# Group7_final_exam_report

1) (3%) Implement a single program to create three threads in the order of T1, T2, and T3. Then configure the threads properly so that `SIGINT`, `SIGTERM`, and `SIGUSR1` are exactly handled by T1, T2, and T3, respectively.

For question 1, so we declare three function called `T1_handler`, `T2_handler`, `T3_handler`, these three handler will do following things :

1. First, it will call `sigemptyset` , `sigaddset` and `pthread_sigmask` to initialize the signal set, add the specific signal to the set, and block the specific set (in here is `SIGINT`, `SIGUSR1` and `SIGTERM`)
2. Next, it calls the `sigwait` to wait for the required signal
3. After the function receives the required signals, it will print out the required word (e.g. T1 handling SIGINT) and terminate the thread.

```c
// T1 needs to handle the signal SIGINT
void *T1_handler(void *arg){
  sigset_t set;
  int sig;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  pthread_sigmask(SIG_BLOCK, &set, NULL);
  sigwait(&set, &sig); // block until a signal is received, and then call
  printf("T1 handling SIGINT\n");
  return NULL;
}
```

For the main function, we first use `sigfillset` and `pthread_sigmask` to block all of the signals. Next we call `pthread_create` to create T1, T2 and T3. Last, we calls the `pthread_join` to make sure that all of the threads will terminate properly

```c
int main(){

  sigset_t set;
  sigfillset(&set);
  pthread_sigmask(SIG_BLOCK, &set, NULL);

  // initialize the thread
  pthread_t T1, T2, T3;
  pthread_create(&T1, NULL, T1_handler, NULL);
  pthread_create(&T2, NULL, T2_handler, NULL);
  pthread_create(&T3, NULL, T3_handler, NULL);

  pthread_join(T1, NULL);
  pthread_join(T2, NULL);
  pthread_join(T3, NULL);
  return 0;
}      You, 43 minutes ago • update q1
```

Below is the execution output

```
mygodimato@RPi400Group7:~/Advanced-UNIX-Programming_Student/final_coding $ ./q1 &
mygodimato@RPi400Group7:~/Advanced-UNIX-Programming_Student/final_coding $ kill -USR1 %1
mygodimato@RPi400Group7:~/Advanced-UNIX-Programming_Student/final_coding $ T3 handling SIGUSR1
kill -INT %1
T1 handling SIGINT
mygodimato@RPi400Group7:~/Advanced-UNIX-Programming_Student/final_coding $ kill -TERM %1
T2 handling SIGTERM
```

---

2) (3%) Implement a function `sleep_us`, which is similar to sleep, but waits for a specified number of microseconds. You may build it upon `select` or `poll`. Instrument your code to measure whether your `sleep_us` returns in the precise time. Please record the time before and after your `sleep_us`, and print the elapsed (sleep) time in microseconds.

**Answer:** To run the file, we do:

```
make q2
./q2 1000000 # or any other required amount of time
```

We use the `select` function in our implementation of this sleep command.

The select function man page states the following:

```
Any of readfds, writefds, and exceptfds may be given as null poi
     no descriptors are of interest.
```

Hence, we can set the three middle arguments of `select` to `NULL` pointers in order to use it as a microsecond timer, targeting zero file descriptors.

The definition of `sleep_usec` is as follows:

```
void sleep_us(long int scs) {
    struct timeval tv;
    tv.tv_sec=scs/1000000;
    tv.tv_usec=scs%1000000;
    select(0,NULL,NULL,NULL,&tv);
}
```

we have to set both fields of the structure for select to work correctly.

Then, profiling the amount of time is done using `gettimeofday` since it returns elapsed time in microsecond precision.

```
struct timeval ct;
struct timeval nt;
gettimeofday(&ct, NULL);
sleep_us(scs);
gettimeofday(&nt, NULL);
long int fin = (nt.tv_sec-ct.tv_sec)*1000000l+(nt.tv_usec-ct.tv_
printf("Sleep time: %lld us\n",fin);
```

---

3) (4%) Use a single timer (either `alarm` or `setitimer`) to implement a set of functions that enables a process to set and clear any number of timers. Please print "*Alarm!*" when the alarm signal trigger. You don't have to implement extra code to parse arguments from the shell. We provide sample function prototypes below. Please use the code (in main function) to test your implementation, and include the results in the report.

When call `setAlarm` we fork another process (since there is only one alarm timer for each process).

Replace the signal handler by `sig_handler` . The handler print "Alarm" when signal `SIGALRM` comes and terminate the process when `SIGUSR2` comes (serve for `clearAlarm`).

After that, register an alarm timer by `alarm(sec)` and then do `pause()` . The child process is now waiting a signal comes.

```
 9 void setAlarm(int sec){
10     /* do something here ... */
11     if (fork() == 0){
12         if (signal(SIGALRM, sig_handler) == SIG_ERR)
13         {
14             printf("error: SIGALRM handler\n");
15             exit(0);
16         }
17         if (signal(SIGUSR2, sig_handler) == SIG_ERR)
18         {
19             printf("error: SIGUSR2 handler\n");
20             exit(0);
21         }
22     alarm(sec);
23     while(1)
24         pause();
25     }
26 }
```

In `clearAlarm` , we just send `SIGUSR2` to all process in this group to terminate all child processes. In this way, the registered timer will act like being “canceled”.

```
27 void clearAlarm(){
28     /* do something here ... */
29     kill(0, SIGUSR2);
30 }
```

The last thing to do is to ignore both `SIGUSR2` and `SIGALRM` in parent process

```
33     if (signal(SIGALRM, SIG_IGN) == SIG_ERR)
34     {
35         printf("error: SIGALRM handler of parent\n");
36         exit(0);
37     }
38     if (signal(SIGUSR2, SIG_IGN) == SIG_ERR)
39     {
40         printf("error: SIGUSR2 handler parent\n");
41         exit(0);
42     }
```

Even the parent process is terminated, the register alarm still works.