ECE 434 Introduction to Computer Systems
Maria Striki Rutgers University Spring 2019
Project 3: 04-15-2019
Synchronization and User Level Threads: (60 = 30 + 30) points
Issue Date: Monday April 15th, Due Date: Saturday May 4th, 13.00

PART 1 (30 points)

In this project you are expected to implement your own thread library. Through your involvement you will gain experience with multi-threaded systems.

User Level Thread Library

For this part you will implement a cooperative User Level Thread (ULT) library for Linux that can replace the default PThreads libary. Cooperative user level threading is conceptually similar to a concept known as coroutines, in which a programming language provides a facility for switching execution between different contexts, each of which has its own independent stack. A very simple program using coroutines might look like this:

```
void coroutine1()
{
       // Some work
       yield(coroutine2);
}

void coroutine2()
{
       // Some different work
       if (!done)
                    yield(coroutine1);
}

int main()
{
       coroutine1();
}
```

Note that cooperative threading is fundamentally different than the preemptive threading done by many modern operating systems in that every thread must yield in order for another thread to be scheduled.

1.1 Basic User Level Thread Library

Write a non-preemptive cooperative user-level thread (ULT) library that operates similarly to the Linux pthreads library, implementing the following functions:

- mypthread_create
- mypthread_exit
- mypthread_yield
- mypthread_join

Please note that we are prefixing the typical function names with ”my” to avoid conflicts with the standard library. In this ULT model one thread yields control of the processor when one of the following conditions is true:

- thread exits by calling mypthread_exit
- thread explicitly yields by calling mypthread_yield

- thread waits for another thread to terminate by calling mypthread_join

You should use the functionality provided by the Linux functions setcontext(), getcontext(), and swapcontext() in order to implement your thread library. See the Linux manual pages for details about using these functions. These functions allow you to get the current execution context, make new execution contexts, and swap the currently running context with one that was stored.

So, what is a context? It is related to *context switches*—when one process (or thread) is interrupted and control is given to another. It's called a context switch because you are changing the current execution context (the registers, the stack, and program counter) for another. We could do this from scratch, using inline assembly, but the **getcontext, makecontext, and swapcontext functions** make it much simpler. For example, when getcontext is called, it saves the current execution context in a struct of type ucontext_t. The man page for getcontext describes the elements of this struct. Some of these elements are machine dependent and you don't have to worry about the majority of them. They may include the current state of CPU registers, a signal mask that defines which signals should be responded to, and of course, the call stack. The call stack is the one element of this struct that you will need to pay some attention to.

The current context must be saved first using getcontext (the new thread's context is based on the saved context). Space for a new stack must be allocated, and the size recorded. Finally, makecontext modifies the saved context, so that when it is activated, it will call a specific function, with the specified arguments.

The newly created context is then activated with either a call to setcontext or swapcontext. The former (setcontext) replaces the current context with a stored context. When successful, a call to setcontext does not return (it begins running in the new context). A call to swapcontext is similar to setcontext, except that it saves the context that was running (a useful thing to do, if you later want to resume the old context later). A successful call to swapcontext also does not return immediately, but it may return later, when the thread that was saved is swapped back in. You probably want to store your thread contexts on the heap.

1.2 Requirements

We will provide a test program (mtsort.c), and simple template (mythread.h). You may not modify the test program, so your library must provide exactly the API that we specify. You should implement all of your code in the provided files mypthread.h, and mypthread.c.

1.3 Coding and Submission Instructions

Coding

For this assignment, you should NOT have to create any other files, simply do your implementation in these existing files (use mythread.c, mythread.h, mtsort.c). Either provide a ReadMe file with instructions how to run this program or generate your own makefile.

Implementation

We provide a mypthread.h that defines the required API. You will need to make changes to the types defined here, but do not change the function call API. You will implement your library in mypthread.c. Once you complete your implementation, you may test it using the test program provided. The test program is implemented in mtsort.c.

Submission

Only one group member should submit the project, but the names of all group members should be entered into the text field on the Sakai submission, and should also appear on the report. A detailed report on justification of your coding and discussion is expected for your project to be complete.

PART 2 (30 points): Solving a Synchronization Problem

For Part 2 you are required to provide a synchronization scheme for the: **"Kindergarten"** Problem.

Regulations require that one teacher per R children is always present (e.g., child/teacher ratio 3: 1) to ensure proper supervision of children. Moreover, your configuration should have the following characteristics:

1. If a teacher attempts to leave but this is not possible, he should return in office (but NOT BLOCK while waiting for the exit condition to be met).

2. Every parent should be able to enter the kindergarten area to verify whether the regulation is met.

The children, the teachers, the parents are simulated with three sort of threads T_C, T_T, and T_P, and each thread type executes the following code:

```
void Teacher()
{
    for (;;) {
        teacher_enter();
        ...critical section...
        teach();
        teacher_exit();
        go_home();
    }
}
```

```
void Child()
{
    for (;;) {
        child_enter();
        ...critical section...
        learn();
        child_exit();
        go_home();
    }
}
```

```
void Parent()
{
    for (;;) {
        parent_enter();
        ...critical section...
        verify_compliance();
        parent_exit();
        go_home();
    }
}
```

Provide the synchronized processes: Teacher, Children, Parents, which satisfy the requirements above, by further defining functions:`_enter(), _exit()`, `verify_compliance()` shown below, or by replacing the latter functions with the proper synchronization code within: `Teacher(), Child(), Parent()`. You will also simulate functions: `teach(), learn(), verify_compliance()` as follows: either apply a for loop or a sleep function with an argument that will be varied during your experiments, and/or combine a sleep function with an actual check to verify compliance.

You are asked to implement the scheme above following two distinct synchronization strategies:

1) Use mutexes/locks, shared variables, and Semaphores only
2) Use mutexes/locks, shared variables, and Condition Variables only.

Your goal is to study which of the above strategies implements the defined framework without any problems. It may be one of the two or both. After your experiments and your theoretical study of both implementation report which strategy provides the correct or optimal implementation and why.

You will create N threads in total where N: a) 15, b) 100, c) 1,000 (experiment will all three values).

You are allowed to create N1 teachers, N2 children, and N3 parents, as long as N = N1 + N2 + N3.

In your experiments and tests you will be varying N1, N2, and N3 and test for combinations that work and for combinations that do not work. You should also vary the intermediate waiting times (e.g., in the sleep calls you will be using) and take notice if the value of the argument plays a role in the experiment or not. You will execute your program under any combinations, observe the results, use the proper printouts and you will document experiment results in your report.

NOTE: In your program you must pass variables: N, N1, N2, and R as arguments.

Questions:

Q1: (5 pts) As you are building your optimal synchronization strategy(ies) also study and observe how the lack of proper synchronization across threads can lead to disastrous results. Experiment with varying the initial parameters such as: N = 12, N2 = 7, N1 = 3, N3 =2, R = 2 or R = 3. This is just an example. Use more such sets of values. Observe the results, document them with printouts, print-screens, and your report. Also, justify analytically/theoretically such results.

Q2: (18 pts) Implement both strategies described (1. Only semaphores, 2. Only condition variables) and provide your code, your experiment sets, and the results of your implementation for both strategies, as described in the introduction above. Also, use some timer function and time the execution with both strategies. If both strategies give you correct results then discuss their optimality and superior performance, based on the following parameters: 1) complexity of code, 2) time of execution (you may run your code for larger N to observe timing differences in both executions: e.g., N = 10,000 or 1,000,000).

Q3: (7 pts) Study your synchronization schemes and your results in theory and justify them: comment if they are the expected results. Towards this direction, you should first investigate if your strategies have correct functionality and outcome – for example whether they lead to deadlock or not – for any sequence of execution across threads (with context switching).

Solution:

What to turn in:

- C files for each problem
- A makefile in order to run your programs.
- Input text file (your test case)
- Output text file (for your test case)
- **Report**: Explain design decisions. Also, please consider providing a very detailed report, as along with your C file deliverables, it corresponds to a substantial portion of your grade.

Logistics:

- For Project 3 please work in groups of 4-5 students.
- You are expected to work on this project using LINUX OS
- Make ONE submission per group. In this submission provide a table of contribution for each member that worked on this project.
- Only students that may be left without peers will be allowed to work in groups of 2 or 3.
- Do not collaborate with other groups. Groups that have copied from each other will BOTH get zero points for this project no matter which copied from another, and will also incur more substantial consequences.