Kernel Threads in `xv6`

Please **read this entire assignment**, before you start working on the code. This is an especially challenging assignment. You really do not want to wait until the night/day before it is due to start on it. Jeepers, just reading all of … … … this assignment is itself enough of a challenge (for a non-engineering student). There may be madness in my method, but there is also method in my madness.

This lab is November 22nd by midnight. Submit a single gzipped `tar` file to **TEACH**. Submitting your solutions before November 22nd will earn you a 10% bonus. If you don't remember how to create a gzipped `tar` file, you need to learn before you submit this assignment. If your submission is not a gzipped `tar` file, I will not grade your assignment.

There are several parts to this assignment. Most are fairly small. Just follow this document like it is a script or recipe and work through all the parts. I recommend you use `#ifdef` sections in your code to make it easier to track where you make changes to the `xv6` source code. I'm sure you have plans to refactor your code after the assignment is due, putting in comments, using mnemonic macros, and using conditional compilation blocks to separate new and old code. Instead, perform that before the due date.

Initially, I'd wanted you to build this in your lottery-scheduler base code. After writing it myself in the lottery-scheduler based code, I don't think that adds much to the assignment and complicates an already challenging assignment. See Part 0 below.

This assignment is done entirely in the `xv6` environment.

This programming project is worth 470 points!!!

Part 0 – Clone the `xv6-kthreads` directory

I have created a directory from which you can easily begin your coding journey. The directory is called `xv6-kthreads`. It is in the same location where I keep the rest of my `xv6` code. You can clone a copy of that directory using the following command:

```
~chaneyr/Classes/cs444/xv6/xv6-clone.bash -s xv6-kthreads -d xv6-kthreads
```

That will create for you a directory called `xv6-kthreads` which contains all the beginning code for your kernel threads `xv6` adventure. Starting with this directory is not a requirement, but it already includes a lot of code to help you get started.

I have marked many/most of the places in the code where you'll need to make some changes or add code. They are marked with `KTHREADS` in `#ifdef/#endif` blocks. Currently within those blocks are `#error` preprocessor directives. If you enable the `KTHREADS` in the `Makefile`, those will cause the compiler to emit the error message and stop. You'll need to remove the `#error` directives as you develop the code. The beginning code will compile and run. The beginning code includes most of the things we've added during class, such as the `shutdown` command and the `kdebug` command.

Part 1 – Add some additional programs – (5 points)

You need to add a couple new user commands/programs into you `xv6-kthreads` directory for some testing. If you are starting with a clone of my `xv6-kthreads` directory, this entire part is already done for you. If not, copy the following programs from my `xv6-kthreads` directory:

`memalgn.c` – this is just a small program you can run to make sure that the steps taken to assure a block of memory is page aligned is correct.
`thtst1.c` – the simplest of simple thread tests. It creates a thread, makes sure that memory between the new thread and the main thread is shared, joins with the thread, and exits. This program uses assert statements to validate values in variables for correctness. See Figure 3, page 13 and Figure 4, page 13.
`thtst2.c` – another small simple test of the thread functions, but this one can run with multiple threads, as given from the command line. I've run it with 10 threads. I don't know how much higher it can go under the current limits of `xv6`. It will simply compute away for a few seconds (usually 30 seconds with 10 threads and 13 seconds with 3 threads, when run in `qemu` emulating 4 CPUs). See Figure 9, page 13.
`thtst3.c` – this is a much smaller version of the multi-threaded matrix multiplication that we did in class. Simplifications include: this is a fixed work allocation (we don't have locks we can use between threads, yet), it generates the same data instead of reading files, it only works on matrices of up to 30x30 (which is the default), it is limited to at most 10 threads, and the output always goes into the same named file (`op.txt`). If you really want to try a larger matrix or more threads, go ahead, but the process memory limit of `xv6` (as we have it configured) will limit you. Interestingly, most of the run time for this program is spent writing the output file. See Figure 8.
`thtst4.c` – this program tests 2 things. 1) is the parent for a thread correctly established. 2) can a thread join with a thread it did not create. There is nothing complex about this code, it simply creates a few threads, runs for a bit, and joins with them. See Figure 7.
`thtst5.c` – this program tests 1 thing. 1) you should receive a graceful error if you try to join with a non-existent thread. See Figure 6.
`thtst6.c` – this program tests 1 thing. 1) if you pass a non-page aligned pointer to memory to `kthread_create()`, it should gracefully reject the new thread creation. See Figure 5.

Table 1: Some simple testing programs.

Now modify your `Makefile` so that the new programs are compiled and built into the file system when `xv6` starts. As mentioned above, if you cloned my `xv6-kthreads` directory, this is already done. Add the new programs into the `UPROGS` variable in the `Makefile`.

Though these are the current set of test/interesting programs for this project. It is very possible that additional test programs are developed. I will make them available in the `xv6-kthreads` directory.

Part 2 – Add some data members to the `proc` structure – (5 points)

You need to add some data members to the `struct proc` data structure that is in the file `proc.h`. The data members you add are:

`oncpu` – we will use this to show on CPUs a thread/process is currently running. Later in the assignment, we will switch from having a single CPU for `qemu` to having multiple CPUs. The `qemu` software will allow up to 8 CPUs, but I've always just used 4. Switching from a single CPU to multiple CPUs is just a small change to the `Makefile`.	
`isThread` – this is use to indicate that a *thingie*[1] taking up a slot in the `ptable` structure is a thread of another process, not a process itself. Since we want the threads scheduling to be handled by the kernel, we are going to make them entries in the `ptable`, just as regular processes are. When determining how an entry in the `ptable` should be handled (by something like the `wait()` call), this will be very helpful.	
`isParent` – indicates that this process has (or had) threads within it. Like the `isThread` data member, this is useful when managing processes.	
`tid` – if this *thingie* is a kernel thread (which is what you are building), this this hold the unique (to the process) identifier for the thread. The `tid` will be unique for a process, but may not be unique for across all processes (just like `PThreads`).	
`nextTid` – the main thread keeps track of what is the next unique `tid` to give to a newly created thread. This value should start at 1 and be incremented each time a new thread for that main thread is created. The first thread created for a process will have a `tid` of 1.	
`threadExitValue` – if this *thingie* is a thread, this is will hold the exit value from that thread (set in `kthread_exit/benny_thread_exit`). Though I had marvelous plans for this, I never took advantage of it.	

Table 2: New data members for `proc` structure.

Be sure to initialize all these data mambers in the `allocproc()` function.

If you cloned my `xv6-kthreads` directory, the date members are already in the structure.

[1] *Thingie* is considered by a few to be a non-technical term. In this case, it is either a process or a kernel thread.

Part 3 – Copy the `benny_thread` code – (5 points)

Copy the `benny_thread.h` and `benny_thread.c` files from my `xv6-kthreads` directory into your development directory. Modify the `Makefile` to build the `benny_thread.o` file from the `benny_thread.c` file. Modify the `Makefile` so that user programs/commands are linked with the `benny_thread.o` object module.

The modification of the `Makefile` to build the `benny_thread.c` module only requires you add `benny_thread.o` into the `ULIB` macro. Doing this will also cause the user programs/commands to link with the `benny_thread.o` file. Not all of them actually need it, but they are fine with it.

Guess what? If you cloned my `xv6-kthreads` directory, this is already done. Otherwise, copy it from my `xv6-kthreads` directory.

What are the `benny_thread` functions? An excellent question. The `benny_thread` functions are just a few **user level** functions that make managing the kernel threads a bit easier. The `benny_thread` functions are all user space functions. All, except `benny_thread_tid()`, make kernel space calls to similarly named functions that run in privileged mode. The `benny_thread` functions are just wrappers that help with the kernel threads; in the same way that `malloc()` is a user level function that makes memory management easier than having to make a bunch of calls to `sbrk()` to handle the heap.

`benny_thread_create`	Return type: **`int`** Parameters: **`vbt`:** the address of a `benny_thread_t` data type (`typedef`-ed in `benny_thread.h`). **`func`**: A function pointer. The function is a `void` return and accepts a single parameter, a `void *`. This should make you think of the `start_routine` parameter to the function `pthread_create`. **`arg_ptr`**: a `void *` pointer that represents a pointer sized value for the single parameter that is passed to the function `func` (from above). This should make you think of `arg` parameter to the function `pthread_create`. This function is where the memory allocated from the heap that was used as the stack for the thread is allocated.
`benny_thread_join`	Return type: **`int`**

	Parameters: **tid**: a benny_thread_t pointer. The tid is passed on to the kthread_exit function. This function is where the memory allocated from the heap that was used as the stack for the thread is deallocated. Any thread can join with another thread, except that no thread can join with the main thread (thread 0 for a process). It is important to note that this is a blocking function.
benny_thread_exit	Return type: **int** Parameters: **tid**: a benny_thread_t pointer. The tid is passed on to the kthread_exit function. This is called by a thread when is it complete and ready to terminate. It is to die for.
benny_thread_tid	Return type: **int** Parameters: **tid**: a benny_thread_t pointer that the benny_thread_tid function will cast back to the benny_thread_s structure and return the tid of the given thread. The benny_thread_t is considered opaque to functions outside of the benny_thread.c module. This is when a benny_thread wants to know "Who is that?" of another thread.

Table 3: The benny_thread functions.

Any code that wants to use the benny_thread functions must include the benny_thread.h file (as thtst[123].c do). While it is possible to directly call the kthread_ functions from user mode (as the benny_thread functions do), it is simpler to use the wrappers. Simplicity is a benny-fit of the functions.

Part 4 – Stub out the `kthread_` functions – (5 points)

In the proc.c file, stub out the following kthread_ functions:

kthread_create	Return type: **int** Parameters:

	`func`: A function pointer. The function is a `void` return and accepts a single parameter, a `void *`. This should make you think of the `start_routine` parameter to the function `pthread_create`. **`arg_ptr`**: a `void *` pointer that represents a pointer sized value for the single parameter that is passed to the function `func` (from above). This should make you think of `arg` parameter to the function `pthread_create`. **`tstack`**: a `void *` pointer to the space that this newly created thread will use as its stack. The tstack pointer must point to a page aligned lump of memory. This is where an actually kernel thread is created. It gets spot in the `ptable`, has a kernel stack allocated, has its state set to `RUNNABLE`, and so much more…
`kthread_join`	Return type: **`int`** Parameters: **`tid`**: an integer that represents the thread identifier (aka `tid`) for the thread within this process for which the calling thread will join. This is where most of the cleanup for a thread is done. It is important to note that this is a blocking function. If the passed thread is not yet complete (called `kthread_exit`), this function will not return until the thread has terminated.
`kthread_exit`	Return type: **`void`** Parameters: **`exitValue`**: an integer that represents the exit status of the thread. A value of 0 is successful termination of the thread. Any value other than zero indicated a non-successful termination of the thread. This is where a thread declares is it done and terminates. However, even though it is done, it must remain in the `ptable` (as a `ZOMBIE`) until another thread joins with it. The brains of the thread are removed and it turns in a zombie. 

Table 4: The `kthread` functions

If you are starting with a clone of my `xv6-kthreads` directory, these functions are already stubbed out in `proc.c`.

You have soooo much already done and it has been so easy. Sorry partner, but the easy part is about to change. It is time to release your inner wild kernel hacker.

Part 5 – Implement `kthread_` Functions – (300 points)

This is where it starts to be challenging and just downright fun. While the following instructions are extensive, they are not intended to represent everything necessary in the `kthread_` functions.

`kthread_create()`

This is going to be a lot like the `fork()` function. In fact, starting with a copy of `fork()` is not a bad idea at all. I'm going to assume you did this and use the variable names from the `fork()` function below.

The first thing the `kthread_create` function must do is to check to make sure the `tstack` pointer is page aligned. Remember the `memalgn` program (mentioned in Table 1), look at the source code for it. If it is not page aligned, return -1.

Next, call `allocproc()`.

The next thing `fork()` does is make a copy of the page table (the call to `copyuvm`). But, that is for a process (where each process has its own page table). This is a thread, so all you need to do is have the `pgdir` member of np point to the `pgdir` of `curproc`. This is one of the most important things about a thread, all threads in a process share a single page table.

The size of the thread (the `sz` member) is the same size as `curproc` (since they are the same process). This is actually an issue, but we address in a future lab.

Make sure you set the `isThread` member for `np`.

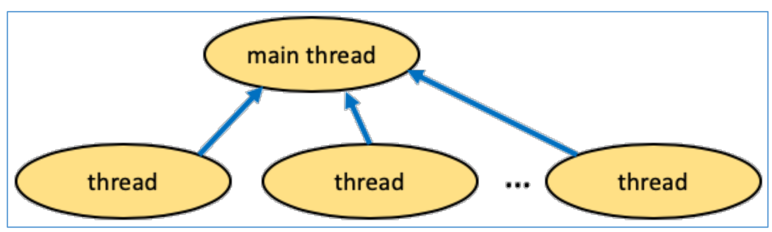


Figure 1: Relationship between newly created threads and the main thread (thread 0). All threads created with `kthread_create()`, for a given process, have the same parent, the main thread.

There is not a parent-child relationship between threads, but we need to keep track of which threads are related to a process. So, set the parent of the new thread `np` to `curproc`, **UNLESS** `curproc` is itself a thread. If `curproc` is a thread, then the parent of `np` is the parent of `curproc`. Use this opportunity to also set the `isParent` member in the process (main thread) and increment the `threadCount` of the parent process (the once running `main()`). When inspecting the parent data member in the proc structure, all threads should point back to the main thread (as shown in Figure 1). You do not want to have a multi-level hierarchy of relationships.

The new thread, `np`, has the copy of `tf` (trap frame) as `curproc`. Do this (yes the * characters are required, that's how it is copied):

```
*np->tf = *curproc->tf;
```

The eax register for the new thread is cleared just as it is for a new process, though frankly it does not really apply here.

The eip register (in the tf structure) represents the instruction pointer for the new thread. You should assign func to it. This is not touched in the fork() function.

We need to assign the esp (extended stack pointer) data member (from the tf structure in np) to the tstack that was passed as a parameter. However, the tstack variable is the opposite end of the stack (as the stack grows low address to high address). So, try something like this:

```
np->tf->esp = ((int) tstack) + PGSIZE;
```

We are going with the assumption that the size of the stack for each thread is a single page (PGSIZE). It would be nice to be able to create threads with different stack size, but that is beyond this assignment.

We need to push a value on the stack. Specifically, we need to push the arg_ptr variable value onto the stack for the thread function to pick up. This will take 2 statements. See if these make sense.

```
np->tf->esp -= sizeof(int);
*((int *) (np->tf->esp)) = (int) arg_ptr;
```

The first line decrements the value for the beginning of the stack pointer. The thread function will pull the value from the esp register beginning at this location. The second line copies the value from within the arg_ptr variable on the stack. We have to do some magic C casts to make sure everything is happy.

We have 2 more manipulations of the stack pointer. We need to push the value of the new thread's tid onto the thread stack. This is the return value from kthread_create that the thread function can pick up. Assign the new tid value to a local variable (called maybe something wild, like tid). Assign the same value to the tid data member for np. The new tid should come from the parent tid data member (np->parent). Make sure you increment the parent's nextTid value as well (as a post increment). Now

```
np->tf->esp -= sizeof(int);
*((int *) (np->tf->esp)) = tid;
```

You are getting to be a real pro at this. We are almost there for kthread_create.

In fork() there is a little loop where the file descriptors from curproc are duplicated (using filedup()) into np. You need to do this for the thread.

You need to idup() the curproc->pwd into np->pwd. As fork() does. This is immediately after the loop above.

As fork() does, do the safestrcpy(), acquire the ptable lock, set the state of the thread to RUNNABLE, and release the lock.

Return the `tid`.

I liberally sprinkled a number of if statements that use the `debugState` variable in the function. Some simply check to see if `debugState` is non-zero while others check to see if `debugState` is greater than 1 or 2. This allows me to change the number of diagnostic messages from the kernel at runtime. I found use of the `kdebug` command very useful.

Whew… done with `kthread_create()`.

`kthread_join()`

The `kthread_join()` function is a bit easier than `kthread_create()`, but it has a few subtle requirements. The `kthread_join()` has a lot in common with the `wait()` function. In fact, you will need to make a change to the `wait()` function as part of developing the `kthread_join()` function. Anything you do in the `kthread_join()` function that makes a change to the `ptable` process array will require a lock on that structure. Make sure that you also unlock it before returning.

There is a lot more room for creativity in development of this function. I'm going to walk through how I did it, but there are a lot of opportunities for variation.

If the `curproc` is a parent (of threads), but has a current thread count of 0, just release and return -1.

If the given `tid` is 0, just release and return -1. As I have designed my code, a thread cannot join with thread 0 (the main thread). This is not a requirement, just how I did my code. In the future, I'd like to change this.

This is a bit tricky here, I want to make sure that I decrement the correct `threadCount` data member, so I check to see if `curproc->isThread` is `TRUE`. If so, I set `curproc = curproc->parent`. Remember back in the `kthread_create()` function, this statement "There is not a parent-child relationship"? We made sure that all threads `parent` pointer went back to the main thread (thread 0, see Figure 1). This is where that becomes important. The `thtst4` program will test this.

Acquire a lock on the `ptable`.

If you look at the code for `wait()`, you'll see that it has an infinite for loop around a loop through the `ptable` at this point. I'm going to take a slightly different approach on this (let me know if you find a flaw in it).

Look through the `ptable` with a `for` loop (I do it in the same way as `wait()` does, where p is the current entry in the table). If `p->parent != curproc || p->tid != tid`, just continue in the loop.

If we get through the test `p->parent != curproc || p->tid != tid`, we know we've found the right `parent` and `tid` combination. I go into the following loop:

```
while (p->state != ZOMBIE) {
      release(&ptable.lock);
      yield();
      acquire(&ptable.lock)
}
```

Basically, if the thread is not marked as a zombie, release the `ptable` lock, and yield the processor. Remember when we talked about yielding the time slice for a process? Once the thread is scheduled again, acquire the `ptable` lock and repeat the test. The thread will be marked as a zombie in the `kthread_exit()` routine.

Once we get through the above loop, we know we have the right thread and that the thread has exited. We need to clean it up.

Decrement the `threadCount` of `curproc` (which is the main thread).

Do the stuff done in `wait()`, **but DO NOT call `freevm()`**. This is the stuff in the if block where `p->state == ZOMBIE`. The main thread owns the virtual memory for the entire process. Only when the main thread exits is the virtual memory freed.

Break out of the loop.

If you found the right `parent` and `tid` combination, return 0. Otherwise, return -1.

Make sure you release the `ptable` lock before the return.

Okay, let's go make that change to the `wait()` function.

Remember that the kernel is periodically looking for zombie processes. As we write these as kernel threads, they look a lot like processes to the kernel. We do not want the kernel doing cleanup on the threads until we are ready for it.

So, there is an `if` block in the `wait()` function that looks like this:

```
if (p->parent != curproc)
      continue;
```

We must change it condition to look like this:

```
if(p->parent != curproc || p->isThread == TRUE)
```

If the *thingie* the kernel (or other process) is inspecting is a thread, just move along. It does not make sense to call `wait()` on a thread, you must join with it.

2 functions down (`kthread_create()` and `kthread_join()`); 1 more to go.

`kthread_exit()`

The code for `kthread_exit()` is pretty straight forward.

Get the `curproc` (as `exit()` does). If `curproc` data member `isThread` is `TRUE`, then:

Close all the open files (see `exit()`).

Cleanup the cwd (exacly as exit() does it with begin_op and end_op). In previous development, I have experienced some issues when cleaning up the cwd, but have not with this development.

Set killed to FALSE, the thread data member threadExitvalue to the passed exitValue, oncpu to -1, and state to ZOMBIE.

Now, acquire the ptable lock and call sched() (not scheduler()). Follow the call to sched() with a panic("kthread_exit") call. Obviously, it should never get to the call to panic.

That's it for kthread_exit().

Time to give a big woohoo!

Part 6 – Refresh the ps/cps() code – (50 points)

We've added a couple data members to the proc structure (Remember Part 2? Seems like ages ago.). We want those to show up when we run the ps command.

You need to add the following to the output from the cps() function: oncpu, isParent, isThread, and threadCount. The header information should be: "cpu", "is par", "is thrd", and "thrd #".

You only show the oncpu value when it is >= 0. If you've followed the instructions from above, this should be easy. Only a RUNNING process/thread should have a value of oncpu that is greater than or equal to 0. Do not show the negative value for oncpu.

```
ps
pid     ppid    name    state   size    cpu     is par  is thrd thrd #
1       1       init    sleep   12288           0       0       0
2       1       sh      sleep   16384           0       0       0
6       5       thtst2  run     45056   1       0       1       0
5       1       thtst2  runble  77824           1       0       6
7       5       thtst2  runble  45056           0       1       0
8       5       thtst2  runble  45056           0       1       0
9       5       thtst2  run     77824   3       0       1       0
10      5       thtst2  runble  77824           0       1       0
11      5       thtst2  run     77824   2       0       1       0
12      2       ps      run     12288   0       0       0       0
$ ps
pid     ppid    name    state   size    cpu     is par  is thrd thrd #
1       1       init    sleep   12288           0       0       0
2       1       sh      sleep   16384           0       0       0
6       5       thtst2  run     45056   3       0       1       0
5       1       thtst2  runble  77824           1       0       6
7       5       thtst2  run     45056   1       0       1       0
8       5       thtst2  runble  45056           0       1       0
9       5       thtst2  run     77824   0       0       1       0
10      5       thtst2  runble  77824           0       1       0
11      5       thtst2  runble  77824           0       1       0
13      2       ps      run     12288   2       0       0       0
```

Figure 2: The ps command with new columns.

For isParent (which means it is a main thread) and isThread, just show 0 for false and 1 for true.

The threadCount value should be zero except for a main thread in a process.

See Figure 2 for how this should look.

The easy way to see this data is to run “`thtst2 6 &`” (in the background) and then run `ps` a couple of time to see the output. If you see a “`zombie!`” after doing this, don’t worry, it is only a flaw in their shell.

Part 7 – Update and Validate on 4 CPUs – (100 points)

Change the `CPUS` macro in the `Makefile` to 4. It is fine for you to do your development with a single cpu in `qemu`. However, you code will be tested with the value of the `CPUS` macro in the `Makefile` set to 4. I highly recommend you test your code this way. In the `Makefile`, look around line 251. The points for this part are not awarded to simply changing the `Makefile`, but for all the test commands working correctly with 4 CPUs.

How It will be Graded

When we grade, we will first run the 6 test commands with a single CPU. We will run 1 test program, then exit `qemu` before running the next test program. I know this stuff is hard and we really don’t have the 6 months to develop a full test suite, that’s why we will exit `qemu` between tests.

We can run `qemu` using a single CPU with the following command:

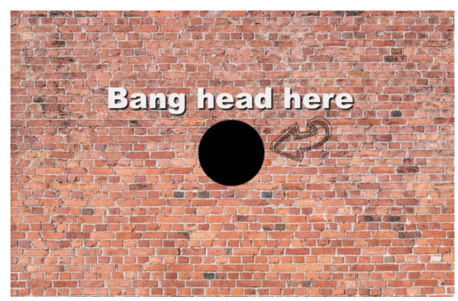
```
make nox CPUS=1
```

We can run `qemu` using 4 CPUs with the following command:

```
make nox CPUS=4
```

The macro on the command line will override the setting within the `Makefile`.

Other Tips



I have to be honest, there are a couple places where I struggled when writing this code. One of the biggest is where I called `kfree()` in the `kthread_join()` function (you can put it in `kthread_exit()`). If you look in the code for `kfree()`, you’ll see that it does a `memset()` on the page of kernel memory to be freed. Doing the `memset()` is an excellent idea, for the reason mentioned in the comment. However, somewhere I must have some boundary conditions messed up. When I run `thtst2` and do the `memset()`, I will usually get “unexpected trap 14 from …”. If `#ifdef` out (or comment out) the call to `memset()` in `kfree()`, all is fine. Many web searches later, it would seem that I have overrun a buffer and stomped all over an instruction pointer. But, I cannot see where I’ve gone astray. I would really like to know what I am doing wrong. ☹ If you find out what my error is, have pity on me and let me know.

Submit to `TEACH`

When you are done with the `Lab5`, submit your code to `TEACH`. Remember how we used the command "`make teach`" to produce a `tar` and gzipped file that you can submit into `TEACH`? Do that and be done.

Final note

The labs in this course are intended to give you basic skills. In later labs, we will ***assume*** that you have mastered the skills introduced in earlier labs. **If you don't understand, ask questions.**

Example Output from Test Programs

```
$ thtst1
global before: 10
i before      : 0xF0F0F0F
rez           : 0x0
global after : 100
i after       : 0xAEAEAEAE
rez           : 0
```

Figure 8: Successful run of thtst1.

```
$ thtst1
global before: 10
i before      : 0xF0F0F0F
rez           : 0x0
global after : 100
i after       : 0xAEAEAEAE
rez           : 0
assert failed: file thtst1.c  line 53
```

Figure 9: Failed run of thtst1

```
$ thtst2
Starting 4 threads
thtst2.c 62: started thread 1
thtst2.c 62: started thread 2
thtst2.c 62: started thread 3
thtst2.c 62: started thread 4
thtst2.c 66: joining with 1
thtst2.c 66: joining with 2
thtst2.c 66: joining with 3
thtst2.c 66: joining with 4
All threads joined
```

Figure 5: Successful run of thtst2.

```
$ thtst3 5
num threads 5
thtst3.c 99: 5
thtst3.c 105: 5
  created thread 1 0
  created thread 2 1
  created thread 3 2
  created thread 4 3
  created thread 5 4
  join thread 1 0
  join thread 2 1
  join thread 3 2
  join thread 4 3
  join thread 5 4
```

Figure 7: Sample test run of thtst3

```
$ thtst4
All threads joined
```

Figure 6: Sample output from thtst4

```
$ thtst5
thtst5.c 53: 0x0 0x2FE8
Starting 2 threads
thtst5.c 64: 1
thtst5.c 64: 2
thtst5.c 72: -1
thtst5.c 75: joining with 1
thtst5.c 75: joining with 2
All threads joined
```

Figure 4: Sample output from thtst5

```
$ thtst6
*** thread stack not page alligned ***
thtst6.c 32: -1
```

Figure 3: Sample output from thtst6