Spring Term 2014

CS 439 - Principles of Computer Systems
Project 2

Assigned on: **Feb 03 2014**
Due by: **Feb 14 2014**

In this project you will learn how a user-level cooperative thread library can be implemented and how a scheduler+dispatcher manages threads.

This project is to be solved through *pair programming* in groups of two. Pair programming is a form of extreme programming and means that two persons work together on the same computer. One is driving (writing code) while the second person is observing, commenting, and making suggestions. The roles are switched every 30 minutes.

1 A Cooperative User-Level Thread Scheduler

We will implement a thread package as a library using a simple round-robin-based scheduler and `setjmp/longjmp` for thread context switching. It is cooperative, i.e., threads need to explicitly give up control (by calling thread_yield) periodically to allow other threads getting scheduled.

You will need to implement at least the following functions:

- thread_create
- thread_add_runqueue
- thread_yield
- thread_exit
- schedule
- dispatch
- thread_start_threading

Each thread should be represented by a `struct thread` which contains at least a function pointer to the thread's function and an argument of type `void*` which is passed to the function when the thread commences execution. The struct should furthermore contain a pointer to the threads stack and two fields storing the current stack and base pointer when the thread yields.

`thread_create` should take a function pointer and a `void*` arg as parameters. It allocates a `struct thread`, a new stack for this thread, and sets default values. It is important that the initial stack

pointer (set by this function) is at an address dividable by 8. The function returns the initialized structure.

`thread_add_runqueue` adds an initialized `struct thread` to the runqueue. Since we implement a round robin scheduler, you can maintain a ring of those structures for simplicity, e.g., by having a next field which always points to the next thread to be scheduled. A static variable (here called `current_thread`) points the thread which is currently scheduled and running.

`thread_yield` suspends the current thread by saving its context to the `struct thread` and calling the scheduler and the dispatcher. If the thread is resumed later, `thread_yield` resumes at the place in the function where yield was called.

`thread_exit` removes the calling thread from the ring, frees its stack and the `struct thread`, sets the `current_thread` variable to the next thread and calls dispatch.

`schedule` decides which thread to run next. This is trivial because of the design of the round robin scheduler.

`dispatch` prepares the scheduled thread for execution. It needs to save the stack pointer and the base pointer of the last thread (to its `struct thread`) restore the stack pointer and base pointer of the scheduled thread. This involves some assembly code which we will elaborate on in the discussion section. In case the thread has never run before the dispatcher may have to do some initializations. If the thread's function returns from execution the thread has to be removed from the ring and the next one has to be dispatched. This can be done by calling `thread_exit` explicitly from the dispatcher as this function already implemented the required behavior.

`thread_start_threading` initializes the thread library by calling schedule and dispatch. This function should by called by your main function (after having added the first thread to the runqueue). It should never return at least as long as there are threads in your system.

In summary creating and running a thread involves the following steps:

```
static void thread_function(void *arg)
{
    ...
    may create threads here and add to the runqueue;
    ...

    while(some condition, maybe forever) {
        do work;
        hread_yield();
        if (bla) {
            may call thread_exit();
        }
    }
}

int main(int argc, char **argv)
{
    struct thread *t1 = thread_create(f1, NULL);
    thread_add_runqueue(t1);
    ...
```

```
    may create more threads and add to runqueue here;
    ...
    thread_start_threading();
    //we shouldn't reach this line
    printf("\nexited\n");
    return 0;
}
```

2 Test you Threads Package

As a second step, implement a main function which creates several threads performing some operations, e.g., writing to the console.

Check if the expected behavior of concurrent execution matches the output.

You can download a simple `main.c` and a **skeleton** of `threads.h` from the course website. The skeleton provides the prototypes assumed by main.c (i.e., the API).

Turnin instructions

Submit your solution (individual source and header files plus documentation) using the `turnin` program on CS machines:

```
turnin --submit jrellerm 439_project2 <files>
```

You are expected to turn in at least the following files: `threads.h`, `threads.c` (your library implementation), `main.c` (an example application using your threads library), and a text file containing the names and CS IDs of the team members (see example below) as well as a log of the pair programming activity (time, duration, who drove, what was done).

```
Name 1
CS ID 1
Name 2
CS ID 2

02/04 10:00a-11:15a
30 min, Name 1 driving, implemented ...
30 min, Name 2 driving, tested ...
15 min, Name 1 driving, worked on ...

...
```