Protecting Your Data in a Multi-Threaded App

Authored by: **John Fehr**
Updated by: **Thomas Fletcher**

The most important part of your application is most likely the data you're manipulating. If that data is corrupted, your application is useless. It's pretty straightforward to make sure your single-threaded application doesn't accidentally corrupt its data. Multi-threaded applications require some additional diligence on the part of the developer however.

The overlap problem

The problem with multi-threaded applications is that you could have one section of code modifying your data, while another section is reading and using that same data. If the write and read overlap, you could have some serious complications!

Why don't you try creating a little application and some data to be modified?

```
#include <stdio.h>
#include <pthread.h>
typedef struct {
 int a;
 int b;
 int result;
 int result2;
 int use_count;
 int use_count2;
 int max_use;
 int max_use2;
} app_data;
```

Pass your **app_data** structure to a thread that reads the data and uses the **a** and **b** variables 100 times whenever **a** is set to five. You'll use a local "**uses**" variable to count 100 of your uses. You'll also increment the application's "**use_count**" in case someone else wants to keep track of how many times your data has been used. You'll add a small **usleep** here to make sure the scheduler gives other threads a chance to run. Here's the code for your data user:

```
void *user_thread(void *data) {
 int uses=0;
 app_data *td=(app_data*)data;
 while(uses<td->max_use) {
  if (td->a==5) {
   td->result+=(td->a+td->b);
   td->use_count++;
   uses++;
  }
  usleep(1);
 }
 return 0;
}
```

Now, make some code that changes your data. You'll want to continue changing **a** and **b** until they've been used elsewhere (your **user_thread**, in this case) 100 times. Your change will be simple: you'll simply toggle your a value between 5 and 50 and fake a CPU-intensive calculation for b with a **usleep**(1000) call. This means that **a** will be changed and ~1ms later **b** will be changed. You'll have a ~1ms gap where **a** and **b** should not be used elsewhere in the application since they haven't been set to what you want yet:

```
void *changer_thread(void *data) {
 app_data *td=(app_data*)data;
 while ((td->use_count+td->use_count2)<(td->max_use+td->max_use2)) {
  if (td->a==5) {
   td->a=50;
   td->b=td->a+usleep(1000);
  } else {
   td->a=5;
   td->b=td->a+usleep(1000);
  }
  usleep(1);
 }
 return 0;
}
```

Wrap it all up in your **main**() by creating the **app_data** instance, spawning your two threads, waiting for them to finish, and printing the results. They should be:

```
Int main(int argc, char **argv) {
 pthread_t ct,ut,st;
 app_data td={5,5,0,0,0,0,100,0};
 void *retval;
 pthread_create(&ut,NULL,user_thread,&td);
 pthread_create(&ct,NULL,changer_thread,&td);
 pthread_join(ct,&retval);
 pthread_join(ut,&retval);
 printf("result should be %d, is %d\n",td.max_use*(5+5),td.result);
 return 0;
}
```

Now build this program and see how she runs:

```
$ gcc assets.c -o assets

$ ./assets
```

The result that should be ~1000, is not, in this case it is 3967 (your result might be different depending on your CPU speed).

Clearly this isn't what you want! Why did you get such a weird result?

The problem is that the **user_thread** is waiting until a is set to 5, and when it is, it adds the **a** and **b** values (which should both be 5) to the result variable. However, since the **b** variable is taking so long to get calculated, it sometimes uses the old **b** value (50) instead of the new one (5). You need to protect your data somehow so that the **user_thread** can't use the data until you've protected it!

Mutexes

Welcome to the wonderful world of the mutex (mutual exclusion). As the name implies, mutexes help you make sure that certain code or data is only accessed by one thread at a time.

Mutexes are very easy to use. Once a mutex has been created, you simply put a **pthread_mutex_lock**() in front of the code you want to protect and **pthread_mutex_unlock**() right after that code. In this case, you want to protect both the modification of the **a** and **b** variables as well as their use. First, though, you need to put in code to create and delete the mutex. Put the mutex into this handy structure:

```
typedef struct {
 int a;
 int b;
 int result;
 int result2;
 int use_count;
 int use_count2;
 int max_use;
 int max_use2;
 pthread_mutex_t mutex;
} app_data;
```

Create and destroy the mutex in your **main**() function. In this example, you'll pass NULL as the **attr** parameter to use the default attributes for the mutex:

```
int main(int argc, char **argv) {
 pthread_t ct,ut,st;
 app_data td={5,5,0,0,0,0,100,0};
 void *retval;
 pthread_mutex_init(&td.mutex,NULL);
 pthread_create(&ut,NULL,user_thread,&td);
 pthread_create(&ct,NULL,changer_thread,&td);
 pthread_join(ct,&retval);
 pthread_join(ut,&retval);
 pthread_mutex_destroy(&td.mutex);
 printf("result should be %d, is %d\n",td.max_use*(5+5),td.result);
 return 0;
}
```

Now, you put the lock/unlock functions around the code you want to protect (when you're reading or writing the **a** or **b** variables). Change the **while**() loop in the user thread to read:

```
while(uses<td->max_use) {
 pthread_mutex_lock(&td->mutex);
 if (td->a==5) {
  td->result+=(td->a+td->b);
  td->use_count++;
  uses++;
 }
 pthread_mutex_unlock(&td->mutex);
 usleep(1);
}
```

Also, change the changer thread's **while**() loop to read:

```
while ((td->use_count+td->use_count2)<(td->max_use+td->max_use2)) {
 pthread_mutex_lock(&td->mutex);
 if (td->a==5) {
  td->a=50;
  td->b=td->a+usleep(1000);
 } else {
  td->a=5;
  td->b=td->a+usleep(1000);
 }
 pthread_mutex_unlock(&td->mutex);
 usleep(1);
}
```

This time when you compile and run it you get a result that should be ~1000, and is ~1000 which is exactly the result you want. :)

What if you have one thread that's changing your assets, but more than one other thread that just wants to read your assets?

Add another thread that needs to read **a** and **b** as well:

```
void *subtracter_thread(void *data) {
 int use=0;
 app_data *td=(app_data*)data;
 while(use<td->max_use2) {
  pthread_mutex_lock(&td->mutex);
  if (td->a==50) {
   td->result2-=(td->a+td->b);
   use++;
   td->use_count2++;
  }
  pthread_mutex_unlock(&td->mutex);
  usleep(1);
 }
 return 0;
}
```

You'll also need to modify **main**() to spawn the new thread and set your **max_use2** variable in your **app_data** structure:

```
int main(int argc, char **argv) {
 pthread_t ct,ut,st;
 app_data td={5,5,0,0,0,0,100,100};
 void *retval;
 pthread_mutex_init(&td.mutex,NULL);
 pthread_create(&ut,NULL,user_thread,&td);
 pthread_create(&ct,NULL,changer_thread,&td);
 pthread_create(&st,NULL,subtracter_thread,&td);
 pthread_join(st,&retval);
 pthread_join(ct,&retval);
 pthread_join(ut,&retval);
 pthread_mutex_destroy(&td.mutex);
 printf("result should be %d, is %d\n",td.max_use*(5+5),td.result);
 printf("result2 should be %d, is %d\n",-
(td.max_use2*(50+50)),td.result2);
 return 0;
}
```

If you compile and run this, you get just what you wanted:

```
result should be 1000, is 1000

result2 should be -10000, is -10000
```

But wait. Shouldn't the **user_thread** and the **subtracter_thread** read **a** and **b** at the same time? (Remember that with mutexes anything wrapped in the lock/unlock pair can't be executed at the same time as another piece of code wrapped in the lock/unlock pair.) Neither of them change **a** or **b**, so it would be nice if you could let both of them read a and b at the same time - just not when the **changer_thread** is changing them.

Rwlocks

With **rwlocks**, another locking mechanism, you can easily implement this kind of behavior. Unlike mutexes, **rwlocks** can be locked either as read or as write. As long as they're not locked for write access, any threads can lock for read and unlock as much as they want. However, once they're locked for write access, all read locks occurring afterwards are blocked until the write lock is unlocked. Sounds perfect?

First you'll need to change your mutex to a **rwlock** in your structure:

```
typedef struct {
 int a;
 int b;
 int result;
 int result2;
 int use_count;
 int use_count2;
 int max_use;
 int max_use2;
 pthread_rwlock_t rwl;
} app_data;
```

Next, modify your **main**() so that the **rwlock** is initialized instead of a mutex. You use NULL again as the **attr** argument to get the default behavior:

```
int main(int argc, char **argv) {
 pthread_t ct,ut,st;
 app_data td={5,5,0,0,0,0,100,100};
 void *retval;
 pthread_rwlock_init(&td.rwl);
 pthread_create(&ut,NULL,user_thread,&td);
 pthread_create(&ct,NULL,changer_thread,&td);
 pthread_create(&st,NULL,subtracter_thread,&td);
 pthread_join(st,&retval);
 pthread_join(ct,&retval);
 pthread_join(ut,&retval);
 pthread_rwlock_destroy(&td.rwl);
 printf("result should be %d, is %d\n",td.max_use*(5+5),td.result);
 printf("result2 should be %d, is %d\n",-
(td.max_use2*(50+50)),td.result2);
 return 0;
}
```

Finally, you change the **pthread_mutex_lock** to **pthread_rwlock_rdlock** and the **pthread_mutex_unlock** to **pthread_rwlock_unlock** in your **user_thread** and **changer_thread**. That takes care of the read access. You also replace the **pthread_mutex_lock** to **pthread_rwlock_wrlock** and the **pthread_mutex_unlock** to **pthread_rwlock_unlock** in your **changer_thread** function, to take care of the write access. Here's your new code:

```
void *user_thread(void *data) {
 int uses=0;
 app_data *td=(app_data*)data;
 while(uses<td->max_use) {
  pthread_rwlock_rdlock(&td->rwl);
  if (td->a==5) {
   td->result+=(td->a+td->b);
```

```
    td->use_count++;
    uses++;
   }
   pthread_rwlock_unlock(&td->rwl);
   usleep(1);
  }
  return 0;
}
void *changer_thread(void *data) {
  app_data *td=(app_data*)data;
  while ((td->use_count+td->use_count2)<(td->max_use+td->max_use2)) {
   pthread_rwlock_wrlock(&td->rwl);
   if (td->a==5) {
    td->a=50;
    td->b=td->a+usleep(1000);
   } else {
    td->a=5;
    td->b=td->a+usleep(1000);
   }
   pthread_rwlock_unlock(&td->rwl);
   usleep(1);
  }
  return 0;
}
void *subtracter_thread(void *data) {
  int use=0;
  app_data *td=(app_data*)data;
  while(use<td->max_use2) {
   pthread_rwlock_rdlock(&td->rwl);
   if (td->a==50) {
    td->result2-=(td->a+td->b);
    use++;
    td->use_count2++;
   }
   pthread_rwlock_unlock(&td->rwl);
   usleep(1);
  }
  return 0;
}
```

That's it!

Now the **subtracter_thread** and the **user_thread** can read the data at the same time, but the **changer_thread** can't access the data at the same time as either the **subtracter_thread** or the **user_thread**.

One final note on the use of **rwlocks**: they are not recursive and therefore are not promotive. In other words, you can't lock your **rwlock** over and over again without unlocking first. You also can't lock your **rwlock** for read and then for write without unlocking first. This tradeoff was done to keep the **rwlock** as lightweight (in terms of size and speed) as possible.

I hope I've helped you figure out how, where, and why to use mutexes and **rwlocks**!