Lab 5

3.3

To check if a detour needs to be arranged in *clkdisp.S* for the current process, *clkhandler()* sets a global variable *alarm_flag* to 1 if the current process has an outstanding alarm event and if the timer has expired. The callback function stored in *prcbf1* is also copied to a global function pointer *alarmcbf* so it can be accessed in *cbfmanager.S*. Lastly, it resets *pralarmreg* to 0, because if everything works as expected, the detour should immediately be arranged once *clkhandler()* returns to *clkdisp.S*.

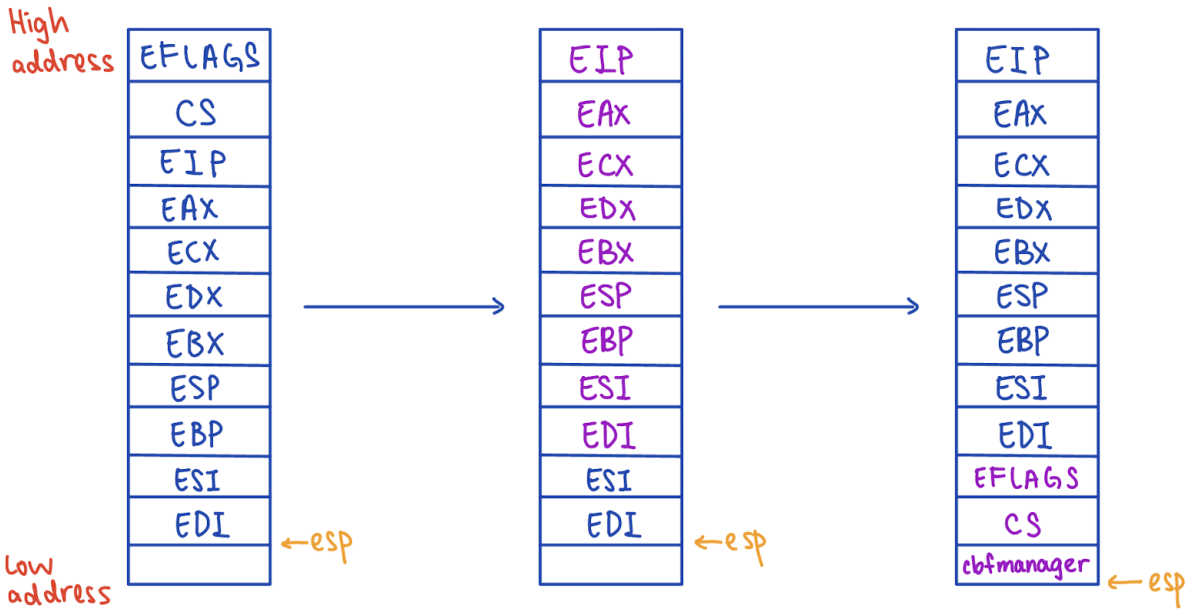
In *clkdisp.S*, it first checks that we have to modify the stack by checking if the global variable *alarm_flag* is 1. The code to modify the stack is then separated into three sections:

- move_setup
  This section is executed if *alarm_flag* is 1. Here, all the variables are setup and stored in registers for future use. The CS and EFLAGS values are stored in register ESI and EDI respectively because we will be overwriting them later. Then, the ESP value is stored in register ECX and 32 is added to it, so ECX now points to the address storing the EIP value. This is because the address stored in ECX is the address where we will be copying stuff from. Then, the address in ECX is copied to register EDX and added 8 to it, which points to the address storing the EFLAGS value. This is because the address stored in EDX is the destination for where we will be copying stuff to. Lastly, I setup a loop counter variable in register EAX that is initialized to 0. After everything has been setup and stored, the code jumps to the “move_loop” section.

- move_loop
  This section executes the shuffling of values to the top of the stack through a loop. The first line checks if the value of EAX is 9 (because we are shuffling the 8 registers and EIP value to the top). If so, it will jump to the “move_cleanup” section. Else, it will keep shuffling the values to the top by copying the value at the address stored in ECX to the address stored in EDX. This is done by indirectly copying the value to register EBX because direct memory-to-memory moves are not possible in assembly. Then, the address stored in ECX and EDX are both subtracted by 4 to shuffle the next value to the top. Lastly, the loop counter variable stored in EAX is incremented before the code jumps back to the beginning of the “move_loop” section.

- move_cleanup
  This section pushes the value of EFLAGS, CS, and *cbfmanager*’s address onto the stack. It copies the EFLAGS value stored in register EDI to the address stored in EDX. Then, it subtracts the address stored in EDX by 4 and copies the CS value stored in register ESI to that address. The code then pushes *cbfmanager*’s function pointer onto the stack. Since it is a “pushl” instruction, the value of ESP will be updated too. Then, *clkcountermsec* is incremented using lab1’s implementation. Lastly, the code calls “sti” and “iret” to restore interrupt status and return from *clkdisp()*. The “popal” call is removed from the original code because we will be doing that in *cbfmanager.S*.

The diagram below shows what happens to the stack when a detour is arranged:



In *cbfmanager.S*, the value of *alarmcbf* is copied to the register EAX and is called using the indirect function call syntax (*%eax). This redirects the execution to the function stored in *prcbf1* and once it finishes executing, it returns back to *cbfmanager.S*. Then, *cbfmanager.S* executes the “popal” instruction to pop the 8 registers we have moved in *clkdisp.S* to restore the original register values, since we have accessed them when modifying the stack in *clkdisp.S*. Lastly, *cbfmanager()* calls “ret” instead of “iret” because we do not have the CS and EFLAGS values anymore. The “ret” call should return to *clkdisp.S* original’s return address EIP as we have moved it to the top of the stack in *clkdisp.S*.

3.5

To test my implementation in 3.3, I created 3 helper functions in *main.c* to verify that a detour occurs after the timer expires. *print()* prints the message “callback” and *print2()* prints the message “another callback”. The main helper function, *testcbf(),* first calls *cbsignal(1, print, 100)* so it would detour to *print()* after 100 milliseconds has pass. I then called *cbsignal(1, print2, 100)* to check that SYSERR will occur. Then, I printed the output returned from both function calls to check that the first call outputs 0 and the second call outputs 1. Next, I implemented a while loop that breaks when *pralarmreg* is 0, meaning that there is no outstanding alarm event. I then printed “in cbf 1” to verify that the detoured callback function prints before this does.

Below is the code described above:

```
void print(void) {
    kprintf("callback\n");
}

void print2(void) {
    kprintf("another callback\n");
}

void testcbf(void) {
    syscall outp1 = cbsignal(1, print, 100);
    syscall outp2 = cbsignal(1, print2, 100);
    kprintf("output 1: %d, output 2: %d\n", outp1, outp2);
    while(1) {
        if (proctab[currpid].pralarmreg == 0) {
            break;
        }
    }
    kprintf("in cbf 1\n");
}
```

After creating and resuming *testcbf()* in *main()*, the results below are as expected:

```
output 1: 0, output 2: -1
callback
in cbf 1
```

To test my implementation in 3.4, I created a helper function, *receiver()*, to setup the receive process. It first calls *cbsignal(2, print, 100)* so it would detour when it wakes up after sleepms(). Then, I called *sleepms(1000)* and printed the receiving message after. After creating and resuming *receiver()* in *main.c*, I called *send()* on the receiver’s PID with the message “hi”. Since the process executing *receiver()* should be in sleep, the *send()* call should be rearranging the stack.

As expected, it prints out “callback” before printing out the message it received, since it detours to *print()* right after it wakes up. The result is as below:

```
callback
1102 ms: Receive 'hi' at pid 5
```