EE442 HOMEWORK 3

User-Level Thread Scheduling

Due: May 7, 2017, 23:55

A. Doğa Hakyemez ARC-201 hdoga@metu.edu.tr

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 they 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 uses user-level threads and schedules them. You will use `<ucontext.h>` header 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 context and the `state` represents element status: ready, running, finished, or empty (if a thread has not been assigned to the element yet). **The array should have a length of 5.** The first element should be reserved for the context of the `main()` function.

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

Your scheduler function will make the context switching (using `swapcontext()` function) in a round-robin fashion with intervals of one second. When the interrupt comes, it will mark the running thread as ready, and the next ready thread will be marked as running. Then it will start/continue execution. If a thread is finished, its place will be marked as empty by the scheduler and the scheduler will free the stack they have been using. For the interrupt, you should use `alarm()` function. Your scheduler function should handle `SIGALRM` signal (using `signal()` function).

You will use `makecontext()` function to indicate the function (and its arguments) which is to be executed by the thread. Your threads will execute a simple function which takes two arguments, "n" and "x". Within a loop, starting from zero up to "n" it will print the count with "x" numbers of tab on the left. After printing each line, it should sleep for 100 milliseconds. Before exiting your threads should mark themselves as finished, and raise a `SIGALRM` signal so that the scheduler can empty their spot.

Your program should take additional inputs from command line. Each input will be the "n" value of a thread. You should create one thread for each of the additional input. "x" values should be given in ascending order, i.e., first created thread will be given "x" value of 1, second thread will be given 2, etc.

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

Specifications:

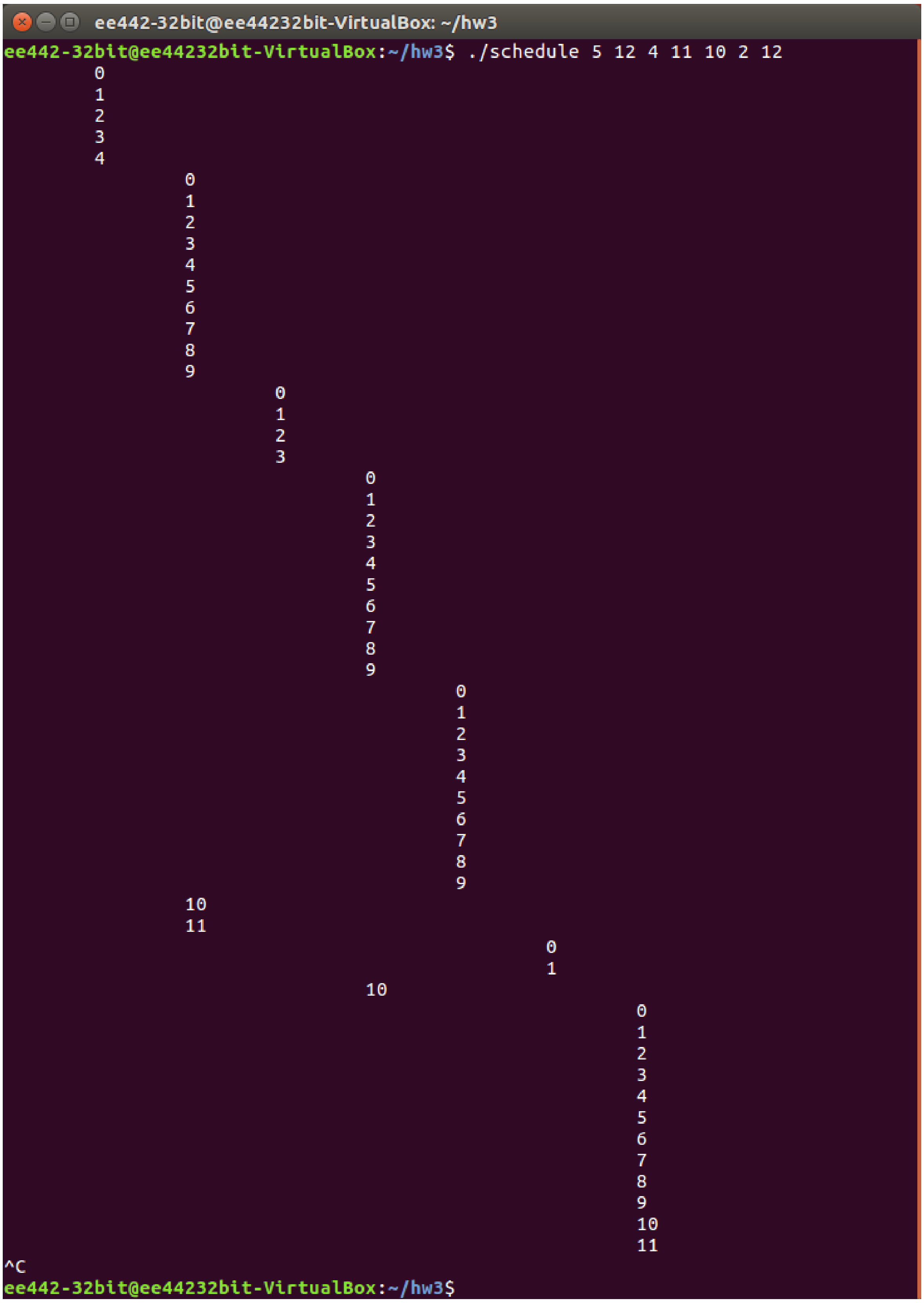
- Your program should be written in C.
- Do not use `<pthread.h>` header.
- You should compile your code with GCC (GNU Compiler Collection).

<u>`<ucontext.h>` example:</u>

The following code shows an example usage of some functions defined in <ucontext.h> header: `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.

```c
#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:



```
ee442-32bit@ee44232bit-VirtualBox:~/hw3$ ./schedule 5 12 4 11 10 2 12
        0
        1
        2
        3
        4
                0
                1
                2
                3
                4
                5
                6
                7
                8
                9
                        0
                        1
                        2
                        3
                                0
                                1
                                2
                                3
                                4
                                5
                                6
                                7
                                8
                                9
                                        0
                                        1
                                        2
                                        3
                                        4
                                        5
                                        6
                                        7
                                        8
                                        9
                10
                11
                                                0
                                                1
                                10
                                                        0
                                                        1
                                                        2
                                                        3
                                                        4
                                                        5
                                                        6
                                                        7
                                                        8
                                                        9
                                                        10
                                                        11
^C
ee442-32bit@ee44232bit-VirtualBox:~/hw3$
```

Remarks:

1. You should insert comments to your code at appropriate places without including any unnecessary detail. Comments will be graded. You have to write to-the-point comments in your code, otherwise it would be very difficult to understand. If your output is wrong, the only way we can grade your homework is through your comments.
2. Send your homework compressed in an archive file with the name "e<student_ID>_HW3" (e.g. e1234567_HW3.tar.gz). The archive file should include your **source file(s)** and **header file(s)** (if they exist).
3. Your work will be graded on its correctness, efficiency and clarity as a whole.
4. Late submissions are welcome, but 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.
    - 4 or more days late submission: HW will not be evaluated.

Good Luck!