EE442 PROGRAMMING ASSIGNMENT 2

User Level Thread Scheduling

Due: May, 28, 2023, 23:59

*** For your questions use forum or send email to** ksert@metu.edu.tr.

Submission

- Send your homework compressed in an archive file with name "**eXXXXXXX_ee442_pa2.tar.gz**", where X's are your **7-digit student ID number**. You will **not** get full credit if you fail to submit your work as required.
- Your work will be graded on its correctness, efficiency, clarity and readability as a whole.
- Comments will be graded. 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 40.
  - 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. Your code will be **checked** for cheating.
- 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 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 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, I/O (if a thread makes some I/O job) 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 a 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:

In this assignment you are going to schedule 7 threads (processors). Each process has 3 processor burst times and 3 I/O burst times.

[1st CPU][1st I/O][2nd CPU][2nd I/O][3rd CPU][3rd I/O]

We are going to assume that we have seven I/O servers (i.e. all processors can make I/O job concurrently) and a single device queue. When the 1st CPU execution is completed, the processor's state is changed from running to I/O and does 1st I/O job. After I/O is completed, the processor returns to the device queue and waits for grabing CPU for second CPU execution.

Scheduler 1: P&WF_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 three seconds. A context switch should take place when the associated interrupt comes every three 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.

Scheduler 2: SRTF_scheduler (), which implements "Shortest Reamining Time First Scheduling" algorithm. This algorithm makes context switching with a switching interval of three seconds. A context switch should take place when the associated interrupt comes every three 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.

** Scheduler functions must print status of all the threads after each interrupt signal.*

Each thread is required to execute a simple counter function that takes two arguments, "*n*" and "*i*", *n* being the CPU burst time for counting and *i* being the thread number. The function counts down starting from "n-1" down to "*zero*". With each decrement, 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 (**You should indicate how user can enter the input clearly!**). If you want, you can get input through .txt file.

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.
- Using `<pthread.h> library` 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 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.

```
#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;
}
```

Expected output for the homework:

Example: Lottery Scheduler, there are 5 threads having one i/o and one processor burst – first five arguments indicate processor bursts and the others indicate i/o bursts, switching interval is two seconds.

```
~/hw3/$ ./ P&WFscheduler 5 4 4 7 10 4 2 2 2 2
Share:
3/16  2/16  2/16  4/16  5/16

Threads:
T1    T2    T3    T4    T5
running>T1  ready>T2,T3,T4,T5 finished>               IO>
4
3
running>T5  ready>T1,T2,T3,T4 finished>               IO>
                        9
                        8
running>T5  ready>T1,T2,T3,T4 finished>               IO>
                        7
                        6
running>T4  ready>T1,T2,T3,T5 finished>               IO>
                  6
                  5
running>T3  ready>T1,T2,T4,T5 finished>               IO>
            3
            2
running>T2  ready>T1,T3,T4,T5 finished>               IO>
      3
      2
running>T1  ready>T2,T3,T4,T5 finished>               IO>
2
1
running>T2  ready>T1,T3,T4,T5 finished>               IO>
      1
      0
running>T4  ready>T1,T3,T5    finished>               IO>T2
                  4
                  3
running>T3  ready>T1,T4,T5    finished>T2             IO>
            1
            0
running>T5  ready>T1,T4       finished>T2             IO>T3
                        5
                        4
running>T5  ready>T1,T2,T3,T4 finished>T2,T3          IO>
                        3
                        2
running>T1  ready>T4,T5       finished>T2,T3          IO>
0
running>T4  ready>T5          finished>T2,T3          IO>T1
                  2
                  1
running>T5  ready>T4          finished>T2,T3          IO>T1
                        1
                        0
running>T4  ready>            finished>T1,T2,T3       IO>T5
                  0
running>    ready>            finished>T1,T2,T3,T5    IO>T4

running>    ready>            finished>T1,T2,T3,T4,T5 IO>
```