**Creating a user-level thread package**

1. **Basic ideas:**

To implement a user-level thread package (the many-to-one model, specifically), several functions should be included, for example, thread creation, work queue creation, scheduling, and thread termination.

Because it is a many-to-one thread package, the CPU would serve only one thread each time. When switching between different threads, the first thing is to switch context, as each thread has its own stack to maintain code segment, local variables and so on. In my project, I use a structure called ucontext\_t to maintain each thread’s statues. When switching between threads, swapcontext() or setcontext() will be called to switch context. By changing context, the current running thread is switched, too.

1. **Design:**

*class tQueue*:

this class use a standard C++ class, *queue*,as a base class to build a higher level thread queue to record all needed threads. The currently running thread is the first element in this queue. When a thread is created, it should not run immediately, instead, it should be added to the tail of the queue and wait for CPU.

*int uthread\_init()*

The library should be initialized first, using this function. When initializing the library, the thread queue should be initialized first and the main thread should be added to the thread queue. Get the current context using getcontext() and add it to the thread queue. Off course, duplication detection should be implemented, too.

*int uthread\_create(void (\*func)(int), int val, int pri)*

When the main thread creates worker threads, this function would be called. The argument *func* is the entry of the new thread, and val is the argument of *func.* A new context variable should be allocated and initialized (stack size, entry and so on). After successfully created the new thread, it would be added to the tail of the queue.

*int uthread\_yield(void)*

When a “yield” function was called, the current thread is relinquished from CPU, and the next thread would be switched to running mode (it should be depended on priority, but in this project, we use FCFS). It is easy to do so, just move the current thread to the tail of the queue, and switch to the first ready thread’s context, and the changing process is done.

*void uthread\_exit(void \*retval)*

The uthread\_exit() function should be the only way a thread should exit. When this function is called by a specific thread, this thread should be deleted from the queue and never returns. Then the CPU should move to next ready thread. When there’s no other ready thread, the whole process should be terminated using exit().

1. **Evaluation and problems:**

The package works well in Debian environment. But there’s several problems at this point.

1. After a thread is terminated, I just deleted it from the work queue, so that the CPU would not work on it anymore. However, I didn’t clean the stack immediately, which is a waste of memory space.
2. Distinguishing different threads by its thread ID is easy, but I didn’t implement this part (thread ID), because I do not think that’s necessary.
3. I built the whole thing using C++, so the compile command should be different with the one published on collab. (Using this please “g++ test.cpp uthread.cpp -o test”)