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 types of executors – the executors are kernel-level threads that run the user-level threads. One type of executor is dedicated to running compute tasks and the other type of executors is dedicated for input-output tasks.

The user-level threads are responsible for running the tasks, which 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 at the end of the task ready 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 types of executors. One type is dedicated for compute tasks and is responsible for carrying out all that is described above. The other type of executor is dedicated for input-output (I/O). For instance, a task would want to write a block of data to a file or read a block of data from a file. This is problematic because input-output can be blocking – that is the thread is blocking until the disk data is fetched. 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 an I/O executor such that the compute executor would not block.

For instance, **`sut_read()`** would read data from the disk much like the **`read()`** you would do. The **`sut_read()`** would not have the data it is trying to retrieve for a while because the disk is slow compared to the CPU. To prevent the whole application stalling, 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 that invoked the **`sut_read()`** 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_write()`** to write data to the file. We follow the same approach and make write non-blocking as well. The asynchronous writing offered by **`sut_write()`** need not give the same performance benefit as **`sut_read()`** because the writes could be just writing to the kernel buffer instead of the disk in the OS provided implementation. Same way we also have **`sut_open()`** and expect the I/O executor to make the connection to the file (i.e., open the file).

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. In the following discussion we assume that we have one compute executor (C-EXEC) and one I/O executor (I-EXEC). This restriction is relaxed in the last part of the assignment. 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, respectively 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 that can alter its state:

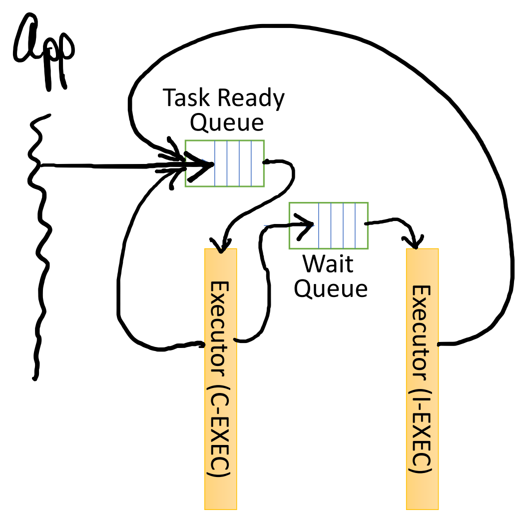
1. Execute **`sut_yield()`**: this causes the C-EXEC to take over the control. That is the user task's context is saved in a task control block (TCB) and we load the context of C-EXEC and start it. We also put the task in the back of the task ready queue.
2. Execute **`sut_exit()`**: this causes the 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_open()`**: 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, which would pick the next user task to run. We put the current task in the back of the wait queue. The I-EXEC would execute the function that would actually open the file and the result of the open would be returned by the **`sut_open()`** function. The result of the open is an integer much like the file descriptor returned by the OS. You can use the OS file descriptor or map it to another integer. If a mapping is done, a mapping table needs to be maintained inside the I-EXEC.
4. 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, which would pick the next user task to run. We put the current task in the back of the wait queue. The I-EXEC thread is responsible for reading the data from the file. Once the data is completely read and available in a memory area that is passed into the function by the calling task. That is, the I-EXEC does not allocate memory – it uses a memory buffer given to it. We assume the memory buffer provided by the calling task is big enough for the data read from the disk.
5. Execute **`sut_write()`**: this causes the C-EXEC to take over the control like the above calls. The I-EXEC thread is responsible for writing the data to the file. It uses the file

descriptor (fd) to select the file that needs to receive the data. The contents of the memory buffer passed into the **sut_write()** is emptied into the corresponding file.

6. Execute **sut_close()**: this causes the C-EXEC to take over the control like the above calls. The I-EXEC thread is responsible for closing the file. We need not have done file closing in an asynchronous manner – so it is done to keep all IO calls asynchronous.

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 just picks the task at the front of the queue and executes it. It 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 sleeps 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 processing all IO functions. It can use message queues to implement its functionality. The actual details of how I-EXEC should implement its operations are left for you as a design exercise. Once the function execution is complete, the I-EXEC puts the task back in the ready queue so it can proceed with the computations.



The **sut_shutdown()** call is responsible for cleanly shutting down the thread library. We need to keep the main thread waiting for the C-EXEC and I-EXEC threads and it one of the important functions of **sut_shutdown()**. In addition, you can put any termination related actions into this function and cleanly terminate the threading library.

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();
int sut_open(char *dest);
void sut_write(int fd, char *buf, int size);
void sut_close();
char *sut_read(int fd, char *buf);
void sut_shutdown();
```

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. The intention of the sample code is to illustrate how you can implement user-level threads – it is not a starter code. You are given full permissions to reuse portions of the sample code as appropriate.

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 (Zhelin) or the professor.

- 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 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.

Part A: One C-EXEC and one I-EXEC

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 30 tasks) will be created. You need to demonstrate that you can have tasks creating other tasks at runtime.
3. Tasks that have read I/O in them.
4. Tasks that have read and write I/O again.

Part B: Two C-EXECs and one I-EXEC

1. Tasks that spawn other tasks (a total of 30 tasks) and read I/O tasks.