HW1-3160102525-赵洁

1.

exercise 1.14

1）What is the purpose of interrupts？

The purpose of interrupts is to alert the CPU to events that require attention.

2）How does an interrupt differ from a trap？

Difference	Interrupt	Trap
generated by	hardware	software
synchronous/asynchronous	asynchronous	synchronous
active/passive	passive, wait for interrupts to happen	active, most of the time, expects the trap to happen and relies on this fact
	/	see as a CPU-internal interrupt

3）Can traps be generated intentionally by a user program? If so, for what purpose?

Traps can be generated intentionally by a user program to call operating system routines or to catch arithmetic errors.

exercise 1.17

Impossible：

We accomplish protection against errant users by designating some of the machine instructions that may cause harm as privileged instructions.

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

It is the distinction in mode of execution at the hardware level that enables the operating system to very strictly control and limit access to the resources.

Possible:

The same mechanism described above which exists at the hardware level can be emulated in software.

To securely execute the user-program, operating system can first scan the entire program (or portions of it) and check to see if the program performs any unauthorized instructions or perhaps a privileged instructions.

If so, the operating system can refuse to run the program.

Or the operating system can run the program in an entirely separate virtual machine so that it will still be able to maintain actual control of the real hardware.

Obviously, such a system will be inefficient and slow as there is a lot of overhead involved.

exercise 1.19

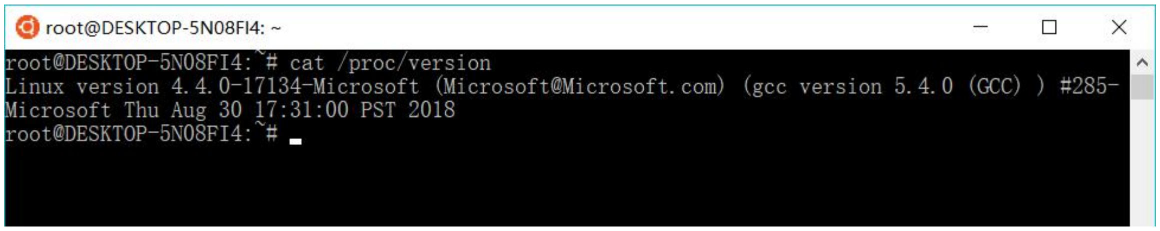
Rank the following storage systems from slowest to fastest:

magnetic tapes
optical disk
Hard-disk drives
nonvolatile memory
main memory
cache
registers

exercise 1.22

The processor could keep track of what locations are associated with each process and limit access to locations that are outside of a program’s extent. Information regarding the extent of a program’s memory could be maintained by using base and limits registers and by performing a check for every memory access.

2.

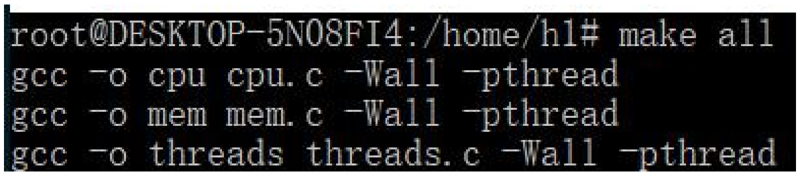


```
root@DESKTOP-5N08FI4:~# cat /proc/version
Linux version 4.4.0-17134-Microsoft (Microsoft@Microsoft.com) (gcc version 5.4.0 (GCC) ) #285-
Microsoft Thu Aug 30 17:31:00 PST 2018
root@DESKTOP-5N08FI4:~#
```

3.

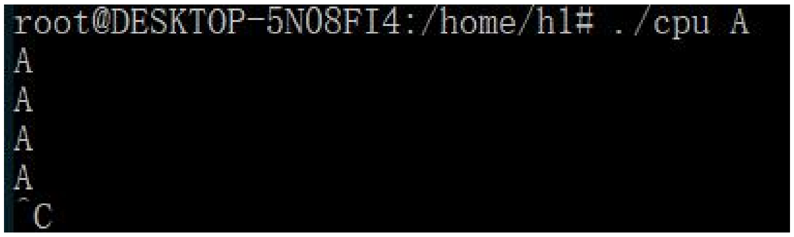
A)

i. Make



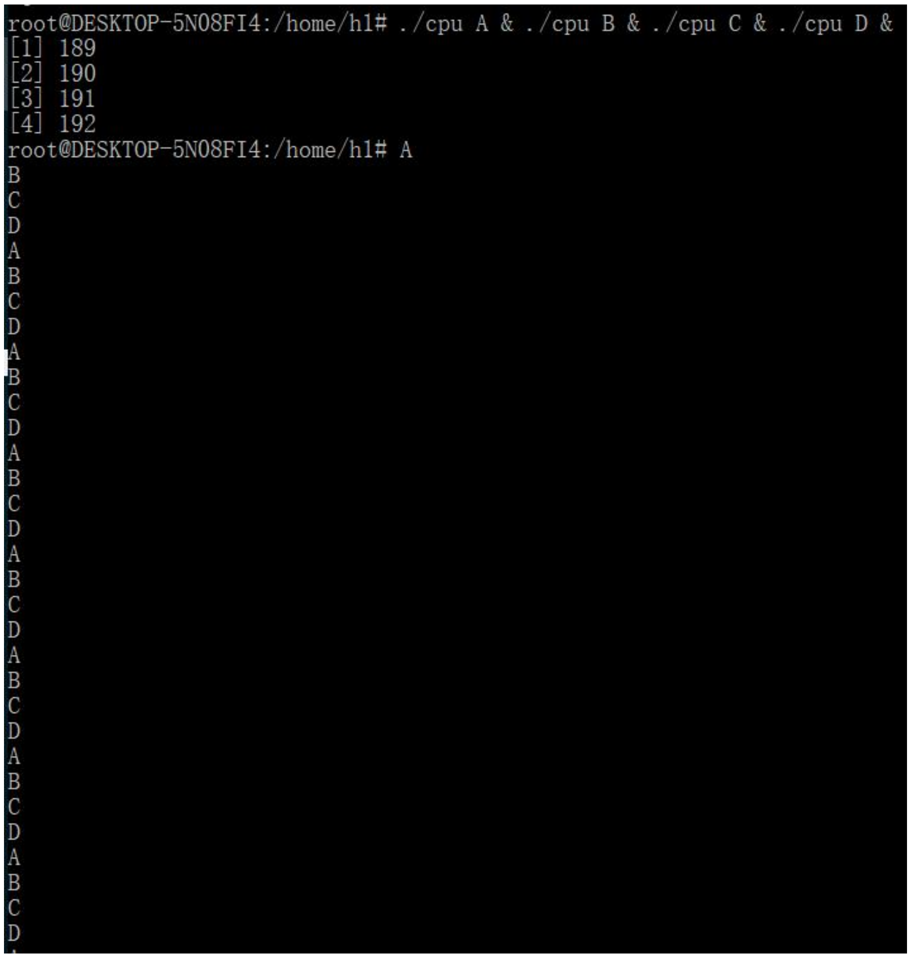
```
root@DESKTOP-5N08FI4:/home/h1# make all
gcc -o cpu cpu.c -Wall -pthread
gcc -o mem mem.c -Wall -pthread
gcc -o threads threads.c -Wall -pthread
```

ii. Run program *cpu* with a single processor



```
root@DESKTOP-5N08FI4:/home/h1# ./cpu A
A
A
A
A
^C
```

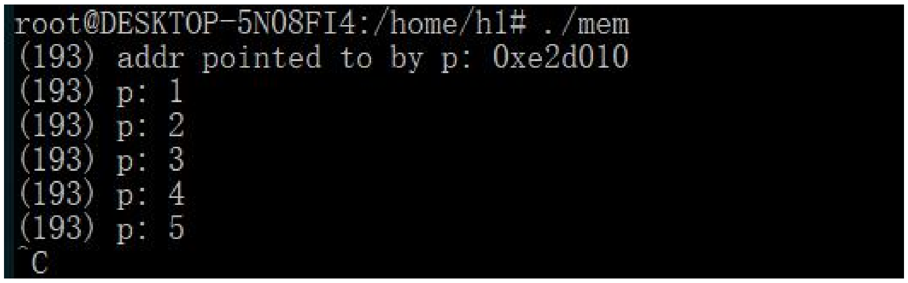
iii. Run four different instances of program *cpu*



```
root@DESKTOP-5N08FI4:/home/h1# ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 189
[2] 190
[3] 191
[4] 192
root@DESKTOP-5N08FI4:/home/h1# A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
A
B
C
D
```

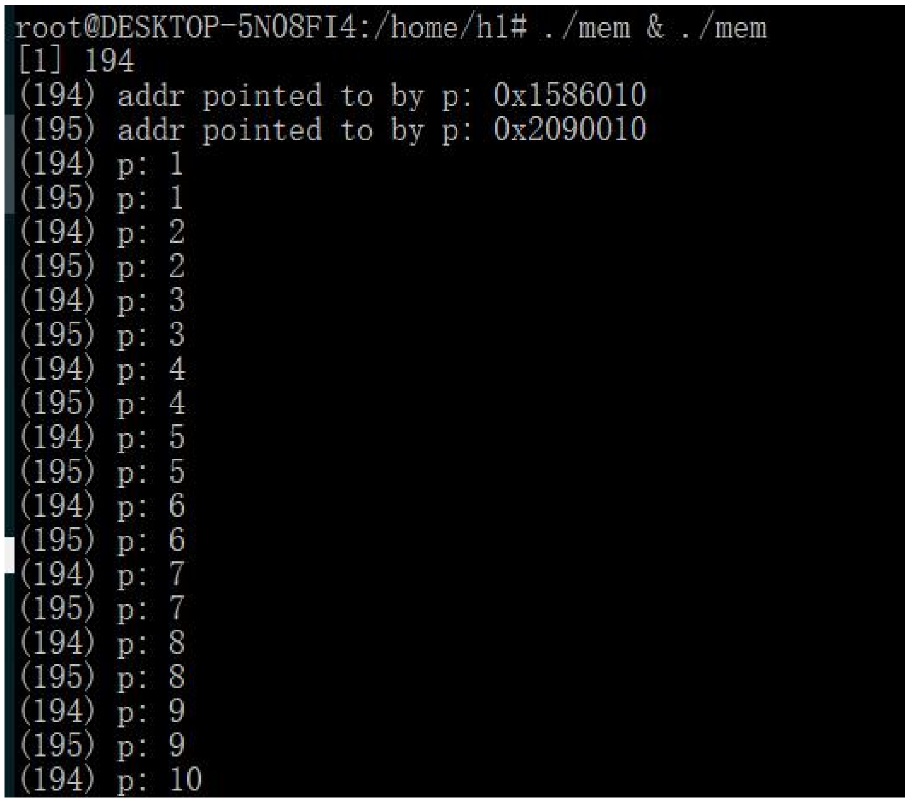
Finally, processes are killed in background, which is not displayed in the above screenshoot.

iv. Run program *mem*



```
root@DESKTOP-5N08FI4:/home/h1# ./mem
(193) addr pointed to by p: 0xe2d010
(193) p: 1
(193) p: 2
(193) p: 3
(193) p: 4
(193) p: 5
^C
```

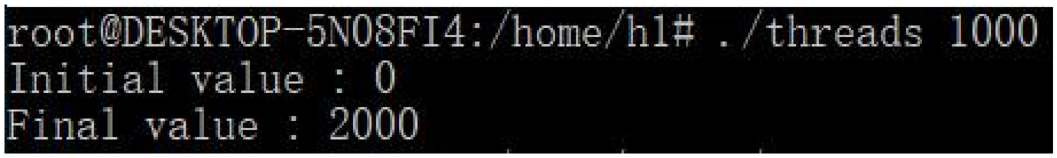
v. Run multiple instances of program *mem*



```
root@DESKTOP-5N08FI4:/home/h1# ./mem & ./mem
[1] 194
(194) addr pointed to by p: 0x1586010
(195) addr pointed to by p: 0x2090010
(194) p: 1
(195) p: 1
(194) p: 2
(195) p: 2
(194) p: 3
(195) p: 3
(194) p: 4
(195) p: 4
(194) p: 5
(195) p: 5
(194) p: 6
(195) p: 6
(194) p: 7
(195) p: 7
(194) p: 8
(195) p: 8
(194) p: 9
(195) p: 9
(194) p: 10
```

Finally, processes are killed in background, which is not displayed in the above screenshoot.

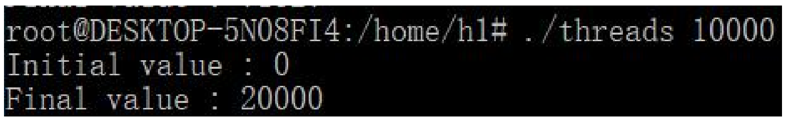
vi. Run program *threads* with the value of loops set to 1000



```
root@DESKTOP-5N08FI4:/home/h1# ./threads 1000
Initial value : 0
Final value : 2000
```

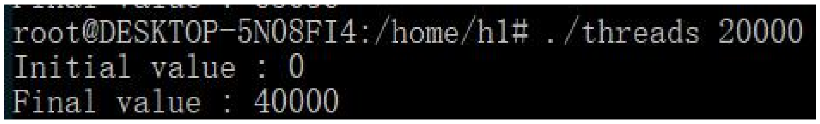
vii. Run program *threads* with higher values for loops

Loops=10000



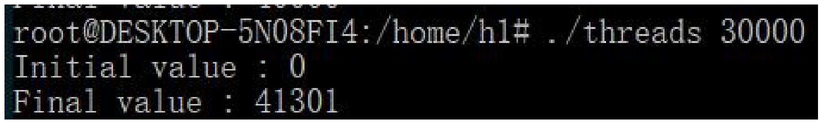
```
root@DESKTOP-5N08FI4:/home/h1# ./threads 10000
Initial value : 0
Final value : 20000
```

Loops=20000



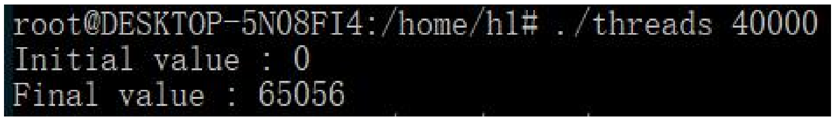
```
root@DESKTOP-5N08FI4:/home/h1# ./threads 20000
Initial value : 0
Final value : 40000
```

Loops=30000



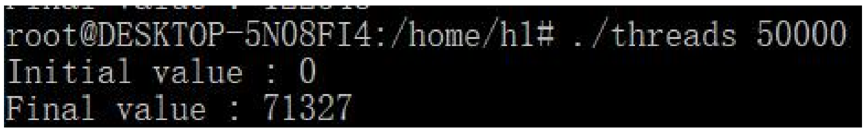
```
root@DESKTOP-5N08FI4:/home/h1# ./threads 30000
Initial value : 0
Final value : 41301
```

Loops=40000



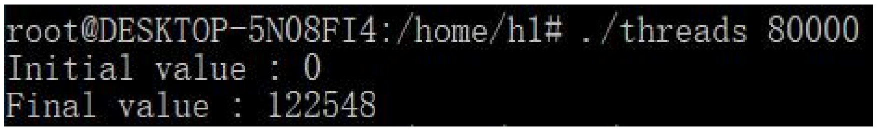
```
root@DESKTOP-5N08FI4:/home/h1# ./threads 40000
Initial value : 0
Final value : 65056
```

Loops=50000



```
root@DESKTOP-5N08FI4:/home/h1# ./threads 50000
Initial value : 0
Final value : 71327
```

Loops=80000



```
root@DESKTOP-5N08FI4:/home/h1# ./threads 80000
Initial value : 0
Final value : 122548
```

Actually, run program *threads* several times with the same high values for loops, the final values differs every time.

B)

i. What I've leart from running program *cpu*

With some help from the hardware, system has a very large number of virtual CPUs. **CPU visualization** turns a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allows many programs to seemingly run at once.

ii. What I've leart from running program *mem*

Operating system makes each running program has its own private memory. Each process accesses its own private virtual address space (sometimes just called its address space), which the OS somehow maps onto the physical memory of the machine. A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself.

iii. What I've leart from running program *threads*

Because the operation *counter++* is not executed atomically, including loading the value from the memory, calculating, and storing the result to the memory, several processes increment a global variable may causes read-write conflict.

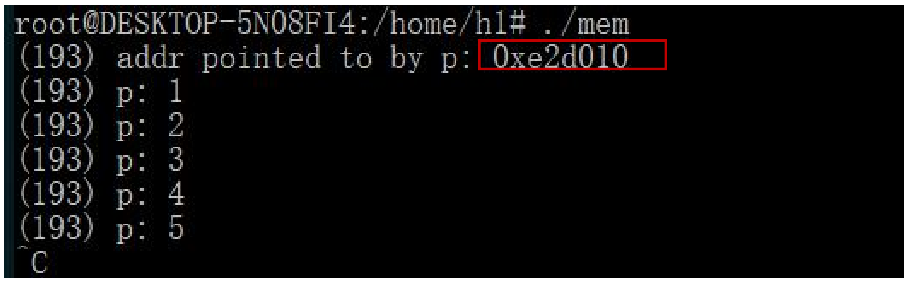
For example, in a situation displayed in the below table, the value of counter increased by 1, not as expected.

Thread 1	Thread 2
Load counter	
	Load counter
counter=counter+1	counter=counter+1
	Store counter
Store counter	

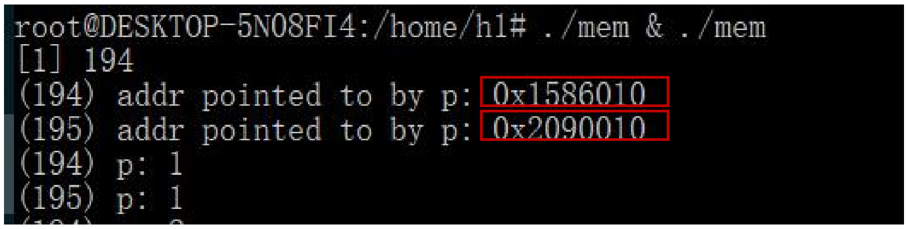
So the problem of concurrency is of vital importance in the implementation of operating system.

C)

No. In the book, the addresses pointed to by p are always the same every time runing the second program. But when I run different instances of the second program, the addresses seem to be random.



```
root@DESKTOP-5N08FI4:/home/h1# ./mem
(193) addr pointed to by p: 0xe2d010
(193) p: 1
(193) p: 2
(193) p: 3
(193) p: 4
(193) p: 5
^C
```

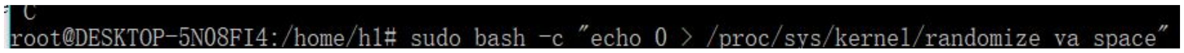


```
root@DESKTOP-5N08FI4:/home/h1# ./mem & ./mem
[1] 194
(194) addr pointed to by p: 0x1586010
(195) addr pointed to by p: 0x2090010
(194) p: 1
(195) p: 1
```

After searching for ASLR(address space layout randomization), I found out that Linux kernel enabled a weak form of ASLR by default since the kernel version 2.6.12, released in June 2005.

Address space layout randomization (ASLR) is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

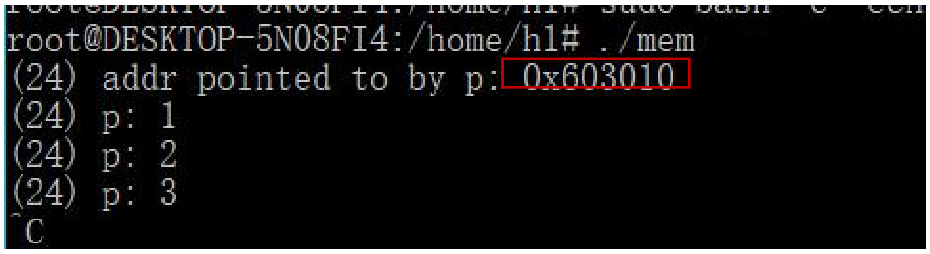
I tried to turn off ASLR to see if the addresses pointed to by p will be the same. Command is as follows:



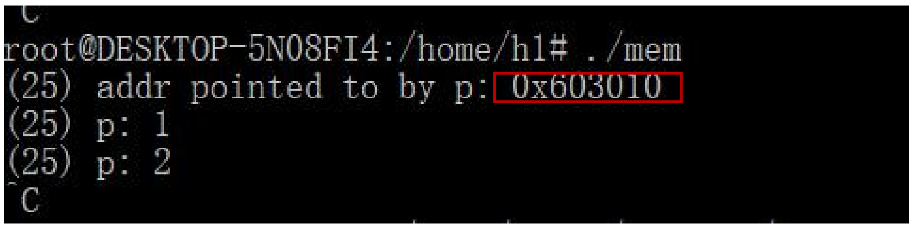
```
^C
root@DESKTOP-5N08FI4:/home/h1# sudo bash -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

This commond means turn off ASLR. And then, run program *mem*.

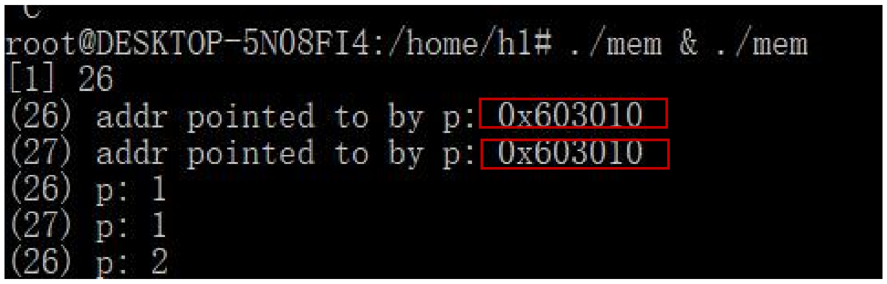
This time, the addresses pointed to by p are the same.



```
root@DESKTOP-5N08FI4:/home/h1# ./mem
(24) addr pointed to by p: 0x603010
(24) p: 1
(24) p: 2
(24) p: 3
^C
```



```
root@DESKTOP-5N08FI4:/home/h1# ./mem
(25) addr pointed to by p: 0x603010
(25) p: 1
(25) p: 2
^C
```



```
root@DESKTOP-5N08FI4:/home/h1# ./mem & ./mem
[1] 26
(26) addr pointed to by p: 0x603010
(27) addr pointed to by p: 0x603010
(26) p: 1
(27) p: 1
(26) p: 2
```

I also found out that there are three values of **/proc/sys/kernel/randomize_va_space**, 0, 1 and 2. When **/proc/sys/kernel/randomize_va_space** is of value 1, the addresses pointed to by p are also the same.

After further searching, I figured out what the values mean.

0 – No randomization. Everything is static.

1 – Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized.

2 – Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.