Q3

main() registers a call back function myrcv(). I tested the correctness of the mechanism by sending messages to main() in other processes. The priority of main() is set to be 2000 before creating a child process to ensure a clear sequence of execution.

Test case 1:

Process created by main() named Lab4_3_debug and pid=4 sends a message “1” to main(). The output is as follows:

```
clktimemilli 26
pid 2 can send message to pid 0, msg=2

clktimemilli 27
pid 4
sending message to main(), msg=1

clktimemilli 33
pid 4 can send message to pid 3, msg=1

clktimemilli 51
pid 3
process has message, msg=1
```

we can see that main()’s parent process sent a message to nulluser(). Since nulluser() has not registered a call back function so it is not invoking a function call (and it may also not become current anymore due to R3 scheduler). Then Lab4_3_debug sent a message “1” to main(), and continued to execute until its timeslice (25 ms) depleted. Then main() become current again, it invokes myrcv() after being context switched in and printed out message.

Test case 2:

main() creates two processes with pid=4 and 5 and goes to sleep. The two processes respectively send a message “1” and “2” to main(). The output is as follows:

```
clktimemilli 1
pid 2 can send message to pid 0, msg=2

clktimemilli 2
pid 4
sending message to main(), msg=1
```

clktimemilli 7
pid 4 can send message to pid 3, msg=1

clktimemilli 26
pid 5
sending message to main(), msg=2

clktimemilli 31
pid 5 cannot send message

clktimemilli 501
pid 3
process has message, msg=1

in this case, the second process created by main() cannot send a message since the first process has already sent a message to main() and main() has not yet received it.

Test case 3:
In this case, main() create a process before it register to a call back function. I expect that main() will not invoke myrcv().

Output:
clktimemilli 1
pid 2 can send message to pid 0, msg=2

clktimemilli 2
pid 4
sending message to main(), msg=1

clktimemilli 7
pid 4 can send message to pid 3, msg=1

it did not invoke myrcv.

Q4
Testing old cases from Q3

Case #1 from Q3 and Output:

clktimemilli 26
pid 2 can send message to pid 0, msg=2

clktimemilli 27
pid 4
sending message to main(), msg=1

clktimemilli 33
pid 4 can send message to pid 3, msg=1

clktimemilli 51
pid 3
process has message, msg=1

Case #2 from Q3 and Output:

clktimemilli 1
pid 2 can send message to pid 0, msg=2

clktimemilli 2
pid 4
sending message to main(), msg=1

clktimemilli 7
pid 4 can send message to pid 3, msg=1

clktimemilli 26
pid 5
sending message to main(), msg=2

clktimemilli 31
pid 5 cannot send message

clktimemilli 501
pid 3
process has message, msg=1

Case #3 from Q3 and Output:
clktimemilli 1
pid 2 can send message to pid 0, msg=2

clktimemilli 2
pid 4
sending message to main(), msg=1

clktimemilli 7
pid 4 can send message to pid 3, msg=1

The output of the three cases outputs are the same compared those in Q3.

Testing three cases on SIGXCPU
Case #1
Set 2500 ms to be the gross cpu usage to be reached. Make sure that only main() is running as the only current process.
Output:

clktimemilli 26
pid 2 can send message to pid 0, msg=2

clktimemilli 2501
pid 3
userhander

clearly, main() ran 2500 ms and invoked the userhandler() which printed out clktimemilli, pid and "userhander". Since main()'s parent become current and ran 1 ms in the middle before terminating, the clktimemilli for main() was 2501 ms.

Case #2
Set 2500 ms to be the gross cpu usage to be reached. Main() calls sleepms(500) before entering while(1) loop.
Output:

clktimemilli 1
pid 2 can send message to pid 0, msg=2

clktimemilli 3001
pid 3
userhander

This time the clktimemilli 3001=2501+500

Case #3
Set 2500 ms to be the gross cpu usage to be reached. Main() creates a testing process Lab4_4_XCPU which has a while(1) loop.
Output:

clktimemilli 26
pid 2 can send message to pid 0, msg=2

clktimemilli 5001
pid 3
userhander

since under R3 scheduling, the two processes are CPU-bound and will run round-robin. So it takes 2*2500=5000 ms for main() to reach the 2500 ms cpu usage.

Testing SIGTIME

Case #1:
Main() calls xalarm(500), then enters infinite loop.
Output:

clktimemilli 27
pid 2 can send message to pid 0, msg=2

clktimemilli 500
pid 3
useralarm

the alarm rings and printed out “useralarm” when clktimemilli=500. This is the case where the alarm handler is invoked in clkhandler().

Case #2

Main Main() calls xalarm(500), then calls sleepms(5000).
Output:

clktimemilli 2
pid 2 can send message to pid 0, msg=2

clktimemilli 5002
pid 3
useralarm

the alarm rings when clktimemilli=5002. This is the case where the alarm handler is invoked in do_shandler().

Case #3
Main Main() calls xalarm(500), then creates a testing process Lab4_4_XCPU which has a while(1) loop.

Output:
clktimemilli 26
pid 2 can send message to pid 0, msg=2

clktimemilli 501
pid 3
useralarm

the creation of the testing process did not affect the execution of alarm handler.

Testing concurrent processes
Main() registers SIGRECV and creates three processes, Lab4_3_debug sending message “1” to main(), Lab4_4_conc_1 registering SIGXCPU with 500 ms gross cpu time and Lab4_4_conc_2 registering SIGTIME with 5000 ms as alarm ringing time.

Output:

clktimemilli 26
pid 2 can send message to pid 0, msg=2

clktimemilli 27
pid 4
sending message to main(), msg=1

clktimemilli 33
pid 4 can send message to pid 3, msg=1

clktimemilli 101
pid 3
process has message, msg=1

clktimemilli 2051
pid 5
userhander

clktimemilli 5000
pid 6
useralarm

After about 1 time slice, Lab4_3_debug became current and sent a message to main().
At 101 ms, which is roughly 4 time slices long (since 4 process were running and scheduled round-robin under R3), main() becomes current and executed the myrcv() call back function.
After approximately 4*500=2000 ms, Lab4_4_conc_1 recived 500 ms cpu time and executed userhandler().
After 5000 ms, the alarm for Lab4_4_conc_2 rang and executed the useralarm().

Bonus
There are three problems.

1. The signal is handled only through the one-time handler which will reset after handling the signal. This make it hard for users to handle a signal in any future time separately. The user will have to reset the signal before using any handlers in the future.
2. ROP implementation allows almost no argument to be passed to the call back function due to the constraint of the size of the stack. The ctxsw()'s stack only has two argument frames that can be used to store temporary information such as the return address, address of the call back function. No argument of a call back function can be stored on the stack.
3. The non-blocking call back function can potentially disrupt the normal behavior of a process. For example, if a call back function is executing some system call, when a signal arrives, it will suspend the execution of the system call which is potentially hazardous to the integrity of a system call.