CS 503 Fall 2022: Lab 4

Suraj Aralihalli

October 2022

3. Asynchronous event handling using callback function

3.4 Testing

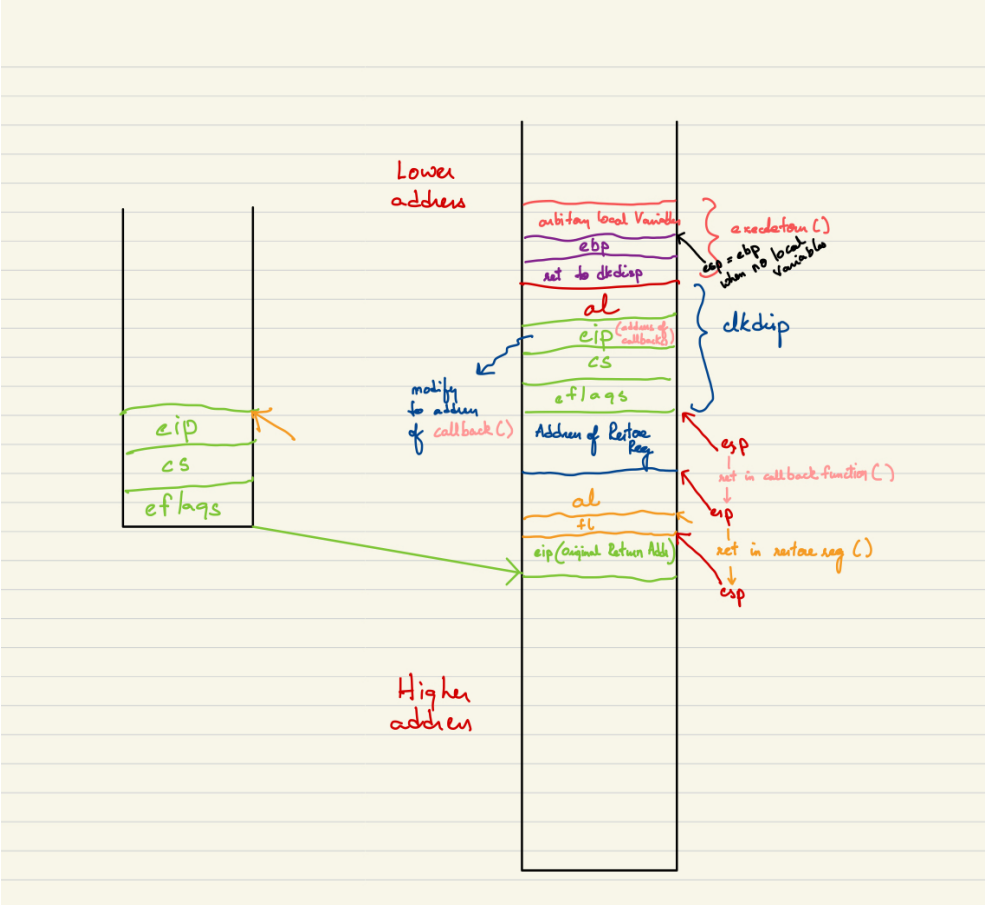

Figure 1: Stack constructed using ROP techniques when prmakedetour is 1

Method:

When prmakedetour value is set to 1 in the wakeup function, I fabricate the stack as depicted in the Figure 1 to support the callback function. The smaller stack on the left in the figure represents the configuration of the stack right before performing iret in the clkdisp.S (when no callback function is registered). However, we have to construct the stack to support jump to callback function along with retaining and restoring the previous general purpose and eflags registers when a callback function is registered.

In my executedetour() function, I build this stack when the prmakedetour variable is set to 1 for a process. I temporarily pop all the values on the stack and store them in static variables (as stack variables are not allocated on same stack, this avoids adding more items on the stack). I populate the stack as shown in Figure 1 so that when iret is executed (in kernel mode), the control flows to callback function (in user mode). After the callback function runs, the control then flows to restoreregs function which is reponsible for poping the general purpose and eflags registers. The restoreregs function exits with ret which takes the control to the original function that was contexted switched.

Testing:

I verified the functionality of my framework using the following test examples:

1. Single process is created that sets an alarm and continues to run: This testcase verifies if the executedetour, wakeup and other fundamental functions are developed as per the instructions.
2. Single process is created that sets an alarm and exits: The alarm should not be triggered (callback function should not be executed) as the userspace of the process does not exist when the alarm goes off.
3. Single process is created that sets an alarm and goes: This testcase verifies if the executedetour2 function is correctly implemented.
4. Single process is created that sets 2 alarms to ring at different times with different callback functions.
   (a) When the second alarm is created before the first one is triggered: The second callback function should run twice.
   (b) When the second alarm is created after the first one is triggered: The two callback functions should run individually.
5. Single process is created that sets 2 alarms to ring at same time with different callback functions: The callback function registered with the second alarm should execute only once.
6. Single process (pid: 5) is created that sets 3 alarms to ring at different times with different callback functions.
   (a) If all three alarms are created immediately one after the other: The third alarmx call should fail with SYSERROR
   (b) If the third alarm is created after the first or second alarm goes off, it should take the place of 105 or 205 in the sleep queue depending on which alarm last rang.
7. Similar testcases with 4 alarms.
8. Testcases to check whether interrupts are enabled when the callback function is running.

4. Preemptible and reentrant IPC support

4.5 Testing

Method:

sendx

1. Returns SYSERR if bad pid
2. Acquires a lock i.e waits on receiver's pripc (Note that all entry points into sendx need acquiring a lock, However we have only such entry point into sendx)
3. Three cases are possibles. In other words, there are multiple exit points in sendx. Lock has to released at all exit points before returning from sendx. Note that each of the following three cases can be considered as individual critical sections.
   (a) Buffer is free: Copy the data into receiver buffer, set relevant variables and make the receiver's process ready if necessary.
   (b) Buffer is not free and no sender is blocked: Copy the data into sender's buffer, set relevant variables and block the sender and call resched().
   (c) Buffer is not free and some sender is blocked: Return SYSERROR

receivex

1. receivex has 2 entry points and 2 exit points. Lock has be acquired at all entry points and released at exit points. In the case of receivx, the code section between any two entry and exit points can be seen as a critical section.
2. If prrecvlen is 0, change the state of receiver process to PR_RECV and call resched(). This is one of the exit points in receivex. When it returns from resched(), acquire the lock again. This is the second entry point of receivex.
3. Copy data from receiver buffer to user buffer.
4. If the pidptr is not NULL, fill it with the sender's pid and also reset the variables like prsenderpid and prrecvlen.
5. Check if any sender is blocked. If true, copy the data from sender's buffer to receiver's buffer and ready the sender.

Testing:
I verified the functionality of my framework using the following test examples:

1. Sender and Receiver processes are created. Sender performs 2 sendx calls one after the other. Sender should get blocked after the first call. After receiver makes the first receivex call, sender should be unblocked.
2. Sender and Receiver processes are created. Sender performs a sendx call. Receiver consumes the data. Sender makes another sendx call, receiver consumes the data again. No one is blocked and sender buffer is not used.

3. Sender and Receiver processes are created. Receiver makes a receivex call and gets blocked (PR_RECV state). Sender makes a sendx call and unblocks the receiver (using ready()).

4. Two processes are created. Both the processes act as both sender and receiver. P1 performs sendx() followed by receivex(). P2 also performs sendx() followed by receivex(). None of the processes are blocked.

5. Two processes are created. P1 performs receivex(), P2 performs receivex(). Both the processes are blocked.

6. Test cases to check if semaphores are working as expected i.e critical sections run atomically.

Bonus problem

The 2 scenarios to be considered while designing the solution are:

1. receivex()/sendx() is in middle of copying data

2. receivex()/sendx() is in blocked state

In both these scenarios, we would like the kernel to handle the interrupt due to myalarmhandler() gracefully.

Let's consider receivex() for example:

1. receivex() is in middle of copying data: Here we have following design choices,

   (a) Prematurely end the receivex() while indicating the amount of data copied before returning and passing control to the myalarmhandler()

   (b) Complete the receivex() call (copy the data completely) and then pass control to myalarmhandler()

   In both these designs, the programmer should be notified on how the situation has been handled. Since myalarmhandler() is time sensitive i.e we cannot compromise on the timeliness of the myalarmhandler() execution, I wouldn't recommend the second approach. Other compelling reason to pick the first design choice is that, it offers flexibility by allowing the programmer to either have the kernel handle copying the remaining data, or the programmer can decide what path needs to be taken in the user code. If the kernel were to handle coping the remnant data, I would re-design the receivex() function to carry out the copy process based on offset passed as an input parameter to the function (Note that this parameter is different from len parameter). Within receivex(), I can have a local variable that is dynamically updated with the latest amount of copied data through out the copy process. In the event of interrupt, this function would prematurely terminate returnimg the local variable indicating the amount of data copied and release the lock. If this returned value is less than the len value passed to the function, the kernel would know that the copy processes was not successfully completed and decide to continue the copy process from where it was left off after executing the myalarmhandler().

2. receivex() is in PR_RECV state

   (a) Prematurely end the receivex() and pass control to myalarmhandler()

   (b) Wait until sender readies the receiver function

As discussed in the previous case, timely execution of myalarmhandler() could be critical. In such cases, waiting until the sender readies the receiver function is not ideal. Consequently, I would recommend prematurely terminating receivex(). However, the receivex() function can be re-designed to release the lock and return -2 if interrupted when alarm goes off. The kernel can capture this return value and decide to either notify the programmer or call receivex() again after executing the myalarmhandler(). Since we are designing a system with support for reentrance and automatic restart, the kernel can make an implicit call to receivex to put the process back in PR_RECV state.