CS 370 Operating Systems

Programming Assignment 3

Multi-Threading

Lead TA: Hisan Naeem

Overview

In this project, you are to implement a thread library which mimics the interface of the pthread library available in C and runs in user space. The purpose of this project is to deepen your understanding of the mechanism and design trade-offs behind threads in operating systems. Additionally, practitioners in industry are sometimes required to implement their own thread library for various reasons: (1) absence of a pthread library on some embedded systems; (2) need for fine-grained control over thread behavior; and (3) efficiency. So far, you have interacted with APIs provided by the operating system in Assignments 1 and 2 (e.g., fork, exec and malloc), but in this assignment you are going to design an API yourself for the user to create and run multiple threads.

Learning Outcomes

In this assignment, you will learn:

- How threads work, each with their own context
- How multiple threads run on a single-core CPU
- How threads are scheduled, and different strategies we can use
- How semaphores are implemented to provide access control

User-level-threads

What is a user-level thread? In this project, it is an independent control flow that can be supported within a process, at the user-level. For example, thread A calls foo(), which calls bar(), thread B calls foo2(), while the main program is waiting for threads A and B to finish. Each thread needs a stack of its own to keep track of its current execution state. Instead of relying on a system library or the operating system to manage the stack and to coordinate thread execution, a user-level-thread is managed at the user-level, including memory allocation and thread scheduling.

API Description

A list of functions that the library exposes. For details, refer to the sections ahead

1. int create_thread(void (*callback))

   Creates a new thread and returns its id. Each thread has a stack size of 4096 Bytes and its program counter register points to the callback function. Each new thread, when created, is in the READY state and placed into the Ready queue.
2. void end_thread()

   This function ends a thread's callback function. See guidelines below for more info on this. Once a thread ends, its state is set to FINISHED and it is placed in the finished queue.
3. int self_id ()

   Returns the Id of the calling thread (which is also the running thread, as the call can only be made when it is running).
4. void sleep(int duration)

   It puts the calling thread into sleep for the given duration. The duration is in milliseconds.
5. void yield()

   A thread calls yield when it wants to voluntarily give up CPU control. When yield is called, the OS immediately schedules another thread and puts the calling thread into the READY state.

6. void block()

   It puts the calling function into the blocked state. A blocked thread does not waste CPU cycles, unlike busy-waiting. The thread is only unblocked when some criteria is met. In the meantime, the control is never given to this thread.

7. void join(int thread_id)

   It requires the thread id of the thread our current thread wants to wait for. The current thread is blocked until the thread that we are waiting for finishes. As the thread finishes, it unblocks any thread that is waiting for it.

8. void sem_init(struct Semaphore* sem, int value)

   Initializes a semaphore to the given value. If the value is 1, the semaphore is essentially reduced to a mutex lock.

9. void sem_wait(struct Semaphore* sem)

   When a thread calls this function, it passes the semaphore it wants to acquire. If sem value > 0, then the thread is given access to the semaphore. Otherwise, it is put into the blocked state, and will only be unblocked when another thread calls sem_post.

10. void sem_post(struct Semaphore* sem)

    When a thread wants to release a semaphore, it calls sem_post with the semaphore as the argument. As a thread leaves a semaphore, one of the blocked threads waiting on the semaphore are unblocked.

While reading the sections ahead, refer to the template code given. The struct definitions and comments will be helpful.

General Guidelines and Info

- **Threads**
  - Each thread is represented by a TCB (Thread control block). It has various properties related to its context and state. Refer to the code for more details
  - Each thread can assume one of the following states

        - READY
        - BLOCKED
        - RUNNING
        - FINISHED
    - When created, a thread is in the READY state.
    - When it has been scheduled by the scheduler, it is in the RUNNING state.
    - When it has been blocked (as a result of *join* or *sem_wait*) it is in the BLOCKED state.
    - When it ends, its state is set to FINISHED
- **Semaphores (used in task 6)**
    - Usage: Create a semaphore instance, then initialize it using the *sem_init* function. (C structs do not have constructors, that is why we use init functions to initialize instances).
    - *void sem_wait(struct Semaphore* sem)*

      checks the semaphore value

      If value > 0 then
      decrement value by 1
      else
      block this thread
    - *void sem_post(struct Semaphore* sem)*

      increment the value by 1
      if any blocked threads waiting on this sem then
      unblock any one of the threads

- **Scheduling**
    - There is a timer running that repeatedly calls the scheduler after a set interval of time.
    - A thread can either give up control voluntarily (yield), or the scheduler can pre-emptively take control away from it.
    - The system should work for any arbitrary number of threads.

- You must disable interrupts when performing something critical, and do not want the scheduler to stop the code midway. Consequently, you must also enable interrupts later, for the alarm interrupt to be able to run.
- Use the *switch_context* function as-is. It takes two threads; the running thread and the next thread, saves the context of the running thread and loads the context of the other thread. The new thread will resume what it was doing the last time there was a context switch.

- **Thread Suspension**
  - Sleep will be implemented via busy-waiting.
  - Semaphores will use the blocking mechanism.
  - You have been given a function called *getTime()* that returns a timestamp in milliseconds. This can be used to calculate time durations.
- **Functions**
  - For information about any system call (or any C library, for that matter), you can search the ***man pages***. (Google Search: *<function_name> man page)*
  - Functions you should read more about:
    - sigsetjmp
    - siglongjmp
    - sigaddset
    - sigemptyset
    - sigprocmask
  - The ***jmpbuf*** struct is the structure where the registers are stored. It supports 8 CPU registers. For our purposes, we will only be using 2 of those: The Stack pointer register, and the Program Counter register.
- **Thread Callback**
  - Due to the nature of the library, there is one limitation: The thread_callback functions **must** call ***end_thread()*** as their last command.

Otherwise, you will end up with Segmentation faults.

```
void callback_function() {
    /*
        Here goes the code for your thread's working
    */

    end_thread();
}
```

- **Main Function**
  - Your main function will change gradually. Initially, it should look something like this:

```
int main() {
    init_lib();
    create_thread(NULL, 1);

    /*
        Create all Worker threads here
    */

    timer_start();
    while(1);
}
```

  - ***init_lib()*** is the function that initializes the library. Any initialization you need should go here, e.g. Queues.
  - *create_thread(NULL, 1)* is used to create the thread corresponding to our main function
  - timer_start() starts the timer that calls the scheduler.
  - Initially, you will not have the *join* function implemented. Thus, to make the main function wait for the child threads, we need to use while(1)

- Once you have implemented the *join* function, your main function will change to this.

```
int main() {
    init_lib();
    create_thread(NULL, 1);

    /*
        Create all Worker threads here
    */

    timer_start();
    /*
        Use the join function to wait on threads
    */
}
```

- **Files**
  - Do all your work in thread.c and thread.h. DO NOT create any other files
  - You can make any helper functions you want, but DO NOT change existing functions unless told to do so.
  - These files will only work on a UNIX based OS, since POSIX libraries are not available in Windows.
  - **Commands**
    - Compiling your code: ***gcc main.c thread.c***
    - Running the executable: ***./a.out***

Tasks

Part A

1. Create a Queue data structure to store TCB*. This will be used to implement queues, such as the *ready* and *finished* queues. You also need to make use of these queues inside the *create_thread* and *end_thread* functions. Refer to the comments in these functions. **(10)**
2. Modify the *scheduler* function to implement a **round robin scheduler**. Once done, your program should be able to run two threads simultaneously, if your implementation is correct. **(10)**
3. Implement the *yield* and *sleep* functions. Sleep uses the busy-waiting strategy, and not the blocking one. **(10)**

At this point, you should make a copy of your thread.c file. Implement the following parts in the copied file, as you will be changing previously written code. SUBMIT BOTH FILES.

Part B

4. Modify your previously implemented scheduler to implement **priority scheduling**. You need to make use of the **priority** property of the **TCB. (20)**
    a. Modify the *create_thread* function to allow the user to pass priority as an argument.
    b. When a thread is chosen to be scheduled next, you need to increase the priority of the threads in the **ready queue** by 1. This can help us to avoid the problem of starvation.
    c. Do not increase the priority of the main thread. Keep its priority to be 1

    Hint: Implement a priority-based Queue for your ready queue.
5. Implement the *block* function. Then the *join* function. **(20)**
    a. You need to maintain a list of blocked threads.

   b. When a thread calls *join* on another thread, it blocks itself and stores its *thread_id* in the waiting_id property of the thread it is waiting on. When the other thread ends, it unblocks the thread waiting on it.

Once you have implemented join, replace while(1) in main with join. The main thread should wait for all the created threads to finish, before it can finish.

6. Implement the functionality of semaphores. Add fields to the Semaphore struct and use that struct for your functions. **(20)**
   a. *sem_init* initializes the semaphore with a value.
   b. *sem_wait* tries to acquire the semaphore. It thread blocks calling the thread if the value is 0. Each semaphore keeps track of the threads waiting on it.
   c. *sem_post* increments the counter and unblocks any one of the blocked threads waiting on this semaphore.
7. Run the test file (readerwriter.c) **(10)**
   a. It is a variant of the reader/writer problem. There can be multiple readers reading shared memory, but only one writer can modify it. Your output should reflect this. Only one writer can access the common variable at a time, and unless it releases the semaphore, no other thread can access this memory. Hence, if the writer yields after acquiring the lock, it is a waste of cpu cycles as all other threads will be waiting on that semaphore, and control will ultimately come back to it. Output can vary depending on which thread internally runs first. Sample output:

```
connor@DESKTOP-KLM42KT:/mnt/c/Users/HisanNaeem/Desktop/OS assignment 3/Assignment3$ ./a.out
Writer 2 modified c to 4
Writer 2 going to yield
Writer 2 back, after yield
Reader 21: read c as 4
Reader 3: read c as 4
Reader 5: read c as 4
Reader 7: read c as 4
Reader 9: read c as 4
Reader 11: read c as 4
Reader 13: read c as 4
Reader 15: read c as 4
Reader 17: read c as 4
Reader 19: read c as 4
Writer 4 modified c to 8
Writer 4 going to yield
Writer 4 back, after yield
Writer 6 modified c to 16
Writer 6 going to yield
Writer 6 back, after yield
Writer 8 modified c to 32
Writer 8 going to yield
Writer 8 back, after yield
Writer 10 modified c to 64
Writer 10 going to yield
Writer 10 back, after yield
Writer 12 modified c to 128
Writer 12 going to yield
Writer 12 back, after yield
Writer 14 modified c to 256
Writer 14 going to yield
Writer 14 back, after yield
Writer 16 modified c to 512
Writer 16 going to yield
Writer 16 back, after yield
Writer 18 modified c to 1024
Writer 18 going to yield
Writer 18 back, after yield
Writer 20 modified c to 2048
Writer 20 going to yield
Writer 20 back, after yield
connor@DESKTOP-KLM42KT:/mnt/c/Users/HisanNaeem/Desktop/OS assignment 3/Assignment3$
```

Submission

You will end up with two sets of **thread.c** and **thread.h**. In one set you will have solved tasks 1-3, and in the other one tasks 4-7. Follow this directory structure closely. (Part a is tasks 1 to 3, Part b is tasks 4-7).

```
<rollnumber>.zip
├── <rollnumber>_part_a
│   ├── thread.c
│   └── thread.h
└── <rollnumber>_part_b
    ├── thread.c
    └── thread.h
```

The assignment is relatively short, but tricky. It will take you some time to understand. So Start Early :))

For any queries, feel free to email me, or reach out on slack.

You have been given the red pill, and you now know multitasking is an illusion created by your CPU……