# CS 2510: COMPUTER OPERATING SYSTEMS

Project 1 – User Level Threads

## Introduction

In this project you will implement a simple user level thread library. This library will export a subset of the common thread API available with most thread packages, but implemented inside a new pet_thread library. The library will support basic thread functions such as creation and deletion, userspsace scheduling, and joining. You will also be required to implement a comprehensive set of test cases, in order to validate the correctness of your implementation.

For this project you will be given a skeleton framework with some of the basic components provided, however you will be responsible for implementing the majority of the thread handling code. Inside the skeleton are a small set of files you will use. The library code will be contained within `pet_thread.c` while the test cases will be implemented inside `drvier.c`. For this project you will only be required to implement cooperative scheduling, however extra credit will be considered if you design and implement a preemptive scheduler.

Each thread will be assigned a unique ID of type `pet_thread_id_t` which is defined as a `uintptr_t`. You will be responsible for assigning IDs and ensuring that they are unique.

The external API you will need to support and test for is described here. For the most part these functions match the basic functionality of the pthread interface, so you should look at the pthread documentation if you are unsure how a given function should behave.

- `pet_thread_create`: Creates a new thread and adds it to the runqueue. After this function returns a new thread will be schedulable and will execute the function passed in. The ID of the new thread is returned by writing to the location of the `thread` argument.
- `pet_thread_join`: Waits (blocks) for a given thread to terminate and stores the return value at the location pointed to by the `ret_val` argument.
- `pet_thread_exit`: Terminates a thread with a given return value.
- `pet_thread_yield_to`: Yields the CPU to the specified thread. This short circuites the thread scheduler and instead explicitly context switches to the given thread.
- `pet_thread_schedule`: Examines the runqueue and selects a thread to run. If the selected thread is different from the currently running thread, this function should yield to the new thread.
- `pet_thread_init`: Initializes the library state. Must be called before any other API call is used.
- `pet_thread_run`: Begins execution of the threads in the runqueue. This function will not return until there are no more threads left to execute.

You will also need to create a thread info structure in `struct pet_thread`. This structure will be used to store all of the state necessary to support switching and scheduling between threads. The most important component of this structure will be the stack of the given thread.

### Skeleton Description

The skeleton contains some of the functionality you will need as well as implementation hints about how to approach the problem. The main component we are giving you is an implementation of a context switching function. Context switching is generally very complex operation, and a lot of time is spent optimizing it and implementing clever tricks to keep things consistent. In this project we instead opted for the easy and naive approach. This function will switch to the execution context of a given thread by switching to its stack. It is assumed that when a thread is not currently running a checkpoint of its execution context (register values) is stored at the bottom of the stack. Thus in our implementation a context switch is achieved by simply pushing the current context onto the stack, switching to a different stack, and popping the execution context that is found there. The popped context "activates" the new thread, while the pushed context "checkpointed" the old one.

The name of the context switch function is `__switch_to_stack()` and takes the following arguments:

- `void * tgt_stack`: A pointer to a variable holding the new stack pointer value. Normally this will be the stack pointer of the thread you are switching to. Note this is not the value of the stack pointer, but a pointer to a variable that holds its value.
- `void * saved_stack`: A pointer to a variable that will be used to store the current stack pointer. This will be where the stack pointer from the currently running context is saved to.
- `pet_thread_id_t current`: The Thread ID of the currently running thread. Note that this is only used to pass as an argument to `pet_thread_cleanup`.
- `pet_thread_id_t tgt`: The Thread ID of the thread being switched to. Note that this is only used to pass as an argument to `pet_thread_cleanup`.

Our implementation also cheats by adding a function call inside the context switching code itself. Normally you would not do this, but it makes things simpler. Immediately after a stack switch occurs, but before the new thread is "activated", a call is made to `pet_thread_cleanup` in `pet_thread.c`. Hint: The important thing to note here is that this function call is being made using the stack of the new thread.

`pet_thread_cleanup()` provides the following arguments:

- `pet_thread_id_t prev_id`: The Thread ID of the thread being switched from. (The value passed as `current` to `__switch_to_stack`).
- `pet_thread_id_t my_id`: The Thread ID of the new thread that was activated and whose stack is currently being used. (The value passed as `tgt` to `__switch_to_stack`).

The skeleton also provides a few more things that you will find useful.

- `thread_run_state_t` contains a set of valid run states that each thread can be assigned. Note that only threads in the `PET_THREAD_READY` state should be executed by the scheduler, and the thread that is currently executing should be in the `PET_THREAD_RUNNING` state.
- `struct exec_ctx` represents the execution context as it will be saved on the stack by `__switch_to_thread`. This provides you an easy way of accessing and modifying the execution context of a thread by casting the stack pointer to this structure type.
- When you call `pet_thread_run` you will be in the main program context. This context exists outside of the threads you will be scheduling, and must be switched back to after all the threads have run. The make things simpler we used the concept of a Master Thread, which refers to this context. The master thread is represented as a pet_thread however it is treated in a special way. It should always be assigned the Thread ID of `0`, as specified in the macro `PET_MASTER_THREAD_ID`. The execution of this master thread should also be stored inside the `master_dummy_thread` variable. Note that you will not need to use all of the fields of this structure, just enough of them to store the execution context of the master thread (or at least enough to allow it to be passed as a target to `__switch_to_thread`).
- The skeleton provides a linked list implemenation from the Linux kernel. While you are not required to use it, we highly recommend doing so. It is generally cleaner and simpler to use than other generic linked list implementations available for C.
- There is also a hashtable implementation provided, should you choose to use hashtables in your implementation. You are free to use others, but this is one of the better ones we have found.

## Advice and Hints

Managing stacks will be the most important part of this project. To do this you will need to understand how stacks are used both by the hardware and the software, and be comfortable with the concepts of how the stack represents the current thread of execution. The first task of this project should be figuring out how to allocate and initialize a threads stack in order to allow it to be scheduled for the first time. A stack can easily be allocated using malloc (or preferably calloc), but it must then be initialized. This will entail setting up the stack to contain a valid execution context that can be "activated" by the context switching code. Remember that whenever a thread is created, the first thing it will do when it begins its execution is to make a function call to the function pointer passed in as an argument. This means that you will need to setup the stack such that it begins its execution in a way that looks like a function call was just made to the function. For my implementation I set `thread_invoker` to be the entry point for every thread. From there I called the thread's function and handled the exit path when that function returned. The trick is to figure out how to invoke `thread_invoker` correctly. For this you should look at the calling conventions for 64 bit Linux and try to figure out a clever way of setting the RIP to the correct value.

Thread exit paths will also be a source of difficulty. You will need to make sure that threads exit cleanly, and any joining threads are notified. The hardest part will be making sure you can safely free the thread's state (specifically its stack) after it has exitted. For this the run_state values will again be useful, as you can assume that any thread which has exitted can be set to the STOPPED state. It might be tempting to postpone freeing memory until after all threads have completed execution. However, this is not acceptable, and will result in grading penalties. Thread state should be freed when the thread it belongs to exits.

For thread scheduling you will not need any complex scheduling algorithm, basic round robin is fine. The scheduler should run a thread as long as there are threads left in the runqueue. If the runqueue has threads in it, but no thread is runnable (i.e. in the READY or RUNNING state), you can consider this an error and exit the program with an error code. Once the runqueue is empty, the scheduler should context switch back to the master thread which will result in returning from `pet_thread_run()`. Failure to cleanly exit the program by returning from `pet_thread_run()` after all the threads have executed to completion will be penalized.

I was able to implement `pet_thread.c` in less than 400 lines of code.

## Important Considerations

You will need to ensure that your library functions correctly for the entire API and for a wide variety of use cases. You will be responsible for developing these use cases as part of your implementation, as they will also be necessary to debug as you go along. For the grading of this project you will need to describe the use cases you develop, and demonstrate they function correctly. However, we will also use our own use cases as part of the grading.

Memory leaks are very easy to have when dealing with thread scheduling. We will be checking to make sure that you are correctly freeing all memory that is allocated during the program's execution. Correctly freeing memory is a large part of the complexity in this project, and you will need to spend time making sure you understand when and how it should be done.