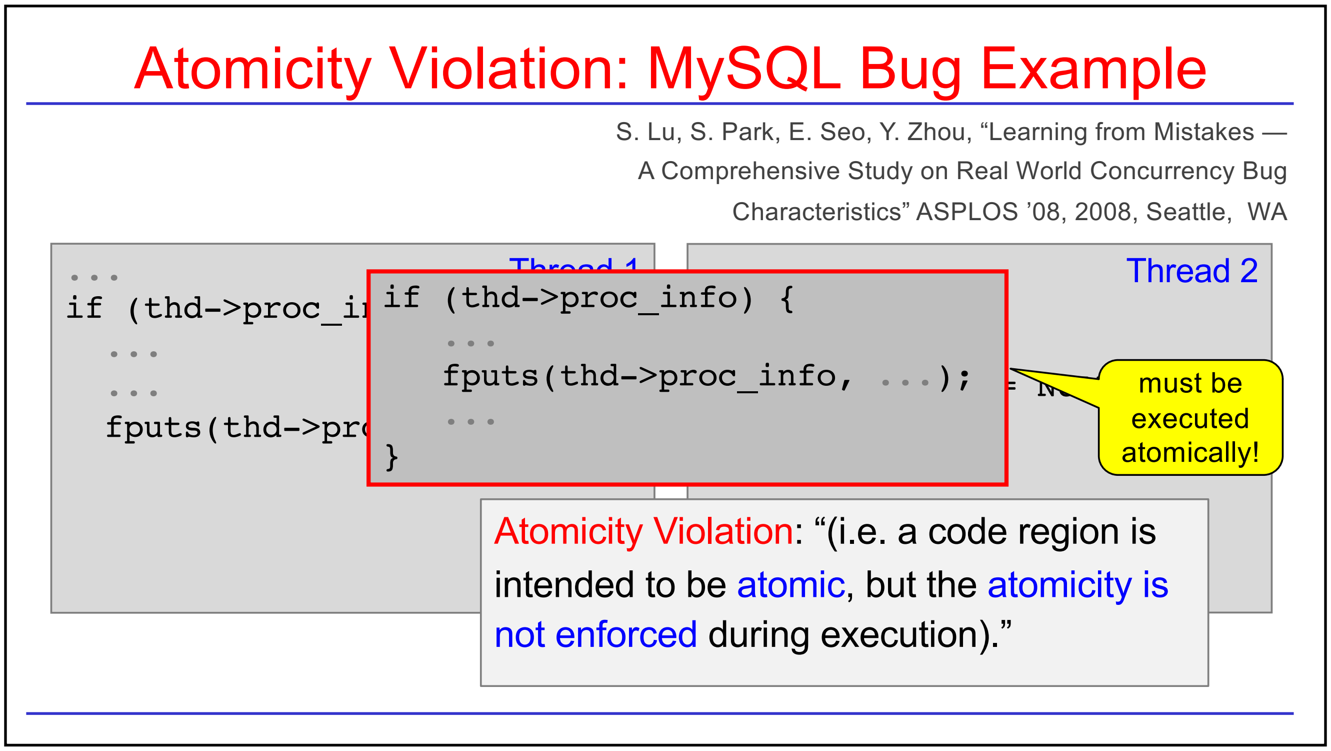
Thread 1

```
...
if (thd->proc_info) {
  ...
  ...
  fputs(thd->proc_info, ...);
```

exception!

Thread 2

```
–
–
–
thd->proc_info = NULL;
–
```



Atomicity Violation: MySQL Bug Example

S. Lu, S. Park, E. Seo, Y. Zhou, “Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics” ASPLOS ’08, 2008, Seattle, WA

Thread 1

```
...
if (thd->proc_i
  ...
  ...
  fputs(thd->pro
```

Thread 2

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

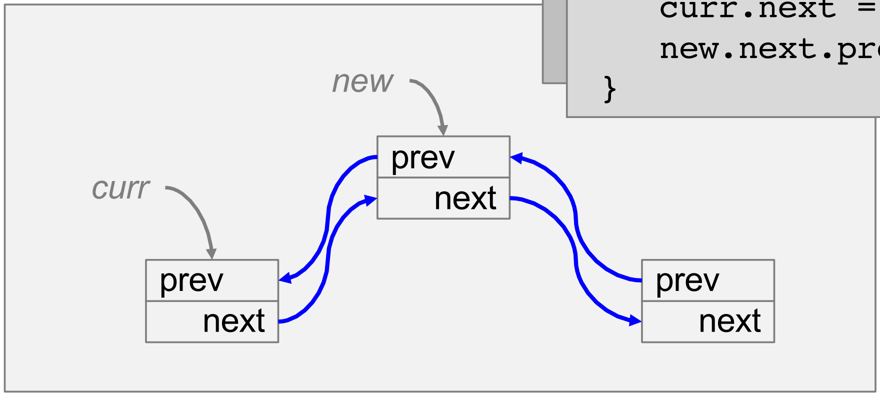
must be executed atomically!

Atomicity Violation: “(i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution).”

Another Atomicity Violation

Insertion of an element into a list.

```
void insert(new, curr) {
   new.next = curr.next;
   new.prev = curr;
   curr.next = new;
   new.next.prev = new;
}
```



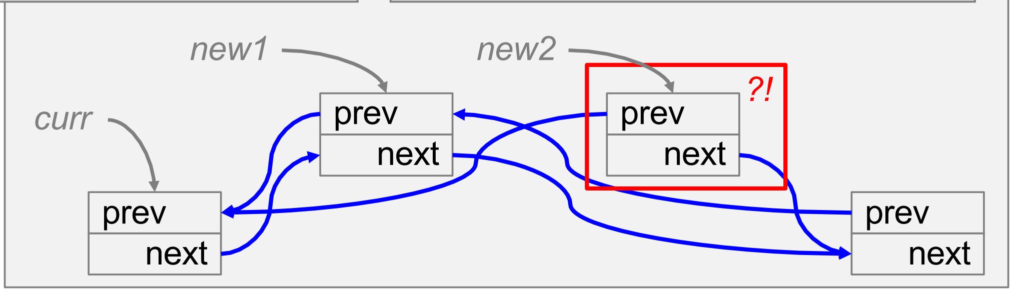
Another Atomicity Violation

Thread 1

```
new1.next = curr.next;
new1.prev = curr;
–
–
–
–
curr.next = new1;
new1.next.prev = new1;
```

Thread 2

```
–
–
new2.next = curr.next;
new2.prev = curr;
curr.next = new2;
new2.next.prev = new2;
–
–
```



Order Violations: Example

S. Lu, S. Park, E. Seo, Y. Zhou, "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics" ASPLOS '08, 2008, Seattle, WA

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

```
void init() {
    ...
    <reg>    = PR_CreateThread(mMain, ...);
    mThread = <reg>;
    ...
}
```

Order Violations: Example

S. Lu, S. Park, E. Seo, Y. Zhou, "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics" ASPLOS '08, 2008, Seattle, WA

Thread 1:

```
<reg>    =
PR_CreateThread(mMain, ...);



mThread = <reg>;
```

Thread 2:

```
mMain() {
   ...
   mState = mThread->State;
```

?!

Order Violations: Example

S. Lu, S. Park, E. Seo, Y. Zhou, “Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics” ASPLOS ’08, 2008, Seattle, WA

Thread 1:

```
<reg>   =
PR_CreateThread(mMain, ...);



mThread = <reg>;
```

Thread 2:

```
mMain() {
  ...
  mState = mThread->State;
```

Order Violation: “The desired order between two (groups of) memory accesses is flipped (i.e., *A* should always be executed before *B*, but the order is not enforced during execution).”

Avoiding Atomicity Violations: Critical Sections

- Execution of critical section by threads must be mutually exclusive.
- Need protocol to enforce mutual exclusion.

```
while (TRUE) {
   enter critical section;
   critical section;
   exit critical section;
   remainder section;
}
```

inter-task invariant violated

Avoiding Atomicity Violations: Critical Sections

- Execution of critical section by threads must be mutually exclusive.
- Need protocol to enforce mutual exclusion.

Thread 1
```
...
enter critical section;
critical section;
exit critical section;
–
–
–
remainder section;
```

Thread 2
```
–
–
–
–
enter critical section;
critical section;
exit critical section;
–
```

Avoiding Atomicity Violations: Critical Sections

- Execution of critical section by threads must be mutually exclusive.
- Need protocol to enforce mutual exclusion.

Thread 1
```
...
enter critical section;
critical section;
–
exit critical section;
remainder section
–
–
```

Thread 2
```
–
–
–
enter critical section;
–
–
enter critical section;
critical section
```

Avoiding Atomicity Violations: Critical Sections

- Execution of critical section by threads must be mutually exclusive.
- Need protocol to enforce mutual exclusion.

```
Thread 1
–
–
–
–
enter critical section;
critical section;
–
exit critical section;
```

```
Thread 2
...
enter critical section;
critical section;
exit critical section;
–
–
remainder section;
–
```

Non-Consecutive Critical Sections

```
while (TRUE) {
   enter critical section;
   critical section CS1;      } inter-task invariant violated
   exit critical section;
   non-critical section;
   enter critical section;
   critical section CS2;      } inter-task invariant violated again
   exit critical section;
   remaining section;
}
```

Non-Consecutive Critical Sections

```
...                            Thread 1
enter critical section;
critical section CS1;
-
exit critical section;
non-critical section;
-
-
enter critical section;
-
enter critical section;
critical section CS2;
exit critical section;
```

```
-                              Thread 2
-
-
enter critical section;
-
-
enter critical section;
critical section CS1;
-
exit critical section;
-
-
-
```

Multiple Critical Sections

inter-task invariant violated

some other inter-task invariant violated

```
while (TRUE) {
   enter critical section A;
   critical section A;
   exit critical section A;
   non-critical section;
   enter critical section B;
   critical section B;
   exit critical section B;
   remaining section;
}
```

Multiple Critical Sections

Thread 1

```
...
enter critical section A;
critical section A;
–
–
exit critical section A;
non-critical section;
enter critical section B;
–
enter critical section B;
critical section B;
exit critical section B;
remainder section;
```

Thread 2

```
–
–
–
enter critical section B;
critical section B;
–
–
–
exit critical section B;
–
–
–
–
```

Multiple Critical Sections

Thread 1

```
...
lock_A.lock();
critical section A;
–
–
lock_A.unlock();
non-critical section;
lock_B.lock();
–
lock_B.lock();
critical section B;
lock_B.unlock();
remainder section;
```

Thread 2

```
–
–
–
lock_B.lock();
critical section B;
–
–
–
lock_B.unlock;
```

```
Lock lock_A, lock_B;
Lock::lock() {...};
Lock::unlock() {...};
```

Synchronization: Review - Summary

- Need for Synchronization
- Typical concurrency problems
  - Atomicity violations
  - Ordering violations
- Introduction to Critical Sections