CSE 451, Introduction to Operating Systems, Autumn 2013

Project 2 - User-Level Threads

Project dates

- Out: Friday October 18
- Parts 1-3 ("project2a"): due Sunday November 3 electronically @ 11:59pm.
- Parts 4-6 ("project2b"): due Sunday November 17 electronically @ 11:59pm.

Outline

For this project, we assume that you will be working in the same groups as for Project 1.

Tasks:

1. Implement a user-level thread manager.
2. Add a set of common synchronization primitives to your thread package.
3. Implement a simple synchronization problem.
4. Add preemption to your thread package.
5. Implement a multithreaded web server to test your thread package.
6. Analyze your design and report test results.

Assignment goals

- To gain understanding of how thread packages are implemented, including data structures used, the interface provided, and common problems they face.
- To make use of common synchronization primitives such as locks and condition variables.
- To understand some of the problems with user-level threads.
- To understand the concept of overlapping I/O and computation.
- To practice discussion and analysis of your designs.

Background

In the beginning (well, the relative beginning), there was UNIX. UNIX supported multiprogramming; that is, there could be multiple independent processes each running with a single thread of control.

As new applications were developed, programmers increasingly desired multiple threads of control within a single process so they could all access common data. For example, databases might be able to process several queries at the same time, all using the same data. Sometimes, they used multiple processes to get this effect, but it was difficult to share data between processes (which is typically viewed as advantage, since it provides isolation, but here it was a problem).

What they wanted was multithreading support: the ability to run several threads of control within a single address space, all able to access the same memory (because they all have the same mapping function). The programmers realized that they could implement this entirely in the user-level, without modifying the kernel at all, if they were clever.

As you'll discover in the last part of this assignment, there were some problems with this approach, which motivated kernel developers to include thread support in the kernel itself (and motivated researchers to do it better; see Scheduler Activations).

`simplethreads` setup

Where to Work

You will be using the `simplethreads` package for this assignment. This project does not require modifying the Linux kernel, so you don't need to use the CSE 451 VM. **We recommend using either `forkbomb` or the CSE home Linux VM for this assignment.** There are three main computing requirements for using `simplethreads`: a Linux environment, a 32-bit or 64-bit x86 architecture, and an up-to-date `autotools` package (the CSE 451 VM does not meet this last requirement, you would have to install autoconf and automake every time you used it). Other machines such as the CSE lab machines or your own Linux computer will probably work for this assignment as well, but we have tested thoroughly on `forkbomb` and the CSE home Linux VM and will use these machines for grading (so you should at least test on one of these before submitting).

Please **do not use attu**.

Create Your Copy of the Project Files

The latest `simplethreads` distribution is available through the same repository used with project 0. Since you and your partner will most likely want to collaborate using Git, your best option is to add the simplethreads code a new shared repository, different than the one you used in project 1. To do this, **one of you** should follow these steps. The first few steps mimic those of setting up a shared repository for the kernel source.

1. Log into forkbomb and go to your project directory

   ```
   forkbomb$ cd /projects/instr/13au/cse451/X
   ```

   Where X is your group's letter.
2. Initialize an empty shared "master" repository called *451repo*:

   ```
   forkbomb$ git init --shared=group --bare 451repo
   Initialized empty shared Git repository in
   /projects/instr/13au/cse451/X/451repo/
   ```

   BEWARE the warning given at the end of Repository setup about changing files here directly
3. Now, on your local machine clone this master repository and populate it with the skeleton code:

   ```
   local$ git clone \
     me@attu.cs.washington.edu:/projects/instr/13au/cse451/X/451repo
   Cloning into '451repo'...
   warning: You appear to have cloned an empty repository.

   local$ cd 451repo

   # Add the official starter files.
   local$ git remote add course \
     username@attu.cs.washington.edu:/cse/courses/cse451/13au/451repo

   local$ git fetch course
   # Enter your password

   # Merge in the code
   local$ git merge course/master

   # Now push your locally added changes into the shared repository
   local$ git push origin master
   ```

Once one partner has followed these steps, the other should clone the master repository to acquire the code:

```
partner-pc$ git clone \
  partner@attu.cs.washington.edu:/projects/instr/13au/cse451/X/451repo
Cloning into '451repo'...
# and so forth
```

The project 2 files are in the `simplethreads` directory. At this point both of you should be all set to share code by pushing/pulling from the Master!

File organization

`simplethreads` contains a lot of files, but you will only need to implement a few skeleton files and do not need to understand the details of the rest of the provided code. Pay attention to these directories and files:

- `lib/`: The `simplethreads` thread library itself.
- `lib/sthread_user.c`: Your part 1 and part 2 implementations go here.
- `lib/sthread_ctx.{c,h}`: Support for creating new stacks and switching. between them.
- `lib/sthread_switch*.h`: Assembly functions for saving registers and switching stacks.
- `lib/sthread_queue.h`: A simple queue that you may find useful. You may wish to use your queue code from project 0 instead.
- `lib/sthread_preempt.h`: Support for generating timer interrupts and controlling them (see part 5).
- `include/`: Contains `sthread.h`, the public API to the library (the functions available for apps using the library).
- `test/`: Test programs for the library.
- `web/`: The webserver for part 4.

Configure the Build

Like many Unix programs, `simplethreads` uses a `configure` script to determine parameters of the machine needed for compilation. (In fact, you'll find many UNIX packages follow exactly the same build steps as `simplethreads`). In the `simplethreads` directory, run `./configure` to generate an appropriately configured Makefile.

Build

After running `./configure`, you can immediately compile and link the distributed code. In the top directory, type `make`. (You can also build an individual part, such as the library, by running `make` in a subdirectory. Or, if you just want to compile one file, run `make myfile.o` from within the directory containing `myfile.c`.) Note that, as you make changes to the source, it is only necessary to repeat the last step (`make`).

Test

Run `make check`. The test programs in `test/` will be run, and messages will be printed to indicate whether or not each test had the correct result. Success at this point is simply having the test programs execute at all -- they will all fail, because you haven't yet completed the implementation of `simplethreads`.

To Remember For Later

A useful feature of `simplethreads` is that the underlying thread implementation can be configured to either use native, kernel-provided threads (pthreads) or the user-level threads that you'll implement (sthreads). Because both provide the same interface (once you've completed parts 1 and 2, anyway), applications that use `simplethreads` don't even know which version they're using. The default is sthreads, but to select pthreads instead, run `configure --with-pthreads`, then `make clean`.

Summary

In summary, the steps to get started are:

1. Acquire the source code from the `451repo` repository.
2. `cd simplethreads`
3. `./configure [--with-pthreads]`
4. `make`
5. `make check` (Run the programs in `test/`. At this point, they should run, but not get correct results unless you specified `--with-pthreads`).

To Add a Source File

If you add a new source file, do the following:

1. Edit the `Makefile.am` in the directory containing the new file, adding it to the `_SOURCES` for the library/executable the file is a part of. E.g., to add a new file in the `lib/` directory that will become part of the sthread library, add it to the `libsthread_la_SOURCES` line.
2. The edited `Makefile.am` has to be processed by the `automake` tool before it can be used by `make`. From

within the top-level directory, run `autoreconf`, which will run `automake` and regenerate the `Makefile.in` file. (**NOTE:** `autoreconf` is part of the GNU Autotools build system, which is present on pretty much all Linux systems. To work with `simplethreads`, your autotools must be reasonably up-to-date. Run `autoconf --version` and check that your autoconf is at least version 2.68; if it is not, then you'll need to update your autotools. Alternatively, you can probably copy your simplethreads code to `forkbomb` or another machine with the proper autotools (this should be easy if you have a code repository set up), run `autoreconf` on that machine, then continue working with the updated files on another machine. See the TAs if you have questions about this.)

3. Also from the top-level directory, run `./configure`.
4. Your file is now added; run `make` as usual to build it.

To add a new test program

1. Edit `test/Makefile.am`. Add your program to the list `bin_PROGRAMS`. Create a new variable `prog_SOURCES` following the examples of the programs already there. For example, if the new program is named `test-silly`, add: `test_silly_SOURCES = test-silly.c` *other sources here*
2. Follow steps 2-4 above.

To add a new arbitrary file

1. Edit the `Makefile.am` in the directory containing your file. Add your file to the list `EXTRA_DIST` (see top-level `Makefile.am` for an example).
2. Follow steps 2-4 above.

The assignment

Part 1: Implement thread scheduling

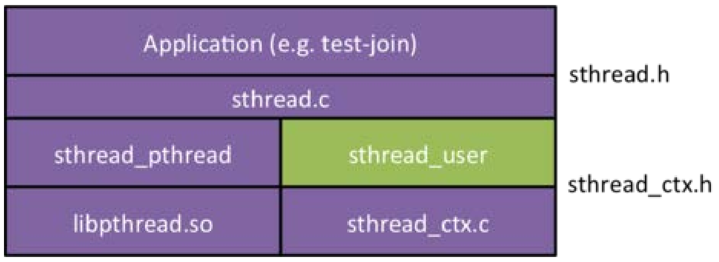
For part 1, we give you:

1. An implementation of a simple queue (`sthread_queue.h`).
2. A context-switch function that, given a new stack pointer, will switch contexts (`sthread_ctx.h`).
3. A new-stack function that will create and initialize a new stack such that it is ready to run.
4. Skeleton routines for a user-level thread system (`sthread_user.c`).

It is your job to complete the thread system, implementing:

1. Data structures to represent threads (`struct _sthread` in `sthread_user.c`).
2. A routine to initialize your data structures (`sthread_user_init()`).
3. A thread creation routine (`sthread_user_create()`).
4. A thread destruction routine (`sthread_user_exit()`).
5. A mechanism for a thread to voluntarily yield, allowing another thread to run (`sthread_user_yield()`).
6. A mechanism for a (single) thread to wait for another to finish (`sthread_user_join()`).
7. A simple non-preemptive thread scheduler.

The routines in `sthread_ctx` do all of the stack manipulation, register storing, and nasty stuff like that. You should be able to use them based on the semantics explained in that file. Rather than focusing on the details of that low level manipulation, this assignment focuses on managing the thread contexts. Viewed as a layered system, you need to implement the green `sthread_user` box below:



At the top layer, applications use the `sthread` package (through the API defined in `sthread.h`). Immediately below that, `sthread.c` performs its function by using either the routines in `sthread_pthread.c` (a thin veneer over `pthreads`) or your implementation in `sthread_user.c`. (The choice depends on a switch you gave (or didn't give) when you ran `configure` before building the `sthread` library.) Your `sthread_user.c`, in turn, builds on the `sthread_ctx` functions (as described in `sthread_ctx.h`).

Applications (the top-layer) may not use any routines except those listed in `sthread.h`. They must not know how the threads are implemented; they simply request that threads be created (after initializing the library), and maybe request yields/exits. For example, they have no access to `sthread_queue`. Nor do they keep lists of running threads around; that is the job of the green box.

Similarly, your green box - `sthread_user.c` - should not need to know how `sthread_ctx` is implemented. Do not attempt to modify the `sthread_ctx_t` directly; just use the routines declared in `sthread_ctx.h`.

Recommended Procedure

1. Figure out what each function that you need to implement (in `sthread_user.c`) does. We have discussed all of these general functions in class. Look at some of the test programs to see usage examples.
2. Examine the supporting routines we've provided (primarily in `sthread_queue.h` and `sthread_ctx.h`).
3. Design your threading algorithm: When are threads put on the ready queue? When are threads removed? Where is `sthread_switch` called?
4. Figure out how to start a new thread and what to do about the initial thread (the one that calls `sthread_user_init`).
5. Discuss any ideas or questions that you have for your design with the TAs, or post on the discussion board if you have any general questions.
6. Implement it.
7. Test it. The test programs provided are not adequate; for full confidence in your implementation, you'll need to create some of your own.

Hints

- All of the sthreads functions that you need to implement have pthreads counterparts; if you are ever unsure of how your function should behave, check the man page for the corresponding pthreads function and imitate its behavior.
- `sthread_create` should not immediately run the new thread.
- Use the provided routines in `sthread_ctx.h`. You don't need to write any assembly or try to directly manipulate registers for this assignment, nor is an understanding of the exact layout of the stack required (though it may help in debugging).
- If the routine passed to `sthread_create()` finishes (returns), you need to make sure the thread's resources are cleaned up (i.e. `sthread_exit()` is (eventually) called).
- The start routine passed to `sthread_new_ctx` does not take any arguments (unlike the routine that your `sthread_user_create` is passed). So you can't create a new stack directly with the user-supplied routine; you need to supply a routine that takes no arguments but somehow winds up invoking the user's routine with the user's argument.
- You should free any resources allocated by your threading library when a thread exits (whether it exits explicitly, by calling `sthread_exit()`, or implicitly, because the start routine returns) and there can be no more threads attempting to `join` with it. However, you should not attempt to free the stack of a running thread (note: to free a stack, use `shread_free_ctx()`). This requires a few tricks.
- Dealing with the initial thread is tricky. You need to make sure that an `sthread_t` struct is created for it at some point, so it can be scheduled like the other threads. However, it wasn't created by your `sthread_user_create()` function. Remember that, while a thread is running, the state stored in the `sthread_t` is mostly garbage (though it is probably important that you have an sthread_t, so you've got some place to put the state when you want to stop the thread).
- While globals are bad in general, there will be places in this assignment where they are necessary.
- When debugging with `gdb(1)`, you may see messages like `[New Thread 1024 (LWP 18771)]`. These messages refer to kernel threads.

Part 2: Implement mutexes and condition variables

For part 2, you'll use the thread system you wrote in part 1, extending it to provide support for mutexes and condition variables. Skeleton functions are again provided in `lib/sthread_user.c`. You need to:

- Design data structures for mutexes (`struct _sthread_mutex`) and condition variables (`struct _sthread_cond`).
- Implement the mutex operations (`sthread_user_mutex_*()`).
- Implement the condition variable operations (`sthread_user_cond_*()`).

So far, your threads are non-preemptive, which gives you atomic critical sections. For this part, get your synchronization primitives working with this non-preemptive assumption and start thinking about where the critical sections are (add comments if you find it useful). When you add preemption, you will have to go back and add appropriate protection to critical sections. The details about this are in part 4.

Hints

- Figure out how to block a thread, making it wait on some queue. How do you get the calling thread? How do you switch out of it? How will you wind up back in it? (Some of this is similar to what you did for `join` in Part 1.)
- Mutexes do not immediately switch threads when unlocked.
- When you finish this part, all tests in the test directory should work except for test-preempt.c

Part 3: Implement a simple synchronization problem

There are several famous synchronization problems in computer science. For part 3, your job is to implement a "food services" problem, an instance of the better-known multiple-producer, multiple-consumer problem. There are N cooks (each a separate thread) that constantly produce burgers. We'll assume for debugging purposes that each burger has a unique id assigned to it at the time it is created. The cooks place their burgers on a **stack**. Each time a cook produces a burger, she prints a message containing the id of the burger just cooked. There are M hungry students (each a separate thread) that constantly grab burgers from the stack and eat them. Each time a student eats a burger, she prints a message containing the id of the burger she just ate. Ensure, for health and sanity reasons, that a given burger is consumed at most once by at most one student.

Note that you are implementing an application now. That means the only interface to the thread system that you should use is that described by `sthread.h` (as distributed in the `tar.gz`). Do not use functions internal to sthreads directly from your solution to this problem.

Place your solution in a new file called `test-burgers.c` **in the test directory** (see the "To Add a Source File" section above for directions to build it). Make the program take 3 command-line parameters: N, M, and the total number of burgers to produce. For example, `test-burgers 3 2 100` should simulate 3 cooks, 2 students, and 100 total burgers.

Hints

- Test the problem with various values of M and N.
- Think about critical points in your code that must contain sthread_yield() calls.
- To test your threads' functionality, don't forget that you can recompile the sthreads library to use POSIX threads (use the `--with-pthreads` argument as shown above). This means you can start working on this part *before* finishing Parts 1 and 2.
- Note that your shell may limit the number of processes and/or threads that you can create; run the `ulimit -a` command to see these limits. If you need to raise these limits, check the ulimit man page.

Part 4: Implement a multithreaded web server

Every web server has the following basic algorithm:

1. Web server starts and listens (see `listen(2)`) for incoming connections.
2. A client (i.e. web browser) opens a connection.
3. The server accepts (see `accept(2)`) the connection.
4. The client sends an `http` request.
5. The server services the request by:
   a. Parsing the request.
   b. Finding the requested file.
   c. Reading the file.
   d. Sending a set of `http` headers, followed by the contents of the file, to the client.
   e. Closing the connection.
6. The client displays the file to the user.

The `sioux` webserver in the `web/` directory implements the server side of the above algorithm.

For this part of the project you will make sioux into a multithreaded web server. It must use a thread pool approach; it should not create a new thread for each request. Instead, incoming requests should be distributed to a pool of waiting threads (this is to eliminate thread creation costs from your experimental data). Make sure your threads are properly and efficiently synchronized. Use the routines you implemented in part 2.

Unfortunately, because sioux does not use asynchronous IO, it will be very difficult to obtain good performance using the user-level sthreads. In some cases, it may be difficult to even get correct behavior at all times (e.g., if no new requests are sent, existing requests may not be serviced at all). Therefore, for this part of the assignment, you should configure `simplethreads` to use pthreads, rather than sthreads (use `./configure --with-pthreads`; don't call pthreads functions directly).

Don't forget that the only interface to the thread system that you should use is that described by `sthread.h`. Do not use functions internal to sthreads directly from sioux.

You should accept a command-line flag indicating how many threads to use.

In testing, you may encounter "Address already in use" errors. TCP connections require a delay before a given port can be reused, so simply waiting a minute or two should be sufficient.

You can use a normal web browser to send requests to your web server. However, note that some web browsers always request a "`favicon.ico`" file along with their page request, which may cause 404 errors to arise from your server. The easiest solution to this problem is to just add a `favicon.ico` file to your server's content directory.

At this point you should perform a sanity-check by doing the "Run the Web Benchmark and Report the Results" portion of Part 6, below.

Part 5: Add Preemption

In this part, you will add preemption to your threads system. This part of the project is not a lot of work (it represents perhaps only 10% of the code you will write), but it's a little tricky. We've made it the last part of the project (aside from the "Report" part) so you won't get stuck on it and fail to get the multi-threaded web server running. (The multi-threaded web server does not require preemption.)

We provide you with:

- A facility to generate timer interrupts
- Primitives to turn interrupts on and off
- Synchronization primitives `atomic_test_and_set` and `atomic_clear`

See `sthread_preempt.h` for more details on the functions you are given.

It is your job to:

1. Add code that will run every time you get a timer interrupt.
2. Add synchronization to your implementation of threads, mutexes, and condition variables.

To initialize the preemption system, you must make a call to `sthread_preemption_init`, which takes two arguments: a function to run on every interrupt, and a period in microseconds, specifying how often to generate the timer interrupts. For example, `sthread_preemption_init(func,50)` will call func every 50 microseconds. You should add a call to `sthread_preemption_init` as the last line in your `sthread_user_init()` function. Make it so that the thread scheduler switches to a different thread on each interrupt.

The hard part will be figuring out where to add synchronization to your thread management routines. Think about what would happen if you were interrupted at various points in your code. For example, we don't want to be preempted when we're inside yield() and in the middle of switching to a different thread. The way to ensure that this never happens is by disabling interrupts. You are provided with a function `splx(int splval)`, where `splx(HIGH)` disables interrupts and `splx(LOW)` enables them. Here is an example:

```
int oldvalue = splx(HIGH);
// disable interrupts; put old
// interrupt state (disabled /
// enabled) into oldvalue

// {critical_section};

splx(oldvalue);
// restore interrupts to
// original state
```

You are also provided with two other synchronization primitives: `atomic_test_and_set` and `atomic_clear.` See `sthread_preempt.h` for a usage example. These will be useful for less important critical sections, such as ready queue manipulation. Note that it is also necessary to synchronize your mutex and condition variable code. For example, in `sthread_user_mutex_lock`, you will want to use `atomic_test_and_set` for grabbing a lock.

A note: timer interrupts are set up so that they only fire when you are executing *your* code (either the user application or the thread library). Interrupts occuring inside printf or other system functions will be dropped. Also, interrupts in the critical assembly code inside `sthread_switch` are also dropped. This simplifies your task greatly.

"Stress testing" is important. It's common to get this part of the project "90% correct." You may forget to disable interrupts in just one or two situations. This may not show up with cursory testing. That's what "race conditions" are all about -- they're devilishly difficult to track down because they're timing-dependent. But eventually they *will* show up!

IMPORTANT! The code that we have provided to support preemption works correctly *only* on Linux machines with an x86 architecture! Do not attempt this portion of the assignment on a Mac or a non-x86 machine! Also, at least one student in the past has reported weirdness in the preemption code when running on an AMD processor, so to be safe you may wish to stick to Intel processors (`cat /proc/cpuinfo` to check).

Hints

- Start by initializing preemption to run a function which only prints something out to the screen, and make it run every second (pass 1000000 to `sthread_preemption_init`). This will check that you indeed can receive interrupts.
- If you disable interrupts in one place, make sure you reenable them on all code paths. This is a common cause if your application suddenly freezes and stops receiving interrupts. Be particularly careful with your scheduler (your yield function). You will probably want to disable interrupts for the whole time you are inside yield, and reenable them after you've completed `sthread_switch`. However, `sthread_switch` could actually return to *two* different places: to the next line after a call to `sthread_switch` and to your user thread starter function when you switch to a new user thread for the first time! Be sure to reenable interrupts in both places.
- You should never execute any application code with interrupts disabled.
- To aid in debugging, you can press ctrl-\ (ctrl-backslash) at any time while your application is running to see the total number of interrupts generated so far.
- There is a test provided for you, test-preempt.c, for testing your library with preemption. It has no yield calls and relies solely on interrupts to make progress. Also, go back to the other tests and check that they still work as expected. In particular, you should see a bit of randomness introduced to your part 3 solution because of preemption.
- It is up to you to set the interrupt period. Good values are around 10-50 microseconds (but you should test your code with shorter and longer periods too, of course).
- To compare programs with and without preemption, it is useful to turn preemption off, which you can do by either commenting out your call to `sthread_preemption_init`, or doing a `./configure --without-preemption` and then `make clean` and `make` from your shell.

Part 6: Report

Include the following in a report to be turned in electronically on the due date. This should be at most 4 pages long.

Design discussion

Briefly describe the design of your user-level threads, synchronization primitives, and webserver. Mention any design decisions you had to make, and why you made them.

Functionality

Does your implementation work? Which parts work and which don't? For the ones that don't work, how do you think you would fix them?

Run the web benchmark and report the results

Consider these two web server configurations:

1. pthreads, 1 thread in thread pool
2. pthreads, 5 threads in thread pool

Using the web benchmark described below, measure the throughput and response time for these two web servers using 1, 5, and 25 clients. Each client should fetch the file `emacs.html` (found under `/cse451/projects/` on `forkbomb`), so you should copy this file into your `web/docs` directory (which is the web server root directory) and include it in urls to pass to the webclient benchmark. For best results, make sure you run `sioux` and `webclient` on *different hosts* (for example, run `sioux` on `forkbomb` and webclient on `attu` - this is the only place where using `attu` is ok!).

Report these results in a few easy to understand graphs. *Explain your results*. How does the multithreaded aspect of the web server affect throughput? What about response time? How would web server performance be affected if you were to use your user threads? (You're welcome to try it and measure it, although this is not required.)

Conclusions

Discuss conclusions you have reached from this project. What did you learn? What do you wish you had done differently?

Using the web benchmark

The WebStone web benchmark tool is located on `forkbomb` at `/cse451/projects/bin/webclient` and in the `simplethreads` package under the `web/` subdirectory; this executable usually works when copied to any Linux machine. `webclient` measures the throughput and latency of a webserver under varying loads. It simulates a number of active clients, all sending requests in parallel (it does this by forking several times). Each client requests a set of files, possibly looping through that set multiple times. When the test is complete, each client outputs the connection delay, response time, bytes transfered, and throughput it observed (depending on the server, the clients may all observe very similar results, or the data may vary widely). The tool takes the following arguments:

- `-n CLIENTS` specify the number of parallel clients to simulate.
- `-l LOOPS` specify the number of times each client should fetch the files.
- `-w SERVER` specify the hostname of the webserver (when `sioux` is started, it prints a URL including the hostname).
- `-p PORT` specify the port number (when `sioux` is started, it prints a port number).
- `-u URLLIST` specify a file containin a list of URLs to fetch, one per line, each followed by a weight (e.g. 1).

All of the above parameters are required. The `URLLIST` file should contain one relative URL (just the file name) per line followed by a space and then a number (the number is the weight representing how often to request the file relative to other files in the list - most likely, 1 is the value you want).

For example, to test a webserver running on `forkbomb`, with two simulated clients each fetching the `index.html` file twice, one would run this command (from any machine):

```
./webclient -w forkbomb.cs.washington.edu -p 12703 -l 2 -n 2 -u ./urls
```

Where the file `urls` would contain the following single line:

```
/index.html 1
```

Submission instructions

Parts 1, 2, 3

First, make sure that you have followed the steps described above to add a new source file / test program / arbitrary file to your code. If you do not follow these instructions and run the `autoreconf` command, then these files may not be included in your submission.

Now, in your top-level simplethreads directory, run the following commands:

```
make distclean
./configure
make dist
```

This will produce a file named `simplethreads-2.01.tar.gz`. Run `tar tzvf simplethreads-2.01.tar.gz` to check that all of the `simplethreads` files, *and any new files you have added*, are included in the archive. You can also extract the archived files via `tar xzvf simplethreads-2.01.tar.gz`. Upload this archive to the Catalyst DropBox for the class.

Parts 4, 5, and 6

Follow the same instructions as for parts 1-3. Turn in a final version of your code, including any scripts or other files you used in part 6 as well as your report. Please ensure that your report is either contained in the archive that you submit or is uploaded as a separate file to the DropBox. Submit your archive and report to the Catalyst DropBox for the class.