EE442 HOMEWORK 2

User-Level Thread Scheduling

Due: May 5, 2019 23:59

Kamil SERT - EA405 ksert@metu.edu.tr

Submission

- Send your homework compressed in an archive file with name "**eXXXXXXX_ee442_hw1.tar.gz**", where X's are your **7-digit student ID number**. You will **not** get full credit if you fail to submit your project folder as required.
- Your work will be graded on its correctness, efficiency, clarity and readability as a whole.
- You should insert comments in your source code at appropriate places without including any unnecessary detail.
- Late submissions are welcome, but are penalized according to the following policy:
  - 1 day late submission : HW will be evaluated out of 70.
  - 2 days late submission : HW will be evaluated out of 50.
  - 3 days late submission : HW will be evaluated out of 30.
  - Later submissions : HW will NOT be evaluated.
- The homework must be written in **C** (**not** in C++ or any other language).
- You should **not** call any external programs in your code.
- **Check** what you upload. Do not send corrupted, wrong files or unnecessary files.
- The homework is to be prepared **individually**. Group work is **not** allowed.
- The design should be your original work. However, if you partially make use of a code from the Web, give proper **reference** to the related website in your comments. Uncited use is unacceptable.
- **METU honor code is essential**. Do **not** share your code. Any kind of involvement in cheating will result in a **zero** grade, for **both** providers and receivers.

Background:

Threads can be separated into two kinds: kernel-level threads and user-level threads. Kernel-level threads are managed and scheduled by the kernel. User-level threads are needed to be managed by the programmer and are seen as a single-threaded process from the kernel's point of view. The user-level threads have some advantages and disadvantages over kernel-level threads.

Advantages:

- User-level threads can be implemented on operating systems which do not support threads.
- Because there is no trapping in kernel, context switching is faster.
- Programmer has direct control over the scheduling policy.

Disadvantages:

- Blocking system calls block the entire process.
- In the case of a page fault, the entire process is blocked, even if some threads might be runnable.
- Because kernel sees user-level threads as a single process, they cannot take advantage of multiple CPUs.
- Programmer has to make sure that threads give up the CPU voluntarily or has to implement a periodic interrupt which schedules the threads.

For this homework, you will write a program which manages user-level threads and schedules them using a preemptive scheduler of your own. You will use `<ucontext.h>` to implement user-level threads.

Description:

The threads can be in three different states: ready, running, or finished (you can omit blocked state for this homework). You will need a global array which will store thread information; you can use the following C structure for the array elements:

```
struct ThreadInfo {
ucontext_t context;
int state;
}
```

`context` is the thread's context and `state` is the thread's status: ready, running, finished, or empty (if a thread has not been assigned to the array element yet). **The array should have a length of 5.** Reserve the first element for the context of the `main()` function.

In your `main()` function, create user-level threads. A newly created thread should be assigned to an empty spot in `ThreadInfo` array. If there is no empty spot, thread creation should wait for a thread to finish and an array spot to be emptied.

Implement the following functions:

initializeThread (), which initializes all global data structures for the thread. You need to define actual data structures but one constraint you have is to accommodate ucontext_t inside your structures.

createThread (), which creates a new thread. If the system is unable to create the new thread, it will return -1 and print out an error message. This function will be used for setting up a user context and associated stack. A newly created thread will be marked as READY when it is inserted into the system.

runThread (), which switches control from the main thread to one of the threads in the thread array, which also activates the timer that triggers context switches.

exitThread (), which removes the thread from the thread array (i.e. the thread does not need to run anymore).

scheduler (), which makes context switching (using `swapcontext()` using a preemptive and weighted fair scheduling structure of your choice (for example you can use lottery scheduling) with a switching interval of two seconds. A context switch should take place when the associated interrupt comes every two seconds. If a thread finishes, its place in the thread array will be marked as empty and the scheduler will free the stack it has used.

Hint: You may use `SIGALRM` signal (<signal.h>) for interrupt creation.

Each thread is required to execute a simple counter function that takes two arguments, "*n*" and "*i*", *n* being the stopping criteria for counting and *i* being the thread number. The function counts up starting from zero up to "*n*". With each increment, the function prints the count value having "*i*" tabs on its left. After each print, the function sleeps for 1 seconds.

Your program may take additional inputs from command line.

In the simplest case, "*n*" value for each thread should be provided. "*i*" values should be given in ascending order, i.e., first created thread will be given "x=1", second thread will be given "x=2", etc. Each thread's "*n*" value is also a measure of the thread's weight for using the CPU.

After creating all threads, `main()` function should simply wait in an infinite loop.

Specifications:

- Your program should be written in C.
- Use of `<pthread.h>` is not allowed.
- You should compile your code with GCC (GNU Compiler Collection).

<ucontext.h> example:

The following code shows an example usage of some of the functions defined in <ucontext.h>, namely, `getcontext()`, `makecontext()`, `swapcontext()`. Note that in this example context switching occurs in `main()` function after each thread returns. In your program, it should happen in the scheduler function.

```
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>

#define STACK_SIZE 4096

ucontext_t c1, c2, c3;

void func1(void) { printf("In func1\n"); }
void func2(int arg) { printf("In func2, argument = %d\n", arg); }

int main()
{
    int argument = 442;

    getcontext(&c1);
    c1.uc_link = &c3;
    c1.uc_stack.ss_sp = malloc(STACK_SIZE);
    c1.uc_stack.ss_size = STACK_SIZE;
    makecontext(&c1, (void (*)(void))func1, 0);

    getcontext(&c2);
    c2.uc_link = &c3;
    c2.uc_stack.ss_sp = malloc(STACK_SIZE);
    c2.uc_stack.ss_size = STACK_SIZE;
    makecontext(&c2, (void (*)(void))func2, 1, argument);

    getcontext(&c3);
    printf("Switching to thread 1\n");
    swapcontext(&c3, &c1);
    printf("Switching to thread 2\n");
    swapcontext(&c3, &c2);

    printf("Exiting\n");
    free(c1.uc_stack.ss_sp);
    free(c2.uc_stack.ss_sp);
    return 0;
}
```

Example output for the homework: (for a lottery scheduler)

```
~/hw3/$ ./schedule 6 4 4 8 10
    share:
     3/16        2/16      2/16      4/16       5/16

   threads:
      T1          T2        T3        T4         T5

       1
       2
                                                  1
                                                  2
                                                  3
                                                  4
                                        1
                                        2
                           1
                           2
                 1
                 2
       3
       4
                 3
                 4
                                        3
                                        4
                           3
                           4
                                                  5
                                                  6
                                                  7
                                                  8
       5
       6
                                        5
                                        6
                                                  9
                                                 10
                                        7
                                        8
```