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 __stuff__ 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.

Overall, this assignment is much – much – much more prescriptive[1] than I'd like. I'd prefer it be more descriptive.[2] Following the document does not absolve you of thinking for yourself and filling in places not directly mentioned.

This lab is May 12th by midnight. Submit a single gzipped tar file to **TEACH**. 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 many parts to this assignment. Many have a lot of steps. 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've heard that before. 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.

This assignment is done entirely in the xv6 environment.

This programming project is worth 500 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. If you prefer to call it Lab4, I'm fine with that. Cloning the xv6-kthread directory is not a requirement, but it already includes a lot of code to help you get started. It will help you be done sooner.

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

[1] Saying exactly what must happen, especially by giving an instruction or making a rule. https://dictionary.cambridge.org/us/dictionary/english/prescriptive

[2] Presenting observations about the characteristics of someone or something. https://www.merriam-webster.com/dictionary/descriptive

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 for class, such as the `halt` 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. You can see sample output from the testing programs (`thtst[1-6]`) at end of this document.

`memalgn.c` – this is just a small program to show what steps are taken to assure a block of memory is page aligned.
`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.
`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).
`thtst3.c` – this is a small version of a multi-threaded matrix multiplication. 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.
`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.
`thtst5.c` – this program tests 1 thing. 1) you should receive a graceful error if you try to join with a non-existent thread.
`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.

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. these programs may show "`zombie!`" when they complete. You can ignore that.

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:

`int oncpu` – we will use this to show the CPU on which 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`. The right default value is for `oncpu` should -1. When the process/thread is `RUNNING`, this should be the CPU number, which you will set in `scheduler()`.

`ushort is_thread` – this is use to indicate that a *thingie* taking up a slot in the `ptable` structure is a thread for a process, not the main process itself. Since we want the thread 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. Only for thingies created through `kthread_create()` will this be set to `TRUE`.

`ushort is_parent` – indicates that this process has (or had) threads within it. Like the `is_thread` data member, this is useful when managing processes. This is `TRUE` only for the "parent" thread. The parent thread is the initial stream of execution for a multi-threaded process. The parent was created by a call to `fork()/exec()`. Only see this to `TRUE`, when a new thread is created by calling `kthread_create()`.

`ushort tid` – if this *thingie* is a kernel thread (which is what you are building), this will 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`). The default value for non-threads is 0. For a thread created by `kthread_create()`, it will be the value of the `next_tid` data member for the parent thread.

`ushort next_tid` – 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, from a call to `kthread_create()`, for a process will have a `tid` of 1.

`int thread_exit_value` – if this *thingie* is a thread, this is will hold the exit value from that thread (set in `kthread_exit`). This is best set with the actual thread exit value in the call to `kthread_exit()`. Initialize it to 0 then set it to the actual value in `kthread_exit()`.

`ushort thread_count` – applies only on the parent thread, it is the count of the number of current threads. For all "child" threads, it should always be 0.

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

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

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/kernel 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: **`abt *`**: 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 is used as the stack for the thread is performed.
`benny_thread_join`	Return type: **`int`** Parameters: **`abt`**: 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. This is also where you can pick up the exit value form a thread.
`benny_thread_exit`	Return type: **`int`** Parameters: **`exit_value`**: the exit value for the thread. This is called by a thread when is it complete and ready to terminate. It is, to die for.
`benny_thread_bid`	Return type: **`int`** Parameters: **`abt`**: 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 abstract/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?.c` test programs 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`. It is modeled after that. **`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 be a page aligned lump

	of memory from user space (NOT kernel space). The tstack must have been allocated before the call to kthread_create() occurred. This is where an actual kernel thread is created. It gets a 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: **exit_value**: an integer that represents the exit status of the thread. A value of 0 generally means a successful termination of the thread. Any value other than zero generally indicates a non-successful termination of the thread. When testing, we can use non-zero values. 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. Only when another thread joins with it is it removed from the ptable (just as happens with a process).

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 innermost wild kernel hacker.

Part 5 – Implement kthread_* Functions – (100 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**.

In addition to the kthread_* functions, we will modify a couple other functions in this section. The functions will be validated with the test programs.

When writing these functions, I put several blocks of code that used the proc_kdebug_level from the kdebug function. It makes it easy to enable and disable diagnostics.

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 or rename variables 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 on page 2), look at the source code for it. If it is not page aligned, return -1.

Next, call allocproc() and assign the value to the np variable (which I renamed newthread). I also renamed the pid variable as tid.

The next thing fork() does is make a copy of the page table (the call to copyuvm, for copy user virtual memory). 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 newthread 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 how we display the size later in this lab.

The tf data member for a proc stands for trap frame (it was not my idea to call it that. I would have used a longer data member name, like ttf, for "the trap frame"). The newthread has a copy of the curproc trap frame. Do this (yes the * characters are required, that's how it is copied):

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

Leave the line that clears the eax data member to zero. It seems to have no effect for the threads.

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, ~~but it is initialized in the allocproc() function. Use the same syntax as is used in allocproc(), but use func instead of forkret.~~

Make sure you set the is_thread member for newthread to TRUE.

There is not a true parent-child relationship between threads, but we need to keep track of which threads are related in a process. So, set the parent of the new thread newthread to curproc, **UNLESS** curproc is itself a thread. If curproc is a thread, then the parent of newthread is the parent of curproc. Use this opportunity to also set the is_parent member in the process (main thread). After assigning the current value of next_tid to the tid data member of newthread, increment the thread_count of the parent process (the one 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. Part of this can be accomplished like this:

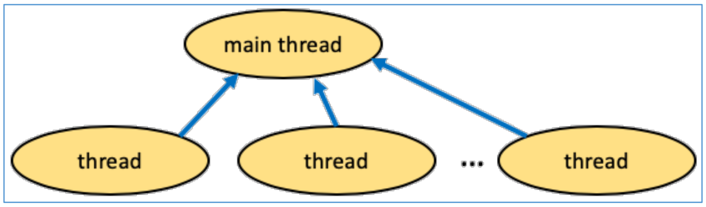


Figure 1: Relationship between newly created threads and the main thread (thread 0). All threads created with kthread_create(), for a given process, have the same parent, the main thread.

```
tid = newthread->tid = newthread->parent->next_tid++;

newthread->parent->thread_count++;
```

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

```
newthread->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 sizes, 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 (do them whether they make sense or not).

```
newthread->tf->esp -= sizeof(void *);
*((int *) (newthread->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 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 calling function can pick up. Consider this

```
newthread->tf->esp -= sizeof(tid);
*((int *) (newthread->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. It is sad to do this, I'd rather just set a pointer to the same file descriptors. But, the file descriptors are a hard-coded array.

Do the same assignment for `cwd` as is done in `fork()`. 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`.

Whew… done with `kthread_create()`. As easy as 1, 2, 3, … ∞. Gotta watch that last step. It's a doozy.

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 later. 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.

Use a variable with a wild name like `return_value` (initialized to `-1`) to hold the return value for the function.

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.

I used a variable called `thread0` as the return from the call to `myproc()`. Finding the right `thread0` is very important, but let's get ahead of ourselves.

If `tid == 0`, bail and return a -1. We cannot join with `tid` 0 (i.e. the `thread0` process).

If `thread0`'s `is_thread` data member is `TRUE`, make `thread0 = thread0->parent`. Remember how we created a single level of threads linked to the parent thread? We are using that here.

Now, acquire the lock on the `ptable` and begin to loop through it, using the `p` variable. We are looking for the thread whose `parent` is `thread0` and whose `tid` data member is equal to the `tid` passed into the function. All other thingies, we ignore.

Now that we've found the `tid` that matches the passes `tid` and it's parent is `thread0`, we need to make sure that thread is complete. For that, we enter a tight `while` loop checking to see when its status becomes `ZOMBIE`. Its status will be set to `ZOMBIE` when it calls `kthread_exit()` on itself. During that while loop, we need to make sure we allow other process/threads to run and don't hold onto the `ptable` lock. I go into the following loop:

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

Once we've completed that loop, we've found the right thread and it has completed.

We must decrement the thread count for thread0.

Free the kstack for the thread. This was allocated through a call to allowproc() while in kthread_create().

After calling kfree(), we just need to clear out the data members from the ptable entry pointed to by p. Set kstack to NULL, pid to -1, parent to NULL, name[0] to NULL, state to UNUSED, sz to 0, and killed to FALSE. Set the return_value local variable to the data member thread_exit_value.

DO NOT CALL freevm(). That would free the virtual memory for the entire process. We do not want to do that here.

Break out of the for(;;) loop, release the ptable lock and return the return_value.

Okay, let's go make a little change to the wait() function. If the calling thingie is a thread, panic. If the calling thingie has a thread_count > 0, panic. It looks like this:

```
if (curproc->is_thread == TRUE) {
    panic("called wait on thread");
}
if (curproc->thread_count > 0) {
    panic("called wait on thread0 with children");
}
```

Luckily, none of the test programs do these nasty things.

Two big functions down (kthread_create() and kthread_join()); 1 more big function to go.

kthread_exit()

The code for kthread_exit() is pretty straight forward.

Get a variable called curproc (as exit() does). If curproc data member is_thread is TRUE, then:

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

Cleanup the cwd (exacly as exit() does it with begin_op and end_op).

Set killed to FALSE, the thread data member thread_exit_value 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. This function does not return, just like you'd not expect a call to exit() to not return.

Okay, let's go make a little change to the exit() function. This is exactly like the code we added into the wait() function. If the calling thingie is a thread, panic. If the calling thingie has a thread_count > 0, panic. It looks like this:

```
if (curproc->is_thread == TRUE) {
      panic("called exit on thread");
}
if (curproc->thread_count > 0) {
      panic("called exit on thread0 with children");
}
```

Luckily, none of the test programs do these nasty things.

That's it for kthread_exit(). Time to give yourself a big woot woot!

kill()

There is a small change we need to make to the kill() function. We need to add a small block of code that makes the exact same check for a thread being killed or if a parent thread with live threads calls kill().

scheduler()

Since we want to know on which CPU a process or thread is running, we need to update that in the scheduler() function.

At the top of the function, create a new variable called something like current_cpu and assign it the value from cpuid().

When a process/thread is chosen to be scheduled, assign is oncpu data member the value of current_cpu.

When the scheduled process/thread completes (following the **switchkvm()**), reset the oncpu data member back to -1.

Sundry files and functions

You are adding 3 new system functions. That means you need to go through the rest of the process of modifying the other files to create the system functions. You should be getting used to this by now. Specifically, you'll need to touch the following files (just as you have for other new system functions):

- defs.h
- syscall.h
- syscall.c
- sysproc.c
- usys.S

Part 6 – Refresh the ps/cps() code – (20 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, thread_count, is_thread, and tid. The header information should be: “cpu”, “thd cnt”, “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 values for oncpu. A CPU number should not show up more than once in the ps listing.

The “thd cnt” column should show the count of active threads only then the process is a parent and has still running threads. Otherwise, show a blank.

The “is thd” column shows a ‘y’ for each kernel thread. Only the active thingies that were created from kthread_create() show up with a ‘y’. All others are an empty space.

Finally, the “thd #” should only show the tid data member value for those thingies that were created from kthread_create(). Anything not created that way are empty.

The easy way to see this data is to run “thtst2 6” and then run ps several times to see the output. If you see a “zombie!” after doing this, don’t worry, it is only a flaw in their shell. You can see an example of running ps with 4 CPUs and the thtst2 6 at the end of the document.

Part 7 – Update and Validate on 4 CPUs – (60 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 256. The points for this part are not awarded to simply changing the Makefile, but for all the test programs 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
```

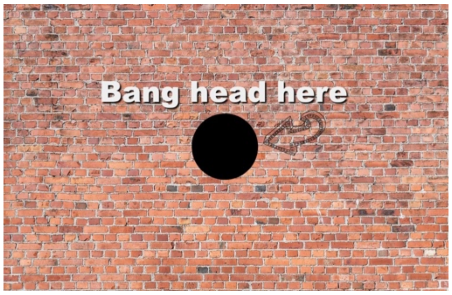
The macro on the command line will override the setting within the Makefile.

Other Tips – Make Sure You Read This

I have to be honest, there are a couple places where I’ve struggled when writing this code. One of the biggest is where I called kfree() in the kthread_join(). 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 …”. This is especially true when I run with more than 1 CPU. 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 tracked address back through kalloc() to kfree() and found nothing. 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.

Another place I struggled was the suptime() function we wrote as part of the lottery scheduler. The suptime() function makes a lock on the tickslock, copies the value of ticks (a kernel wide global variable), and releases the lock. No muss – no fuss. Yet, when run with more than 1 CPU, the kernel will hang when trying to acquire that lock. I’ve stepped through this code for hours trying to find what I can do to prevent it. I’ve found nothing. NOTHING!!! You can take the path of least resistance and just enable the DTICKS macro in the Makefile to get around the issue. If you read this only after spending a few hours debugging, I’m sure you learned something. If you did find how to truly fix this, please let me know.



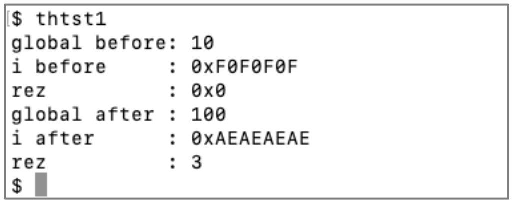
Submit to TEACH

When you are done with the Lab4, 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.

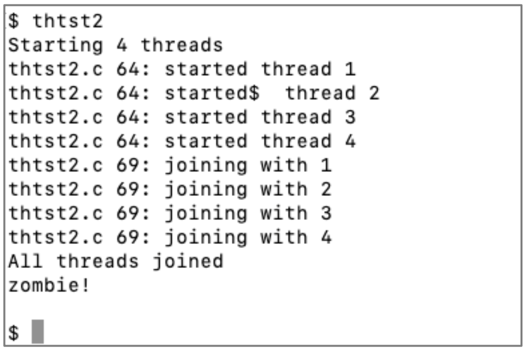
Final note

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 and ps

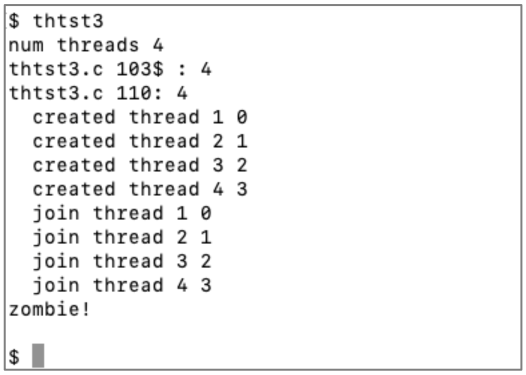


```
$ thtst1
global before: 10
i before      : 0xF0F0F0F
rez           : 0x0
global after : 100
i after       : 0xAEAEAEAE
rez           : 3
$ 
```



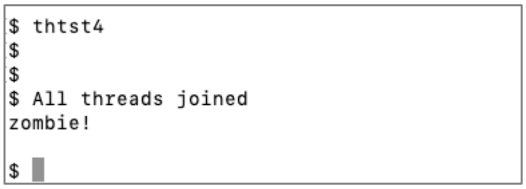
```
$ thtst2
Starting 4 threads
thtst2.c 64: started thread 1
thtst2.c 64: started$  thread 2
thtst2.c 64: started thread 3
thtst2.c 64: started thread 4
thtst2.c 69: joining with 1
thtst2.c 69: joining with 2
thtst2.c 69: joining with 3
thtst2.c 69: joining with 4
All threads joined
zombie!

$ 
```



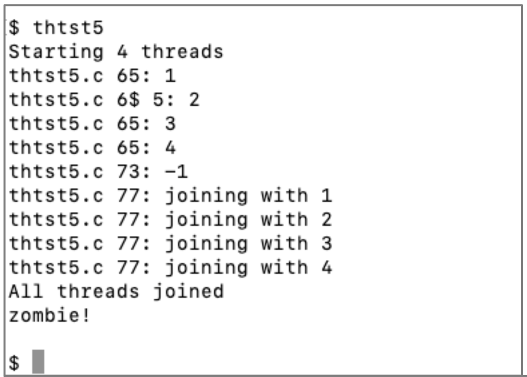
```
$ thtst3
num threads 4
thtst3.c 103$ : 4
thtst3.c 110: 4
  created thread 1 0
  created thread 2 1
  created thread 3 2
  created thread 4 3
  join thread 1 0
  join thread 2 1
  join thread 3 2
  join thread 4 3
zombie!

$ 
```



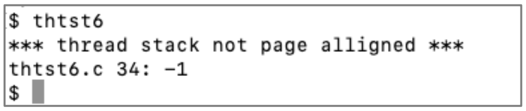
```
$ thtst4
$
$
$ All threads joined
zombie!

$ 
```

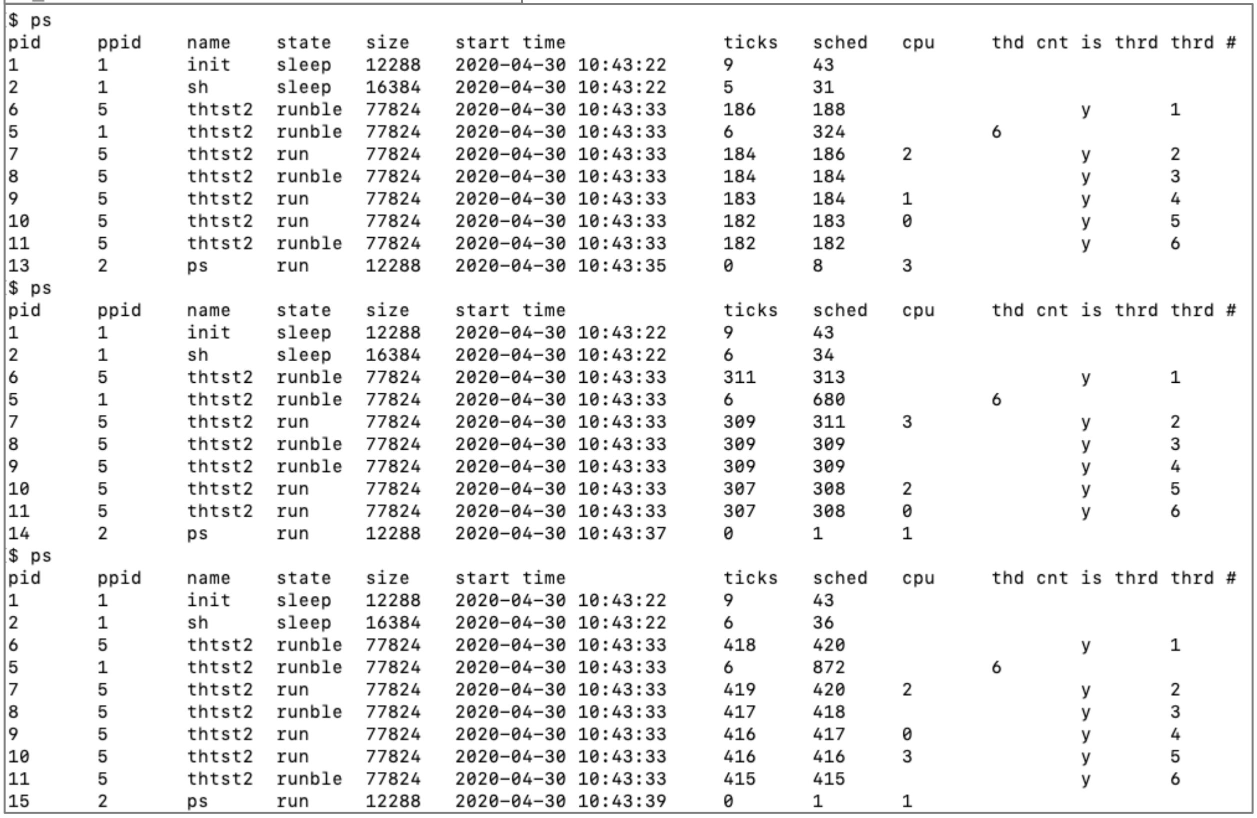


```
$ thtst5
Starting 4 threads
thtst5.c 65: 1
thtst5.c 6$ 5: 2
thtst5.c 65: 3
thtst5.c 65: 4
thtst5.c 73: -1
thtst5.c 77: joining with 1
thtst5.c 77: joining with 2
thtst5.c 77: joining with 3
thtst5.c 77: joining with 4
All threads joined
zombie!

$ 
```



```
$ thtst6
*** thread stack not page alligned ***
thtst6.c 34: -1
$ 
```



```
$ ps
pid     ppid    name    state   size    start time              ticks   sched   cpu     thd cnt is thrd thrd #
1       1       init    sleep   12288   2020-04-30 10:43:22     9       43
2       1       sh      sleep   16384   2020-04-30 10:43:22     5       31
6       5       thtst2  runble  77824   2020-04-30 10:43:33     186     188                     y       1
5       1       thtst2  runble  77824   2020-04-30 10:43:33     6       324             6
7       5       thtst2  run     77824   2020-04-30 10:43:33     184     186     2               y       2
8       5       thtst2  runble  77824   2020-04-30 10:43:33     184     184                     y       3
9       5       thtst2  run     77824   2020-04-30 10:43:33     183     184     1               y       4
10      5       thtst2  run     77824   2020-04-30 10:43:33     182     183     0               y       5
11      5       thtst2  runble  77824   2020-04-30 10:43:33     182     182                     y       6
13      2       ps      run     12288   2020-04-30 10:43:35     0       8       3
$ ps
pid     ppid    name    state   size    start time              ticks   sched   cpu     thd cnt is thrd thrd #
1       1       init    sleep   12288   2020-04-30 10:43:22     9       43
2       1       sh      sleep   16384   2020-04-30 10:43:22     6       34
6       5       thtst2  runble  77824   2020-04-30 10:43:33     311     313                     y       1
5       1       thtst2  runble  77824   2020-04-30 10:43:33     6       680             6
7       5       thtst2  run     77824   2020-04-30 10:43:33     309     311     3               y       2
8       5       thtst2  runble  77824   2020-04-30 10:43:33     309     309                     y       3
9       5       thtst2  runble  77824   2020-04-30 10:43:33     309     309                     y       4
10      5       thtst2  run     77824   2020-04-30 10:43:33     307     308     2               y       5
11      5       thtst2  run     77824   2020-04-30 10:43:33     307     308     0               y       6
14      2       ps      run     12288   2020-04-30 10:43:37     0       1       1
$ ps
pid     ppid    name    state   size    start time              ticks   sched   cpu     thd cnt is thrd thrd #
1       1       init    sleep   12288   2020-04-30 10:43:22     9       43
2       1       sh      sleep   16384   2020-04-30 10:43:22     6       36
6       5       thtst2  runble  77824   2020-04-30 10:43:33     418     420                     y       1
5       1       thtst2  runble  77824   2020-04-30 10:43:33     6       872             6
7       5       thtst2  run     77824   2020-04-30 10:43:33     419     420     2               y       2
8       5       thtst2  runble  77824   2020-04-30 10:43:33     417     418                     y       3
9       5       thtst2  run     77824   2020-04-30 10:43:33     416     417     0               y       4
10      5       thtst2  run     77824   2020-04-30 10:43:33     416     416     3               y       5
11      5       thtst2  runble  77824   2020-04-30 10:43:33     415     415                     y       6
15      2       ps      run     12288   2020-04-30 10:43:39     0       1       1
```