CS50300: Operating Systems
Lab1 Answers

Zichen Wang
wang4113@purdue.edu

September 12, 2018

1 Objectives

2 Readings

3 Running XINU

3.1

An alternative way to 'idle' is to call **halt()** defined in **intr.S**. The assembly code of **halt()** only jumps to itself, which means looping forever.

3.2

The **welcome()** function is under **system/welcome.c**.

4 XINU system calls

4.1

The C type of syscall, defined in **include/kernel.h**, is **int**.

4.2

If the parent process has terminated, the process table field **prstate** should be *PR_FREE*.

5 XINU's run-time environment

5.1 Changing byte order using assembly code

The instruction **bswapl** can change byte order of a 32-bit register directly.

5.2 Checking segment boundaries

The addresses of the end of the text, data, and bss segments of the XINU OS are defined by $\&etext - 1$, $\&edata - 1$ and $\&ebss - 1$ respectively. The **printsegaddress()** function is under **system/printsegaddress.c**.

5.3 Run-time stack: process creation and function call

We will use the inline assembly code to print the top of run-time stack, by copying the value of the register **esp**.

The address and content of the top of the run-time stack inside main() will remain the **same** at the time before myprogA() is created and after myprogA() has been created and resumed. It is because myprocA() is a new independent process, and able to run along with the main() process. The stack of myprocA() will be created in a new free memory area when the process spawns and will **not** change the stack structure of its parent process. Therefore, in these two printing, main() process's stack and main() function's stack frame are being printed, and the results are same.

When myprogA() calls the function myfuncA(), things will be different. Before myfuncA() is called, myprocA() process's stack and myprocA() function's stack frame are being printed. Then, myprogA() will use its own stack to maintain a function call to myfuncA(), pushing the arguments of myfuncA() and return address to the stack. After that, myprogA() will jump to myfuncA(), but the process remains unchanged. In function myfuncA(), the stack belongs to the myprogA() process. So, after myfuncA() is called, myprocA() process's stack and myfuncA() function's stack frame are being printed.

5.4 Comparing two run-time stacks

As shown in the console.

6 Hijacking a process via stack smashing

The strategy for finding the return address of sleepms() function is very simple. In myfuncA() process, it is easy to get the process id of myprogA() process, i.e. PPID of myfuncA() process. Then, we use the PPID to retrieve the stack parameters of myprogA() process and the whole stack from *prstkbase* to *prstkptr* can be printed. There are too many addresses at the first glance on the stack information, but the system call *sleepms* has a paramater '3000'. From the system perspective, when a process make a system call or function call, it will push the return address just following the arguments of the function. According to this convention, I find that '0x00000BB8'

('3000' in hex) in the stack and the content under '0x00000BB8' is just the return address of *sleepms* function. In my case, the true return address is '0x001034F7', and the location of the true return address is $proctab[ppid].prstkptr + 120$, since 'prstkprt' will be updated to the **esp** register when context switching. We could just replace this content with the malwareA()'s address. This offset '120' is fixed because the number of stack allocation for the system call 'sleepms()' is always '120' bytes. The stack frame of myprogA() has been printed in the console.

However, things will not finish. In order to let myprocA() process exit normally, which means that its resources can be released and other process with lower priority can continue to run, myprocA() process should return to the *INITRET* address after malwareA() returns. From the convention, the process will use **ret** instruction to pop the return address from the stack to the **eip** register. Therefore, we may put the *INITRET* address just before the malwareA()'s address in the stack. The *INITRET* address is located in $proctab[ppid].prstkbase - 4$ defined by *create.c*, so $*(int*)(proctab[ppid].prstkptr + 124) = *(int*)(proctab[ppid].prstkbase - 4)$ would work.

7 Bonus problem

As detailed above, we may rewrite the place just before the sleepms() return address in the stack because it is only saving the argument '3000' of sleepms(). Thus, we could temporarily save the true return address in that place, i.e.

$$*(int*)(proctab[ppid].prstkptr + 124) = *(int*)(proctab[ppid].prstkptr + 120);$$

Then, move the malwareB() address into $proctab[ppid].prstkptr + 120$.

To minimize the disruption on the myprocA() process, we cannot modify any register and the content of the running stack after malwareB() returns. First, in the normal situation, the address $proctab[ppid].prstkptr + 124$ would become useless after the sleepms() returns, so saving the true return address there before jumping to malwareB() is fine. Second, by looking up the assembly code of malwareB(), the **ebx** register which belongs to callee saved has been modified. Thrid, we cannot just use the true return address saved in $proctab[ppid].prstkptr + 124$ to go back to myprocA(), because by doing so, the value of the **esp** register will not be equal to the value in the normal situation. Therefore, the only way to make an unusual return is to use inline assembly codes. Before malwareB() returns, we should restore the value of **ebx** which has saved in the stack by the function. Then, have the stack frame registers restored normally. Instead of returning now, we must push the true return address from $proctab[ppid].prstkptr + 124$ to $proctab[ppid].prstkptr + 120$, in order to imitate the return situation of sleepms(). Next, we can do a return by the **ret** instruction. All these return steps have been achieved by inline assembly codes.