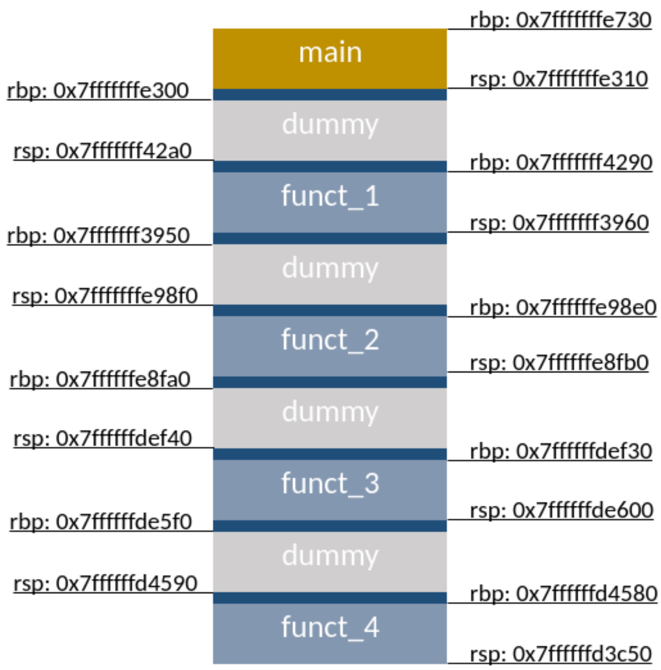
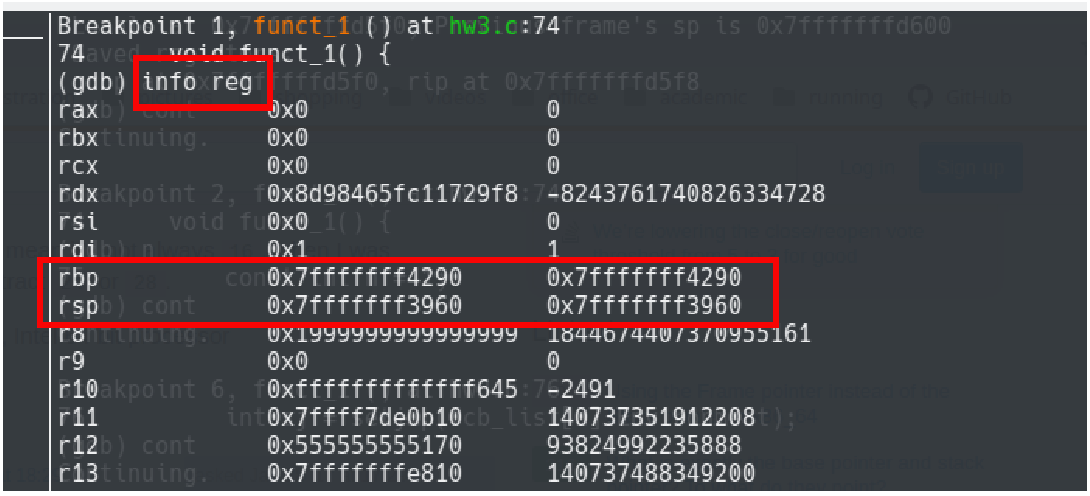
Programming Assignment #3

Student Name: 林楷恩
Student ID: b07902075

a. Below is an illustration of the arrangement of stack frames:



The current address of `rbp`(base pointer) and `rsp`(stack pointer) can be obtained by typing `info reg` in `gdb`. The Below screenshot is the demonstration of how I get the `rbp/rsp` of `funct_1`:



b. Yes, the values of the local variables will be the same when we jump back to the function, because when the program leaves a function it does not really clean up the contents of the top stack frame, but simply move the *base pointer* and *stack pointer*. Though now it regards that block of memory as free and may push some other data on it, our implementation make sure that it never does such things(I will explain it in the next question). Thus, when we restore the `rbp/rsp` with `longjmp()`, our variables remains the same values.

c. `dummy()` creates a "padding" between stack frames which is large enough to serve as a buffer, which prevent the functions from polluting the stack frames of the other functions by pushing new data on the top of their stack frames. This is because we still have to call some functions in `main`, `funct_1`, `funct_2`, `funct_3`, `funct_4`. In `main`, we call `Scheduler()`; in `funct_1`, `funct_2`, `funct_3`, `funct_4`, we call `sleep()` and `longjmp()`. All of these function calls push new stack frames on the top of the current stack frames and overwrite something. If we don't add `dummy()`, some important variables in `funct_1..4` may lose their original values. If we add `dummy()`, then the variables (like `int a[10000]`) in `dummy()` may be corrupted, but we do not care about it.

d. No, it cannot follow this path and return to main, because of the **GCC Stack Protection Mechanism**. In my experiment, the program is aborted when it returns from `funct_5(name=4)` to `funct_3`, as following screenshot shows:

```
(gdb) s
funct_5 (name=4) at hw3.c:25
25                case 4: funct_4(); break;
(gdb) n
28        }
(gdb) n
*** stack smashing detected ***: <unknown> terminated
```

To prevent stack overflow attack, GCC adds a protection mechanism which is described as follow:

"The basic idea behind stack protection is to push a 'canary' (a randomly chosen integer) on the stack just after the function return pointer has been pushed. The canary value is then checked before the function returns; if it has changed, the program will abort."

In my program, the "canary" is changed because `funct_3()` considers itself on the top of the stack, so when it calls functions (in this case, `sleep()` and `longjmp()`), it pushes new stack frames on the position where `funct_5(name=4)` lies in. These new stack frames overwrite the "canary", triggering the protection mechanism and making the program terminates. The same thing does not happen when we use `longjmp()` because this protection mechanism only triggered when a function **return**.

By the way, there is a trick to disable such a protection mechanism, by adding `-fno-stack-protector` flag when compiling the program, then it will not add this to the executable file.

e. (a) **The Structure of** `func_1..4`:

i. If it is the first time `funct_n` is called, first call `setjmp` and then call `funct_5(n + 1)` if `n != 4` else `longjmp` to `main`.

ii. If the scheduler switch to `funct_n`, enter the "big loop"

iii. In big loop:

(1) check the lock, if `mutex != 0` or `mutex != n`, `setjmp` and jump back to scheduler

(2) acquire the lock and pop `n` from the queue if it is in queue

(3) run small loop

(4) For Task 2, check if it should release the lock and if it does, `setjmp` and jump back to scheduler

(5) For Task 3, check pending signals, if a signal is pending, `setjmp` and remember the old signal mask then unblock the pending signal by `sigprocmask`.

iv. After finishing the big loop, release the lock and `longjmp` to scheduler with -2, indicating this function is finished.

(b) main function of `hw3.c`

i. parse the command line arguments, if the task number is not 2, set `K` to 1000000007 such that `i % K` never equals to 0

(c) main function of `main.c`

i. I block the 3 signal before calling `fork()` to avoid race condition.