

Lab Thread

Learn how xv6 switch threads and create a user-space thread library. Use the popular thread library pthread.

Environment Preparation

Run the following commands:

```
$ git fetch
$ git checkout thread
$ make clean
```

Lab Tasks

uthread

In this exercise you will design the context switch mechanism for a user-level threading system, and then implement it. To get you started, your xv6 has two files `user/uthread.c` and `user/uthread_switch.S`, and a rule in the `Makefile` to build a uthread program. `uthread.c` contains most of a user-level threading package, and code for three simple test threads. The threading package is missing some of the code to create a thread and to switch between threads.

Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, `make grade` should say that your solution passes the `uthread` test.

Once you've finished, you should see the following output when you run `uthread` on xv6 (the three threads might start in a different order):

```
$ make qemu
...
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
...
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
```

```
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

This output comes from the three test threads, each of which has a loop that prints a line and then yields the CPU to the other threads.

At this point, however, with no context switch code, you'll see no output.

You will need to add code to `thread_create()` and `thread_schedule()` in `user/uthread.c`, and `thread_switch` in `user/uthread_switch.S`. One goal is ensure that when `thread_schedule()` runs a given thread for the first time, the thread executes the function passed to `thread_create()`, on its own stack. Another goal is to ensure that `thread_switch` saves the registers of the thread being switched away from, restores the registers of the thread being switched to, and returns to the point in the latter thread's instructions where it last left off. You will have to decide where to save/restore registers; modifying `struct thread` to hold registers is a good plan. You'll need to add a call to `thread_switch` in `thread_schedule`; you can pass whatever arguments you need to `thread_switch`, but the intent is to switch from thread `t` to `next_thread`.

Some hints:

- `thread_switch` needs to save/restore only the callee-save registers. Why?
- You can see the assembly code for `uthread` in `user/uthread.asm`, which may be handy for debugging.
- To test your code it might be helpful to single step through your `thread_switch` using `riscv64-linux-gnu-gdb`. You can get started in this way:

```
(gdb) file user/_uthread
Reading symbols from user/_uthread...
(gdb) b uthread.c:60
```

This sets a breakpoint at line 60 of `uthread.c`. The breakpoint may (or may not) be triggered before you even run `uthread`. How could that happen?

Once your `xv6` shell runs, type `"uthread"`, and `gdb` will break at line 60. If you hit the breakpoint from another process, keep going until you hit the breakpoint in the `uthread` process. Now you can type commands like the following to inspect the state of `uthread`:

```
(gdb) p/x *next_thread
```

With `"x"`, you can examine the content of a memory location:

```
(gdb) x/x next_thread->stack
```

You can skip to the start of `thread_switch` thus:

```
(gdb) b thread_switch
(gdb) c
```

You can single step assembly instructions using:

```
(gdb) si
```

On-line documentation for gdb is [here](#).

pthread

In this assignment you will explore parallel programming with threads and locks using a hash table. You should do this assignment on a real Linux or MacOS computer (not xv6, not qemu) that has multiple cores. Most recent laptops have multicore processors.

This assignment uses the UNIX `pthread` threading library. You can find information about it from the manual page, with `man pthreads`, and you can look on the web, for example [here](#), [here](#), and [here](#).

The file `notxv6/ph.c` contains a simple hash table that is correct if used from a single thread, but incorrect when used from multiple threads. In your main xv6 directory (perhaps `~/xv6-labs-2021`), type this:

```
$ make ph
$ ./ph 1
```

Note that to build `ph` the Makefile uses your OS's gcc, not the 6.S081 tools. The argument to `ph` specifies the number of threads that execute put and get operations on the the hash table. After running for a little while, `ph 1` will produce output similar to this:

```
100000 puts, 3.991 seconds, 25056 puts/second
0: 0 keys missing
100000 gets, 3.981 seconds, 25118 gets/second
```

The numbers you see may differ from this sample output by a factor of two or more, depending on how fast your computer is, whether it has multiple cores, and whether it's busy doing other things.

`ph` runs two benchmarks. First it adds lots of keys to the hash table by calling `put()`, and prints the achieved rate in puts per second. Then it fetches keys from the hash table with `get()`. It prints the number keys that should have been in the hash table as a result of the puts but are missing (zero in this case), and it prints the number of gets per second it achieved.

You can tell `ph` to use its hash table from multiple threads at the same time by giving it an argument greater than one. Try `ph 2`:

```
$ ./ph 2
100000 puts, 1.885 seconds, 53044 puts/second
1: 16579 keys missing
0: 16579 keys missing
200000 gets, 4.322 seconds, 46274 gets/second
```

The first line of this `ph 2` output indicates that when two threads concurrently add entries to the hash table, they achieve a total rate of 53,044 inserts per second. That's about twice the rate of the single thread from running `ph 1`. That's an excellent "parallel speedup" of about 2x, as much as one could possibly hope for (i.e. twice as many cores yielding twice as much work per unit time).

However, the two lines saying `16579 keys missing` indicate that a large number of keys that should have been in the hash table are not there. That is, the puts were supposed to add those keys to the hash table, but something went wrong. Have a look at `notxv6/ph.c`, particularly at `put()` and `insert()`.

Why are there missing keys with 2 threads, but not with 1 thread? Identify a sequence of events with 2 threads that can lead to a key being missing. Submit your sequence with a short explanation in `answers-thread.txt`

To avoid this sequence of events, insert lock and unlock statements in `put` and `get` in `notxv6/ph.c` so that the number of keys missing is always 0 with two threads. The relevant pthread calls are:

```
pthread_mutex_t lock;           // declare a lock
pthread_mutex_init(&lock, NULL); // initialize the lock
pthread_mutex_lock(&lock);       // acquire lock
pthread_mutex_unlock(&lock);     // release lock
```

You're done when `make grade` says that your code passes the `ph_safe` test, which requires zero missing keys with two threads. It's OK at this point to fail the `ph_fast` test.

Don't forget to call `pthread_mutex_init()`. Test your code first with 1 thread, then test it with 2 threads. Is it correct (i.e. have you eliminated missing keys)? Does the two-threaded version achieve parallel speedup (i.e. more total work per unit time) relative to the single-threaded version?

There are situations where concurrent `put()`s have no overlap in the memory they read or write in the hash table, and thus don't need a lock to protect against each other. Can you change `ph.c` to take advantage of such situations to obtain parallel speedup for some `put()`s? **Hint: how about a lock per hash bucket?**

Modify your code so that some `put` operations run in parallel while maintaining correctness. You're done when `make grade` says your code passes both the `ph_safe` and `ph_fast` tests. The `ph_fast` test requires that two threads yield at least 1.25 times as many puts/second as one thread.

barrier

In this assignment you'll implement a [barrier](#): a point in an application at which all participating threads must wait until all other participating threads reach that point too. You'll use pthread condition variables, which are a sequence coordination technique similar to xv6's sleep and wakeup.

You should do this assignment on a real computer (not xv6, not qemu).

The file `notxv6/barrier.c` contains a broken barrier.

```
$ make barrier
$ ./barrier 2
barrier: notxv6/barrier.c:42: thread: Assertion `i == t' failed.
```

The 2 specifies the number of threads that synchronize on the barrier (`nthread` in `barrier.c`). Each thread executes a loop. In each loop iteration a thread calls `barrier()` and then sleeps for a random number of microseconds. The assert triggers, because one thread leaves the barrier before the other thread has reached the barrier. The desired behavior is that each thread blocks in `barrier()` until all `nthreads` of them have called `barrier()`.

Your goal is to achieve the desired barrier behavior. In addition to the lock primitives that you have seen in the `ph` assignment, you will need the following new pthread primitives; look [here](#) and [here](#) for details.

```
pthread_cond_wait(&cond, &mutex); // go to sleep on cond, releasing lock mutex,
acquiring upon wake up
pthread_cond_broadcast(&cond);    // wake up every thread sleeping on cond
```

Make sure your solution passes `make grade`'s `barrier` test.

`pthread_cond_wait` releases the `mutex` when called, and re-acquires the `mutex` before returning.

We have given you `barrier_init()`. Your job is to implement `barrier()` so that the panic doesn't occur. We've defined `struct barrier` for you; its fields are for your use.

There are two issues that complicate your task:

- You have to deal with a succession of barrier calls, each of which we'll call a round. `bstate.round` records the current round. You should increment `bstate.round` each time all threads have reached the barrier.
- You have to handle the case in which one thread races around the loop before the others have exited the barrier. In particular, you are re-using the `bstate.nthread` variable from one round to the next. Make sure that a thread that leaves the barrier and races around the loop doesn't increase `bstate.nthread` while a previous round is still using it.

Test your code with one, two, and more than two threads.

Submit the Lab

Report

Put your report about this lab in a new file `report.txt`. You can write down the errors you met when doing the lab, or some techniques you found that helps you to finish the lab better. You can use both **English** and **Chinese**.

Don't forget to `git add` and `git commit` the file.

Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

Check

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make tarball`

Submit

Run `make tarball`, you should see a new file in the root directory called `lab-thread-handin.tar.gz`. Upload this file to the corresponding section in course [canvas page](#).