CSE 536: Advanced Operating Systems

Assignment 3: User-Level Thread Management

Deadline: 11:59 PM on March 29, 2023

(100 points)

1 Brief Description

In this assignment, you will implement user-level threads (also called self-threads) for xv6 processes and make scheduling decisions inside the process based on different policies.

An xv6 process starts with only one *kernel-supported* thread. Within the process, we will divide this thread into several user-level threads. We will maintain one of these user-level threads as a *user-level scheduler thread*. This thread will make decisions regarding which user-level thread should execute at a certain time, based on different scheduling algorithms that we will choose.

GitHub Repo. Please clone as follows:

git clone -b lab3-thread https://github.com/ASTERISC-ASU/cse536-release.git

Important note

- **If your assignment does not compile or your upload is corrupted, you will get a zero.** Always double-check your submission.
- **Submit a zip file titled by your ASU username**. For instance, if your username is adil, your file should be adil.zip.

2 User-Level Threading Library (ULTLib) (100 points)

This library will create user-level threads for a process, schedule the threads to be executed, yield the CPU from a certain thread after a period of time, and destroy user-level threads when their task is completed. You must write the code for this library in the files: `ulthread.c`, `ulthread_swtch.S`, and `ulthread.h`.

Please follow the steps below to complete this task:

Library initialization. Important housekeeping must be done whenever a library is initialized. For this task, you must create a data structure (similar to `proc` kept by the kernel) to keep track of each user-level thread. You must decide on the contents of this data structure based on the information provided in this document. During initialization, also assign the first *kernel-provided* thread to be the *user-level scheduler thread*. Write the code for this part in the `ulthread_init()` function.

Thread creation. To create a thread, we need the following: (a) the starting function address where the thread begins execution, (b) initial arguments for the starting function, (c) the location of a stack that the thread will use during its execution, (d) the thread's priority, and (e) a context save location (`context`) where the registers of the thread will be saved and restored on context switches.

(a)–(d) will be provided by the user process when it calls the function `ulthread_create()`. However, the context save location must be created and maintained by the ULTLib. *Hint: check how the xv6 kernel maintains context save locations for kernel self-threads.* Once the thread is created, ULTLib should also track that the thread is now RUNNABLE for scheduling decisions.

Thread switch. The *user-level scheduler thread* must be called to schedule threads after creation and each time a user-level thread yields its CPU to a different thread. At these times, the *user-level scheduler thread* must (a) save its registers in it's `context` and (b) restore registers from the next scheduled thread's `context`. The thread whose registers are restored is chosen based on the scheduling policy.

Write an assembly function (`ulthread_context_switch()` in `ulthread.S`) that takes two pointers—previous context and next context—as arguments. The function must save the current thread's registers at the location of previous context, then load the registers from the next context's location. It must also set-up registers in such a way that function arguments are provided correctly when a thread starts.

Suggested reading(s)

RISC-V Function Argument Calling Conventions.

Thread yield and destroy. A user-level thread can give up its execution (using `ulthread_yield()`) and ask the *user-level scheduler thread* to schedule a different thread. Additionally, at the end of it's execution, a thread will explicitly call `ulthread_destroy()` to signal that it has completed its task. At these points, ULTLib should track that the thread's state has changed either to YIELD or FREE. Then, it should perform housekeeping tasks to maintain data structures, and finally switch back to the execution of the scheduler.

A user-level thread will decide when to yield based on clock timing information that it receives from the xv6 OS. To achieve this, you must write a new xv6 system call (`ctime`) that traps to the xv6 OS and returns the time from the RISC-V register `r_time`.

Thread scheduling decisions. The *user-level scheduler thread* decides which thread to schedule based on three algorithms: (a) round-robin, (b) first-come-first-serve, and (c) priority scheduling. The specific algorithm to use is provided during library initialization. Your goal is to write a scheduler that is aware of all these algorithms and decides the correct thread to execute whenever it is called based on the policy.

Testcases. You are only provided a single testcase for this assignment (namely `test1`). Please rigorously test your code using your own testcases for corner situations.

3 Miscellaneous

3.1 GIT Diff of Changes Required

Provided below is a sample of how many changes would be required to complete this assignment. Note that this would vary based on code styles.

```
kernel/syscall.c                  |   6 ++++++
kernel/syscall.h                  |   1 +
kernel/sysproc.c                  |  10 +++++++++-
user/ulthread.c                   | 217 +++++++++++++++++++++++++++++++++++++++++++++--------------------
user/ulthread.h                   |  34 +++++++++++++++++++++++++++---
user/ulthread_swtch.S             |  38 ++++++++++++++++++++++++++++++++
user/usys.pl                      |   3 ++-
```

3.2 Submitting your Assignment

Please zip the entire provided code directory and submit it to the Canvas under "Assignment 3: CODE".