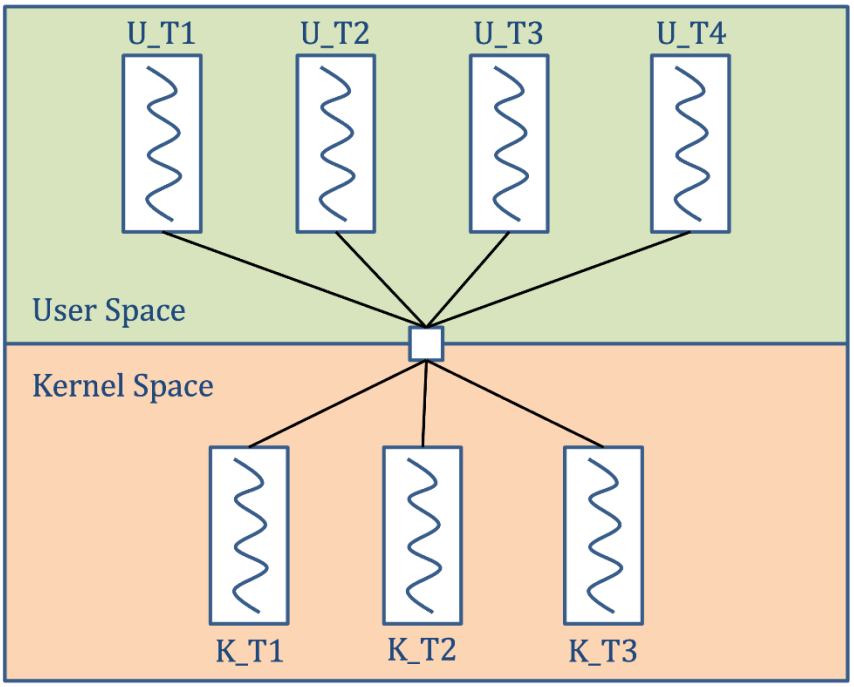
Operating Systems 202.1.3031

Spring 2022/2023 Assignment 2

Threads in XV6 and Synchronization

Responsible Teaching Assistants:
Roee Weiss-Lipshitz, Pan Eyal



Ben-Gurion University of the Negev

Contents

1. Introduction **2**

2. Submission Instructions **3**

3. Task 0: Compiling and Running xv6 **4**

4. Task 1: User Level Thread Framework **5**

4.0. Apply the ULT framework patch 5

4.1. Application programming interface 6

4.2. Scheduling policies 7

5. Task 2: Kernel Level Thread Framework **8**

5.0. Apply the KLT support patch 8

5.1. The process and kernel thread control blocks (PCB & KTCB) 9

5.2. Supporting single threaded execution 11

5.3. Supporting multi-threaded execution 14

6. Task 3: Test your implementation! **17**

1. Introduction

Welcome back to another exciting chapter in the operating systems world!
We hope that by now you are more familiar with **xv6** and ready to delve deeper into the OS.

In this assignment you are going to implement two types of thread frameworks on **xv6**: **user level threads (ULT)** and **kernel level threads (KLT)**. Each approach has pros & cons; the goal of this assignment is to understand the differences between the 2 approaches.

The assignment consists of the following parts:

1. Cooperative User-level thread framework
2. Kernel-level thread framework
3. Testing the two thread frameworks

Remember:

An operating system, even a relatively simple one like xv6, is a complex piece of software. Such low-level code is often challenging at first. This stuff takes time, but it can be fun and rewarding! We hope this class will serve to bring together a lot of what you've learned in your degree program here at BGU. **Take a deep breath and be patient with yourself.** When things don't work, keep calm, and start debugging!

Good luck and have fun!

2. Submission Instructions

- Make sure your code compiles without errors and warnings and runs properly!

- We recommend that you comment on your code and explain your choices, if needed. This would also be helpful for the discussion with the graders.

- You should submit your code (the entire xv6 folder, including files you did not modify) as a single **.tar.gz** or **.zip** file.

- We advise you to work with git and commit your changes regularly. This will make it easier for you to track your progress, collaborate and maintain a working version of the code to compare to when things go wrong.

- Submission is allowed only in **pairs** and only via **Moodle**. Email submissions will not be accepted.

- Before submitting, run the following command in your xv6 directory:

```
$ make clean
```

  This will remove all compiled files as well as the obj directory.

3. Task 0: Compiling and Running xv6

Start by downloading the clean xv6 source code from the class GitHub.

Tasks will appear in this assignment in orange boxes like this one:

Task 0 - Download and compile xv6

1. Open a shell or terminal in your virtual machine. We will use the **bash** shell in this document.
2. Point your shell to the directory where you want to download the xv6 source code. For example:

```
$ cd ~/projects/os
```

3. Run the following command in your terminal:

```
$ git clone https://github.com/BGU-CS-OS/xv6-riscv.git
```

4. You can then move into the source directory to inspect the source code or open it in your favorite editor:

```
$ cd xv6 -riscv
```

5. Build xv6 by running the following command:

```
$ make
```

6. Make xv6 run in QEMU by the following command:

```
$ make qemu
```

Note: Every terminal or shell command in this document is precededby a dollar sign ($). This is a convention used to distinguish between commands and their output. You should not type the dollar sign when running the commands. Also, your terminal prompt may be different than the one shown here.

4. Task 1: User Level Threads Framework

User level threads (ULT) avoid using the kernel (and are transparent to it). They manage their own tables and their own scheduling algorithm (thus allowing more efficient, problem-specific scheduling). ULT are usually cooperative (i.e., threads must voluntarily give up the CPU by calling the *uthread_yield* function).

4.0. Apply the ULT framework patch

In order to manage user threads within a process, a struct is required to represent a user thread (much like a PCB for a process). We will refer to this as user thread control block (UTCB). Such a struct should look something like the following:

```
struct uthread {
    char                ustack[STACK_SIZE];  // the thread's stack
    enum tstate         state;          // FREE, RUNNING, RUNNABLE
    struct context      context;        // uswtch() here to run process
    enum sched_priority priority;       // scheduling priority
};
```

You may add any field you see fit to the above struct. This type and the prototypes of the ULT package API functions are defined for you in *uthread.h* file, which will be applied with the patch we provide.

The user threads framework must maintain a table of the threads that exist within the process. You can define a static array for this purpose, whose size will be defined by the MAX_UTHREADS macro (the MAX_UTHREADS macro is defined in *uthread.h* as well).

Task 1.0 – apply the 'ULT' framework patch

1. Download from the assignment page in the Moodle the *ULT-framework.patch* file into a fresh repository of xv6.
2. Open a terminal in the project directory and run the following command:

```
$ git apply --ignore-space-change --ignore-whitespace ULT-framework.patch
```

4.1. Application programming interface

The application programming interface (API) of your user threads package must include the following functions:

*int uthread_create(void (*start_func)(), enum sched_priority priority);*

This function receives as arguments a pointer to the user thread's start function and a priority. The function should initialize the user thread in a free entry in the table, but not run it just yet. Once the thread's fields are all initialized, the user thread's state is set to runnable. The function returns 0 on success or -1 in case of a failure.
Note: do not forget to set the *'ra'* register to the *start_func* and *'sp'* to the **top** of the relevant *ustack*.

Consider what happens when the user thread's start function finishes: does it have an address to return to?
Hint: A *uthread_exit(…)* call should be carried out explicitly at the end of each given *start_func(…)*.

void uthread_yield();

This function picks up the next user thread from the user threads table, according to the scheduling policy (explained later) and restores its context. The context of a cooperative ULT is identical to the *context* field in the xv6 PCB, which we've seen in the practical sessions.
Therefore, the context switch between the user threads can occur in the same way as the regular context switch (look at *swtch.S* and *uswtch.S*).
If you are interested in understanding why the specific registers in the *context* are saved, you can read about callee and caller saved registers.

After the context switch occurs, the newly selected thread will run the code starting from the address stored by the '*ra*' register in the same way as in switching between two processes in the kernel.

void uthread_exit();

Terminates the calling user thread and transfers control to some other user thread (similarly to yield). Don't forget to change the terminated user thread's state to free. When the last user thread in the process calls *uthread_exit* the process should terminate (i.e., *exit(…)* should be called).

enum sched_priority uthread_set_priority(enum sched_priority priority);
enum sched_priority uthread_get_priority();

The first function sets the priority of the calling user thread to the specified argument and returns the previous priority.
The second function returns the current priority of the calling user thread.
Use the given *enum* defined in *uthread.h* for these functions.

int uthread_start_all();

This function is called by the main user thread after it has created one or more user threads. It is similar to yield; it picks the first user thread to run according to the scheduling policy and starts it. If successful, this function never returns, and any code beyond it will never be executed (much like *execvp*).
Since the main user thread was not created using *uthread_create*, and hence has no entry in the user thread table, its context will never be restored. Any subsequent calls (after the user threads were already started) to *uthread_start_all* should not succeed. In such a case, this function should return -1 to indicate an error. Hint: this functionality can be implemented using a global variable or static variable similarly to *forkret* in *proc.c*.

uthread_t uthread_self();*

Returns a pointer to the UTCB associated with the calling thread.

4.2. Scheduling

Your threads package should support a **priority scheduling policy** where threads are scheduled according to their priorities. In case of several threads having the same priority, they should be scheduled in a round-robin manner.

The scheduler should perform the context switch between two *uthread*s by using the *uswtch(…)* function we implemented for you in the patch. The function resides in *uswtch.S* file and it is identical to the *swtch(…)* function that is under the kernel. Because user libraries do not usually have direct access from the user-space to the kernel-space, we duplicated this function to use it in the user-space. The function has the following signature:

```
void uswtch(struct context *old, struct context *new)
```

Task 1.1 – write the ULT framework

The following steps will guide you through the process of adding the ULT framework:

1. Create a new file under the user directory named *uthread.c*.
   include *uthread.h* and any other necessary header files in it, and add the following global variables:
   - An array for all the process threads
   - A pointer for the current thread
2. Implement in *uthread.c* all remaining functions of *uthread.h* as described above.
3. Support the *uthread* library in the Makefile by adding a $U/uthread.o rule to ULIB variable on line 90.

5. Task 2: Kernel Level Thread Framework

Kernel level threads (KLT) are implemented into and managed by the kernel. Our implementation of KLT will be added on top of the processes in xv6 and will use a new struct you will need to define (similar to task 1) called *kthread*. One key characteristic of threads is that all threads of the same process share resources such as memory and open files. Thus, a process will now be able to contain more than a single thread. That is, the PCB will store the *kthread* structs of its threads as well as its own shared resources which are shared between its threads.

5.0. Apply the KLT support patch

In order to avoid dealing with a lot of uncovered material, we are supplying you with a patch to add to your code. This patch includes changes in the following files: *defs.h, exec.h, memlayout.h, param.h, proc.h, proc.c, syscall.c, trampoline.S* and *trap.c*. It also adds two new files: *kthread.h* and *kthread.c*. These changes in the code do several things:

- It modifies how the *trapframe* is being referenced, from statically to dynamically. Originally, xv6 could 'get away' with statically referencing the *trapframe* since each process has only one *trapframe*. Now, in a multi-threaded framework, this is no longer true. Each kernel thread executes separately; therefore, it should have its own *trapframe* inside the process memory. This means that *trampoline* needs to receive the pointer to the correct *trapframe* at runtime.
- It modifies the use of the *trapframe* in some functions that we don't want you to deal with. This needs to be done because in this section the *trapframe* will move from the *proc* struct to the new *kthread* struct.
- It allocates kernel stacks for each of the kernel threads. This is done because of architectural choices of xv6. (Since we haven't yet reached the memory chapter in the course, we don't want you to deal with it.)
- It adds two new files containing the foundations for the *kthread* struct and functionalities. That is, some of the functions and the new struct will be changed later by you.

Task 2.0 – apply the 'KLT' support patch

1. Download from the assignment page in the Moodle the *KLT-support.patch* file into your repository of xv6.
2. Open a terminal in the project directory and run the following command:

```
$ git apply --ignore-space-change --ignore-whitespace KLT-support.patch
```

5.1. The process and kernel thread control blocks (PCB & KTCB)

Since a process may now contain more than a single thread, we will need to make some changes to the PCB as well as create a new KTCB for managing threads under the process.

The **KTCB** should contain all the data required by a thread.
This includes:

- A **lock** (similarly to the PCB)
- The current **state** of the thread (the possible states should be the same as in the original PCB)
- A **channel** field (a pointer that can be used to wake up the thread in case it is sleeping on it)
- The **killed** flag of the kernel thread
- The **exit status** of the thread
- The unique (per group) **thread ID** (similar to PID for processes)
- A pointer to the **PCB** it belongs to
- A pointer to the **kernel** stack
- A pointer to its **trapframe**
- The **context** (needed for context switch)

Most of the above should be moved from the original PCB (represented by the *proc* struct). This is because the original process was the executable unit of the operating system. Now, in the multi-threaded framework, threads are the executable units and should hold the execution information in their control block.

The **PCB** should be modified. Some responsibilities are alleviated from it, but it also acquires some new ones.
The required information is:

- A **lock** for the public properties of the PCB (including the process' threads table)
- A **counter** and a **lock** for allocating thread IDs (that will be used similarly to *allocpid* in *proc.c*)
- The current **state** of the process (just: unused, used or zombie)
- The **killed** flag of the process
- The **exit status** of the process
- The unique **process ID**
- The **kthread group table**
- A pointer to the **base of the threads trapframes page** (*base_trapframes*) in memory (more on that later)
- A pointer to its **parent** process
- The **size** of the process' memory
- The **page table**
- An array of pointers to the process' **open files**
- A pointer to the **current working directory**
- The process' **name**

In this way, the PCB holds all the information shared between its threads and all of them may access it.

Task 2.1 – implement the new PCB and KTCB

The following steps will guide you through the process of writing the new PCB and KTCB:

1. Update inside *kthread.h* the struct named *kthread* that represents the KTCB with the required fields as mentioned above. Note: most of them should be moved from *proc* (and some were already moved by us). When you move or copy a field make sure you also move or copy the necessary structs, enums or macros it depends on.
2. Remove all unnecessary fields from the *proc* struct (if you haven't done so already in the previous sub-task) and notice the two newly added ones.
3. Move the *cpu* struct and cpus array from *proc.h* to *kthread.h*. Edit the struct so that it will hold the running thread instead of a process.
4. Notice inside the *param.h* file that a new macro named *NKT* was added with a value of 1 (similarly to *NPROC*). This macro represents the number of kthreads that will be supported per process (the value will be changed in later tasks).

WARNING: you will have some duplicate definitions and fields (e.g., states in thread and process). Make sure you give each of them a unique and descriptive name so that it will be clear to you when you use them.

5.2. Supporting single threaded execution

Now, we need to modify our OS to support our new structs and move to a single-threaded framework. This will be performed by only using the first entry of the *kthread* table field added in the *proc* struct.
Note: The main (first) thread for each process should always be the thread in the first entry of that *kthread* table.

NOTE: We have three locks that might collide; proc lock, thread lock and wait lock. Make sure to acquire them in the correct order: **wait lock → proc lock → thread lock** (that are under the aforementioned proc). Make sure to release them in the opposite order.

Let us start defining the **KTCB** (*kthread*) related functions. Some of these functions are modified versions of originally PCB (*proc*) related functions, and some are completely new.

Initialization function for the *kthread* table (*kthreadinit(…)*)

Given a *proc*, it initializes the lock in charge of thread ID allocation, then, it initializes for every thread in the process table its lock, state (to unused), process (to be the given *proc* it belongs to) and sets the *kthread* stack pointer in the same way as in *procinit(…)* (the last part has already been done for you). This function will be called once for each process at the initialization time of xv6. Add your code to the already existing *kthreadinit(…)* function in *kthread.c*.

Related original *proc* function: *procinit(…)*

Function for getting current kernel thread (*mykthread(…)*)

Fetches and returns the current running thread from the current cpu's *cpu* structs. Replace the already existing *mykthread(…)* function in *kthread.c* with your implementation.

Related original *proc* function: *myproc(…)*

Function for allocating a new kernel thread ID

Given a *proc*, it allocates a unique kernel thread ID using the counter and lock inside the *proc*.

Related original *proc* function: *allocpid(…)*

Function for allocating a new kernel thread

Given a *proc*, it finds an unused entry in its kernel thread table. It then allocates a new kernel thread ID for it, sets its state to used, assigns it its trapframe, initializes the *context* to zeroes, changes the '*ra*' register in *context* to *forkret* address, and the '*sp*' register in *context* to the top of the stack. Finally, it returns a pointer to the newly allocated *kthread* with its lock acquired.
Note: the pointer to the trapframe should be fetched by using the *get_kthread_trapframe(…)* function from *kthread.c*.

Related original *proc* function: *allocproc(…)*

Function for freeing an allocated kernel thread

Given a *kthread*, it sets its fields to *null / zero*, and the state to unused.

Related original *proc* function: *freeproc(…)*

Now, let's look at the existing functions that should be modified for the **PCB** in *proc.c*. In the following, we describe the required changes:

allocproc

Once an unused *proc* is found, initialize the counter used for kernel thread ID allocation to 1, then allocate the first kernel thread using the allocation function you wrote (without using the returned value from this function). Remember that upon returning from this function, the proc lock and the kthread lock are still acquired (as they should be).
Note: right after applying the patch you can see a new helper function called at the end of *allocproc(…)*. The function is named *allocproc_help_function(…)*. The only purpose of the helper function is to allow the OS to compile and run right after applying the patch. During or after making your changes, you should remove the helper function and its usage. It should be removed from *proc.c*, *kthread.c* and *defs.h*.

freeproc

Free all the kernel threads inside the thread table using the function you wrote, free the *base_trapframes* field and set it to zero.

userinit

Set the state of the first *kthread* state to runnable. Also, release the first *kthread* lock that was acquired in *allocproc(…)* (in addition to the *proc* lock).

fork

Clone the calling thread into the main kernel thread of the newly allocated *proc*, similarly to how it was done in the original *fork(…)* (some of the work has already been done). Make sure you hold and release any necessary locks, and in the correct order! (Keep in mind this Note).

exit

This function exits the entire process (including all of its kernel threads). Therefore, it saves the exit status in the *proc* and changes its state to zombie. The kernel thread state should be changed into zombie as well.

kill

Find the proc to kill based on its *pid* and wake up all of the *proc* sleeping kernel threads after setting the process' *killed* flag value to 1.

procdump

You should remove deprecated *proc* states. This is a function used for debugging.

Look at all of the other functions in *proc.c* file and consider if they should be modified to address the new structs.
Finally, in *kerneltrap(…)* that sits in *trap.c*, before giving up the CPU by calling *yield(…)*, the testing condition should reference the desired *kthread* and its state (as opposed to a *proc*).

Task 2.2 – Supporting single threaded execution

The following steps will guide you through the process of writing the new PCB and KTCB:

1. Create/move inside *kthread.c* all the functions we mentioned above that are related to the KTCB (and any other functions you see fit). You may choose all of the new functions' names in this task, except existing ones we mentioned.
2. Modify all remaining functions in *proc.c* as described previously in this section (and any other function you think that would need a modification as well).
3. Add to *defs.h* file the kthread struct definition, and all of the new functions you added in this section.
4. Make the necessary changes in *kerneltrap(…)*.
5. Make sure the helper function, mentioned in *allocproc(…)* description above, is not present in your OS code.

If this task is done properly, you should be able to run the system now without any issues (running the *usertests* and *forktest* programs inside the OS is a good way to check that everything works).

5.3. Supporting multi-threaded execution

In this part of the assignment, you will edit some more existing kernel code, as well as add system calls that will allow applications to create kernel threads. These changes will support a multi-threaded framework.

Preliminary changes provided in the patch:

Kernel stack allocation:

xv6 originally allocates kernel stacks statically at system initialization (as can be seen in the original *proc_mapstacks(…)*, before applying the patch) and assigns a single allocated stack to each process entry. This is done during initialization, also at initialization (as can be seen in the original *procinit(…)* as well). For a single threaded system, allocating a single kernel stack per process was fine since each process had a single kernel thread. In order to support the change, we needed to do both of those things for each *kthread* entry in each *proc*'s *kthread* table.

Trapframe allocation and mapping:

xv6 allocates a single page in memory for the trapframe of each process (originally seen in *allocproc(…)*) and the trapframe can be referenced from kernel space simply by using the pointer in the *proc* struct. However, the trapframe is sometimes needed when in user mode, and thus it is mapped in the user space as well, somewhere at top of the user memory (you will understand memory mapping better when we discuss memory management later in this course). Again, this is fine in a single-threaded framework since every process has one thread and thus one trapframe.

Now, in a multi-threaded framework, we need to support multiple trapframe per-process. Each trapframe needs to have allocated space in the kernel memory, as well as a mapping in the user space. The *trapframe* struct size is 288 bytes, while a page (which is the smallest unit that can be allocated in the kernel) is of size 4096 bytes. This means that a single memory page can hold up to 14 *trapframe*s. Keeping this in mind, we can use the single page allocated in *allocproc(…)* to hold all of the *trapframe*s of the kernel threads in the same process. Our page was allocated into the *base_trapframes* field (which should now make more sense). We use this pointer to access the *trapframe*s of all the process' *kthread*s, based on the **indices** of the *kthread* entries they belong to. For mapping in user space, our patch edited the existing macro TRAPFRAME (in *memlayout.h*) to receive an argument, which is again the **index** of a *kthread* entry inside the *kthread* table of a *proc*. The macro translates the given index into the address in user memory that the corresponding *trapframe* is mapped to. The initial mapping of the base pointer is done in *proc_pagetable(…)*, which we provided for you in the patch.

More changes in additional files were added to support the above. However, the ones we describe here are enough for understanding the main idea.

System calls:
Now, let's get familiar with the following new system calls, which will allow users to use the multi-threaded framework:

int kthread_create(void *(*start_func)(), void *stack, uint stack_size);

Calling *kthread_create(…)* will create a new thread within the context of the calling process. The newly created thread state will be runnable. The caller of *kthread_create(…)* must allocate a stack for the new thread to use (using malloc). The stack size should be 4000 bytes (as in the ULT). It would be best practice to define a new macro for the size. Define it in a file that is accessible from the user space.
start_func is a pointer to the entry function, which the thread will start executing. Upon success, the identifier of the newly created thread is returned. In case of an error -1 is returned.
This function should set the user-space *'epc'* register to point to *start_func*, and the user-space *'sp'* register to the **top** of the stack.
Note: make sure that the function pointed to by *start_func* calls *kthread_exit(…)* at the end, similarly to the requirement in ULT.

Related original *proc* system call: *fork(…)*

int kthread_id();

Upon success, this function returns the caller thread's id.

Related original *proc* system call: *getpid(…)*

int kthread_kill(int ktid);

This function sets the *killed* flag of the *kthread* with the given *ktid* in the same process. If the kernel thread is sleeping, it also sets its state to runnable. Upon success this function returns 0, and -1 if it fails (e.g., when the *ktid* does not match any *kthread*'s *ktid* under this process).
Note: in order for this system call to have any effect, the *killed* flag of the *kthread* needs to be checked in certain places and *kthread_exit(…)* should be called accordingly.
Hint: look where *proc*'s *killed* flag is checked in *proc.c*, *trap.c* and *sysproc.c*.

Related original *proc* system call: *kill(…)*

void kthread_exit(int status);

This function terminates the execution of the calling thread. If called by a thread (even the main thread) while other threads exist within the same process, it shouldn't terminate the whole process. The number given in *status* should be saved in the thread's structure as its exit status.

Related original *proc* system call: *exit(…)*

int kthread_join(int ktid, int *status);

This function suspends the execution of the calling thread until the target thread (indicated by the argument *ktid*) terminates. When successful, the pointer given in *status* is filled with the exit status (if it's not *null*), and the function returns zero. Otherwise, -1 should be returned to indicate an error.
Note: calling this function with a *ktid* of an already terminated (but not yet joined) *kthread* should succeed and allow fetching the exit status of the *kthread*.

Related original *proc* system call: *wait(…)*

Changes to exit and exec:

Now that we implemented the framework functions to help us control our threads, we can use them to address the expected behavior of *exit(…)* and *exec(…)* in an OS that supports threads. As taught in class, when calling *exit(…)* or *exec(…)*, all kernel threads, other than the one calling the function, should terminate. Only after every other kernel thread terminated, the calling kernel thread can continue the execution of *exit(…)* or *exec(…)*.

You will need to implement this behavior, but you will need to find the correct place in the function to perform it. In the case of *exit(…)*, since it should never fail, the thread termination can occur at the beginning of the function before any other operation (already done) in the function. In *exec(…)* however, this is not the case. *exec(…)* can fail and return to the original process, which should allow all other threads to continue their execution normally. This means we need to make sure we are beyond the point of failure in *exec(…)* before we terminate all other threads.

Task 2.3 – supporting the system calls

The following steps will guide you through the process of adding the system calls that will allow for creating/managing/terminating additional kernel threads in the OS.

1. Implement all of the system calls in these sections as they are described above.
2. Edit *exit(…)* and *exec(…)* to support the new multi-threaded system we've created. (GUIDANCE: use the functions you created for the system calls)

Make sure, that after these changes are implemented, your OS still works properly with NKT values of up to 10.

6. Task 3: Test your implementation!

At the frontal checks, we will run a user program that will use both the ULT and KLT frameworks. Test both frameworks to make sure all requirements are met.

We supply you with two simple tests that will expand the *usertests* file that is already in the xv6 OS. These tests are named *ulttest* and *klttest* and will pass **only after** completing all previous tasks.

Task 3.0 – apply the thread tests patch and run the tests

1. Download from the assignment page in the Moodle the *threads_tests.patch* file into your repository of xv6.
2. Open a terminal in the project directory and run the following command:

```
$ git apply --ignore-space-change --ignore-whitespace threads_tests.patch
```

3. Run the tests by calling the *usertests* user program and make sure all tests pass (this will also run the *ulttest* and *klttest* we provided. If you want to run them individually you can by calling *usertests* with one specific test name as its argument).
4. The tests we provided are very basic. Write more comprehensive tests that will check all of your ULT and KLT functions with more difficult scenarios.