System Programming 2019 Fall

Programming Assignment # 3

Due: 23:00 Tue, Dec 17, 2019

1. Problem Description

In this assignment, you need to simulate a user-thread library by using the longjmp(), setjmp(), etc. We use a function to represent a thread and there are 5 threads in this assignment. For simplicity, we use the term "function" instead of "thread" in the following description. The goal of this assignment is to practice how to use the system call to simulate user-thread library and use signal to control the process. You are required to do the following subtasks:

(1) Implement a scheduler to do the context switch between different function.
(2) Implement lock mechanism for different functions so they cannot access the same variable simultaneously.
(3) Use different signals to control the process.

2. Task 1 ---- Implement a Scheduler

In task 1, we need to implement a scheduler to do the context switch between different functions. First, we define the structure of Process Control Block (FCB)

```
typedef struct FCB_Node
{
        jmp_buf Environment;
        int Name;
        FCB_ptr Next;
        FCB_ptr Previous;
}FCB;
typedef struct FCB_Node *FCB_ptr;
```

FCB_ptr is a pointer which points to a FCB.
Each function can be represented as a FCB. Suppose there are 5 functions which are called funct_1, funct_2, funct_3, funct_4, and dummy, the schedule need to do the context switch between funct_1, funct_2, funct_3 and funct_4 only, not including the dummy function. There are four attributes in the structure of FCB. Environment is a jmp_buf variable and can be used to restore the environment

of the function. The scheduler will use longjmp(Environment,...) to switch to a specific function.
The next attribute of FCB is “Name” and its type is an integer. Name can be seemed as the ID of each function. For example, the “Name” of funct_1, funct_2, funct_3, funct_4, and dummy should be 1, 2, 3, 4, 5 respectively.
You need to construct a circular linked list by using the FCB of funct_1, funct_2, funct_3 and funct_4. Hence, the other two attributes of FCB are used to point to the next FCB or previous FCB of the linked list. Below shows the example of circular linked list.

funct_1 → funct_2 → funct_3 → funct_4 → funct_1 → …

Before constructing this circular linked list, you need to call funct_1, funct_2, funct_3 and funct_4. When these functions are called at the first time, you should call setjmp() in these functions, so the scheduler can longjmp to these functions afterwards. To initialize the stack frame correctly, you should call the functions in the following order:
main → dummy → funct_1 → dummy → funct_2 → dummy → funct_3 → dummy → funct_4 →→ main → scheduler

P/S: A → B means function A calls function B. A→→ B means function A longjmp to function B.

After initializing the stack frame as the order above, the scheduler should start calling longjmp to switch to the funct_1 which is also the head of the linked list. When funct_1 longjmp back to the scheduler, the scheduler should longjmp to the next function by following the circular linked list.

For funct_1, funct_2, funct_3 and funct_4, they need to append character ‘1’, ‘2’, ‘3’, ‘4’ respectively to a global array “arr”, which is char type. **Remember that the name of this global array is “arr” and you cannot change the name of this array**. The pseudo code of funct_1 is shown as below:

```
void funct_1(int name)
{
        int i, j
        // do the longjmp or setjmp
        for(j = 1; j <= P; j++)           // We call this for loop as “Big loop” in the
following description
        {
                // something else
                for(i = 1; i <= Q; i++)    // We call this for loop as “Small loop” in
the following description
                {
                        sleep(1);     // You should sleep for a second before append
to arr
                        arr[idx++] = '1'
                }
                //something else
        }

        // do the longjmp or setjmp
}
```

The pseudo_code of funct_2, funct_3, funct_4 should similar to funct_1.
We do not reveal the usage of dummy function here. It should be a question in your report. The pseudo code of dummy function is shown as below:

```
void dummy(int name)
{
        int a[10000];  // This line must not be changed.
        // call other functions
}
```

From the pseudo code above, we realize that the dummy function only declare a local variable and then call other functions. Remember that you should allocate a local variable (ex. a[10000] in the pseudo_code above) which needs a lot of space in the stack.

./hw3 5 3 1 0
You should execute your code by using the command above. There are 5 arguments in the command. The first argument is "./hw3", which is the filename of the executable file. The second argument is 5 which represents variable P in the above pseudo code. The third argument is 3 which represents variable Q in the above pseudo code. The fourth argument is 1 to specify this is task 1. The fifth argument is 0 and this argument is not used in task 1. So, you can pass a 0 to fifth argument.
We assume that the all of the arguments is valid. The fourth argument for task 1 will always be 1. We will change the value of second and third arguments to check the correctness of your code.

In task 1, the scheduler needs to longjmp to funct_1, funct_2, funct_3 funct_4 once and print out the content of the arr. According to the circular linked list, the scheduler should longjmp to funct_1 first, and funct_1 will append P*Q 1's to the arr. Then, funct_1 longjmp to scheduler and scheduler will longjmp to funct_2. Similarly, funct_2 will append P*Q 2's to the arr and jump back to the scheduler. So, the output of the command will be one line as follows

```
111111111111111222222222222222333333333333333444444444444444\n
```

There are fifteen 1’s are followed by fifteen 2’s. Fifteen 2’s are followed by fifteen 3’s. Fifteen 3’s are followed by fifteen 4’s.

3. Task 2 ---- Implement lock mechanism

Task 2 can be viewed as the extension of task 1. In this task, we need to implement lock mechanism in the task 1. A lock can be implemented by a global variable “mutex”.
For example, when funct_1 wants to append 1’s to the global array arr, it needs to make sure that the value of the mutex is equal to zero. If the value of the mutex is equal to zero, it means that there is no any function obtain the lock. So funct_1 can set the value of mutex to 1, and then access the global variable arr.
P/S: Remember that if the value of mutex is equal to zero, then funct_2, funct_3, funct_4 should set the value of the mutex to 2, 3, 4 respectively to get the lock. Release the lock by setting the value of mutex to zero.

If funct_1 finds out the value of mutex is equal to 4, this means that the lock is obtained by funct_4. Hence, funct_1 cannot get the lock and cannot access global variable arr. Then, funct_1 should longjmp to the scheduler, and the scheduler should jump to the next function according to the circular linked list.

./hw3 4 3 2 2

The meaning of each argument are mentioned in the task 1. We only explain the meaning of fifth argument in here. Fifth argument is 2, it means that when the function runs the small loop twice, the function should release the lock and jump back to the scheduler. Then, the scheduler will run to next function according to the circular linked list. Finally, print the content of arr and the output format should be one line as follows

```
111111222222333333444444111111222222333333444444\n
```

By knowing P=4, Q=3, we can know there must be twelve 1’s, 2’s, 3’s, 4’s in the arr. Because of the fifth argument is 2, the small loop is run twice before releasing the lock. So, the first ‘111111’ is the result of the small loop of funct_1. Then, funct_1 jump back to scheduler and scheduler longjmp to funct_2. Similarly, the small loop of funct_2 is run twice before releasing the lock. So, ‘111111’ is followed by ‘222222’. Due to P is equal to four, each function should run the small loop for four times totally. Funct_1 run the small loop twice again after the scheduler longjmp to funct_1 again.

4. Task 3 ---- Signal Control

In this task, we need to use signal to control the process. There are two processes in this task, “main” and hw3. “Main” is the parent and fork a child.

The child will execute ./hw3 10 3 3 0. The meaning of each arguments is already mentioned above. The fifth argument is not been used in this task, so you can pass the fifth argument as zero. “Main” and hw3 communicate with a pipe, hw3 write the message to stdout and “main” should be able to read the message from pipe.
Before explaining the details of this task, you should add

#define SIGUSR3 SIGWINCH

to your code.

There are three signals in this task which are SIGUSR1, SIGUSR2, SIGUSR3. SIGUSR1 is used to tell the scheduler to switch the function. SIGUSR2 is used to tell the function to release lock. SIGUSR3 is used to tell hw3 print the queue to the stdout and “main” read the message from pipe and print it out.

SIGUSR1:

For example, when funct_2 receives SIGUSR1, it should jump to the signal handler, the signal handler should jump to the scheduler. The scheduler should jump to the next function according to the circular linked list.

SIGUSR2:

For example, when funct_3 receives SIGUSR2, it should release the lock and jump to the signal handler. The signal handler should jump to the scheduler and the scheduler should jump to the next function according to the circular linked list.

SIGUSR3:

For example, when funct_4 receives SIGUSR3, it should jump to the signal handler. The signal handler should print the id of the functions which are in the queue to stdout. Then, The signal handler should jump to the scheduler and the scheduler should jump back to the same function instead of the next function.

The pseudo code of funct_1 is shown as below:
```
void funct_1(int name)
{
    int i, j
    // do the longjmp or setjmp
    while(1)
    {
        if the lock is available
        {
                for(j = 1; j <= P; j++)
```

```
                        {
                                // something else
                                for(i = 1; i <= Q; i++)
                                {
                                        sleep(1);
                                        arr[idx++] = '1'
                                }
                                // use sigpending and sigismember to check
                        which signal is pending
                                // if SIGUSR2 is pending, release the lock
                                // unblock signal
                        }
                }
                else
                {
                        //put funct_1 into the queue
                }
        }
        // release the lock
        // jump back to the scheduler (*)
        // do the longjmp or setjmp
}
```

For SIGUSR2, you must release the lock before unblocking the signal. The signals are blocked in the whole process, and you can ONLY check pending signal after each small loop. If there is pending signal, the signals are unblocked and the signals will be delivered . Before leaving the signal handler, the signal handler should call sigprocmask to block the signals again. The signal handler should send an ACK message to its parent so the parent knows the signal is delivered successfully.

The scheduler should delete a FCB block from circular linked list if the function is already run the big loop once. **Remember that the scheduler can only delete the FCB from circular linked list if the scheduler is jumped back from function but not the signal handler**.

When parent receive the ACK message from child, it should sleep for 5 seconds. When it wakes up, it will send a new signal to the child and read the ACK message from pipe again. Finally, the parent should print the content of arr which is sent by the child from pipe.
So the pseudo code of the parent is shown as below:

```
int main()
```

```
{
        // initialize
        scanf(); //read the input
        for 1 to number_of_signal
        {
                sleep(5);
                kill(); // send signal
                read() // read ACK message;
        }
        read() // read the content of arr
        waitpid()//
        close pipe
}
```

The input format of task 3 is

```
10 3
10
1 3 1 3 2 3 1 3 2 2
```

There are three line in the input. The first line has the two integers, which represents P and Q. The second line has one integer R, this integer specify how many signal will be sent by the parent to the child. The third line is the signal id and each signal id is separated by space. The signal id of SIGUSR1, SIGUSR2, SIGUSR3 are 1, 2, 3 respectively.

In the example above, there are 4 SIGUSR3 will be sent by the parent. So, the output will have 5 lines. The first 4 lines are the function id which is in the queue. Each function id is separated by a space. The fifth line is the content of arr.

We will show a sample execution for task 3 in the section below.

For all three task, $1 \leq P \leq 10$, $1 \leq Q \leq 5$, $1 \leq R \leq 10$. You can assume all of the arguments and input are valid. In task 3, you can assume that there are always have some functions in the queue when the process receive SIGUSR3. You can also assume that there is always a function get the mutex when the process receive SIGUSR2

5. Sample Execution

In this section, we will show a sample execution for each task. The green lines are the details of the execution while the red lines are the output that should be shown on the terminal.

Task 1: ./hw3 5 3 1 0

```
switch to funct_1
switch to funct_2
switch to funct_3
switch to funct_4
```

```
1111111111111111222222222222222233333333333333334444444444444444\n
```

Task2: ./hw3 4 3 2 2

```
switch to funct_1
append 3 1's to arr
append 3 1's to arr
funct_1 release lock
switch to funct_2
append 3 2's to arr
append 3 2's tp arr
funct_2 release lock
switch to funct_3
append 3 3's to arr
append 3 3's to arr
funct_3 release lock
switch to funct_4
append 3 4's to arr
append 3 4's tp arr
funct_4 release lock
switch to funct_1
append 3 1's to arr
append 3 1's to arr
funct_1 release lock
switch to funct_2
append 3 2's to arr
append 3 2's tp arr
funct_2 release lock
switch to funct_3
append 3 3's to arr
append 3 3's to arr
funct_3 release lock
switch to funct_4
append 3 4's to arr
append 3 4's tp arr
funct_4 release lock
111111222222333333444444111111222222333333444444\n
```

Task 3: ./main

Input:

```
5 3
5
1 3 2 3 1
```

Switch to funct_1
Append 6 1’s to arr
Receive SIGUSR1, jump to signal handler
Signal handler send ACK message to parent via pipe, then jump to scheduler from signal handler.
Switch to funct_2 (Unable to get the lock, put funct_2 in the queue)
Switch to funct_3 (Unable to get the lock, put funct_3 in the queue)
Switch to funct_4 (Unable to get the lock, put funct_4 in the queue)
Switch to funct_1
Append 6 1’s to arr
Receive SIGUSR3, jump to signal handler
Signal handler send the function id which are in the queue to parent via pipe, then jump to scheduler from signal handler.
Scheduler jump back to funct_1
Parent read the function id from pipe and print to the standard output. (print 2 3 4)
Append 3 1’s to arr
Jump to scheduler, switch to funct_2 (remove funct_2 from the queue)
Append 3 2’s to arr
Receive SIGUSR2, funct_2 releases the lock. Jump to the signal handler.
Signal handler send ACK message to parent via pipe, then jump to the scheduler and the scheduler switch to funct_3.
Funct_3 append 6 3’s to arr (remove funct_3 from the queue)
Receive SIGUSR3, jump to signal handler
Signal handler send the function id which are in the queue to parent via pipe, then jump to scheduler from signal handler. (print 4)
Scheduler jump back to funct_3
Parent read the function id from pipe and print to the standard output. (print 4)
Append 6 3’s to arr
Receive SIGUSR1, jump to signal handler
Signal handler send ACK message to parent via pipe, then jump to scheduler from signal handler.
switch to funct_4 (Unable to get the lock, put funct_4 in the queue)
switch to funct_2 (Unable to get the lock, put funct_2 in the queue)
switch to funct_3
append 3 3’s to arr
switch to funct_4
append 15 4’s to arr
switch to funct_2
append 12 2’s to arr

```
2 3 4\n
4\n
11111111111111222333333333333334444444444444422222222222\n
```

6. Grading

There are 3 subtasks in this assignment. You can get 7 points if you finish all of them. We will test your code on csie workstation. We will provide scheduler.o and scheduler.h for you to test your code.

The object file contains a scheduler function and the pseudo code is shown as below:

```
void Scheduler()
{
        // setjmp
        // if setjmp return a specific value, this means one of the function finish its big loop
        // so you need to remove the function from linked list or print the content of global array arr
        // switch to next function
        // longjmp to the function

}
```

1. **(2 point) Task 1.**

You need to write hw3.c and link hw3.c with our object file and header file. You must be able to compile and execute your program by

```
gcc scheduler.o hw3.c -o hw3
./hw3 5 3 1 0
```

2. (2 points) Task 2.

You need to write hw3.c and link hw3.c with our object file and header file. You must be able to compile and execute your program by

```
gcc scheduler.o hw3.c -o hw3
./hw3 4 3 2 2
```

3. (3 points) Task 3.

You need to write main.c, hw3.c and link hw3.c with our object file and header file. You must be able to compile and execute your programs by

```
gcc main.c -o main
gcc scheduler.o hw3.c -o hw3
./main
```

Bonus. (5 points) Report

These 5 points are optional in this assignment. You can try to submit a report to get the bonus. Page limit is 1 page and the format should be pdf. In this report, you must answer these question:

a. Draw the stack frame of funct_1, funct_2, funct_3, funct_4 and dummy. Can you use gdb or other methods to find out where is the stack pointer and base pointer of each function?
b. In funct_1, funct_2, funct_3, funct_4, if we declare a local variable and give it a value, is the value will be same when the scheduler jump back to the same function again? Please give the reason for your answer
c. Can you figure out what is the usage of the dummy function in this assignment?
d. If the scheduler switch to funct_4 and call return in funct_4, can the program follows this path and return to main?

funct_4 → dummy → funct_3 → dummy → funct_2 → dummy → funct_1 → dummy → main

Please give the reason for your answer and use gdb to prove your reason is correct.

e. Please briefly state how do you finish your program and something valuable you want to explain.

7. Submission

Your assignment should be submitted to CEIBA before the deadline, or you will receive penalty. At least 4 files should be included:

1.main.c

2. hw3.c

3. Makefile (as well as other *.c files)

4. readme.txt

Since we will directly execute your Makefile, therefore you can modify the names of .c files, but Makefile should compile your source code (bidding_system.c, host.c, player.c) into two executable files named main and hw3. In readme.txt, please teach us how to run your program in case we could not run your code successfully.

These files should be put inside a folder named with your student ID (in lower case) and you should compress the folder into a .tar.gz before submission. Please do not use .rar or any other file types. The commands below will do the trick. Suppose your student ID is b07902000:

```
$ mkdir b07902000
$ cp Makefile readme.txt *.c b07902000/
$ tar -zcvf SP_HW3_b07902000.tar.gz b07902000/
$ rm -r b07902000/
```

Please do NOT add executable files to the compressed file. Errors in the submission file (such as files not in a directory named with your student ID (in lower case), executable files not named correctly, and so on) may cause deduction of your credits. Submit the compressed file **SP_HW3_b07902000.tar.gz** to CEIBA.

8. Notes

The results about signal processing will be unstable if the workload of system is heavy. So, please run your program for multiple times to make sure your program can output the desired results.

9. Reminder

1. Plagiarism is **STRICTLY** prohibited.
2. Your credits will be deducted 5% for each day delay, but a late submission is still better than absence.
3. If you have any question, please feel free to contact us via email *ntucsiesp@gmail.com* or come to R302 during TA hours.
4. Please start your work ASAP and do not leave it until the last day!
5. **Good luck!**