Synchronization, Part 3: Working with Mutexes And Semaphores

Arun Prakash Jana edited this page on Mar 8 · 1 revision

What is an atomic operation?

To paraphrase Wikipedia, "An operation (or set of operations) is atomic or uninterruptible if it appears to the rest of the system to occur instantaneously." Without locks, only simple CPU instructions ("read this byte from memory") are atomic (indivisible). On a single CPU system one could temporarily disable interrupts (so a sequence of operations cannot be interrupted) but in practice atomicity is achieved by using synchronization primitives, typically a mutex lock.

Incrementing a variable (`i++`) is *not* atomic because it requires three distinct steps: Copying the bit pattern from memory into the CPU; performing a calculation using the CPU's registers; copying the bit pattern back to memory. During this increment sequence, another thread or process can still read the old value and other writes to the same memory would also be over-written when the increment sequence completes.

How do I use mutex lock to make my data-structure thread-safe?

Note, this is just an introduction - writing high-performance thread-safe data-structures requires it's own book! Here's a simple data structure (a stack) that is not thread-safe:

```
// A simple fixed-sized stack (version 1)
int count;
double values[count];

void push(double v) { values[count++] = v; }
double pop() { return values[--count]; }
int is_empty() { return count == 0; }
```

Version 1 of the stack is not thread-safe because if two threads call push or pop at the same time then the results or the stack can be inconsistent. For example, imagine if two threads call pop at the same time then both threads may read the same value, both may read the original count value.

To turn this into a thread-safe data structure we need to identify the *critical sections* of our code i.e. which section(s) of the code must only have one thread at a time. In the above example the `push` , `pop` and `is_empty` functions access the same variables (i.e. memory) and all critical sections for the stack.

While `push` (and `pop`) is executing, the datastructure is an inconsistent state (for example the count may not have been written to, so may still contain the original value). By wrapping these methods with a mutex we can ensure that only one thread at a time can update (or read) the stack.

A candidate 'solution' is shown below. Is it correct? If not, how will it fail?

```
// An attempt at a thread-safe stack (version 2)
int count;
double values[count];
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void push() { pthread_mutex_lock(&m1); values[count++] = values; pthread_mutex_un
double pop() { pthread_mutex_lock(&m2); double v=values[--count]; pthread_mutex_u
int is_empty() { pthread_mutex_lock(&m1); return count == 0; pthread_mutex_unlock
```

The above code ('version 2') contains at least one error. Take a moment to see if you can the error(s) and work out the consequence(s).

If three called `push()` at the same time the lock `m1` ensures that only one thread at time manipulates the stack (two threads will need to wait until the first thread completes (calls unlock), then a second thread will be allowed to continue into the critical section and finally the third thread will be allowed to continue once the second thread has finished).

A similar argument applies to concurrent calls (calls at the same time) to `pop`. However version 2 does not prevent push and pop from running at the same time because `push` and `pop` use two different mutex locks.

The fix is simple in this case - use the same mutex lock for both the push and pop functions.

The code has a second error; `is_empty` returns after the comparison and will not unlock the mutex. However the error would not be spotted immediately. For example, suppose one thread calls `is_empty` and a second thread later calls `push`. This thread would mysteriously stop. Using debugger you can discover that the thread is stuck at the lock() method inside the `push` method because the lock was never unlocked by the earlier `is_empty` call. Thus an oversight in one thread led to problems much later in time in an arbitrary other thread.

A better version is shown below -

```
// An attempt at a thread-safe stack (version 3)
int count;
double values[count];
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void push(double v) {
  pthread_mutex_lock(&m);
  values[count++] = v;
  pthread_mutex_unlock(&m);
}
double pop() {
  pthread_mutex_lock(&m);
  double v = values[--count];
  pthread_mutex_unlock(&m);
  return v;
}
int is_empty() {
  pthread_mutex_lock(&m);
  int result= count == 0;
  pthread_mutex_unlock(&m);
  return result;
}
```

Version 3 is thread-safe (we have ensured mutual exclusion for all of the critical sections) however there are two points of note:

- `is_empty` is thread-safe but its result may already be out-of date i.e. the stack may no longer be empty by the time the thread gets the result!
- There is no protection against underflow (popping on an empty stack) or overflow (pushing onto an already-full stack)

The latter point can be fixed using counting semaphores.

The implementation assumes a single stack. A more general purpose version might include the mutex as part of the memory struct and use pthread_mutex_init to initialize the mutex. For example,

```
// Support for multiple stacks (each one has a mutex)
typedef struct stack {
  int count;
  pthread_mutex_t m;
  double *values;
} stack_t;

stack_t* stack_create(int capacity) {
  stack_t *result = malloc(sizeof(stack_t));
  result->count = 0;
  result->values = malloc(sizeof(double) * capacity);
  pthread_mutex_init(&result->m, NULL);
  return result;
}
void stack_destroy(stack_t *s) {
  free(s->values);
  pthread_mutex_destroy(&s->m);
  free(s);
}
// Warning no underflow or overflow checks!

void push(stack_t *s, double v) {
  pthread_mutex_lock(&s->m);
  s->values[(s->count)++] = v;
  pthread_mutex_unlock(&s->m); }

double pop(stack_t *s) {
  pthread_mutex_lock(&s->m);
  double v = s->values[--(s->count)];
  pthread_mutex_unlock(&s->m);
  return v;
}

int is_empty(stack_t *s) {
  pthread_mutex_lock(&s->m);
  int result = s->count == 0;
  pthread_mutex_unlock(&s->m);
  return result;
}
```

Example use:

```
int main() {
    stack_t *s1 = stack_create(10 /* Max capacity*/);
    stack_t *s2 = stack_create(10);
    push(s1, 3.141);
    push(s2, pop(s1));
    stack_destroy(s2);
    stack_destroy(s1);
}
```

When can I destroy the mutex?

You can only destroy an unlocked mutex

Can I copy a pthread_mutex_t to a new memory locaton?

No, copying the bytes of the mutex to a new memory location and then using the copy is *not* supported.

What would a simple implementation of a mutex look like?

A simple (but incorrect!) suggestion is shown below. The `unlock` function simply unlocks the mutex and returns. The lock function first checks to see if the lock is already locked. If it is currently locked, it will keep checking again until another thread has unlocked the mutex.

```
// Version 1 (Incorrect!)

void lock(mutex_t *m) {
  while(m->locked) { /*Locked? Nevermind - just loop and check again!*/ }

  m->locked = 1;
}
void unlock(mutex_t *m) {
  m->locked = 0;
}
```

Version 1 uses 'busy-waiting' (unnecessarily wasting CPU resources) however there is a more serious problem: We have a race-condition!

If two threads both called `lock` concurrently it is possible that both threads would read 'm_locked' as zero. Thus both threads would believe they have exclusive access to the lock and both threads will continue. Ooops!

We might attempt to reduce the CPU overhead a little by calling `pthread_yield()` inside the loop - pthread_yield suggests to the operating system that the thread does not the

CPU for a short while, so the CPU may be assigned to threads that are waiting to run. But does not fix the race-condition. We need a better implementation - can you work how to prevent the race-condition?

How can I force my threads to wait if the stack is empty or full?

Use counting semaphores! Use a counting semaphore to keep track of how many spaces remain and another semaphore to keep to track the number of items in the stack. We will call these two semaphores 'sremain' and 'sitems'. Remember `sem_wait` will wait if the semaphore's count has been decremented to zero (by another thread calling sem_post).

```
// Sketch #1

sem_t sitems;
sem_t sremain;
void stack_init(){
  sem_init(&sitems, 0, 0);
  sem_init(&sremain, 0, 10);
}


double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  ...

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  ...
```

Sketch #2 has implemented the `post` too early. Another thread waiting in push can erroneously attempt to write into a full stack (and similarly a thread waiting in the pop() is allowed to continue too early).

```
// Sketch #2 (Error!)
double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  sem_post(&sremain); // error! wakes up pushing() thread too early
  return values[--count];
}
void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  sem_post(&sitems); // error! wakes up a popping() thread too early
  values[count++] = v;
}
```

Sketch 3 implements the correct semaphore logic but can you spot the error?

```
// Sketch #3 (Error!)
double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  double v= values[--count];
  sem_post(&sremain);
  return v;
}

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  values[count++] = v;
  sem_post(&sitems);
}
```

Sketch 3 correctly enforces buffer full and buffer empty conditions using semaphores. However there is no *mutual exclusion*: Two threads can be in the *critical section* at the same time, which would corrupt the data structure (or least lead to data loss). The fix is to wrap a mutex around the critical section:

```
// Simple single stack - see above example on how to convert this into a multiple
// Also a robust POSIX implementation would check for EINTR and error codes of se

// PTHREAD_MUTEX_INITIALIZER for statics (use pthread_mutex_init() for stack/heap

pthread_mutex_t m= PTHREAD_MUTEX_INITIALIZER;
int count = 0;
double values[10];
sem_t sitems, sremain;

void init() {
  sem_init(&sitems, 0, 0);
  sem_init(&sremains, 0, 10); // 10 spaces
}

double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);

  pthread_mutex_lock(&m); // CRITICAL SECTION
  double v= values[--count];
  pthread_mutex_unlock(&m);

  sem_post(&sremain); // Hey world, there's at least one space
  return v;
}

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);

  pthread_mutex_lock(&m); // CRITICAL SECTION
  values[count++] = v;
  pthread_mutex_unlock(&m);

  sem_post(&sitems); // Hey world, there's at least one item
}
// Note a robust solution will need to check sem_wait's result for EINTR (more ab
```

What are the common Mutex Gotchas?

- Locking/unlocking the wrong mutex (due to a silly typo)
- Not unlocking a mutex (due to say an early return during an error condition)
- Resource leak (not calling `pthread_mutex_destroy`)
- Using an unitialized mutex (or using a mutex that has already been destroyed)
- Locking a mutex twice on a thread (without unlocking first)
- Deadlock and Priority Inversion (we will talk about these later)