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 first-come, first-served (FCFS) thread scheduler. The library will have two types of executors: one for running compute tasks and another for input-output (IO) tasks. Each executor is a kernel-level thread, while the tasks that run are user-level threads. By providing IO tasks their own executor, compute tasks will not be blocked.

The tasks are C functions, and the library is expected to run the tasks provided in this assignment. A task is executed by the scheduler until it completes or yields. Once created, each task is placed in a FCFS ready queue. The provided tasks do not complete and have a **while (true)** loop in their bodies. A task can only stop running by yielding or terminating itself. A yielding task is placed 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, which means they share the process' memory. Variables in the tasks follow C scoping rules. Local variables can be declared in the C function that forms the "main" of the task, while global variables are accessible from all tasks.

The threading library has two types of executors: one for compute tasks and another for input-output (I/O) tasks. For example, a task might need to read or write data to/from a file. This can be problematic because I/O operations can be blocking, causing the thread to wait until the data is fetched from the disk. To avoid this, a dedicated executor is used for I/O. The idea is to offload I/O operations to the I/O executor, allowing the compute executor to continue running without being blocked.

When a task issues a read request, it is added to a request queue along with the corresponding task that invoked it. When the response arrives, the task is moved from the wait queue to the task ready queue, and it will be scheduled to run at a future time. Similar handling applies to write operations and file opening/closing.

Overall Architecture of the SUT Library

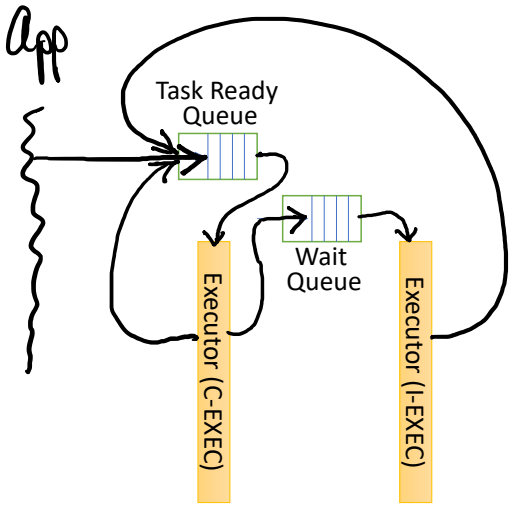
The simple user-level threading (SUT) library being developed in this assignment consists of several major components. In the following discussion, we assume the presence of one compute executor (C-EXEC) and one I/O executor (I-EXEC). The C-EXEC is responsible for most activities in the SUT library, while the I-EXEC handles I/O operations. During initialization of the SUT library, the first action is to create two kernel-level threads to run C-EXEC and I-EXEC, respectively.

Creating a task is done by calling **sut_create().** This involves creating a task structure with the specified C task-main function, stack, and appropriate values filled into the structure. Once the task structure is created, it is inserted into the task ready queue. The C-EXEC pulls the first task from the task ready queue and starts executing it. The executing task can take actions that can alter its state:

1. Yield **sut_yield()**: This causes the C-EXEC to take control. The user task's context is saved in a task control block (TCB), and the context of C-EXEC is loaded and started. The task is then placed at the back of the task ready queue.
2. Exit/Terminate **sut_exit()**: This causes the C-EXEC to take control, similar to the previous case. The major difference is that the TCB is not updated and the task is not inserted back into the task ready queue.
3. Open file **sut_open()**: This causes the C-EXEC to take control. The user task's context is saved in a TCB, and the context of C-EXEC is loaded and started, allowing the next user task to run. The current task is placed at the back of the wait queue. The I-EXEC executes the function that opens the file, and the result of the open is returned by **sut_open()**. This result is an integer, similar to the file descriptor returned by the operating system. The OS file descriptor can be used, or it can be mapped to another integer using a mapping table maintained inside the I-EXEC.
4. Read from file **sut_read()**: This causes the C-EXEC to take control. The user task's context is saved in a TCB, and the context of C-EXEC is loaded and started, allowing the next user task to run. The current task is placed at the back of the wait queue. The I-EXEC thread is responsible for reading data from the file. The data is read into a memory area provided by the calling task. It is assumed that the memory buffer is sufficiently large to hold the data read from the disk.
5. Write to file **sut_write()**: This causes the C-EXEC to take control, similar to the previous calls. The I-EXEC thread is responsible for writing data to the file using the file descriptor (fd) to select the destination file. The contents of the memory buffer passed into **sut_write()** are written to the corresponding file.
6. Close file **sut_close()**: This causes the C-EXEC to take control, similar to the previous calls. The I-EXEC thread is responsible for closing the file. File closing is done synchronously in this case.

After the SUT library completes initialization, it starts creating tasks and pushing them into the task ready queue. Tasks can also be created at runtime by user tasks using the **sut_create()** function. The task scheduler may find that there are no tasks to run in the task ready queue. For example, the only task in the task ready queue may issue a read operation and enter the wait queue. To reduce CPU utilization, the C-EXEC sleeps briefly using the **nanosleep()** command (a sleep of 100 microseconds is appropriate). After the sleep, the C-EXEC checks the task ready queue again.

The I-EXEC is primarily responsible for processing all I/O functions. The specific implementation details of I-EXEC operations are left for you to design. Once an I/O operation is completed for a particular task, the I-EXEC puts the task back into the ready queue so that C-EXEC can continue its computations.



Finally, the library includes one more function to close the entire thread library. The **sut_shutdown()** call is responsible for gracefully shutting down the thread library. In this function, you can include any termination-related actions to cleanly terminate the threading library. It is important to keep the main thread waiting for the C-EXEC and I-EXEC threads to finish executing all tasks in the queues before terminating the executors.

The SUT Library API and Usage

The SUT library should provide the following API. To facilitate testing, your library implementation needs to adhere to this API.

void sut_init();

This function initializes the SUT library. It should be called before making any other API calls.

bool sut_create(sut_task_f fn);

This function creates a task with the given function as its main body. It returns True (1) on success, and False (0) otherwise.

void sut_yield();

This function allows a running task to yield execution before completing its function.

void sut_exit();

This function terminates the execution of a task. If there are multiple tasks running, calling this function will only terminate the current task.

- *int sut_open(char fname);*

This function requests the system to open the file specified by the name. It returns a negative value if the file does not exist, and a non-negative value otherwise.

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

This function writes the bytes in the buffer `buf` to the disk file that is already open. Write errors are not considered in this call.

void sut_close(int fd);

This function closes the file pointed to by the file descriptor `fd`.

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

This function is provided with a pre-allocated memory buffer. The calling program is responsible for allocating the memory. The `size` parameter specifies the maximum number of bytes that can be copied into the buffer. If the read operation is successful, a non-NULL value is returned. Otherwise, it returns NULL.

void sut_shutdown();

This function completely shuts down the executors and terminates the program.

Context Switching and Tasks

To manage user-level thread creation, switching, and more, you can utilize the **makecontext()** and **swapcontext()** functions. The **YAUThreads** package includes sample code that demonstrates the usage of user-level context management in Linux. It is important to note that the provided sample code serves as an illustration of implementing user-level threads and should not be used as a starting point. However, you have the freedom to reuse relevant portions of the sample code as needed.

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 provided you with some libraries for implementing data structures.

Grading

Your assignment will be evaluated in stages.

1. Simple computing tasks: We will spawn several tasks that perform basic computations, such as printing messages and yielding. Your task is 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 additional tasks. A total of 30 tasks, or fewer, will be created. Your goal is to demonstrate that you can have tasks creating other tasks during runtime.
3. Tasks with read I/O: These tasks involve reading input/output operations.
4. Tasks with read and write I/O: These tasks involve both reading and writing input/output operations.