

# CS169 Programming Assignment 1: Kernel 2

## Threads, Synchronization, and Processes

---

*Assignment Out:* Monday, 29 Sept. 2008  
*Assignment Due:* Monday, 6 Oct. 2008 (11:59 pm)

---

## 1 The Assignment

If you're reading this, you've presumably read the previous handout on Kernel 1, regarding the low level device drivers and implementing them. We also assume that you've read the information handout on the overview of the assignment.

After this stage you should have a fully functional threads library and synchronization primitives for threads. You will be able to support processes that contain one kernel thread.

The following is a list of the features which you will be adding to your semi-functional Weenix kernel:

- Threads and processes
- Scheduling and context switching
- Exiting
- Synchronization primitives

As you've learned in the previous step, each of the functions that is still unfinished will have the macro `NOT_YET_IMPLEMENTED`.

## 2 Booting, revisited

As described in the previous handout, we are in the boot process, and we don't have a context. If you've completed Kernel 1, your `bootstrap` was a testbed for your device drivers, and everything was executing in a fake thread (fake in that there was no metadata for the thread; it was just raw

code executing). The goal of the `bootstrap` function is now to set up the first kernel thread and process, and to have it call the `idleproc` function within this new thread. This should be a piece of cake once you've developed and implemented the functionality for threads and the scheduler.

Inside of `idleproc` is where all of your test code will be placed. When your operating system is nearly complete, this will execute a binary program in user mode, but for now be content to put together your own testing system for threads and processes.

Functions you need to revisit in `boot.c`:

- `void bootstrap(void)`
- `void idleproc(long arg1, void *arg2)`

### 3 Processes

Functions you need to write in `proc.c`:

- `void proc_init(void)`
- `void init_new_proc(proc_t *p, int i)`
- `void proc_new_special(int i)`
- `int proc_new(void)`
- `void proc_add_thread(proc_t* p, kthread_t* thr)`

### 4 Threads

Each process should have only one kernel thread associated with it. There are not multiple kernel threads in a single process. There are a few hacks regarding context creation when writing `kthread_create`. See the comments for the functions as well as the simulator manual for details. These most likely will not come up until Virtual Memory. Don't worry about them now.

Note that you may need to implement the scheduler (described in the next section) in order to understand and write some of these functions.

Functions you will need to write in `kthread.c`:

- `thread_t* kthread_create(struct proc* proc, kthread_func_t func, int arg1, void* arg2, int initial_psr)`

- `void sleep_on(kqueue_t* q)`
- `kthread_t* wakeup_on(kqueue_t* q)`
- `void do_thr_exit(int status)`
- `int do_thr_cancel(kthread_t *thr, int status)`

## 5 Scheduler

Once you have created processes and threads, you will need a way to run these threads and switch between them. If you choose to write a yield function, it is advisable to name it `sched_yield()` or similar, as `sched_yield()` is the name of an existing Linux system call.

Functions you will need to write in `sched.c`:

- `void sched_make_runnable(kthread_t* thr)`
- `void sched_switch(void)`

## 6 Exiting

After finishing this section, you should make sure that you can have multiple processes with multiple threads running, and that exiting threads and processes works correctly.

Functions you will need to write in `exit.c`:

- `void proc_exit(proc_t *proc, int status)`
- `pid_t do_wait(int *status)`
- `int do_exit(int pid, int status)`

## 7 Synchronization Primitives

Since the kernel is multithreaded, we need some way to ensure that certain critical paths are not executed simultaneously by multiple threads. One mechanism you will need (and are required to implement) is the mutex. Feel free to implement semaphores (and condition variables!) as well if you so desire, though they are in no way necessary to have a functioning kernel.

## 7.1 Mutexes

Functions you will need to write in `kmutex.c`:

- `void kmutex_init(kmutex_t* mtx)`
- `void kmutex_lock(kmutex_t* mtx)`
- `void kmutex_lock_cancellable(kmutex_t* mtx)`
- `void kmutex_unlock(kmutex_t* mtx)`

## 8 Closing Remarks

We know that this assignment can seem overwhelming. It is also possible that this is your first exposure to a large code base. You should definitely spend some time poring over the existing code, thinking about what it is that we are asking you to implement. For a richer understanding, it might be helpful to try and understand where your code fits into the existing support code.

Finally, be sure and ask questions. Your Mentor TAs worked through this last year and over the summer to try and simplify things. If you're not sure how something works or why, ask! Constant communication is the key for this course.

Good Luck!