Simple User-Level Thread Scheduler

Check My Courses for Due Date

High-Level Description

In this project, you will develop a simple one-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 – one dedicated to running compute tasks and the other dedicated for input-output (IO) tasks. Each executor is a kernel-level thread and the tasks that run are user-level threads. Because the threading library provides IO tasks their own executor, they would not block compute 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. Each task, once created, is placed in a FCFS ready queue. The tasks we provide 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** or **terminating** itself. A task that is yielding is put back at the end of the ready queue and 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.

All tasks execute in a single process (i.e., the process has two kernel-level threads that run as executors which run the tasks). 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.

Recall that the threading library has two types of executors - one dedicated for compute tasks and the other type 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. To prevent this, 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 need not block.

When a task issues a read, we send the request to a request queue and the task that invoked it is also 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. A similar case would be for write to a file and closing/opening a 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). 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.

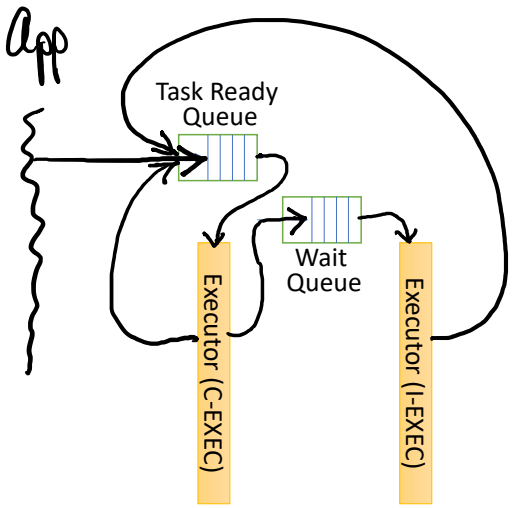
Creating a task is done by calling **sut_create().** We need to create a task structure with the given C task-main function, stack, and the appropriate values filled into the task structure. Once the task

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 these actions that can alter its state:

1. Yield **`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. Exit/Terminate **`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. Open file **`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. Read from file **`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. Write to file **`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. Close file **`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 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 I/O functions. The actual details of how I-EXEC should implement its operations are left for you to design. Once the I/O operation is done for a particular task, the I-EXEC puts the task back in the ready queue so that C-EXEC can proceed with it's computations.



Lastly, the library has one more function to close the entire thread library function. 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 to terminate (you can put any termination related actions into this function and cleanly terminate the threading library). The executors terminate only after executing all tasks in the queues.

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();
```

This call initializes the SUT library. Needs to be called before making any other API calls.

```
bool sut_create(sut_task_f fn);
```

This call creates the task. The main body of the task is the function that is passed as the only argument to this function. On success this returns a True (1) else returns False (0).

```
void sut_yield();
```

In SUT a running task can yield execution before the completion of the function by issuing this.

```
void sut_exit();
```

You call this function terminate the task execution. That is, if there are multiple SUT tasks running in the system, exiting a task using this call is just going to terminate that task.

```
int sut_open(char *fname);
```

This is an I/O function. You are requesting the system to open the file specified by the name. If there is no such file, you will get a negative return value. Otherwise, a non-negative value will be returned.

```
void sut_write(int fd, char *buf, int size);
```

We write the bytes in buf to the disk file that is already open. We don't consider write errors in this call.

```
void sut_close(int fd);
```

This function closes the file that is pointed by the file descriptor.

```
char *sut_read(int fd, char *buf, int size);
```

This function is provided a pre-allocated memory buffer. It is the responsibility of the calling program to allocate the memory. The size tells the function the max number of bytes that could be copied into the buffer. If the read operation is a success, it returns a non NULL value. On error it returns NULL.

```
void sut_shutdown();
```

Shuts down the executors completely and terminates the program.

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 (Akshay) 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.

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.