User Level Thread Library Project

By:
Rohit Sindhu – sindh010
Aravind Alagiri Ramkumar – alagi005
Aparna Mahadevan – mahad028

Answers to Questions

A. What (additional) assumptions did you make?
Answer:

- The tid of the main thread will always be 0
- The system will never exceed the maximum number of threads (MAX_THREADS) allowed
- The TCB of a thread contains an extra field which is the tid of the thread on which it is waiting. (defaults to -1 if it is not waiting).
- There will be no zombie threads because in actuality only one thread runs. Once that thread ends, all the other "threads" we have created will also end and will not be childless.

B. Describe your API, including input, output, return value
Answer:

uthread_create
Input: void*(*start_routine)(void*) - a function pointer that takes in any number of arguments
void* arg - argument for start routine
Output: a new thread is created
Return Value: int tid (newly created thread's tid)
Description: uthread_create will create a new thread and put it in the ready queue (sch_queue). It takes a function pointer and arguments and use them to initialize a new thread TCB and move it to the ready state. It will initialize the stack pointer and program counter of the thread and store it in the thread's ucontext_t structure.

uthread_self
Input: void
Output: returns tid of currently running thread

uthread_yield
Input: void
Output: Swaps the context to a new thread
Return Value: void
Description: If the ready queue is not empty, swaps the context by calling swapContext (described in answer to c. below)

uthread_join
Input: int tid - ID of the thread we are joining on (i.e. waiting for)
void **retval - pointer to store the return value of the child if it is completed

Output: If the thread you are joining on is completed, then return and no longer wait. Otherwise suspend until woken up
Return Value: void
Description: Join checks the complete queue to see if the thread is completed or not. If it is, it will store the return value in the retval pointer. If not, it will check the ready queue and the suspend queue for that thread's tid. If the thread is in either of those queues, then the calling thread will suspend itself.

uthread_init
Input: int time_slice - the time to which the timer should be set
Output: Starts the timer at the specified time slice
Return Value: void
Description: Sets a signal handler to the timer handler code. Associates SIGVTALRM signal to the signal handler

uthread_terminate
Input: int tid - ID of the thread to be terminated.
Output: The thread to be terminated wakes up the parent thread that is joined(waiting) on it, and is stopped. A new thread is run.
Return Value: 0
Description: The tid of the thread to be terminated is put into the complete queue. The next element in the ready queue is chosen to run and the context is switched to the chosen thread's context.

uthread_suspend
Input: int tid - ID of the thread to be suspended.
Output: Suspend the thread and set the context to that of the next thread in the ready queue.
Return Value: 0 if there is a thread to run in the ready queue, -1 otherwise
Description: Save the context of the thread to be suspended in the global context array. Then place it in the suspend queue, and set the context to that of the next thread in the ready queue.

uthread_resume
Input: int tid - ID of the thread to be resumed
Output: Starts running the suspended thread
Return Value: 0 if the thread is suspended (and is ready to be resumed), -1 otherwise
Description: If the tid exists in the suspend queue, then remove it from the suspend queue. Set the context to the context of the thread whose ID is tid.

async_read
Input: int fildes – file pointer of the file being read
void *buf – the buffer in which the data (which is read from the file) will be written
size_t nbytes – number of bytes we will read from the file
Output: Reads the file and puts it in the buffer for the thread to consume
Return Value: Number of bytes read (-1 otherwise)

Description: async_read will issue an async IO using aio_read. It will poll and yield the calling thread until the read is complete.

C) Did you add any customized APIs? What functionalities do they provide? Answer:

We implemented swap context which is called by thread yield:
swapContext
Input: void
Output: The currently running thread is put at the back of the queue and the next element of the queue is chosen to run.
Return Value: void
Description: The currently running thread (run_TCB) is set to the ready state. Its context is saved in the global context array, and it is pushed to the back of the ready queue. We then run the next element of the ready queue by calling setcontext with the context of the next element in the ready queue.

D) How did your library pass input/output parameters to a thread entry function?
Answer:

When the thread is created, the function pointer (start_routine), arguments and the return pointer are saved inside the TCB. And when the thread is scheduled, we use the function pointer and arguments stored in the TCB to invoke the function from the stub. And when the function completes, it will be returned to the stub where we will store the returned value inside the TCB again.

E) How did you implement context switch? E.g. using sigsetjmp or getcontext?
Answer:

The context switch is implemented using getcontext and setcontext methods.

F) How do different lengths of time slice affect performance?
Answer:

If the time slice is too small, then there will be a lot of overhead because the threads will be switched very frequently and it may not be close to completing before it is switched. If the time slice is too large, then the round robin scheduler becomes redundant and become equivalent to FIFO. This is because the threads may fully complete within one time slice and the next thread in the ready queue will run.
Essentially, if time slice is very small, the response time of the large threads are very high. And if the time slice is very large, the response time of the smaller threads are very high if they are waiting behind a large thread.

G) What are the critical sections your code has protected against interruptions and why?
Answer:

1. Uthread_create -> modifies the ready queue.
2. Uthread_yield -> modifies the ready queue
3. Uthread_join -> modifies the completed queue
4. Uthread_terminate -> modifies the ready queue and completed queue
5. Uthread_suspend -> modifies the ready queue and suspend queue
6. Uthread_resume -> modifies the suspend queue and ready queue.

All the functions mentioned above needs to be protected against the interruptions because the timer may interrupt while making changes in the queue.

H) If you have implemented more advanced scheduling algorithm, describe them.
Answer:

Not applicable. It has the round robin scheduler.

I) How did you implement asynchronous I/O? Answer:

We implemented it through polling.

J) Does your library allow each thread to have its own signal mask, or do all threads share the same signal mask? How did you guarantee this?
Answer:

No. We call the sigemptyset, which will ensure that all signals are set to empty. And then we are registering a single timer signal for all the threads.