Atomicity Violations

Godmar Back

Atomicity Violations

- Are a type of concurrency bug that can occur even if proper locking discipline is followed (that is, if a lock is consistently held while accessing or updating shared data)
- Occurs if updates that should be atomic are not done under the protection of the same lock *when combined*
- Possible pattern
  a. Acquire Lock protecting S
  b. Extract information I from shared state S
  c. (possibly) Release Lock protecting S
  d. …. Commit to action based on information I ….
  e. Acquire Lock protecting S'
  f. Act on shared state S' based on saved information I
  g. Release Lock protecting S'

Fails if information I obtained in b) is out of date in f) due to an update to shared state. S may be the same as S'

Atomicity Violation Example

```
char  *p = ….;  /* shared variable */
pthread_mutex_t lp; /* protects 'p' */
….
int getplen() {
  pthread_mutex_lock(&lp);
  int len = strlen(p);
  pthread_mutex_unlock(&lp);
  return len;
}
…
int nchars = getplen();            // obtain length (safe by itself)
char *copy = malloc(nchars + 1);  // commit to allocate memory
pthread_mutex_lock(&lp);
strcpy(copy, p);                   // copy now possibly out-of-date version
pthread_mutex_unlock(&lp);
```

Atomicity Violation Example (2)

```
public synchronized StringBuffer append(StringBuffer sb) {
    // note: StringBuffer.length() is synchronized
    int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length)
       expandCapacity(newcount);
    // StringBuffer.getChars() is synchronized
    sb.getChars(0, len, value, count);
    count = newcount;
    return this;
}
```

Obtain length

StringIndexOutOfBoundsException if length has changed

Found by Flanagan/Freund: Atomizer: A Dynamic Atomicity Checker for Multi-Threaded Programs, POPL 2004

Atomicity Violation Example (3)

```
pthread_mutex_lock(&worker->lock);    // protects worker queue
....
  struct listelem *e = list_pop_front(&worker->queue);  // grab task from worker queue
  struct future *f = list_entry(e, struct future, elem);
  pthread_mutex_lock(&f->lock);  // lock task to update state
  f->state = RUNNING;     // mark as running
  // run task
```

```
// somewhere in future_get()....
// lock protects state
pthread_mutex_lock(&future->lock);
if (future->state == NEW) {
    // lock containing list
   pthread_mutex_lock(&future->owner->lock);
   list_remove(&future->elem);
   pthread_mutex_unlock(&future->owner->lock);
   f->state = RUNNING;
   // run task
}
```

Top: worker locks queue, removes task and commits to running it.

Left: joiner sees fresh task, commits to running it, removes it from queue

Task is run twice!

Atomicity requirement: changing the task's state and removing it from the queue it is on must be one atomic operation

How to acquire 2 locks in opposite order while avoiding deadlock

```
A.lock()

…. find B

B.lock()

// holding A & B
```

```
retry:

   B.lock()

   …. find A

   if (!A.trylock())

         { B.unlock(); goto retry; }

// holding A & B
```