Assignment 5: M:N Threading

Due Date: Wednesday, December 10th

Overview

So far, our system has been an "N:1" system; that is, it has been *N* user-level threads running on a single kernel thread. This is suitable for so-called "cooperative task management", which is useful for separating the compute-bound and I/O-bound portions of a program. You can do quite a bit with N:1 threading, like constructing a web server or GUI application.

However, N:1 threading is not suitable for applications that want to take advantage of multi-processor parallelism. A parallel application could instead do "1:1" threading, where each thread it wants to create runs on its own kernel thread. Linux's implementation of the `pthread_create` interface does exactly this. There is a problem with this, however. Imagine we wanted to write a simple parallel version of mergesort in which each sub-array was handled by its own thread. Sorting a large array would constitute a recursive "fork bomb" that would quickly overwhelm a system with work, grinding it to a halt.

One solution would be to force the application to keep track of the number of kernel threads it creates, but this complicates the parallel algorithm with book-keeping work that is irrelevant to the problem it's trying to solve. Fortunately, there is a solution that allows the application to be simple without overwhelming the system: "M:N" threading, sometimes called "hybrid" threading, is an approach where *M* user-level threads are multiplexed over *N* kernel threads. *M* might be as high as the application wants, but the library can set an upper bound on *N* that is appropriate for the underlying hardware. If *M* is greater than *N*, the excess work will be automatically queued up on a ready list, just as it is in an N:1 system.

In this assignment, we will extend our user-level threads library into an M:N threading system. This a is long, difficult and complex task (it is the hardest assignment yet; do not wait until the eleventh hour to begin!), but we'll try to break it down and consider it one step at a time to make it manageable.

Design and Implementation Considerations

We have quite a few choices regarding how to design and implement an M:N system with our existing library. To narrow the space a little bit, we suggest some decisions below.

Scheduler Design

Have a look at Figure 1, which summarizes the operation of our current N:1 scheduler.

One of the first questions we should ask when adding kernel threads is "what we should do with them?" Figure 2 suggests a potential design that makes minimal changes to the existing scheduler.

The "`current_thread`" Problem

Have a look at your scheduler's code. The global symbol `current_thread` is used all over the place in

queue operations and thread state management.

Now have a look at Figure 2 again. Each kernel thread has its **own** notion of a "current thread". Thus `current_thread` can no longer be a global variable. But the same user-level thread might be executing on different kernel threads at different times, and it still has to execute the same code. How can we make `current_thread` a symbol that refers to different things depending on which kernel thread a user-level thread is executing on?

The answer is that `current_thread` must be a function call instead of a global variable. This function should get the current kernel thread id (using the `gettid` system call), and then query a table keyed by kernel thread id to get a pointer to the current user-level thread. This means that we must also have a function to set the current user-level thread.

To save you some time, we've supplied a file `threadmap.c` that contains the following functions:

```
void set_current_thread(struct thread_t *);
struct thread_t * get_current_thread(void);
```

If you want to make minimal changes to your existing code, you can define the following macro in `scheduler.h`:

```
#define current_thread (get_current_thread())
```

This will transparently replace any instances of `current_thread` with the `get_current_thread` function call. However, you will still have to manually replace any *assignments* you make to `current_thread` with the `set_current_thread` function call.

Thus something like:

```
current_thread->state = READY;
```

Need not change, but:

```
current_thread = next_thread;
```

Would become:

```
set_current_thread(next_thread);
```

Creating a New Kernel Thread

A fundamental implementation choice we have to make is how to add a new kernel thread. We could use `pthread_create`, since it creates a new kernel thread, but for our purposes it's more educational to get our hands dirty with Linux's kernel thread interface directly via the `clone` system call. `clone` is used under the hood by glibc to implement both `pthread_create` and `fork` by supplying it with different flags.

`clone` looks a slightly lower level version of our `thread_fork` function: it takes a target function (`fn`) to execute, and an initial argument (`arg`), but it also requires that you supply a region of memory to be used for the child stack (`child_stack`). This parameter should be a pointer to the *end* of a region of memory, just like how we created our user-level thread stacks in `thread_fork`.

The `flags` parameter is what determines much of the behavior of `clone`. Read the `clone` man page to get a sense for what these flags do. In our case, we'll want `flags` to at least consist of the following:

```
CLONE_THREAD | CLONE_VM | CLONE_SIGHAND | CLONE_FILES | CLONE_FS | CLONE_IO
```

The first three are straightforward (and have to be supplied together). Think about why we would want the last three if an application did any I/O.

In order to use `clone`, your implementation file (`scheduler.c`) must `#include <sched.h>`, but it must first `#define _GNU_SOURCE`, as follows:

```
#define _GNU_SOURCE
#include <sched.h>
...etc
```

Your Tasks

This assignment proceeds incrementally, in stages. Ideally you should use a version control system, or at least make a separate directory for each stage so that you can preserve and study your previous work as you go. Each stage should be tested independently, and all stages should be handed in at the end.

Part 0: Preparation

Adding concurrency can be a task fraught with strange bugs, so it's important to make sure you have a sound system before starting. If you have any issues with your scheduler that have been nagging at you, be sure to fix them before proceeding.

In addition, it would be wise to bring `threadmap.c` into your project and do the substitutions described above in "The "`current_thread`" Problem" before going any further. The `get_current_thread` and `set_current_thread` functions do not need multiple kernel threads to work correctly, so you should test that your existing applications that import `scheduler.h` still work properly with the substitution in place.

Part 1: Adding Concurrency

Let's start by adding a second kernel thread. Once we've done that, we can scale up to adding *N* kernel threads.

We know that we need to put a call to `clone` somewhere. Where that call goes and what function the `fn` should be is quite a large design space, so let's fix some choices. It makes sense for the `clone` call to go in `scheduler_begin`, since that is an explicit call our client must make to start the thread library anyway.

Now we need to decide what function will be the initial function of the new kernel thread. Let's call it `kernel_thread_begin`. Much like `scheduler_begin`, it should initialize data structures local to that kernel thread. Assuming the design in Figure 2 above, the only kernel-thread local data structure is its notion of `current_thread`. So, `kernel_thread_begin` should create an empty thread table entry, set its state to `RUNNING`, and then set the current thread to that thread table entry. We do not need to allocate a stack for it or set an initial function (think about why that is).

Next, `kernel_thread_begin` should enter the infinite loop described in Fig. 2, yielding forever. An astute student might have some concerns about efficiency here, but we'll delay discussion of those until later.

This is a difficult stage to test independently. Go ahead and try to run one of your existing applications at this point. You will likely find that it does not work at all; this is because we now have multiple processors making unsynchronized concurrent accesses to the ready list, which will lead to data

corruption and undefined behavior. You should preserve this failed test and hand it in.

Part 2: Atomic Operations and Spinlocks

The simplest thing to do to prevent corruption of the ready list is protect it with mutual exclusion. However, we can't use our existing blocking mutex primitive for this. The correctness of its implementation was dependent on the fact that it was running on a single kernel thread, and that it did not yield or get preempted in the middle of its operation. This is analogous to an operating system running on a single CPU with interrupts disabled.

With a second kernel thread, new situations become possible. For example, if user-level thread A is executing on kernel thread K, and another user-level thread B is executing on a second kernel thread L, then:

1. K and L could be on different CPUs, in which case A and B would execute simultaneously.
2. K's CPU could be preempted to run L, in which case A's execution could be suspended at an arbitrary point to run B. This is possible even though our user-level scheduler has no preemption, because kernel threads can still be preempted. Note that this can happen regardless of whether or not the underlying hardware is a uniprocessor or multiprocessor.

The first situation is true concurrency, while the second is pseudo-concurrency; but from the perspective our user-level threads, there is no difference. From A's perspective, B's code could run at an arbitrary time in either situation. This is analogous to an operating system running on a multiple CPUs.

Since we can now have arbitrary interleaving of instruction sequences, we will need atomic instructions in order to build mutual exclusion primitives. Atomic instructions are "indivisible". This means that a kernel thread cannot be preempted in the middle of an atomic instruction, and if two kernel threads attempt to execute an atomic instruction simultaneously, they will be forced to run in sequence.

The simplest mutual exclusion primitive using atomic instructions is a spinlock, which requires an atomic "test-and-set" instruction. The C language has no notion of atomic instructions, so in order to use a test-and-set in C code on an architecture that supports it, we would need to call assembly code directly. Furthermore, not all architectures have test-and-set. Some, like x86, have more powerful instructions like "compare-and-exchange", though this can be used to implement test-and-set.

Fortunately for you, some brave souls have put a lot of work into making a portable library for atomic operations that abstracts a lot of these fiddly details. It's called `libatomic_ops`.

Among many other things, `libatomic_ops` provides the following types, values, functions, and macros:

```
/*
 * AO_TS_t            type for a test-and-settable variable
 * AO_TS_INITIALIZER  initial value for an AO_TS_t
 * AO_TS_SET          value of type AO_TS_VAL_t indicating a held AO_TS_t
 * AO_TS_CLEAR        value of type AO_TS_VAL_t indicating a free AO_TS_t
 */

/*
 * Atomically test-and-set an AO_TS_t,
 * returning its old value as an AO_TS_VAL_t.
 */
AO_TS_VAL_t AO_test_and_set_acquire(AO_TS_t *);
```

```
/*
 * Atomically clear an AO_TS_t (this is a macro).
 */
AO_CLEAR(AO_TS_t *);
```

You are required to implement a spinlock using these features of `libatomic_ops`. Your spinlock should have the interface:

```
void spinlock_lock(AO_TS_t *);
void spinlock_unlock(AO_TS_t *);
```

`libatomic_ops` is not a standard library and must be compiled from the source code. To save you time, I've already compiled a version of the library for you to link against. To do this, `#include <atomic_ops.h>` in your implementation file, and then when compiling, use the following flag:

```
gcc -I ~kstew2/local/include ...
```

We'd like to be able to test our spinlock implementation independently of the scheduler before proceeding. To save you some time, we've provided a test program, spinlock_test.c, that does not need to be linked with the scheduler; just copy your spinlock implementation into the space indicated in the code. This program also has an example usage of `clone`.

Now that your spinlock is complete, uncomment the indicated line at the top of `threadmap.c`; this will allow the mapping data structure to be protected by your spinlock.

Aside on `malloc` and `free`

If you take a look at the man pages for `malloc` and `free`, you'll notice that they claim to be protected by their own internal mutex locks that allow them to work correctly with multi-threaded programs. I did not find this to be true. To save you time and frustration, I've provided a wrapper for `malloc` and `free` that protects all calls with the same spinlock. To use it, add the following definitions to `scheduler.h`:

```
extern void*safe_mem(int, void*);
#define malloc(arg) safe_mem(0, ((void*)(arg)))
#define free(arg) safe_mem(1, arg)
```

And the following to `scheduler.c`:

```
#undef malloc
#undef free
void * safe_mem(int op, void * arg) {
  static AO_TS_t spinlock = AO_TS_INITIALIZER;
  void * result = 0;

  spinlock_lock(&spinlock);
  if(op == 0) {
    result = malloc((size_t)arg);
  } else {
    free(arg);
  }
  spinlock_unlock(&spinlock);
  return result;
}
#define malloc(arg) safe_mem(0, ((void*)(arg)))
#define free(arg) safe_mem(1, arg)
```

NOTE: As a consequence of this workaround, you'll need to make sure that `scheduler.h` is included **last** in any file which `#includes` it. This is an unsatisfying solution but will work for our purposes.

Part 3: Protecting the Scheduler

Now that we have an effective spinlock, we can use it to protect the scheduler from concurrent accesses. Right now, client applications have several ways to call into the scheduler:

- `scheduler_begin`
- `scheduler_end`
- `yield`
- `thread_fork`
- Synchronization primitives like `mutex_lock`, etc.

We'll worry about the synchronization primitives in the next section. For now, let's focus on protecting the other routes.

`scheduler_begin` is straightforward, because any work that needs to be done to the ready list (e.g. initialization) can be done before a second kernel thread is created.

`scheduler_end` needs to be modified, because calling `is_empty` on the ready list requires exclusive access; otherwise we might observe the ready list in an inconsistent state. This means we can no longer write code like this, because it does not acquire a spinlock:

```
while(is_empty(&ready_list)) { yield(); }
```

You will need to figure out how to modify `scheduler_end` to work properly.

Now, both `yield` and `thread_fork` perform context switches. These make our lives more complicated. Obviously, these routines should acquire a spinlock before putting the current thread on the ready list. But when should that spinlock be released? If we release it before doing the switch, then another kernel thread might see the thread we had just enqueued on the ready list and try to run it. But it would run an old version, since the switch had not saved its registers and stack pointer yet! Clearly, we have to release the spinlock *after* the context switch.

The complication here is that `thread_switch` returns into a different thread than the one that called it, so how can we make sure that this next thread releases the lock?

We could modify the `thread_start` and `thread_switch` assembly routines to release the ready list lock before returning, but we could also enforce the invariant that all paths into `thread_switch` and `thread_start` must have the ready list lock acquired, and all paths out of `thread_switch` and `thread_start` release the ready list lock. This requirement should be an explicit comment in the implementation of `thread_start` and `thread_switch`.

That is, the statement after `thread_switch` and `thread_start` must be a release of the ready list lock.

This is a bit of a brain teaser, so let's break it down case by case. Consider first the case where a thread is `yield`ing to a thread that itself got switched out because it called `thread_switch`:

```
Locked:     Thread A:                Thread B:
                                        .
```

```
          // Thread A code                .
                                          .
          enter yield                     .
  x         lock ready                    .
  x         enter switch(A, B)            .
  x           save A's regs+sp            .
  x           load B's sp+regs            .
  x           return ---------------> exit switch(B, A)
  x              .                      unlock ready
                 .                    exit yield
                 .
                 .                    // Thread B code
                 .
                 .                    enter yield
  x              .                      lock ready
  x              .                      enter switch(B, A)
  x              .                        save B's regs+sp
  x              .                        load A's sp+regs
  x         exit switch(A, B) <-------- return
  x         unlock ready                    .
          exit yield                        .
                                            .
          // Thread A code                  .
                                            .
```

Note that a crucial property here is that the ready list lock is *not* held whenever a thread is executing normal code that does not manipulate the ready list (indicated by the lines `Thread A code`, `Thread B code`, etc).

What about the other case, where the thread we are switching to never got switched out, but is a new thread? In our current implementation, `thread_start` jumps straight into the new thread's initial function. This is what that would look like:

```
Locked:     Thread A:                Thread B:

          // Thread A code

          enter thread_fork
    x       lock ready
    x       enter start(A, B)
    x         save A's regs+sp
    x         jmp to B -----------> enter initial_function
    x            .                    // Thread B code
    x            .                    enter yield
    x            .                      lock ready

                     Deadlock!
```

Clearly we need some way for a newly started thread to release the ready list lock. The answer is to jump to a wrapper function that does just that, and then calls the initial function instead of jumping to it directly. This looks as follows:

```
Locked:     Thread A:                  Thread B:

          // Thread A code

          enter thread_fork
```

```
x          lock ready
x          enter start(A, B)
x            save A's regs+sp
x            jmp to thread_wrap ---> enter thread_wrap
x               .                      unlock ready
                .                      enter initial_function
                .                        // Thread B code
                .
                .                        enter yield
x               .                          lock ready
x               .                          enter switch(B, A)
x               .                            save B's regs+sp
x               .                            load A's sp+regs
x            exit start(A, B) <------------- return
x            unlock ready                       .
           exit thread_fork                     .
                                                .
           // Thread A code                     .
                                                .
           enter yield                          .
x            lock ready                         .
x            enter switch(A, B)                 .
x             save A's regs+sp                  .
x             load B's sp+regs                  .
x             return ------------------->  exit switch(B, A)
x               .                          unlock ready
                .                        exit yield
                .
                .                        // Thread B code
                .
                .                      exit initial_function
                .                      current_thread->state = DONE
                .
                .                      enter yield
x               .                        lock ready
x               .                        enter switch(B, A)
x               .                          save B's regs+sp
x               .                          load A's sp+regs
x            exit switch(A, B) <---------- return
x            unlock ready
           exit yield

           // Thread A code
```

Implement `thread_wrap` and modify your `thread_start` assembly routine to jump to `thread_wrap` instead of to the new thread's initial function. Notice that this also means we can eliminate `thread_finish`, since it is no longer necessary.

At this point your implementation should be complete enough to write an interesting test application (provided it does not attempt to use any blocking primitives, which are discussed in the next section).

Part 4: Multiprocessor Blocking Primitives

Earlier, we noted that using blocking primitives (e.g. `mutex_lock`) on a multiprocessor requires using a spinlock to protect the waiting queue of the blocking primitive.

Clearly a thread about to block needs to acquire that spinlock to add itself to the waiting queue. But much

like with the ready list, we face the question of when to release that spinlock. If a thread releases the spinlock before blocking, then another thread could pull it off the waiting queue and add it to the ready list, even though its stack pointer and registers had not been saved by a context switch.

The solution is to release the waiting queue's spinlock only after the ready list lock has been acquired -- this will prevent a thread from enqueuing a blocking thread on the ready list until after its state has been saved.

However, we cannot acquire the ready list lock twice, e.g. by locking the ready list and then calling `yield`, as this will self-deadlock. So we can add a new function:

```
void block(AO_TS_t * spinlock)
```

Which is exactly the same as `yield`, except that it releases the `spinlock` parameter after acquiring the ready list lock.

Thus, an execution sequence for a pair of concurrent `mutex_lock` and `mutex_unlock` operations might proceed correctly as follows:

```
Thread A:                  Thread B:

mutex_lock(m)
  lock m->s
  find that m is held
  add A to m->waiting      mutex_unlock(m)
  A->state=BLOCKED           lock m->s
  block(m->s)                  spin ...
    lock ready                 spin ...
    unlock m->s                spin ... acquire!
    switch(A, C)             find that m->waiting is non-empty
                             grab A from m->waiting
Thread C:                    lock ready
                               spin ...
    unlock ready               spin ... acquire!
                             A->state=READY
                             m->held = A
                             add A to ready list
                             unlock ready
                             unlock m->s
```

Your task is to implement `block`, and then upgrade at least your mutex primitive to work on a multiprocessor. Test your implementation thoroughly.

Part 5: Scalability

The design we have suggested has many inherent scalability limits. Begin by parameterizing `scheduler_begin` to allow the client program to request *N* kernel threads instead of just creating a second one, and then study the performance of the system as *N* increases.

Then, complete one of the following tasks:

- Compose a written report (at least two pages, in your own words) analysing the scalability and performance of this system. Identify some of the scalability bottlenecks and propose some design alternatives that you believe would improve performance. Be sure to provide sufficient detail about why you think your alternatives would work.

- Implement an optimization that you will believe will improve the system's performance, and conduct experiments to prove that it actually does. Produce a brief written report of your results.

Testing

You should produce appropriate tests for each stage whenever possible, and maintain a version of the library for each stage, so that the repeatability of the test is preserved and can remain isolated from further changes. This will help us grade your progress and separate early successes from later shortcomings.

Successful tests will usually hammer on the code, repeating an operation hundreds or thousands of times in order to encourage all possible scheduling orders. This will help reveal race conditions and potential deadlocks. Your test should have some known, verifiable result, and you should check for result corruption.

Testing scalability limits in part 5 is difficult without writing an application that can take advantage of parallelism. I suggest writing an application that can operate on disjoint portions of the problem simultaneously (e.g. mergesort). If you decide you need to turn to the internet to find and adapt a parallel algorithm, **you must provide a citation of the code you are adapting**.

A note: parallel algorithms require communication of results between threads. Since we don't have `join`, you'll have to figure out a way for one thread to wait for another's result. This can be done with semaphores, condition variables, or some form of Dekker's algorithm (see the spinlock test for an example of the latter).

What To Hand In

If you are working with a partner, please prepare just one submission for the both of you. Make sure your (and your partner's) names are included in each file you submit.

You should submit:

1. All code, including tests, for each stage that you complete.
2. A written report, including:
    1. A description of what you did and how you tested it.
    2. The results of your efforts from Part 5, above.

Please submit your code files *as-is*; do not copy them into a Word document or PDF.
Plain text is also preferred for your write-up.
You may wrap your files in an archive such as a .tar file.

Email your submission to the TA at kstew2 at cs.pdx.edu on or before the due date. The subject line should be "CS533 Assignment 5".

Need Help?

If you have any questions or concerns, or want clarification, feel free to contact the TA by coming to

office hours or sending an email.

You may also send an email to the class mailing list, but please pose discussion questions only; **do not** email any code to the class mailing list! (It is okay to send code directly to the TA, however).