EX2 - User-Level Threads

Due: 07.04.2022

Note: This exercise is complex.
Start early, and write code that you'll be happy to debug (not less miserable to, but actually happy).

Part 1: Theoretical Questions (10 pts)

The following questions are here to help you understand the material, not to trick you. Please provide straight forward answers.
All questions worth an equal amount of points.

1. Describe one general use of user-level threads and explain why user-level threads are a reasonable choice for your example.
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)?
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
4. What is the difference between 'real' and 'virtual' time? Give one example of using each.
5. Describe what the functions *sigsetjmp* and *siglongjmp* do.

Part 2: Coding Assignment (90 pts)

Introduction

In this assignment you'll create a *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 its public interface: the uthreads.h header file.
Your task is to implement all the functions that make up the uthreads API, according to their documentation. 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.

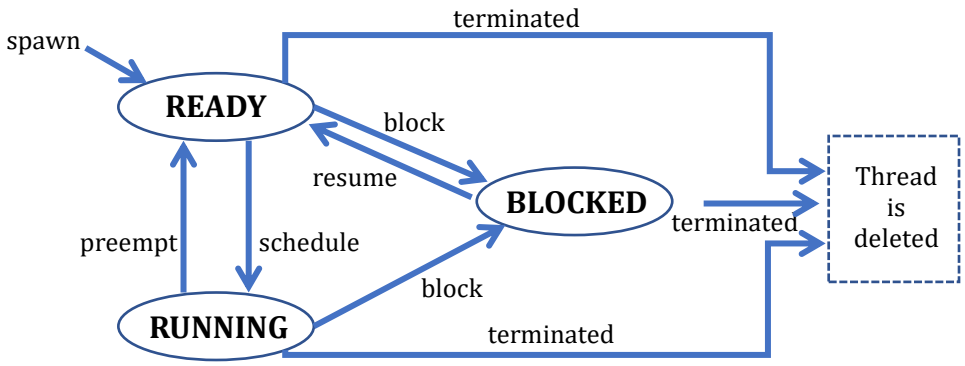
Remark: the "clear and readable code" requirement is for you, not for us.

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 thread 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.
The scheduler you'll create will implement the Round-Robin (RR) scheduling algorithm.

States

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

Time

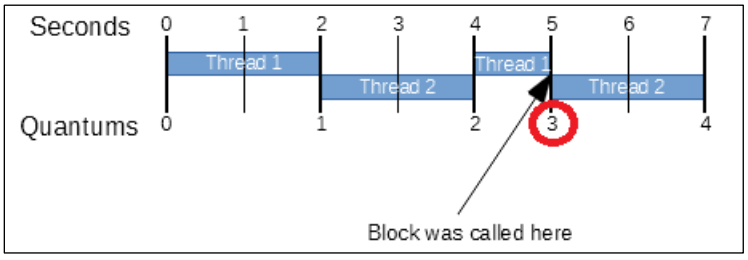
The process running time is measured by the Virtual Timer.
An example of using this timer can be found here.

Note: unless stated otherwise, 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.

Algorithm

The Round-Robin scheduling policy should work as follows:

1. 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*.
2. The RUNNING 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 some time has passed – more details below).
    c) It is terminated.
3. When the RUNNING thread is preempted, do the following:
    1. If it was preempted because its quantum has expired, move it to the end of the READY threads list.
    2. Move the next thread in the queue of READY threads to RUNNING state.
4. 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.
5. 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.



Library functions

The file uthread.h contains the API of the library. 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.

Notes

1. You are required to manage a queue, list, or a similar data structure of the READY threads. You can use more data structures for other purposes.
2. When a thread gets both blocked and is sleeping, it needs to wait for both the sleeping period to expire and another thread to resume it before going back to the READY threads list.
3. The main thread (tid == 0) uses the same stack, PC and registers that were used when **uthread_init** is called. There's no need to allocate a stack for it, or to set its SP and PC manually. Note that this doesn't mean it should behave differently when switching to / from this thread, but just that there is no need to worry about this aspect of its creation.
4. Valgrind doesn't work well with sigsetjmp and siglongjmp. **That's OK**. You are not required to configure Valgrind to solve these issues.
   You are, however, required to implement your code correctly, and with no memory issues / leaks.

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)
   - sigprocmask (2)
   - sigemptyset, sigaddset, sigdelset, sigfillset, sigismember (3)
   - sigpending (2)
   - sigwait (3)
2. These examples may help you in your coding:
   - demo_singInt_handler.c which contains an example of using signalaction.
   - demo_itimer.c which contains an example of using the virtual timer.
   - 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.

Submission

Submit a tar file named ex2.tar that contains:

1. README file built according to the course guidelines, with the answers to the theoretical questions.
2. All relevant source code files (except *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.

Late Submission Policy

Submission time	**14.04, 23:55**	**17.04, 23:55**	**18.04, 23:55**	**19.04, 23:55**	**20.04, 23:55**	**21.04, 23:55**
Penalty	**0**	**-3**	**-10**	**-25**	**-40**	**-100**

Good luck!