OS 2015

Ex2: User-Level Threads

Due to: 16/04/2015

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

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

Part I: Coding Assignment (90 pts)

Introduction

In this assignment you are required to deliver a functional *static* library, for user-level threads creation and management.

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.

You must include the original header file, and 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. But you **are** required to write clear, readable and efficient code.

remark: read the exercise description the code examples and the man pages thoroughly, it will help you to make things clear.

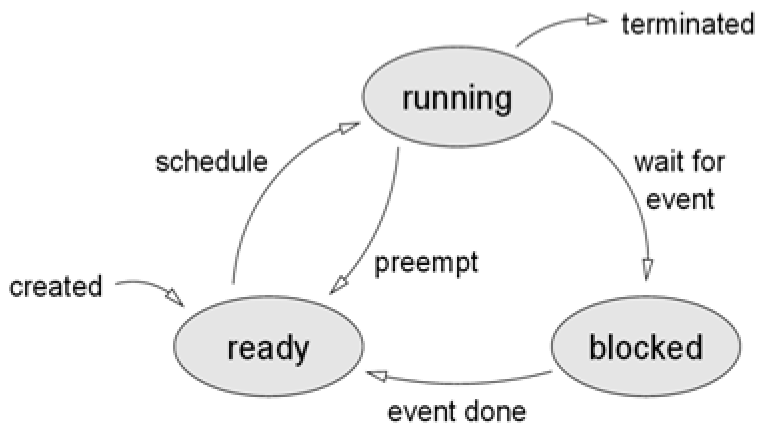
The Threads

Initially, a program comprises of a 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. The maximal number of threads the library should support (including the main thread) is MAX_THREAD_NUM.

Furthermore each thread will be provided with one of three given priorities (explained below). The main thread will be of ORANGE priority.

Threads 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 policy.
You will implement a simple expansion of the Round-Robin (RR) scheduling algorithm.
In order to decrease the obscurance level you will implement the following RR+ version, and only this.

Priorities

You will implement a slightly more complicated version of RR, which is an implementation of RR with a READY list being a priority queue.
To simplify things instead of implementing MAX_INT priorities, you will only implement 3 priorities: RED, ORANGE and GREEN, which define the highest priority mid priority and low priority respectively.
Repeat: RED - high, ORANGE - mid, GREEN - low.

Time

Note that; 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 in the resources below.

Algorithm

Our Round-Robin+ algorithm consists of three lists of threads (RUNNING, READY, BLOCKED) and a predefined time quanta.
With the READY list being a priority queue, with priorities as explained above.

The RR+ scheduling policy should be as follows:

- Every time a thread is moved to the RUNNING state, it is allocated a predefined number of microseconds to run. This time interval is called a quantum.
- A RUNNING thread is preempted (moved from being currently RUNNING) if any of the following occurs:
    a) Its quantum expires.
    b) It is blocked and consequently waiting for an event (some other thread that will resume it).
    c) It has terminated.
- If the RUNNING thread is preempted for any reason, the next thread in the list of READY threads is moved to the RUNNING state.
- 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 according to its priority.
  Which means that: when you put a thread in the ready queue it becomes the last one of its priority, meaning that in the queue the RED threads are always first, in the order that they entered the list, then the ORANGE threads in the order they entered, and last are the GREEN prioritised threads.
- There are no 'compensations' for quanta that were not fully used (as in the case of a thread that suspended itself).

- All the threads that are not RUNNING are currently held in the READY list according to their priority, or held in the BLOCKED list waiting for some other thread to resume them.
- Each quantum (a singular of quanta) the READY top-of-list thread is moved to the RUNNING list.
- If a thread is blocked (it waits for a resume event), it is moved to the BLOCKED list, either from the READY or the RUNNING lists.
  A thread in the BLOCKED list will wait until the event it is waiting for will be done (i.e. some other thread will resume it),; then it will move back to the READY list and will be inserted according to it's priority, i.e. it will be the last one of it's priority.

Library functions

Following is the list and description 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 thread's 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.
Return value: On success, return 0. On failure, return -1.

`int uthread_spawn(void (*f)(void), Priority pr)`
Description: This function creates a new thread, whose entry point is the function f with the signature `void f(void)`, And with priority pr (either RED, ORANGE or GREEN). The uthread_spawn function should fail if it would 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. Terminating the main thread (tid == 0) will result in the termination of the entire process using exit(0).
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. It is an error to try to block the main thread (tid == 0). If a thread blocks itself, a scheduling decision should be made. Blocking a thread in the 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. Resuming a thread in the RUNNING or READY state has no effect and is not considered an error.
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 that were started for the thread with ID tid, including the current quantum. 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.
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) 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 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, 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 done in the demo, otherwise your code will not work correctly.
    - demo_itimer.c which contains an example of using the virtual timer.

PART II: Theoretical Questions (10 pts)

The following questions are here to help you understand the material. We don't try to trick or fail you, so please provide a short straightforward answer and not a full thesis.

1. In this exercise you implemented a combination between RR and priority queue scheduling policies.
   Write down a con and a pro for each policy (RR and PQ), in regards of threads scheduling.
   Furthermore state how the cons\pros that you mentioned change or not in our combined policy (the RR+ you just implemented). [**No more than 5 lines**] (6 pts)
2. In real-life scheduling threads or processes are not simply blocked and resumed, they wait for an event and are blocked until it happens. Give a simple example of a user-level event that will imply blocking a thread (or a process). [**No more than 2 lines**] (4 pts)

Submission

Submit a tar file named ex2.tar, containing:
1. README file built according to the course guidelines. Remember to add your answers to the README file.
2. Source code.
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 do compile.

Guidelines

1. The general guidelines, as well as the coding guidelines regarding exercises are relevant to this exercise as well. **Read the 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 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.

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