CS50300: Operating Systems

Lab4 Answers

Zichen Wang
wang4113@purdue.edu

October 27, 2018

1 Asynchronous message receive

The sender is defined in `system/test_sender.c`.

The receiver is defined in `system/test_receiver.c`.

the sender sends 10 random numbers to each receiver asynchronously. After sending a message, the sender will run a 500,000 busy loop and then go to sleep for 500 ms.

1. Case 1: one sender and one receiver. The result is shown in table 1. This simple scenario demonstrates that the basic function of asynchronous receive works well. The receiver process receives the message only when the context switch to it.

2. Case 2: one sender and multiple receivers. This scenario is kind of complicated but the performance is approximately the same as scenario 1, since only one message can be sent at any time, and only one receiver can receive the message at any time. The part of result is shown in table 2.

3. Case 3: multiple senders and multiple receivers. Each receiver randomly chooses a receiver process and sends message to it. Hence, there might be some errors occurred when a sender sends a message to a receiver who has already been sent a message by another sender before. Moreover, preemption may occur since the sender has higher priority. Since there are too many lines in the result, we can hardly show them in the PDF.

No.	Time (ms)	Message
1	64	Process 5 sent '0' to Process 4.
	104	Process 4 received '0'.
2	579	Process 5 sent '579' to Process 4.
	620	Process 4 received '579'.
3	1095	Process 5 sent '2190' to Process 4.
	1111	Process 4 received '2190'.
4	1611	Process 5 sent '4833' to Process 4.
	1652	Process 4 received '4833'.
5	2127	Process 5 sent '8508' to Process 4.
	2143	Process 4 received '8508'.
6	2643	Process 5 sent '3215' to Process 4.
	2684	Process 4 received '3215'.
7	3159	Process 5 sent '8954' to Process 4.
	3175	Process 4 received '8954'.
8	3675	Process 5 sent '5725' to Process 4.
	3716	Process 4 received '5725'.
9	4191	Process 5 sent '3528' to Process 4.
	4207	Process 4 received '3528'.
10	4707	Process 5 sent '2363' to Process 4.
	4748	Process 4 received '2363'.

Table 1: The result of asynchronous message receive in case 1.

No.	Time (ms)	Message
1	239	Process 7 sent ‘0’ to Process 4.
	279	Process 4 received ‘0’.
2	754	Process 7 sent ‘754’ to Process 5.
	820	Process 5 received ‘754’.
3	1270	Process 5 sent ‘2540’ to Process 6.
	1336	Process 6 received ‘2540’.
4	1786	Process 5 sent ‘5358’ to Process 4.
	1877	Process 4 received ‘5358’.
5	2393	Process 5 sent ‘9208’ to Process 5.
	2818	Process 5 received ‘9208’.
6	2818	Process 5 sent ‘4090’ to Process 6.
	2909	Process 6 received ‘4090’.
7	3334	Process 5 sent ‘4’ to Process 4.
	3350	Process 4 received ‘4’.
8	3849	Process 5 sent ‘6943’ to Process 5.
	3865	Process 5 received ‘6943’.
9	4365	Process 5 sent ‘4920’ to Process 6.
	4381	Process 6 received ‘4920’.
10	4881	Process 5 sent ‘3929’ to Process 4.
	4922	Process 4 received ‘3929’.
...		
...		

Table 2: The part of result of asynchronous message receive in case 2.

PID	Time (ms)	CPU usage
4	1964	500
5	2989	750
6	4014	1000

Table 3: The result of SIGXCPU

2 Signal handling subsystem

2.1 Overview

The return address of ctxsw() is located in `prptr -> prstkptr + 40`. Since the two arguments of ctxsw() are the addresses of old process pointer and new process pointer which are useless after context switch, so we can use the places of these two arguments to save an argument and the original return address for do_handler() respectively.

2.2 SIGRECV

- We test three cases of asynchronous message receive again. For case 1, the result is totally the same. For case 2 and case 3, the performance is very similar but not exactly the same, since the overhead of these two approaches are not the same.
- The implementation of this signal registration is simple and the same as asynchronous message receive. We just need to make sure the handler pointer is not NULL.

2.3 SIGXCPU

- We create three processes, each of which registers SIGXCPU callback handler. The timer is 500, 750 and 1200 respectively. The result is in table 3. The process is defined in `test_xcpu.c`.
- The implementation is not very complicated. In `signalreg.c`, we also need to make sure the callback pointer is not NULL. Moreover, the time value should not be less than or equal to $pgrosscpu + currproctime$. In `clkhandler.c`, if $pgrosscpu + currproctime$ is equal to the time set before, we should enable interrupts and call then handler function, and disable interrupts again.

2.4 SIGTIME

- We create three processes, each of which registers SIGTIME callback handler. The xalarm() time value is set 1000 for all test processes, since we can test both cases in one experiment. The result is shown in table 4. The process is defined in `test_xalarm.c`.
- The implementation of SIGTIME is the most complicated. We need to not only deal with SIGTIME itself, but also work well with multiple signals.

PID	Time (ms)
4	1000
5	1014
6	1039

Table 4: The result of SIGTIME

1. First, in xalarm(), we need to make sure the time value is greater than `clktimemilli`, and then save it in the process table. xalarm() will return error if there is no callback handler registered.
2. Second, in `clkhandler.c`, we have to go through all processes and check its alarm time. If the time of a process is equal to `clktimemilli` and the process if not the current process, we should perform ROP technique. If it is the current process, we would run it after checking all other processes, since this callback function may take some time and interfaces the checking in `clkhandler.c`. Hence, we should set a local variable `curr_alarm_flag` to indicate whether the current process needs to run SIGTIME callback handler.
3. Third, in `do_shandler.c`, we need to distinguish which callback function should be called, SIGRECV, SIGTIME or both. To deal with that, we set an argument `uint32 arg` for do_shandler() by manipulating the stack. `arg` is a 2-bit interger. If the first bit is 1, SIGRECV callback handler should be called, and if the second bit is 1, SIGTIME callback handler should be called.
4. Finally, if `curr_alarm_flag` is set to true during the checking, the SIGTIME callback function of the current process should be called after the SIGXCPU callback function, since we need to check $pgrosscpu + currproctime$ for SIGXCPU which is sentitve to time change in clkhandler(). If the SIGTIME callback function is called first, the `currproctime` may change when coming back, where the SIGXCPU callback function might not be called forever.

2.5 Different signals with multiple concurrent processes

In this test, we create 6 processes that have registered SIGRECV and SIGXCPU callback functions, 6 processes that have registered SIGRECV and SIGTIME callback functions, and another 3 processes that have registered all three callback function. We also create a sender process to send messages to all these 15 processes. The result is very complex and cannot be shown and analyzed one by one. As a whole, it is correct since the message in sender and receiver is the same, and the alarm rings at the time which is equal to or slightly larger than the timer we set before. SIGXCPU callback handler also works well since the signal time and gross CPU usage are make sense. The test code can be found in `main.c`, `test_receiver.c`, `test_xcpu.c`, `test_xalarm.c`, `test_signals.c`, and `test_sender.c`.

3 Bonus problem

1. The delay of a time event calling a event handler will be very large if there are multiple processes with higher priority running concurrently. One strategy, we believe, is to provide the process whose callback function should be called with a higher priority temporarily. Hence, the scheduler can decide to run this process and call the handler as soon as possible.

2. SIGRECV may cause stack overflow. For example, process A registers a signal handler for SIGRECV, and process B sends message to process A constantly. When process A is running the handler for SIGRECV and has read a message by receive(), the clock interrupt happens and the scheduler decides to run process B. Now, the context of process A has been saved in the stack. Process B sends a message again. After that, process A will go to the handler again and read a message by receive(). If the clock interrupt happens just after process A called receive() and process B sends a message again, the stack will be overflow in the end.

3. We cannot cancel a signal callback handler. For example, we has registered a callback handler for SIGRECV, and captured a asynchronous message. After that, however, we would like to cancel it and want to receive a message synchronously. We have not implemented cancellation in our design.