

# CS314-001 Operating Systems Programming Project #2 Description, Spring 2024

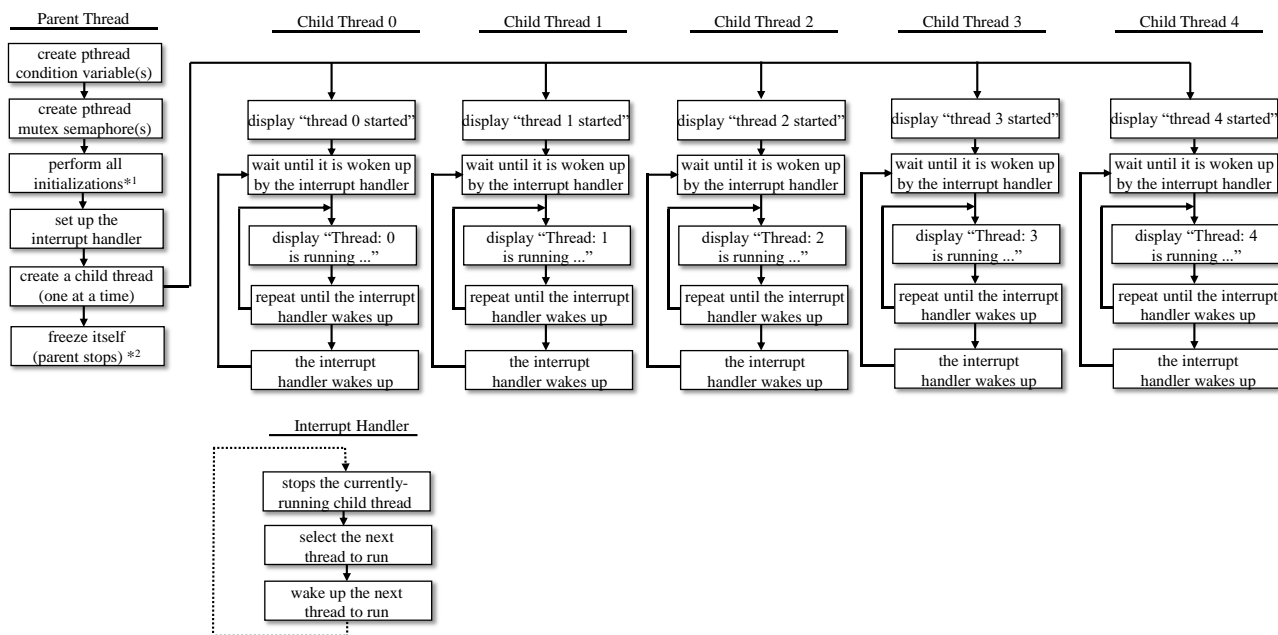
Project Due: 9:30 a.m. on April 16, 2024

## I. Objectives

This is the final programming project in this course. The objective in this programming project is to have practical experience for thread programming in UNIX operating systems, using "pthread". Pthread is a UNIX user-mode thread library and it is one of the most popular thread implementations for UNIX platforms. In this project, we control the execution timings of five threads using an interrupt handler. The interrupt handler manages (schedules) threads using "thread schedule vector", which is an array of integers to determine the pattern of thread executions. As we already discussed, user-mode threads are not preemptive by an operating systems. Because of the limitation, it is necessary for us to use semaphore(s) and a thread controlling mechanisms that are designed specifically for user-mode threads (i.e., "pthread mutex(es)" and "pthread conditional variable(s)").

## II. Requirements

- (1) Each submission should meet both "the output requirements (see the appendix for the required outputs)" and "the program structure requirements".
- (2) Figure 1 shows the required program structure for Project #2.



**Figure 1** - the required program structure for Project #2

- (3) The main (the parent) thread should create five child threads after it is started.
- (4) Project #2 requires the following structure (Figure 2) for each child thread primarily for testing purposes by the TA, which also posted to the course home.

```

#define SLOWDOWN_FACTOR    6000000    /* child thread display slowdown */

/* child thread implementation ----- */
void child_thread_routine (void * arg)
{
    int myid = *(int *) arg;    // child thread number (not ID)
    int my_counter = 0;        // local loop counter

    /* ----- */
    /* if you would like to add your local variable(s), you can */
    /* add one (them) here.                                     */
    /* ----- */

    /* declarare the start of this child thread (required) ---- */
    printf("Child thread %d started ...\n", myid);

    /* ----- */
    /* if you would like to perform some initialization to your */
    /* local variable(s), you can do so here.                   */
    /* ----- */

    /* infinite loop (required) ----- */
    while (1)
    {
        /* declare working of myself (required) ----- */
        my_counter ++;
        if ((my_counter % SLOWDOWN_FACTOR) == 0)
        { printf("Thread: %d is running ...\n", myid); }

        /* ----- */
        /* Here is your working space (to implement some mechanism) */
        /* to synchronize with the interrupt handler.                */
        /* ----- */
    }
}
/* END OF A CHILD THREAD ===== */

```

**Figure 2** – the required structure for each child thread

- (5) Figure 3 shows a sample of the child thread schedule vector (presented in the next page) required in each submitted source code file. The child thread schedule vector is implemented as an array of integers, where the first array element (i.e., “schedule\_vector[0]”) holds the ID of a child thread (0, 1, 2, 3, or 4) that should be executed first. Similarly, the second array element (i.e., “schedule\_vector[1]”) holds the ID of a child thread (0, 1, 2, 3, or 4) that should be executed second, and so on. The sample child thread schedule vector shown in Figure 3 will execute the five child threads in the order of thread #4, #3, #2, #1, and #0 (in the descending order of the child thread ID’s). The followings are the requirements about the child thread schedule vector:

- (a) As presented in “p3\_sample.c”, the child thread schedule vector should be declared at the beginning of each submitted source code file as a global variable. Any submission

without (or any submission that does not properly implement) the child thread schedule vector will not be graded.

- (b) The child thread schedule vector should be initialized by the parent thread before the parent thread starts creating five child threads.
- (c) Each submitted source code file will be tested by the TA after the TA changes the contents in the child thread schedule vector. Multiple different child thread scheduling patterns will be tested.
- (d) Each element in the child thread schedule vector can take any thread IDs (0 through 4). For example, the TA will test some cases where a child thread ID appears more than once in the child thread schedule vector.
- (e) The TA will NOT apply any thread IDs other than 0, 1, 2, 3, and 4.

```
/* initialize the schedule vector ----- */
schedule_vector[0] = 4;           // the first thread
schedule_vector[1] = 3;           // the second thread
schedule_vector[2] = 2;           // the third thread
schedule_vector[3] = 1;           // the fourth thread
schedule_vector[4] = 0;           // the fifth thread
```

**Figure 3** – a sample of the child thread schedule vector

(6) Other requirements

- (a) No spin-wait by the parent thread, all five child threads, and the interrupt handler.
- (b) The interrupt interval should be declared by a label, “#define SCHEDULE\_INTERVAL”, which specifies the interrupt interval in second (e.g., “#define SCHEDULE\_INTERVAL 1” means that the interrupt handler will be activated once in each second).
- (c) The outputs from all threads (including those from the interrupt handler) should follow the sample outputs posted to the course home. No output should be made except by the required outputs (i.e., the required *printf* in the required structure). Any deviation from this requirement may be considered as cheating.
- (d) Once a child thread is scheduled by the interrupt handler, no other child thread(s) should run (until the interrupt handler switches to another child thread).
- (e) When the interrupt handler wakes up, it should display:  
  
“I woke up on the timer interrupt (%d)”, where ‘d’ represents how many times the interrupt handler was activated so far (since the beginning of the program run). Please see the sample outputs posted to the course home.

- (f) The interrupt handler is the only subject that can wake up child threads. None of the five child threads should wake up (i.e., start running) other child thread(s). Figure 4 shows the required structure of the interrupt handler.

```
#define SCHEDULE_INTERVAL 1          /* scheduling interval in second */

void clock_interrupt_handler(void);

/* global variables ----- */
pthread_mutex_t condition_mutex; // pthread mutex_condition
pthread_cond_t t_condition; // pthread condition
int loop_counter; // loop counter for the interrupt handler

/* The main (the parent thread) ----- */
int main(int argc, char *argv[])
{
    int i = 0; // the loop counter for the parent thread
    loop_counter = 0; // initialize the loop counter for the interrupt handler

    /* specify the clock interrupt to be sent to this process --- */
    signal(SIGALRM, clock_interrupt_handler);

    /* set the interrupt interval to 1 second --- */
    alarm(SCHEDULE_INTERVAL);

    pthread_mutex_init (&condition_mutex, NULL);
    pthread_cond_init (&t_condition, NULL);

    /* infinite loop for the parent thread ===== */
    while (1)
    {
        printf("the parent thread is here (%d) ....\n", i);

        /* just for keeping the parent thread sleep forever ----- */
        pthread_mutex_lock(&condition_mutex);
        pthread_cond_wait(&t_condition, &condition_mutex);
        pthread_mutex_unlock(&condition_mutex);

        i = i + 1;
    }

    /* The main (parent) thread terminates itself ----- */
    return(0);
}

/* The interrupt handler for SIGALM interrupt ----- */
void clock_interrupt_handler(void)
{
    printf("I woke up on the timer interrupt (%d) .... \n", loop_counter);
    loop_counter = loop_counter + 1;

    /* scheduler wakes up again one second later ----- */
    alarm(SCHEDULE_INTERVAL);
}
```

**Figure 4** - the required interrupt handler structure

- (g) You can add (use) your own global variable(s). You can use as many pthread condition variables as you like.

### III. Testing by the TA

The TA will test each submission by changing the following parameters:

- (a) The interrupt interval “SCHEDULE\_INTERVAL”
- (b) The child thread schedule vector
- (c) The child thread display slow-down factor (“SLOWDOWN\_FACTOR”). In the sample, it is currently set to 6,000,000, but it can be changed (slowdown more or slightly make it less).
- (d) The number of the child thread will NOT be changed. Thus, it is OK for you to “hard code” the number of the child threads in your source code.

### IV. Required Submission

Program source code file (named as “p3\_ddd.c” where ‘ddd’ represents the last three digits of your SIUE 800- ID number (some of you are expected to specify four digits, instead of three)) should be submitted to Moodle by the deadlines.

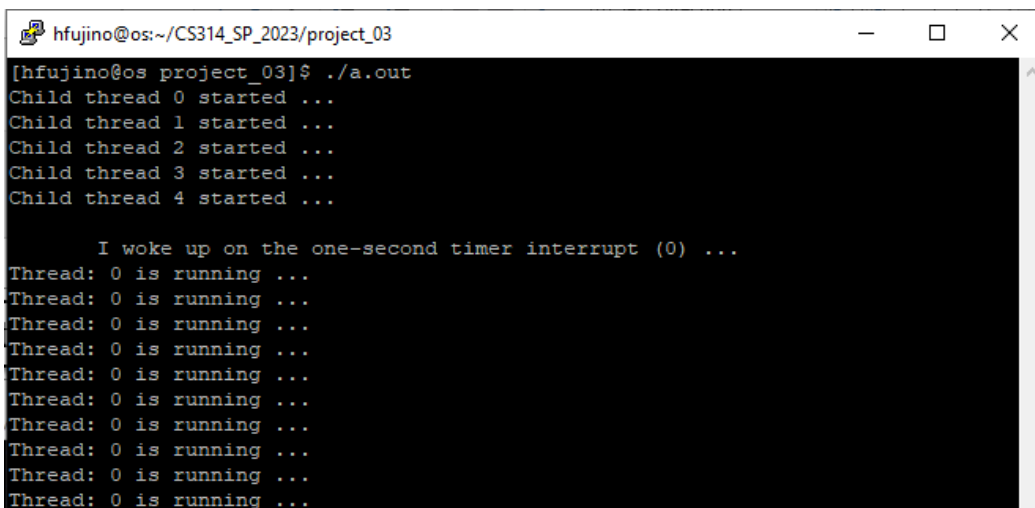
- (a) **Final submission deadline: 9:30 a.m., on April 16th (Tuesday)**
- (b) **Late submissions within 48 hours past the final submission due:** -10% for each 12 hours after the final submission deadline.
- (c) **Late submissions over 48 hours past the final submission due:** will not be accepted

### V. Other Requirements

- To be posted, if any.

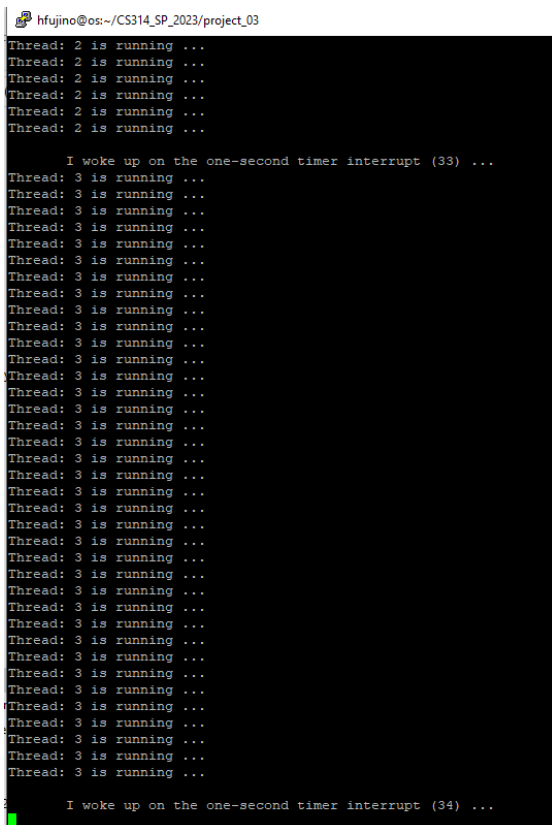
APPENDIX (the required outputs):

(a) Right after the program is started:



A terminal window titled 'hfujino@os:~/CS314\_SP\_2023/project\_03'. The prompt is '[hfujino@os project\_03]\$ ./a.out'. The output consists of five lines: 'Child thread 0 started ...', 'Child thread 1 started ...', 'Child thread 2 started ...', 'Child thread 3 started ...', and 'Child thread 4 started ...'. After a gap, the main thread prints 'I woke up on the one-second timer interrupt (0) ...'. This is followed by ten lines of 'Thread: 0 is running ...'.

(b) While the program (the five child threads and the interrupt handler) is in progress:



A terminal window titled 'hfujino@os:~/CS314\_SP\_2023/project\_03'. The output shows six lines of 'Thread: 2 is running ...'. Then, the main thread prints 'I woke up on the one-second timer interrupt (33) ...'. This is followed by 33 lines of 'Thread: 3 is running ...'. Finally, the main thread prints 'I woke up on the one-second timer interrupt (34) ...'.