Simple User-Level Thread Scheduler

Check My Courses for Due Date

High-Level Description

In this project, you will develop a simple many-to-many user-level threading library with a simple first-come-first-serve (FCFS) thread scheduler. The threading library will have two executors – these are kernel-level threads that run the user-level threads. One executor is dedicated to running compute tasks and the other executor is dedicated for input-output tasks.

The user-level threads are responsible for running tasks. The tasks are C functions. Your threading library is expected to at least run the tasks we provide as part of this assignment. A task is run by the scheduler until it completes or yields. The tasks we provide here do not complete (i.e., they have **`while (true)`** in their bodies). The only way a task can stop running once it is started is by yielding the execution. A task calls **`sut_yield()`** for yielding (i.e., pausing) its execution. A task that is yielding is put back in the task ready queue (at the end of the queue). Once the running task is put back in the ready queue, the task at the front of the queue is selected to run next by the scheduler.

A newly created task is added to the end of the task ready queue. To create a task we use the **`sut_create()`** function, which takes a C function as its sole argument. When the task gets to run on the executor, the user supplied C function is executed.

All tasks execute in a single process. Therefore, the tasks share the process' memory. In this simple user-level threading system, variables follow the C scoping rules in the tasks. You can make variables local to a task by declaring them in the C function that forms the "main" of the task. The global variables are accessible from all tasks.

The **`sut_yield()`** pauses the execution of a thread until it is selected again by the scheduler. With a FCFS scheduler the task being paused is put at the back of the queue. That means the task would be picked again after running all other tasks that are ahead of it in the queue. We also have a way of terminating a task's execution by calling **`sut_exit()`**, which stops the execution of the current task like **`sut_yield()`** but does not put it back into the task ready queue for further execution.

So far, we described what happens with the compute tasks. The threading library has two executors (i.e., kernel-level threads). One is dedicated for compute tasks and is responsible for carrying out all that is described above. The other executor is dedicated for input-output (I/O). For instance, a task would want to send a message to outside processes or receive data from outside processes. This is problematic because input-output can be blocking. We have a problem when a user-level thread blocks – the whole program would stall. To prevent this problem, we use a dedicated executor for I/O. The idea is to offload the input and output operations to the I/O executor such that the compute executor would not block.

For instance, **`sut_read()`** would read data from an external process much like the **`read()`** you would do on a socket. The **`sut_read()`** can stall (delay) the task until the data arrives. With user-level threading, stalling the task is not a good idea. We want to receive the data without stalling the executor with the following approach. When a task issues **`sut_read()`** we send the request to a request queue and the task itself is put in a wait queue. When the response arrives, the task is

moved from the wait queue to the task ready queue and it would get to run in a future time. We have **`sut_send()`** to send data to the remote process. However, the send is non-blocking. To simplify the problem, we don't even wait for an acknowledgement of the sent data. We copy the data to be sent to local buffer and expect the I/O executor to reliably send it to the destination. Same way we also have **`sut_open()`** and expect the I/O executor to make the connection to the remote process without error. Error conditions such as remote process not found is not willing to accept connections are not handled. We assume all is good regarding the remote process and just open the connection and move to the next statement.

Overall Architecture of the SUT Library

The simple user-level threading (SUT) library that you are developing in this assignment has the following major components.

It has two kernel-level threads known as the executors. One of them is the compute executor (C-EXEC) and the other is the I/O executor (I-EXEC). The C-EXEC is responsible for most the activities in the SUT library. The I-EXEC is only taking care of the I/O operations. Creating the two kernel-level threads to run C-EXEC and I-EXEC is the first action performed while initializing the SUT library.

The C-EXEC is directly responsible for creating tasks and launching them. Creating a task means we need to create a task structure with the given C task-main function, stack, and the appropriate values filled into the task structure. Once the task structure is created, it is inserted into the task ready queue. The C-EXEC pulls the first task in the task ready queue and starts executing it. The executing task can take three actions:

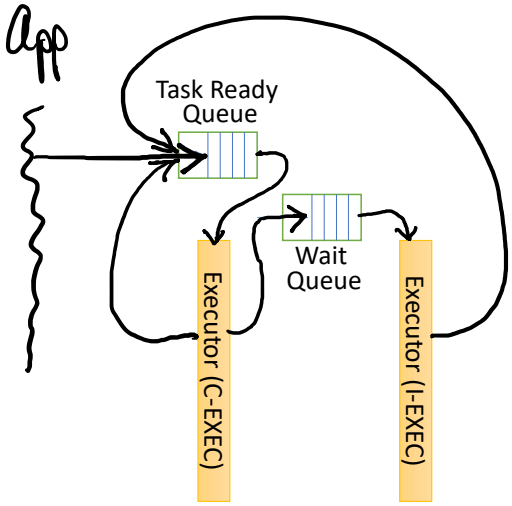
1. Execute **`sut_yield()`**: this causes the C-EXEC to take over the control. That is the user task's context is saved in a task control block (TCB) and we load the context of C-EXEC and start it. We also put the task in the back of the task ready queue.
2. Execute **`sut_exit()`**: this causes the C-EXEC to take over the control like the above case. The major difference is that TCB is not updated or the task inserted back into the task ready queue.
3. Execute **`sut_read()`**: this causes the C-EXEC to take over the control. We save the user task's context in a task control block (TCB) and we load the context of C-EXEC and start it. We put the task in the back of the wait queue.

When a task executes **`sut_open()`**, it sends a message to the I-EXEC by enqueuing the message in the To-IO queue. The message will instruct the I-EXEC to open a connection to the specified destination. In a subsequent statement, the task would issue **`sut_write()`** to write the data to the destination process. We would not wait for confirmation from open or write as a way of simplifying their implementations. The destination process we open for I/O is used by **`sut_read()`** as well. Because we assume a single active remote process there is no need to pass a connection handle like a file descriptor for the **`sut_read()`** or **`sut_write()`** call. We will have a **`sut_close()`** terminate the connection we have established with the remote process.

After the SUT library is done with the initializations, it will start creating the tasks and pushing them into the task ready queue. Once the tasks are created, the SUT will pick a task from the task ready queue and launch it. Some tasks can be launched at runtime by user tasks by calling the **`sut_create()`** function. The task scheduler, which is a very simple scheme – just pick the task

at the front of the queue, might find that there are no tasks to run in the task ready queue. For instance, the only task in the task ready queue could issue a read and go into the wait queue. To reduce the CPU utilization the C-EXEC will go take a short sleep using the nanosleep command in Linux (a sleep of 100 microseconds is appropriate). After the sleep, the C-EXEC will check the task ready queue again.

The I-EXEC is primarily responsible for reading the To-IO message queue and executing the appropriate command. For instance, I-EXEC needs to open a connection to the remote process in response to the **`sut_open()`** call. Similarly, it needs to process the **`sut_write()`** call as well. With the **`sut_read()`** call, I-EXEC get the data and needs to put the data in the From-IO message queue and transfer the task from the wait queue to the task ready queue. This transfer is done by I-EXEC because the task does not need to wait any more. The data the task wanting to read has arrived.



The SUT Library API and Usage

The SUT library will have the following API. You need to follow the given API so that testing can be easy.

```
void sut_init();
bool sut_create(sut_task_f fn);
void sut_yield();
void sut_exit();
void sut_open(char *dest, int port);
void sut_write(char *buf, int size);
void sut_close();
char *sut_read();
```

Context Switching and Tasks

You can use the **`makecontext()`** and **`swapcontext()`** to manage the user-level thread creation, switching, etc. The sample code provided in the **`YAUThreads`** package illustrates the use of the user-level context management in Linux.

Important Assumptions

Here are some important assumptions you can make in this assignment. If you want to make additional assumptions, check with the TA-in-charge (Jason) or the professor.

- The read and write tasks are launched by the application. We assume that there is only one outstanding read/write at any given time. With the read API provided in SUT, we can have multiple outstanding read calls. However, processing multiple outstanding read calls can be complicated. Therefore, we assume only one read or write call could be pending at any given time. That also means that the wait queue for I/O can have only one task at any given time (do we actually need a queue?).
- There are no interrupts in the user-level thread management to be implemented in this assignment. A task that starts running only stops for the three reasons given in Section 2.
- You can use libraries for creating queues and other data structures – you don't need to implement them yourself! We have already given you some libraries for implementing data structures.

Grading

Your assignment will be evaluated in stages.

1. Only simple computing tasks. We spawn several simple tasks that just print messages and yield. You need to get this working to demonstrate that you can create tasks and they can cooperatively switch among them.
2. Tasks that spawn other tasks. In this case, we have some tasks that spawn more tasks. In total a bounded (not more than 15 tasks) will be created. You need to demonstrate that you can have tasks creating other tasks at runtime.
3. Tasks that have I/O in them. We have some tasks write to a remote server. We don't mix write and read, just one type – write from the tasks to an external server.
4. Tasks that have I/O again. This time the tasks have read in them.