

Please read the code shared after the lectures and Chapter 4 (you can skip 4.4.2, 4.4.3, 4.5.2, 4.5.3, 4.7.1) of the textbook and then answer the following, trying not to look at your notes or at the textbook. Quiz #5, on Thursday 2nd November, will consist exclusively of questions taken from the Part I of this homework.

Part I — Short Questions

Question 1

Give one similarity and one difference between thread and process.

Question 2

Give two benefits provided by threads, and one difficulty the programmer has to face to use them.

Question 3

From a programmer perspective, what makes a program easy to divide between threads?

Question 4

What is the difference between data parallelism, and task parallelism?

Question 5

Why is there normally no need to “protect” one thread from the others within the same process?

Question 6

What does the C statement `pthread_join(tid, NULL)` do?

Question 7

What is the datatype used to store the id of a thread?

Question 8

If the C statement `pthread_create(&var_pthread, NULL, func, &y)` returns the value 0, what can you conclude? You can use `man pthread_create` to find out.

Question 9

What is implicit threading?

Question 10

Give three signal numbers and explain their role.

Question 11

What statement in C would you use to send the signal 14 to a process with id 234?

Question 12

What does the C statement `pthread_exit()` do?

Question 13

What is the name of the model where many user-level threads are all mapped onto a single kernel thread?

Question 14

Name the two sorts of signal handlers that exist.



Part II — Problems

This time, the three problems require a computer. They are rather short, but critical for your understanding of this lecture. As usual, I'll assume that you will have successfully completed them by the time Homework #6 is released (Thursday 2nd November), so don't wait and let me know if you had difficulties doing them.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void* print1() {
    int i = 0;
    while (i < 5) {
        printf("Hi, how do you do?\n");
        sleep(0.1);
        i++;
    }
    pthread_exit(0);
}

void* print2() {
    int i = 0;
    while (i < 5) {
        printf("Hello, how are you?\n");
        sleep(0.1);
        i++;
    }
    pthread_exit(0);
}

int main() {
    pthread_t tr1, tr2;
    pthread_create(&tr1, NULL, print1, NULL);
    pthread_create(&tr2, NULL, print2, NULL);
    int i = 0;
    while (i < 5) {
        printf("Let me introduce you.\n");
        sleep(0.1);
        i++;
    }
    pthread_join(tr1, NULL);
    pthread_join(tr2, NULL);
    exit(0);
}
```

Listing 1: thread_pb1.c

Problem 1

Open VirtualBox, select the OSC-2016 image, and click on “Settings”. Then, click on “System”, and then

“Processor”, and make sure that the virtual machine is running with more than one processor. Then, run your virtual machine, created a “05” folder in your “Desktop/HW/” folder, and copy the code displayed in Listing 1, or download it from http://spots.augusta.edu/caubert/teaching/2017/fall/csci3271/hw/05/thread_pb1.c. Answer the following:

- Compile the program, using `gcc -l pthread thread_pb1.c`, and execute it *multiple times*, using `./a.out`. Is the result always the same? How do you explain this behavior?
- Create a copy of the file, and edit it, so that the variable declaration `int i = 0;` appears only once, instead of three. We want the variable to be shared among all the threads of this program, so where should that variable declaration go? What do you observe when you run the program: are the printing instructions performed five time, overall? Why, or why not? Using statement like `printf("%d\n", i);`, “track” the value of `i`.

Problem 2

Run your virtual machine, and open two terminal.

- In the first one, execute `top`. In the second terminal, find the pid of `top`. The right combination of `ps` and `grep` should give you that answer easily.
- In that second terminal, execute `man kill`, and find the instruction to list all the signal names. Once you found it, exit the manual by typing `q`.
- Send the signal 26 to `top` using `kill`. What do you observe?
- Run `top` again in the first terminal, and send it another signal.

Problem 3

Copy the code displayed in Listing 2, or download it from http://spots.augusta.edu/caubert/teaching/2017/fall/csci3271/hw/05/signal_pb1.c. Do the following:

- Compile and run the program, then send it the SIGINT signal, using `Ctrl + C` or the `kill` program.
- Add the statement `signal(29, sig_handler);` after `signal(SIGINT, sig_handler);`, compile and run the program, then send it the signal 29.
- Modify your program so that it would print “Time’s over” then quit when it received the signal SIGALRM.
- Now, write a handler for SIGKILL, compile and run your program, and send it a SIGKILL signal. You should get a “Illegal instruction” error message printed (or, at least, your handler won’t be executed). Can you guess why? Look for the answer if needed.

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

void sig_handler(int signo)
{
    printf("Caught signal %d, coming out...\n", signo);
    exit(1);
}

int main(void)
{
    printf("My pid is %d.\n", getpid());
    signal(SIGINT, sig_handler);
```

```
        while(1)
        {
            sleep(1);
        }
        exit(0);
    }
```

Listing 2: signal_pb1.c

