Assignment 4:
Threads

1 Overview

In this assignment you will become acquainted with threading basics. The threading system you'll be using is the standard *pthreads* package. pthreads is standard on UNIX systems, and because C doesn't contain any intrinsic threading primitives, pthreads is just a library of functions that interface to OS syscalls for creating and destroying threads. Because pthreads is a separate library, you'll have to compile you're code a little differently. Essentially, you just have to add the ***-lpthreads*** flag to your gcc invocation. For example, if your source file was named ***threadprog.c***, you would compile it so:

$gcc -lpthread -o threadprog threadprog.c

Additionally, you will need to include the *pthreads* header file *pthread.h* (unsurprisingly). Note about testing your code. Testing multi-threaded code is a very different proposition from testing single-threaded code. Many multi-threaded errors result from specific inter leavings, and as such, are very hard to duplicate. The code provided for this assignment will allow you to modify the configuration of threads at the command line. This makes it easy to test scenarios with a single thread, or just a few threads. Additionally, it can be helpful to insert print statements throughout the code. If you find your code hanging, this is most likely from deadlock. If you print out something like ”lock x locked” and ”lock x unlocked” before each lock and unlock call, you can more easily

determine which thread is hanging on which lock
request.

An additional note, this assignment can be done on any machine that has a C compiler and the *pthreads* library.

2 pthread_create

In *pthreads*, new threads are created by calling the function pthread_create. The actual definition of pthread_create is quite verbose:

```
int pthread_create(pthread_t *thread,
                              const pthread_attr_t *attr, void
                              *(*start_routine)(void*), void *arg);
```

Let’s start at the beginning. pthread create returns an int, which is used to determine if a thread was successfully created (0 for yes, non-zero for no). The first argument is a pointer to a *pthread_t* structure. This structure holds all kinds of useful information about a thread (mostly used by the threading internals). So, for each thread you create, you’ll need to allocate one of these guys to store all of its information. The second argument is a pointer to a *pthread_attr_t* struct. This structure contains the attributes of the thread. This has even more information about the thread. The only really interesting thing, for us at least, is that the *pthread_attr_t* struct tells the system how much memory to allocate for the thread’s stack. By default, this is usually 512K. This is actually quite large, and if you run with lots of threads you can run out of memory just from allocating thread stacks. Therefore, your code will have to initialize an attribute structure and set the stack size, like so:

```
#define SMALL_STACK 131072 //128K for stack

pthread_attr_t thread_attr;

pthread_attr_init(&thread_attr);
pthread_attr_setstacksize(&thread_attr,
                                        SMALL_STACK)
                                        ;
```

Fortunately, you can reuse an attribute structure, so you really only need to do this once.

The third argument to *pthread_create* is even more mysterious. Lets look at it again: *void* (*start routine)(void*)*. This is actually the function that the thread should start executing. This argument is an instance of what C programmers call a function pointer. The name of the argument is actually start routine, and it is a function that returns *void** and takes a *void** as an argument. The last argument to *pthread_create* is a void* which is the argument to be passed to start routine when the thread gets going. So, why a void*? Because in C, void* basically means anything. Remember you can cast pointers to void*, but you can also cast ints, chars, etc. to a void*. Which means that it is really the only way to specify a generic argument in C.

2.1 **An Example**

Here’s an example use of pthread_create. This code will create a thread that executes the function fn.

```
#include<stdio.h> #include<stdlib.h>
#include<pthread.h> #define SMALL_STACK 131072
//128K for stack

pthread_attr_t thread_attr;

void* fn(void* arg);

int main(int argc, char** argv){
   pthread_attr_init(&thread_attr);
   pthread_attr_setstacksize(&thread_attr,
                                             SMALL_STACK)
                                             ;
pthread_t th; pthread_create(&th, &thread_attr, fn, (void*)14);

void* val; pthread_join(th,&val); return 0; } void* fn(void* arg){

printf("arg = 0x%x\n", (int)arg); return NULL; }
```

For the moment, just ignore the pthread_join call at the bottom. It just makes the main thread wait until the child terminates (think of it like a waitpid for threads). This code performs all the attribute initialization voodoo, and then creates a single child thread that starts executing the fn function. As you can see, I cast the integer value 14 to a void* to pass it to fn. It looks a little weird, but that’s how you do it in *pthreads*!

Q1: To pass a single argument to a start routine simply requires some casting.

But how would you pass multiple arguments? HINT: Think about command line arguments as an example

Q2: Remove the call to pthread_join in the code above. Now recompile and run it several times. Does it always print the same thing? Why or why not?

Calling pthread_create this way can get annoying, so I created a wrapper function make thread that handles most of the gorier details. This code relies on a global pthread_attr_t structure that has been initialized. So remember to initialize the structure in main before you call make thread!

```
pthread_t* make_thread(void *(*start_fn)(void *), void*
   arg){ pthread_t* thread_info =
   malloc(sizeof(pthread_t));

   if(!pthread_create(thread_info, &thread_attr, start_fn,
      arg)){ return thread_info;


   } return NULL;


}
```

Now the mainline code no longer has to explicitly allocate pthread_t structures, the two lines that created the thread above can be replaced by: **pthread_t* th = make thread(fn, (void*)14);**

3 Mutexes and Condition Variables

It will come as no surprise that *pthreads* supports explicit mutexes and condition variables. Mutexes are fairly simple. A pthread mutex has the type pthread_mutex_t. However, they must be initialized by the pthread mutex init function. The following code creates and initializes a pthreads mutex:

```
pthread_mutex_t mut;
pthread_mutex_init(&mut,NULL);
```

The second argument to the init routine is NULL because that specifies the mutex attributes. NULL tells *pthreads* to use the defaults (which are just fine here). *pthreads* mutexes are simple souls. You lock them with the pthread mutex lock function and unlock them with the pthread mutex unlock function.

Q3: Take the code from boundedBuffer.c (available on the website) and add the appropriate lock and unlock calls to the consume and produce methods. Remember to compile your code with the -lpthread flag! The mainline for this code allows you to change the number of operations and threads at the command line. The format is: boundedBuffer <numops> <num consumers>

<num producers>. So an invocation like: boundedBuffer 100 1 4 would create 1 consumer thread, 4 producer threads, and would perform 100 con-sumes and 100 produces in total.

3.1 Condition Variables

pthreads also supports condition variables. A *pthreads* condition variable is stored in a pthread_cond_t variable that must be initialized by a call to pthread cond_init, like so:

pthread_cond_t my_cond;
pthread_cond_init(&my_cond
,NULL);

pthread condition variables can be waited on, signaled to (wake up one waiting thread), broad-cast to (wake up all waiting threads). That much is as we’ve discussed in class. However, pthread conditions are a little different. The function to wait on a condition variable is called pthread_cond_wait and its signature is:

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

The first argument is just the condition to wait on, but the second argument is a mutex. Why is there a mutex involved? The call to pthread_cond_wait puts the thread to sleep and unlocks the mutex. This is necessary to prevent deadlock (a sleeping thread can’t unlock mutexes without waking up). When the thread awakes (i.e. the call to pthread cond wait returns) it is guaranteed to be holding the mutex again. The mutex argument ensures that the programmer won’t instantly deadlock her program by forgetting to unlock the mutex before calling pthread_cond_wait. Conditions are signaled by pthread_cond_signal, which takes a pointer to a condition as its one and only argument. In *pthreads*, signaling ensures that the awakened thread will hold the mutex (essentially, signal puts the thread on the wait queue for the mutex). This leads to the following sequence of actions for using a condition:

Thread 1 Thread 2 lock mutex; lock mutex; (blocks waiting for thread1) wait on cond, mutex locks mutex; waiting on cond, mutex do some work waiting on cond, mutex signal cond

waiting on mutex unlock mutex
re-locks mutex ...

Note the order:

1. Lock before you wait (and wait on a mutex you locked)
2. Signal before you unlock

Q4: Extend the code from question 3 to handle the full and empty queue conditions. You will have to add both the if test as well as the call to pthread_cond_wait.

4 Coarse-Grained Locking

In the world of multi-threaded programming, there is often a distinction made between coarse-grained locking and fine-grained locking. With coarse-grained locking, there is usually one or two locks that protect the entire shared data-structure. Or, put another way, there are only a few mutexes that guard all of the critical sections for a given data structure. This is convenient and intuitive. Usually, with coarse-grained locks, you lock the mutex when you enter a critical section and then you unlock the mutex just before you return. For the next few questions we'll be referring to the example of a doubly-linked list. The code for the linked list is contained in the file doubleList.c. A doubly-linked list node is defined as:

```
struct ListNode{
struct ListNode* prev; struct ListNode*
next; void* data; }; typedef struct ListNode
ListNode_t;
```

And the list itself has a head and tail pointer, defined as:

```
typedef struct{
ListNode_t* head;
ListNode_t* tail;

   pthread_mutex_t* list_mutex;
}DoubleList_t;
```

The list can store any data, however at the moment the data fields are just ints. Because the list is meant to be stored in sorted order from least to greatest, there is a comparison function **int dataLT(void* data1, void* data2)**, which returns 1 if data1 is less than data2 and 0 otherwise. Similarly dataEQ(void*, void*) returns 1 if the two data fields are equal and 0 otherwise. Note that the DoubleList_t struct has the mutex for the list as one of its elements. This mutex protects the entire list. There are three operations for this list: find, insert and delete. Now lets look at the code for find:

```
ListNode_t* find(DoubleList_t* ls, void* data){
  ListNode_t* prev;
  ListNode_t* current;

  pthread_mutex_lock(ls- >list_mutex); prev
  = current = ls- >head; while(current !=
  NULL){

    if(dataEQ(current->data,
      data)){ break;

    } if(dataLT(data, current-
      >data)){ break;

    } prev = current;

    current = current->next;

  } pthread_mutex_unlock(ls-
  >list_mutex); return current;

}
```

As you can see, first find grabs the list mutex and then proceeds to traverse the list. After finding the node (or running out of list to inspect), the loop exits and then find unlocks the list mutex before returning. The code, when compiled is invoked with three arguments: the number of insert threads, the number of delete threads, and the number of find threads. All the code for creating these threads is already implemented in the mainline of doubleList.c.

Q5: Implement the insert function (void insert(DoubleList t*, void* data)) in a thread-safe fashion (use the mutex). Remember that the list is sorted, so you may have to traverse a bit to find the correct spot in the list. Also be sure to deal with empty lists, lists with one element, and inserting at the head or tail of the list.

5 Read-Write Locks

In the example of the doubly linked list, there were three operations: find, insert, and delete. But only two of them actually modified the list (insert and delete). find had to lock the list to prevent an insert or delete from interfering with list structure while the find was going on. But, because 2 concurrent finds won't be changing the list structure they won't interfere. But, because of the semantics of a pthread mutex t, only one find may execute on a particular list at once. This is inefficient (it needlessly reduces concurrency). However, have no fear, *pthreads* has a solution. Namely: the read-write lock.

The basic idea behind a read-write lock is that it allows many readers to hold the lock, but only one writer. This means that many readers can be executing their critical sections, but only one writer (and no readers) can execute a write critical section. In general, this means that operations that merely scan a data-structure need only acquire the lock in read mode; but operations that need to modify the data structure must acquire the lock in write mode.

A read-write lock introduces more complexity in handling mutexes. With a normal mutex, the lock is awarded to the threads in the order they requested it. However, a read-write lock can allow multiple threads to hold the lock in read-mode. If the lock is held by some readers, the read-write lock can allow any other readers to acquire the lock. But if a writer wants the lock, it must wait until all the readers have finished. At this point, the implementation of a read-write lock has a choice. It can either keep granting read locks to any readers that request it, or it can block them, allowing them through only after the waiting writer has finally finished. The first case is the simplest to implement, but is clearly unfair. A writer can be postponed indefinitely if readers keep coming along. The second case is more complicated but awards the lock fairly. The default, in *pthreads*, is for a fair-style read-write lock. In essence, arriving readers will queue up behind a waiting writer (if there is one).

In *pthreads*, a read-write lock is of the type pthread_rwlock_t. Read-write locks are initialized with **pthread rwlock init(pthread rwlock_t*, pthread rwlock attr*)**, if the second arg to the initialization function is NULL, the defaults will be used. To acquire the lock in read mode, you use the function: pthread_rwlock_rdlock(pthread rwlock t*). To acquire the lock in write mode, you use: pthread_rwlock_wrlock(pthread rwlock t*). To unlock, from either read or write mode, you use: pthread_rwlock_unlock(pthread rwlock_t*).

Q6: Assume there are 8 threads, T0 through T7. There is a read-write lock: lck. Assuming that the lock is awarded fairly, and that the request arrive, in order, as: T0:rdlock, T1:rdlock, T2:wrlock, T3:wrlock, T4:rdlock, T5:wrlock, T6:rdlock, T7:rdlock. In what order will the threads execute? Your answer should have the form: Tx:acquire read ; Tx:release read ; Ty:acquire write ; Ty:release write, use the semi-colons to indicate an ordering (in this example Tx acquires and then releases the lock in read mode, then Ty acquires and releases the lock in write mode.

Q7: Modify the code from question 5 so that list mutex is a pthread_rwlock_t* rather than a normal mutex. Remember to acquire the lock in the correct mode for the type of operation.

6 Fine-Grained Locking

Both the normal mutex and the read-write lock lists are examples of coarse-grained implementations. Read-write locks increased concurrency by allowing more non-conflicting operations to execute concurrently. Mutexes prevented finds from executing concurrently, read-write locks allowed this. However, read-write locks still unnecessarily inhibit concurrency. Consider the case where there is a large number of nodes in the list already. Can two inserts proceed concurrently? They can actually, if each insert is inserting into separate parts of the list. For example, if one insert is inserting right after the first node, and the second is inserting right before the last node. As long as there were at least 4 nodes in the list, the inserts shouldn't overlap at all. In order to exploit this kind of concurrency, programmers have to adopt so-called fine-grained locking techniques.

In the case of the doubly-linked list, rather than locking the entire list (the coarse-grained style), a thread should lock a region of the list that it's currently operating on. Only threads that traverse through that region will be prevented from executing concurrently. Because the locking is happening at a finer level of detail, this style of programming is called fine-grained locking. In practical terms, fine-grained locking on a list means that each list node will have a separate lock. As threads traverse the list, they'll grab the node locks as they move ahead and release the ones behind. In actuality a thread traversing the list should hold at most 2 node locks, one lock for the current node being inspected, and one for the previous node. These two locks will prevent insertions and deletions from interfering, but only if the insertions/deletions are operating over the same section of the list.

How is this implemented? Well, the first step is adding a mutex to each list node, like so:

```
struct ListNode{
struct ListNode* prev; struct ListNode* next; void* data; pthread_mutex_t*

node_mutex; }; typedef struct ListNode ListNode_t;
```

Because pthreads mutexes have to be initialized, list node allocation is best moved to a separate function, makeNode that handles both memory allocation and mutex

initialization:

```
ListNode_t* makeNode(void* data){
ListNode_t* node = malloc(sizeof(ListNode_t)); node->prev = NULL;
node->next = NULL; node->data = data;
node->node_mutex = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(node->node_mutex, NULL);
return node;

}
```

The actual DoubleList t structure still contains a read-write lock, although now most opera-tions grab it in read mode only. This lock is necessary because occasionally an insert or delete will change the head and/or the tail of the list. In those cases (removing from the front, or inserting into an empty list) the read-write lock will need to be grabbed in write mode. However, the actual node locks will also have to be grabbed. The locks will need to be acquired in a "hand-over-hand" or "crabbed" fashion. Meaning that the code should release the previous node's lock before grabbing the next node's lock. The following graphic illustrates a few iterations of hand-over-hand locking on a list: As you can see, the idea is: given that you are holding locks on two neighboring list

prev current

Step 1: Holding the

locks on prev Lock Lock

and current

prev current

Step 2: Unlock prev THEN advance the pointers

Step 3: Lock the node now pointed at by current

Figure 1: Hand-over-hand locking

nodes, first you release the least node’s lock (usually called prev in the code), then you acquire the next node’s lock. When you have that lock you’ve safely moved one slot in the list.

In code, this means that any loop which traverses the list must grab and release locks on every iteration. Consider:

```
prev = current = ls->head;

while(current != NULL){
   pthread_mutex_lock(current-
   >node_mutex);

   if(prev != current){
      pthread_mutex_unlock(prev-
      >node_mutex);
```

```
    } prev = current;

    current = current->next;

}
```

Loc
k

prev current

Lock Lock

```
/* release locks
*/ if(current !=
NULL)

    pthread_mutex_unlock(current->node_mutex);

if((prev != NULL) && (prev !=
    current))
    pthread_mutex_unlock(prev-
    >node_mutex);
```

This code implements a version of hand-over-hand locking. The lock on the current node is grabbed, then the lock on the previous node is released. Then the pointers a readjusted. When the loop iterates the next node’s lock will be grabbed. This code implements the ’locking rythym’ for hand-over-hand locking, namely: start holding current and prev, release prev, grab current->next, release current, grab current->next->next, etc. etc. Of course, this code doesn’t do anything particularly useful, it just goes from one end of the list to the other. Real operations would have more logic in the loop body.

Q8: Extend the code contained in doubleList_finegrain.c to implement find. Remember to grab both the list read-write lock as well as implement hand-over- hand locking during list traversal. Remember to unlock the list elements when you're done! Also remember that the list is sorted in increasing order.

The mainline in doubleList_finegrain.c initializes the global structures. The binary is invoked like so: **doubleList_finegrain num_inserts num_deletes num_finds**. It will create the number of threads specified. Each will perform 256 operations, some will fail (e.g. deleting a value not in the list), but that's ok.

Q9: Extend the code from Q8 to implement the size function. Size should lock the entire list, to ensure that the size is consistent (i.e. that the list doesn't change size while size is running). Size will need to traverse the list, front-to- back in order to calculate the list size.

Q10: Instead of having to traverse the list each time the size needed to be calculated, it would make sense to just have an int value in the DoubleList t structure that contained the

1

0

current size. How would this value be protected? What additional code would you have to add to insert and delete to update this value? Would this impact concurrency (assuming size is never queried)?

6.1 Lock upgrading

The fine-grained doubly-linked list has both a list-wide read-write lock as well as individual node locks. For most operations, the node locks are sufficient protection, but every now and then, an operation needs to modify the list in such a way that it changes the head and/or tail of the list itself. In those cases, such an operation needs to lock the DoubleList_t* read-write lock in write mode. However, such list-wide modifications are fairly rare. The only operations that would have to change these fields are inserts and removes that operate at the head or the tail of the list. In a list with many elements, most inserts and deletes would fall somewhere in the middle and so won't need to grab the write lock on the DoubleList_t structure. If the whole point of fine-grained locking is to increase concurrency, making every insert and delete grab the list lock in write mode seems like a bad idea.

Fortunately, in *pthreads*, read-write locks can be upgraded. A lock upgrade, in this case, means transforming a read lock into a write lock. The logic goes like this: first you grab the list read-write lock in read mode. Then you traverse the list, if you find that you have to modify the tail or head of the list, you then upgrade your lock to a write lock (thereby protecting the list as a whole). At that point you can perform the insert or delete without interfering with other threads. Then, the thread must release the upgraded lock. In *pthreads*, a read-write lock is upgraded by requesting the write lock on a read lock the thread already holds. When this write lock is granted, the thread holds both a write lock and a read lock (and consequently must call pthread rwlock unlock twice). Consider the following (simplified) insert code:

```
ListNode_t* prev;
ListNode_t* current;

ListNode_t* val = makeNode(data); int
hold_wr = 0;

pthread_rwlock_rdlock(ls->list_mutex);

while(1){

    ins_traverse(ls, data, &prev, &current);

    val->next = current;
    val->prev = prev;
    if(((prev == NULL) || (prev == current)
       || (current == NULL)) && (hold_wr == 0)){ /*need to modify the whole list
       (need to change head and/or tail) *release our locks (so we don’t
       deadlock), and then * upgrade our list lock */

       hold_wr = 1; /* Release node locks */
       pthread_mutex_unlock(prev->node_mutex);
       pthread_mutex_unlock(current->node_mutex);

       /* upgrade the whole list lock */
       pthread_rwlock_wrlock(ls->list_mutex); /*
       re-traverse the list */
```

```
        continue;

      } /* Actually inserting the node at the current location */ ···

      /* Releasing the node locks, if any are held */
      pthread_mutex_unlock(prev->node_mutex);
      pthread_mutex_unlock(current->node_mutex);

      break;

    } if(hold_wr){ pthread_rwlock_unlock(ls-
      >list_mutex);

    } pthread_rwlock_unlock(ls->list_mutex);
```

For this example, ins traverse is a function that does the hand-over-hand traversal of the list, writing the values of the prev and current pointers into the local variables prev and current. The basic logic is this: first insert grabs the list lock in read mode. Then insert traverses the list. If the list is either empty, or the node to be inserted goes at the head or tail, insert has to upgrade the lock. First a flag is set (hold wr), this is used later to ensure that the read-write lock is fully unlocked. Then the node locks held are released (to allow other threads to continue, in case we have to block waiting for the write lock). Then the list lock is acquired in write mode. The next step is tricky. Because we can't make any assumptions about how many operations were completed between our request for the write lock and when it was granted, the list may have changed out from under us. That's why the list is re-traversed (the continue directive that restarts the loop). At this point, we're the only thread executing in the list so we can change whatever we like. After the second traversal, we then perform the insert, and then release the node locks. Then we exit the loop and release the read-write lock (at least once, perhaps twice).

A more efficient implementation would do two things differently: first, rather than just simply re-traverse the list, the values of prev and current could be inspected to ensure that at least that portion of the list hadn't changed (and only re-traverse if they had). Second, the second traversal (if needed) doesn't need to do hand-over-hand locking. The write lock ensures that we're the only thread operating on the list, so we could do a much faster simple traversal.

A note about lock upgrading. Lock upgrading can actually cause deadlock (for instance, if I didn't release the node locks first). Second, although *pthreads* is supposed to support it the way I've shown it, on some systems it will hang sometimes if you make the call to pthread rwlock wrlock (for example, on my mac). In those cases, first you must release the read lock before grabbing the write lock like so:

```
/* upgrading the lock */
pthread_rwlock_unlock(ls->list_mutex);
pthread_rwlock_wrlock(ls->list_mutex);
```

On the plus side, this scheme means that you only have to call unlock once outside the loop.

Q11: Implement the **ListNode_t* head(DoubleList_t* ls)** function. head should re-move and return the head of the list. However, if the list is empty, head should return NULL. Use lock upgrading to make sure that your implementation of head only acquires a write lock if the list is non-empty.

Q12: Extra Credit Extend the insert function so that it is more efficient using the ideas described above. Make sure that the list is only re-traversed when absolutely necessary, and that when the list is traversed while the write lock is held, that the node locks are not acquired.

The code mentioned in the assignment can be found here https://goo.gl/OsAVQe