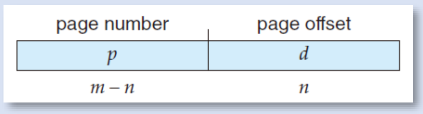
Operating System Homework 9

Jinyan Xu, 3160101126, Information Security

Part I

9.2 Why are page sizes always powers of 2?

Answer:



The page size is defined by the hardware. Paging is implemented by dividing an address into a page number and page offset. Using binary address is the most efficient way to get page number and page offset, instead of calculating with decimal arithmetic. Because each bit position represents a power of 2, the final address is also power of 2.

9.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

Answer:

Page offset 10 bits, for there 1024(2^{10}) words in each page, the logical address space is 64(2^6), physical is 32(2^5). So,

a. Logical address is 6 + 10 = 16 bits long.

b. Physical address is 5 + 10 = 15 bits long.

9.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?

Answer:

Two entries in a page table to point to the same page frame in memory means these two processes share code and data. When many large processes need to copy some large same memory, make entries of page table point to the same memory will save much memory space. But each little change made by one process will influence others.

9.6 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 12 5 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

Answer:

First-fit Algorithm:

Process	300KB	600KB	350KB	200KB	750KB	125KB
P1 = 115KB	P1(185KB)					
P2 = 500KB	P1(185KB)	P2(100KB)				
P3 = 358KB	P1(185KB)	P2(100KB)			P3(392KB)	
P4 = 200KB	P1(185KB)	P2(100KB)	P4(150KB)		P3(392KB)	
P5 = 375KB	P1(185KB)	P2(100KB)	P4(150KB)		P3, P5(17KB)	

Best-fit Algorithm:

Process	300KB	600KB	350KB	200KB	750KB	125KB
P1 = 115KB						P1(10KB)
P2 = 500KB		P2(100KB)				P1(10KB)
P3 = 358KB		P2(100KB)			P3(392KB)	P1(10KB)
P4 = 200KB		P2(100KB)		P4(0KB)	P3(392KB)	P1(10KB)
P5 = 375KB		P2(100KB)		P4(0KB)	P3, P5(17KB)	P1(10KB)

Worst-fit Algorithm:

Process	300KB	600KB	350KB	200KB	750KB	125KB
P1 = 115KB					P1(635KB)	
P2 = 500KB					P1, P2(135KB)	
P3 = 358KB		P3(242KB)			P1, P2(135KB)	
P4 = 200KB		P3(242KB)	P4(150KB)		P1, P2(135KB)	
P5 = 375KB	×	P3(242KB)	P4(150KB)	×	P1, P2(135KB)	×

! : P5 will have no enough memory to place.

9.15 Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:

a. External fragmentation

b. Internal fragmentation

c. Ability to share code across processes

Answer:

The contiguous memory allocation scheme:

External fragmentation: As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous.

Internal fragmentation: System breaks the physical memory into fixed-sized blocks and allocate memory in units based on block size, internal fragmentation exists, when the memory allocated to a process may be slightly larger than the requested memory.

Ability to share code across processes: Can't, because its virtual memory segment is not noncontiguous segments.
Paging:
External fragmentation: No, physical memory breaks into fixed-sized blocks called frames and logical memory breaks into blocks of the same size called pages.
Internal fragmentation: If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
Ability to share code across processes: Yes, paging allows two entries in different page tables to point to the same page frame in memory to share memory.

9.19 Explain why address-space identifiers (ASIDs) are used in TLBs.

Answer:
ASIDs provide address space protection in the TLB as well as supporting TLB entries for several different processes at the same time. Without ASIDs, then every time context switch, the TLB must be flushed to ensure that the next executing process does not use the wrong translation information from the last process.

9.20 Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?

a. Contiguous memory allocation
b. Paging

Answer:
Both Contiguous-memory allocation and Paging have internal fragmentation, process allocate entire virtual address space when it runs, and memory is allocated by blocks, so there're always some space is not used at the end of the address space, so the structure that data and code saved in low address, stack saved at high address can greatly improve the utilization of space.

9.25 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?
b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

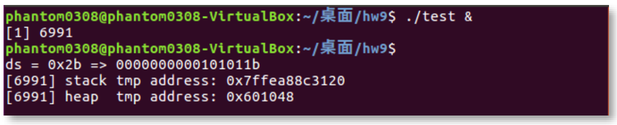
Answer:
a. 50 + 50 = 100 nanoseconds, 50 nanoseconds to access the entry in page table and 50 nanoseconds to access the word in memory.
b. Effective access time = 0.75 × (50 + 2) + 0.25 × (50 + 50 + 2) = 64.5 nanoseconds.

Part II

Write a kernel module and a user-mode program to show how the logical address of a variable in the user-mode program is converted to the physical address in the x86-64 environment.

Screenshot:

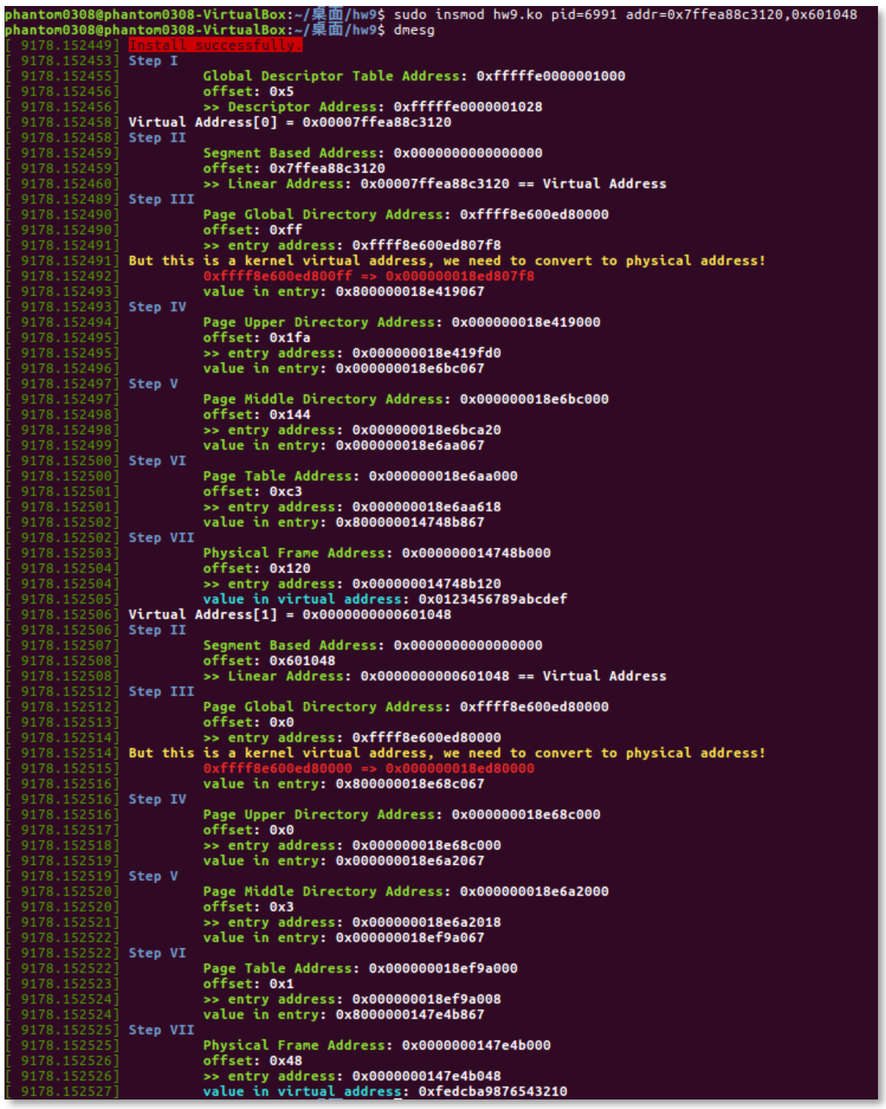
User-mode program:



```
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$ ./test &
[1] 6991
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$
ds = 0x2b => 0000000000101011b
[6991] stack tmp address: 0x7ffea88c3120
[6991] heap  tmp address: 0x601048
```

- (value in stack: 0x0123456789abcdef, in heap: 0xfedcba9876543210)

Kernel module:



```
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$ sudo insmod hw9.ko pid=6991 addr=0x7ffea88c3120,0x601048
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$ dmesg
[ 9178.152449] Install successfully.
[ 9178.152453] Step I
[ 9178.152455]         Global Descriptor Table Address: 0xfffffe0000001000
[ 9178.152456]         offset: 0x5
[ 9178.152456]         >> Descriptor Address: 0xfffffe0000001028
[ 9178.152458] Virtual Address[0] = 0x00007ffea88c3120
[ 9178.152458] Step II
[ 9178.152459]         Segment Based Address: 0x0000000000000000
[ 9178.152459]         offset: 0x7ffea88c3120
[ 9178.152460]         >> Linear Address: 0x00007ffea88c3120 == Virtual Address
[ 9178.152489] Step III
[ 9178.152490]         Page Global Directory Address: 0xffff8e600ed80000
[ 9178.152490]         offset: 0xff
[ 9178.152491]         >> entry address: 0xffff8e600ed807f8
[ 9178.152491] But this is a kernel virtual address, we need to convert to physical address!
[ 9178.152492]         0xffff8e600ed800ff => 0x000000018ed807f8
[ 9178.152493]         value in entry: 0x800000018e419067
[ 9178.152493] Step IV
[ 9178.152494]         Page Upper Directory Address: 0x000000018e419000
[ 9178.152495]         offset: 0x1fa
[ 9178.152495]         >> entry address: 0x000000018e419fd0
[ 9178.152496]         value in entry: 0x000000018e6bc067
[ 9178.152497] Step V
[ 9178.152497]         Page Middle Directory Address: 0x000000018e6bc000
[ 9178.152498]         offset: 0x144
[ 9178.152498]         >> entry address: 0x000000018e6bca20
[ 9178.152499]         value in entry: 0x000000018e6aa067
[ 9178.152500] Step VI
[ 9178.152500]         Page Table Address: 0x000000018e6aa000
[ 9178.152501]         offset: 0xc3
[ 9178.152501]         >> entry address: 0x000000018e6aa618
[ 9178.152502]         value in entry: 0x800000014748b867
[ 9178.152502] Step VII
[ 9178.152503]         Physical Frame Address: 0x000000014748b000
[ 9178.152504]         offset: 0x120
[ 9178.152504]         >> entry address: 0x000000014748b120
[ 9178.152505]         value in virtual address: 0x0123456789abcdef
[ 9178.152506] Virtual Address[1] = 0x0000000000601048
[ 9178.152506] Step II
[ 9178.152507]         Segment Based Address: 0x0000000000000000
[ 9178.152508]         offset: 0x601048
[ 9178.152508]         >> Linear Address: 0x0000000000601048 == Virtual Address
[ 9178.152512] Step III
[ 9178.152512]         Page Global Directory Address: 0xffff8e600ed80000
[ 9178.152513]         offset: 0x0
[ 9178.152514]         >> entry address: 0xffff8e600ed80000
[ 9178.152514] But this is a kernel virtual address, we need to convert to physical address!
[ 9178.152515]         0xffff8e600ed80000 => 0x000000018ed80000
[ 9178.152516]         value in entry: 0x800000018e68c067
[ 9178.152516] Step IV
[ 9178.152516]         Page Upper Directory Address: 0x000000018e68c000
[ 9178.152517]         offset: 0x0
[ 9178.152518]         >> entry address: 0x000000018e68c000
[ 9178.152519]         value in entry: 0x000000018e6a2067
[ 9178.152519] Step V
[ 9178.152520]         Page Middle Directory Address: 0x000000018e6a2000
[ 9178.152520]         offset: 0x3
[ 9178.152521]         >> entry address: 0x000000018e6a2018
[ 9178.152522]         value in entry: 0x000000018ef9a067
[ 9178.152522] Step VI
[ 9178.152522]         Page Table Address: 0x000000018ef9a000
[ 9178.152523]         offset: 0x1
[ 9178.152524]         >> entry address: 0x000000018ef9a008
[ 9178.152524]         value in entry: 0x8000000147e4b867
[ 9178.152525] Step VII
[ 9178.152525]         Physical Frame Address: 0x0000000147e4b000
[ 9178.152526]         offset: 0x48
[ 9178.152526]         >> entry address: 0x0000000147e4b048
[ 9178.152527]         value in virtual address: 0xfedcba9876543210
```

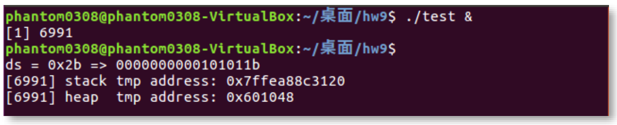
Analysis:

Address can be divided into logic address, linear address, physical address. For this part, our mission is to convert logic address to physical address step by step.

When operating system has loaded, CPU changes its model from real mode to protected mode. In protected model, if we want to get physical memory, we must get linear address first.

Ⅰ: Convert logical address to linear address

First, we need to get the segment and offset. Run our test program:



```
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$ ./test &
[1] 6991
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$
ds = 0x2b => 0000000000101011b
[6991] stack tmp address: 0x7ffea88c3120
[6991] heap  tmp address: 0x601048
```

Now, we get: Segment: **0x2b**, Offset: **0x7ffea88c3120**

We should notice that in protected model, **the value in segment register doesn't mean segment address**, it is a selector that selects a descriptor from a descriptor table.

Here is the format of a selector:

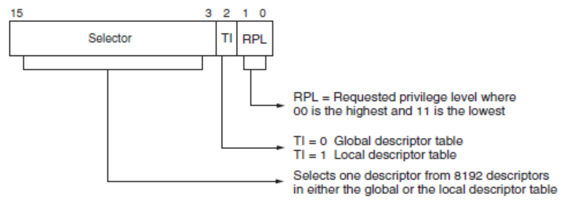
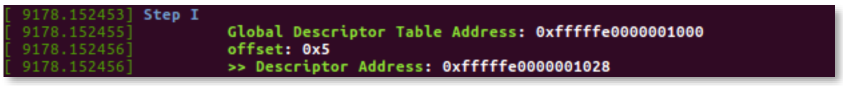

15
3
2
1
0
Selector
TI
RPL
RPL = Requested privilege level where 00 is the highest and 11 is the lowest
TI = 0 Global descriptor table
TI = 1 Local descriptor table
Selects one descriptor from 8192 descriptors in either the global or the local descriptor table

FIGURE 2–8 The contents of a segment register during protected mode operation of the 80286 through Core2 microprocessors.

Recall the value in ds is 0x2b => 101011b

So we choose global descriptor table (GDT), and t selector number (offset of GDT) is 5.



```
[ 9178.152453] Step I
[ 9178.152455]          Global Descriptor Table Address: 0xfffffe0000001000
[ 9178.152456]          offset: 0x5
[ 9178.152456]          >> Descriptor Address: 0xfffffe0000001028
```

But the size of a descriptor is 2^3 bytes, so we need to multiply 8 to get the true address, 5 * 8 = 40 = **0x28**, so the descriptors' address is **0xfffffe0000001028**.

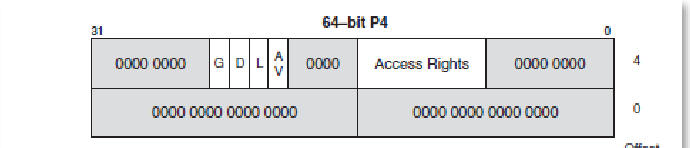
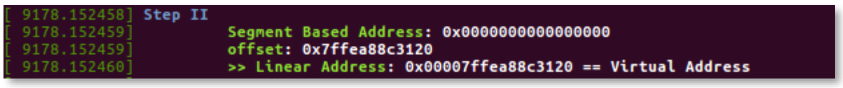


FIGURE 2–6 The 80286 through Core2 64-bit descriptors.

Above is the format of a descriptor, the gray area is represented segment based address.



```
[ 9178.152458] Step II
[ 9178.152459]          Segment Based Address: 0x0000000000000000
[ 9178.152459]          offset: 0x7ffea88c3120
[ 9178.152460]          >> Linear Address: 0x00007ffea88c3120 == Virtual Address
```

So, we can find that in x86-64, linear address is equal to virtual address.

II : Convert linear address to physical address

In x86-64, system only uses 48 bits, address is divided into following format:

47	38	29	20	11 0
Page Global Directory	Page Upper Directory	Page Middle Directory	Page Table	Offset

The bits 48–63 are called **sign extension** bits and must be copies of bit 47. The following 36 bits define the page table indexes (9 bits per table) and the last 12 bits specify the offset in the 4KiB page.

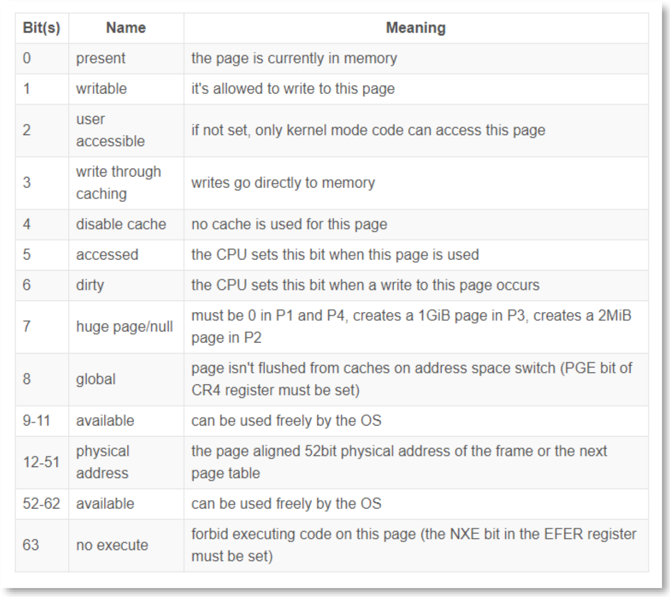
We can divide our virtual address (linear address) as following:

0x7ffea88c3120

= 011111111 111111010 101000100 011000011 000100100000b

= 0xff 0x1fa 0x144 0xc3 0x120

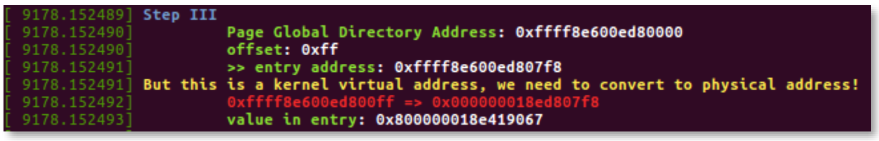
Next, we will model the contained physical address and the various flags. Remember, entries have the following format:



Bit(s)	Name	Meaning
0	present	the page is currently in memory
1	writable	it's allowed to write to this page
2	user accessible	if not set, only kernel mode code can access this page
3	write through caching	writes go directly to memory
4	disable cache	no cache is used for this page
5	accessed	the CPU sets this bit when this page is used
6	dirty	the CPU sets this bit when a write to this page occurs
7	huge page/null	must be 0 in P1 and P4, creates a 1GiB page in P3, creates a 2MiB page in P2
8	global	page isn't flushed from caches on address space switch (PGE bit of CR4 register must be set)
9-11	available	can be used freely by the OS
12-51	physical address	the page aligned 52bit physical address of the frame or the next page table
52-62	available	can be used freely by the OS
63	no execute	forbid executing code on this page (the NXE bit in the EFER register must be set)

But before the calculate begin, we need to get the page table directory address first. We can't directly read CR3, this is because the CR3 now is for our kernel mode.

Luckily, this address has been saved in PCB, we get the value from PCB.



```
[ 9178.152489] Step III
[ 9178.152490]         Page Global Directory Address: 0xffff8e600ed80000
[ 9178.152490]         offset: 0xff
[ 9178.152491]         >> entry address: 0xffff8e600ed807f8
[ 9178.152491] But this is a kernel virtual address, we need to convert to physical address!
[ 9178.152492]         0xffff8e600ed800ff => 0x000000018ed807f8
[ 9178.152493]         value in entry: 0x800000018e419067
```

Because our address is 64 bits, 2^3 bytes, we need to multiply 8 on the offset to get the true address. 0xff * 8 = 0x7f8, the entry address is **0xffff8e600ed807f8**.

Before next step, we need know that: for the virtual address space:

The bottom part (0x00000000 00000000 to 0x008FFFFF FFFFFFFF) to processes.

The top part (0xFFFF8000 00000000 to FFFFFFFF FFFFFFFF) to the kernel.

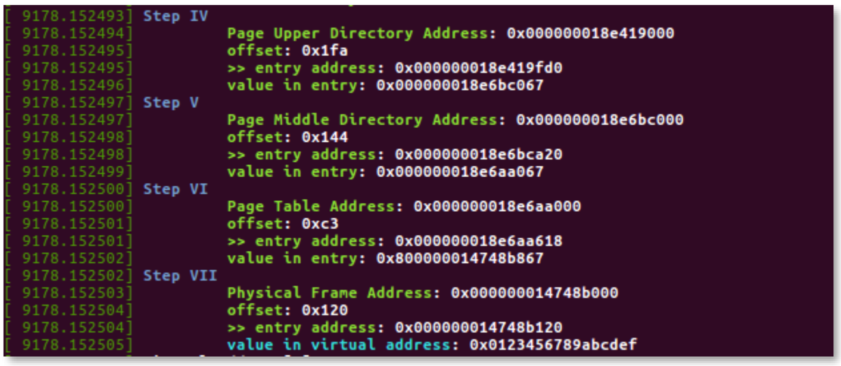
So, the address we get: **0xffff8e600ed807f8** is a kernel virtual address. Remember now we are in kernel, the page for our test user-model process may not in kernel's page table now, so we need to use kmap() to add them in, otherwise we can't access directly.

This kernel virtual address is mapped to a continue physical address, so wo can get its physical address by add/sub an offset.

0xffff8e600ed807f8 => 0x000000018ed807f8

After convert the address to physical address, now we can get the next page table based address.

Next steps are same:

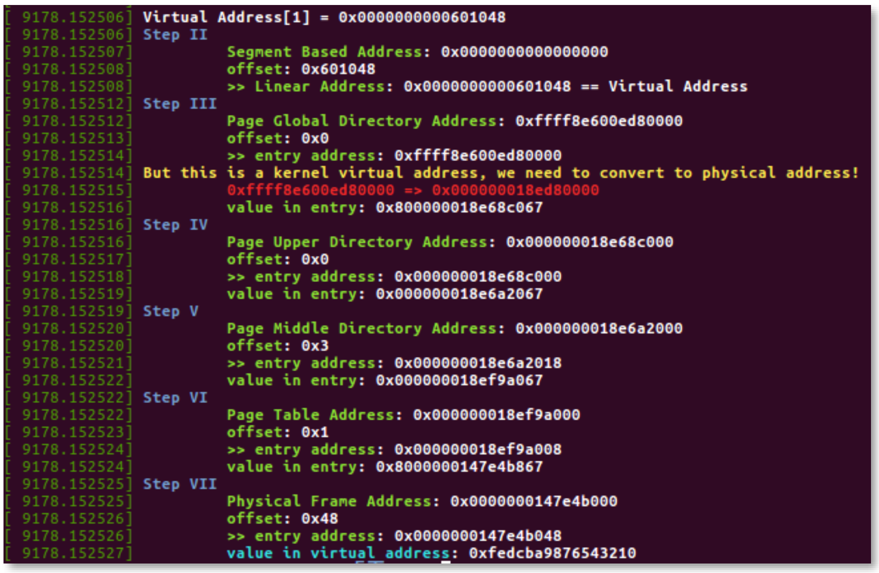


```
[ 9178.152493] Step IV
[ 9178.152494]          Page Upper Directory Address: 0x000000018e419000
[ 9178.152495]          offset: 0x1fa
[ 9178.152495]          >> entry address: 0x000000018e419fd0
[ 9178.152496]          value in entry: 0x000000018e6bc067
[ 9178.152497] Step V
[ 9178.152497]          Page Middle Directory Address: 0x000000018e6bc000
[ 9178.152498]          offset: 0x144
[ 9178.152498]          >> entry address: 0x000000018e6bca20
[ 9178.152499]          value in entry: 0x000000018e6aa067
[ 9178.152500] Step VI
[ 9178.152500]          Page Table Address: 0x000000018e6aa000
[ 9178.152501]          offset: 0xc3
[ 9178.152501]          >> entry address: 0x000000018e6aa618
[ 9178.152502]          value in entry: 0x800000014748b867
[ 9178.152502] Step VII
[ 9178.152503]          Physical Frame Address: 0x000000014748b000
[ 9178.152504]          offset: 0x120
[ 9178.152504]          >> entry address: 0x000000014748b120
[ 9178.152505]          value in virtual address: 0x0123456789abcdef
```

- A little difference is that in the last round we don't need to multiply 8, this is because the unit of memory is 8 bytes.

We can get the final value.

I also test the variable in heap.



```
[ 9178.152506] Virtual Address[1] = 0x0000000000601048
[ 9178.152506] Step II
[ 9178.152507]          Segment Based Address: 0x0000000000000000
[ 9178.152508]          offset: 0x601048
[ 9178.152508]          >> Linear Address: 0x0000000000601048 == Virtual Address
[ 9178.152512] Step III
[ 9178.152512]          Page Global Directory Address: 0xffff8e600ed80000
[ 9178.152513]          offset: 0x0
[ 9178.152514]          >> entry address: 0xffff8e600ed80000
[ 9178.152514] But this is a kernel virtual address, we need to convert to physical address!
[ 9178.152515]          0xffff8e600ed80000 => 0x000000018ed80000
[ 9178.152516]          value in entry: 0x800000018e68c067
[ 9178.152516] Step IV
[ 9178.152516]          Page Upper Directory Address: 0x000000018e68c000
[ 9178.152517]          offset: 0x0
[ 9178.152518]          >> entry address: 0x000000018e68c000
[ 9178.152519]          value in entry: 0x000000018e6a2067
[ 9178.152519] Step V
[ 9178.152520]          Page Middle Directory Address: 0x000000018e6a2000
[ 9178.152520]          offset: 0x3
[ 9178.152521]          >> entry address: 0x000000018e6a2018
[ 9178.152522]          value in entry: 0x000000018ef9a067
[ 9178.152522] Step VI
[ 9178.152522]          Page Table Address: 0x000000018ef9a000
[ 9178.152523]          offset: 0x1
[ 9178.152524]          >> entry address: 0x000000018ef9a008
[ 9178.152524]          value in entry: 0x8000000147e4b867
[ 9178.152525] Step VII
[ 9178.152525]          Physical Frame Address: 0x0000000147e4b000
[ 9178.152526]          offset: 0x48
[ 9178.152526]          >> entry address: 0x0000000147e4b048
[ 9178.152527]          value in virtual address: 0xfedcba9876543210
```

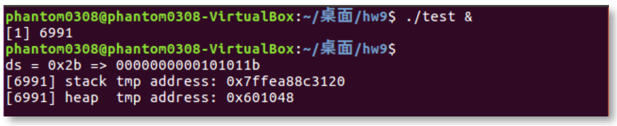
We can see that the result is right.

How to Run:

Your input:

> make
> ./test &

You will get like following:



```
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$ ./test &
[1] 6991
phantom0308@phantom0308-VirtualBox:~/桌面/hw9$
ds = 0x2b => 0000000000101011b
[6991] stack tmp address: 0x7ffea88c3120
[6991] heap  tmp address: 0x601048
```

The value in [ ] is pid, you need to seed pid and virtual addresses to kernel module. My kernel module can accept more than one addresses, but the max number is 8.

> sudo insmod hw9.ko pid=/*your pid*/   addr=/*va1*/,/*va2/,/va3/…

Code:

Web Link：https://pan.baidu.com/s/1E2lGgwOGKkjbFwH3GqdI7g
Password：gx3c

Reference:

Intel Microprocessors Architecture, Programming, and Interfacing 8th, Barry B. Brey

Great thanks to teacher, Yajin Zhou. Thanks!

2018-12-15