OS 2018

Ex2: User-Level Threads

Supervisor – Eytan Lifshitz

Due: 10.5.2018

Note: This exercise takes a lot of time. Start early!

As stated in the guidelines, the deadline will not be extended.

Part 1: Coding Assignment (90 pts)

Introduction

In this assignment you are required to deliver a functional *static* library, that creates and manages user-level threads.

A potential user will be able to include your library and use it according to the package's public interface: the uthreads.h header file.

Your task is to implement all the detailed functions, as explained below. You will probably find it necessary to implement internal functions and data structures. These should not be visible outside the library, as they are the private part of your implementation. You are not restricted in their number, signatures, or content.

You **are** required to write clear, readable and efficient code.

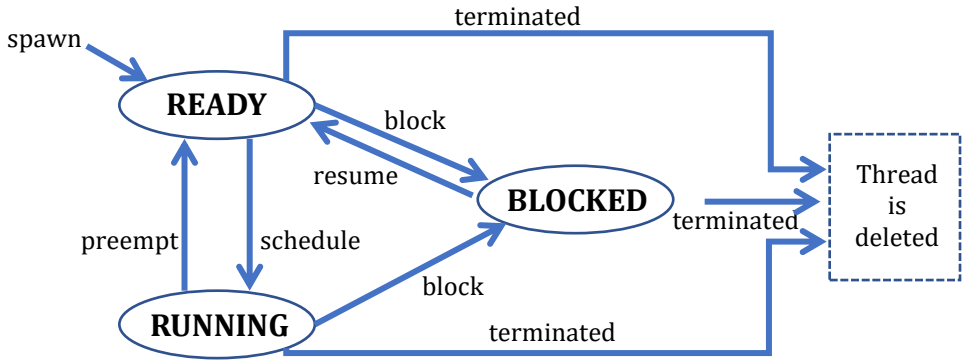
Remark: read the exercise description, the code examples and the man pages thoroughly, they will help make things clear.

The Threads

Initially, a program is comprised of the default main thread, whose ID is 0. All other threads will be explicitly created. Each existing thread has a unique thread ID, which is a non-negative integer. The ID given to a new thread must be the smallest non-negative integer not already taken by an existing thread (i.e. if thread with ID 1 is terminated and then a new thread is spawned, it should receive 1 as its ID). The maximal number of threads the library should support (including the main thread) is MAX_THREAD_NUM.

Thread State Diagram

At any given time during the running of the user's program, each of the threads in the program is in one of the states shown in the following state diagram. Transitions from state to state occur as a result of calling one of the library functions, or from elapsing of time, as explained below. This state diagram must not be changed: do not add or remove states.



Scheduler

In order to manage the threads in your library, you will need some sort of scheduling.
You will implement a Round-Robin (RR) scheduling algorithm.

States

Each thread can be in one of the following states: RUNNING, BLOCKED, or READY.

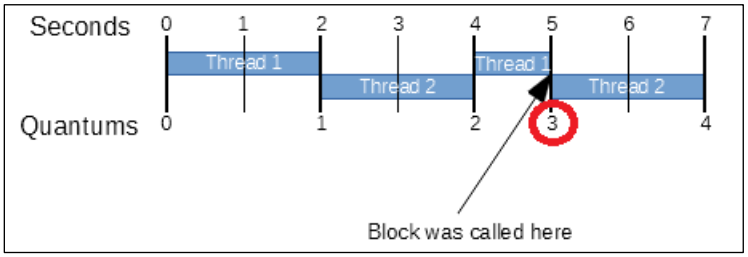
Time

Note: whenever we mention *time* in this exercise, we mean the running time of the process (also called the *virtual time*), and not the real time that has passed in the system.
The process running-time is measured by the Virtual Timer.
An example of using this timer can be found here.

Algorithm

The Round-Robin scheduling policy should work as follows:

- Every time a thread is moved to RUNNING state, it is allocated a predefined number of micro-seconds to run. This time interval is called a *quantum*.
- A thread is preempted if any of the following occurs:
    a) Its quantum expires.
    b) It changed its state to BLOCKED and is consequently waiting for an event (i.e. some other thread that will resume it or after a specific thread had terminated – more details below).
    c) It is terminated.
- Every time a thread moves to the READY state from any other state, it is placed at the end of the list of READY threads.
- If the RUNNING thread is preempted, do the following:
    1. If the RUNNING thread is preempted because its quantum is expired, move the preempted thread to the end of the READY threads list.
    2. Move the next thread in the list of READY threads to RUNNING state.
- Every time a thread is terminated, add all the threads that finished their sync dependency to the end of the READY threads list (in the same order they called the *sync function* with the terminated thread as an argument). Don't forget to change their states.
- When a thread doesn't finish its quantum (as in the case of a thread that blocks itself), the next thread should start executing immediately as if the previous thread finished its quota.
  In the following illustration the quantum was set for 2 seconds, Thread 1 blocks itself after running only for 1 second and Thread 2 immediately starts its next quantum.



- You are required to manage a READY threads list. You can use more lists for other purposes.
- On each quantum the READY top-of-list thread is moved to the RUNNING state.

Library functions

Following is the list and descriptions of all library functions. Calling these functions may result in a transition of states in the *state diagram* shown above. A thread may call a library function with its own ID, thereby possibly changing its own state, or it may call a library function with some other threads ID, thereby affecting the other thread's state.

int uthread_init(int quantum_usecs)

Description: This function initializes the thread library. You may assume that this function is called before any other thread library function, and that it is called exactly once. The input to the function is the length of a quantum in micro-seconds. It is an error to call this function with non-positive *quantum_usecs*.
Return value: On success, return 0. On failure, return -1.

int uthread_spawn(void (*f)(void))

Description: This function creates a new thread, whose entry point is the function f with the signature *void f(void)*. The thread is added to the end of the READY threads list. The *uthread_spawn* function should fail if it will cause the number of concurrent threads to exceed the limit (MAX_THREAD_NUM). Each thread should be allocated with a stack of size STACK_SIZE bytes.
Return value: On success, return the ID of the created thread. On failure, return -1.

int uthread_terminate(int tid)

Description: This function terminates the thread with ID *tid* and deletes it from all relevant control structures. All the resources allocated by the library for this thread should be released. If no thread with ID *tid* exists it is considered an error. Terminating the main thread (*tid* == 0) will result in the termination of the entire process using *exit(0)* (after releasing the assigned library memory).
Return value: The function returns 0 if the thread was successfully terminated and -1 otherwise. If a thread terminates itself or the main thread is terminated, the function does not return.

int uthread_block(int tid)

Description: This function blocks the thread with ID *tid*. The thread may be resumed later using *uthread_resume*. If no thread with ID *tid* exists it is considered an error. In addition, it is an error to try blocking the main thread (*tid* == 0). If a thread blocks itself, a scheduling decision should be made. Blocking a thread in BLOCKED state has no effect and is not considered an error.
Return value: On success, return 0. On failure, return -1.

int uthread_resume(int tid)

Description: This function resumes a blocked thread with ID *tid* and moves it to the READY state if it's not synced. Resuming a thread in a RUNNING or READY state has no effect and is not considered an error. If no thread with ID *tid* exists it is considered an error.
Return value: On success, return 0. On failure, return -1.

int uthread_sync(int tid)

Description: This function blocks the RUNNING thread until thread with ID *tid* terminates (i.e. right after *uthread_termiante(tid)* is called). It is considered an error if no thread with ID *tid* exists, if thread *tid* calls this function, or if the main thread *(tid*==0*)* calls this function. Immediately after the RUNNING thread transitions to the BLOCKED state a scheduling decision should be made.
Return value: On success, return 0. On failure, return -1.

int uthread_get_tid()

Description: This function returns the thread ID of the calling thread.
Return value: The ID of the calling thread.

int uthread_get_total_quantums()

Description: This function returns the total number of quantums that were started since the library was initialized, including the current quantum. Right after the call to *uthread_init*, the value should be 1. Each time a new quantum starts, regardless of the reason, this number should be increased by 1.
Return value: The total number of quantums.

int uthread_get_quantums(int tid)

Description: This function returns the number of quantums the thread with ID *tid* was in RUNNING state. On the first time a thread runs, the function should return 1. Every additional quantum that the thread starts should increase this value by 1 (so if the thread with ID *tid* is in RUNNING state when this function is called, include also the current quantum). If no thread with ID *tid* exists, it is considered an error.
Return value: On success, return the number of quantums of the thread with ID *tid*. On failure, return -1.

Simplifying Assumptions

You are allowed to assume the following:

1. All threads end with **uthread_terminate** before returning, either by terminating themselves or due to a call by some other thread.
2. The stack space of each spawned thread isn't exceeded during its execution.
3. The main thread and the threads spawned using the *uthreads* library will not send timer signals themselves (specifically SIGVTALRM), mask them or set interval timers that do so.

Error Messages

The following error messages should be emitted to *stderr*.
Nothing else should be emitted to *stderr* or *stdout*.

When a system call fails (such as memory allocation) you should print a **single line** in the following format:
"system error: *text*\n"
Where *text* is a description of the error, and then *exit(1).*

When a function in the threads library fails (such as invalid input), you should print a **single line** in the following format:
"thread library error: *text*\n"
Where *text* is a description of the error, and then return the appropriate return value.

Background reading and Resources

1. Read the following man-pages for a complete explanation of relevant system calls:
    - setitimer (2)
    - getitimer (2)
    - sigaction (2)
    - sigsetjmp (3)
    - siglongjmp (3)
    - signal (3)
    - sigprocmask (2)
    - sigemptyset, sigaddset, sigdelset, sigfillset, sigismember (3)
    - sigpending (2)
    - sigwait (3)
2. These examples may help you in your coding:
    - demo_jmp.c which contains an example of using *sigsetjmp* and *siglongjmp* as demonstrated in class. Note that you must use *translate_address* in your code as seen in the demo, otherwise your code will not work correctly.
    - demo_itimer.c which contains an example of using the virtual timer.

Part 2: Theoretical Questions (10 pts)

The following questions are here to help you understand the material, not to trick you. Please provide straight forward answers.

1. Describe one general use of user-level threads and explain why user-level threads are a reasonable choice for your example. (2.5 pts)

2. Google's Chrome browser creates a new process for each tab. What are the advantages and disadvantages of creating the new process (instead of creating kernel-level thread)? (2.5 pts)
3. Interrupts and signals:
    a. Open an application (for example, "Shotwell" on one of the CS computers). Use the "ps -A" command to extract the application's *pid* (process ID).
    b. Open a shell and type "kill *pid*"
    c. Explain which interrupts and signals are involved during the command execution, what triggered them and who should handle them. In your answer refer to the keyboard, OS, shell and the application you just killed

    (2.5 pts)

4. What is the difference between 'real' and 'virtual' time? Give one example of using each (2.5 pts).

Submission

Submit a tar file named ex2.tar that contains:

1. README file built according to the course guidelines. Remember to add your answers to the README file.
2. Source code (don't include *uthreads.h*).
3. Makefile - your makefile should generate a **static** library file named: *libuthreads.a* when running 'make' with no arguments.

Make sure that the tar file can be extracted and that the extracted files compile.

Guidelines

1. **Read the course guidelines**.
2. Design your program carefully before you start writing it. Pay special attention to choosing suitable data structures.
3. Do not forget to take care of possible signal races - protect relevant code by blocking and unblocking signals at the right places.
4. Encapsulate important actions such as performing a thread switch and deciding which thread should run next. Make sure each action works on its own before combining them.
5. Always check the return value of the system calls you use.
6. Test your code thoroughly - write test programs and cross test programs with other groups.
7. During development, use asserts and debug printouts, but make sure to remove all asserts and any debug output from the library before submission.
8. Verify that your program has **at most one memory leak** (which may be caused by the last terminated thread).

Late Submission Policy

Submission time	**10.5, 23:55**	**13.5, 23:55**	**15.5, 23:55**	**16.5, 23:55**	**17.5, 23:55**	
Penalty	**0**	**3**	**10**	**25**	**40**	**Course failure**

This exercise is **Difficult**. Start early!
Good luck!