HW9 MainMemory

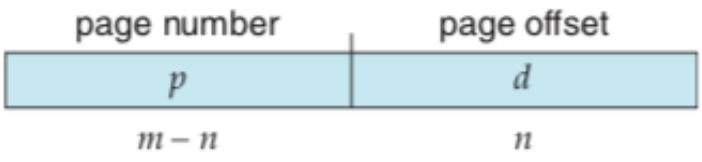
Exercises on Operating System Concept

9.2

> *Why are page sizes always powers of 2?*

Answer:

- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. That is, if you don't obey this rule, the construction of the logical address will be tedious. When the size is 2^n , the bits used for demonstrating the page offset can be utlized fully.
- Below is what the logical address like



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

- Logical address = [page number] || [page offset] Consider that there are 64 pages, then 6 bits are needed for page number field. Each page has 1024 words, then 10 bits are needed for page offset field <u>The total is 16 bits</u>
- What need to be concerned is that we only have 32 frames in physical memory. So there must be some pages refer to the same frame. Physical address = [frame number] || [frame offset] 5 bits are enought for frame number The offset is as same as the page case, 10 bits <u>The total is 15 bits</u>

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

- This case is indeed stand for the seperation of logical and physical address, for programmer may think they have more address which is not true. Additionally, this can be used for sharing data and code for different processes.
- When facing situation the process need to copy a large amount of memory from one place to another, the target virtual address may never do rewriting but just refer to the same place in physical address. It can speed up the copying work, however, when the modification occurs, the real copy work has to be done.
- If not operating carefully, the updating on one page can influence the content in another page. That can be insecure because hackers can use this to do arbitrary write.

9.6

> *Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 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

Let see it one by one

- For **first-fit** algorithms, the allocation will be done when it finds suitable block.
    - 115 KB need <== 300 KB allocation
    - 500 KB need <== 600 KB allocation
    - 358 KB need <== 750 KB allocation
    - 200 KB need <== 200 KB allocation
    - 375 KB need <== no memory can be allocated but wait for others to free

- For **best-fit** algorithms, the allocation will be done after finding the most close one
    - 115 KB need <== 125 KB allocation
    - 500 KB need <== 600 KB allocation
    - 358 KB need <== 750 KB allocation
    - 200 KB need <== 200 KB allocation
    - 375 KB need <== no memory can be allocated
- For the **worst-fit**, always the allocate the larget hole
    - 115 KB need <== 750 KB allocation
    - 500 KB need <== 600 KB allocation
    - 358 KB need <== no memory can be allocated
    - 200 KB need <== 350 KB allocation
    - 375 KB need <== no memory can be allocated

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

- For **External fragmentation**:
    - The contiguous memory allocation suffers the problem of external fragmentation! There are many example cases in the book, most of which facing a situation after allocation and free of small-size memory, the larger memory cannot be allocated for there is no **contiguous** free space but small Scattered memory.
    - The paging is designed for solve the problem of external fragmentation, and it works! By using paging, the programmer beleives the memory is contiguous but it is not in physical layer. Permitting noncontiguous logical address psace of processes allow a process to be allocated physical memory wherever such memory is available.
- For **Internal fragmentation**:
    - Unfortunately, these two methods both suffer from internal fragmentation. Frankly speaking, the internal fragmentation issue is related to the process requirement instead of the strategy we adopted in allocating.
- For the **ability to share code across processes**:
    - The paging method's ability of sharing is better thant he contiguous

memory allocation. This is indeed what we talked about in 9.5. Because there are cases where some pages refer to the same physical address, the sharing can be done easily.

9.19

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

Answer

An ASID uniquely identified each process and is used to provide **address-space protection** for that process.

Because the TLB may contains different processes' mapping and entries at the same time, we need an identifier to distinguish them, that is, we will make sure that the ASID for curently running process matches the ASID associated with the virtual page.

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

- For contiguous memoruy allocation
    - This stragtegy can be benefited because the structure divide different part for different usage. The allocation can allocate contiguous memory for the code part and date part, respectively. Taking the contiguous stack for example, it is great to use contiguous allocation for this segment. However, the disadvantage is that the whole memory need to be allocated before starting and may costs wasting.

- For Paging
    - Different segment can also help with do paging, and paging doesn't require the operating system to allocate the maximun extent of the space at startup time. By using methods like **Demand Paging** to speed up the allocation. However, when program needs to be extended in stack or heap it needs to allocate a new page.

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:

- Because the page table is stored in memory, we need to fetch it before we access the real target. The answer should be **100 nanoseconds**.
- After TLB is added,
    - If the entry is present: (2+50) nanoseconds are needed for finding entry and fetch memory
    - If the entry is not present: (2+50*2) nanoseconds are needed for firstly look up the TLB and then fetech page table and fetch memory
    - The final answer is: 2 + 50 * 0.75 + 100 * 0.25 = **64.5 nanoseconds**.

X86-64 address transform

如何传递参数给内核模块

和之前实验实现的内核模块编写不一样，这次需要动态的传递参数以跟踪我们想要的值

- 内核模块参数的传递办法主要依靠宏：

  ```
  module_param(var_name, var_type, var_mode);
  ```

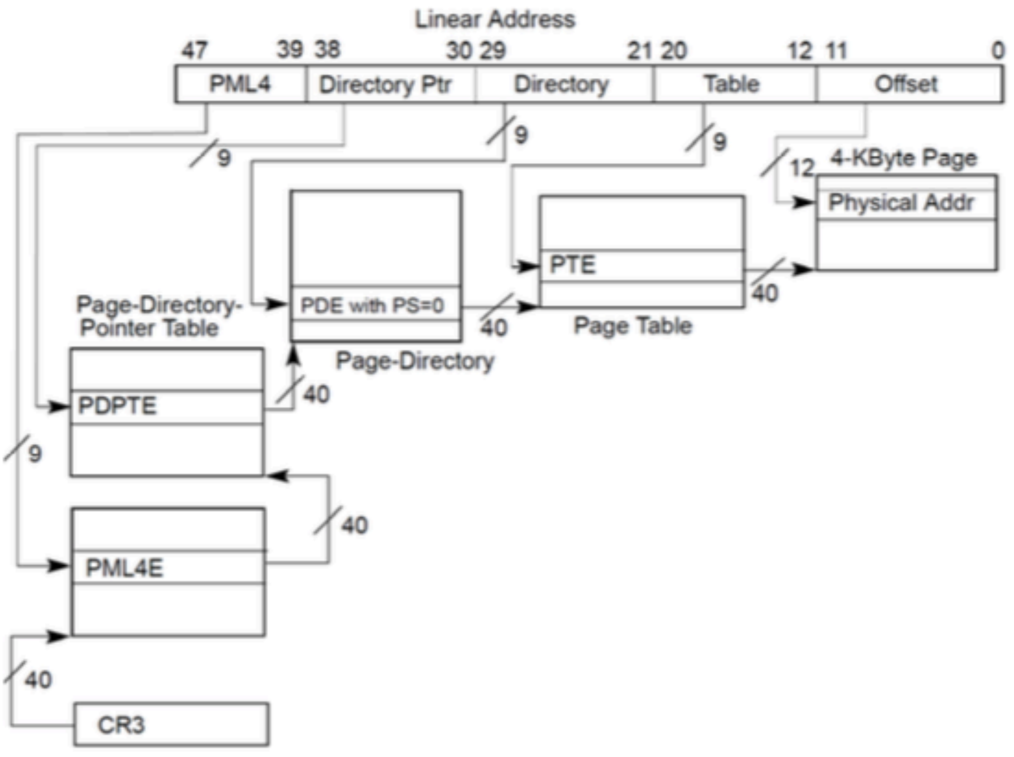
  同时可以通过函数宏描述说明参数

```
MODULE_PARM_DESC()
```

- 参考博客 http://blog.51cto.com/freshpassport/615725

内核模块编写

- 主要依据（参见Linux 内核在 x86_64 CPU 中地址映射）



- 地址转换部分
    - 分析请见注释内容

```
//=================================================================
=======
    // Get global descriptor table (gdt) entry value
    store_gdt(&gdtr);
    gdt_entry_address = gdtr.address + (__USER_DS - 3); // _USER_DS
== (GDT_ENTRY_DEFAULT_USER_DS * 8 + 3) so need to subtract 3
    gdt_entry = *(unsigned long*)(gdt_entry_address);
    printk(KERN_INFO "[*] step 1: gdtr %lx get entry [%lx]: %lx \n",
gdtr.address, gdt_entry_address, gdt_entry);

//=================================================================
=======
    // Calculate the linear address
    /*
     * gdt_entry[63:52] <= Base[31:24]
     * gdt_entry[39:32] <= Base[23:16]
     * gdt_entry[31:16] <= Base[15:00]
     * Why simply addition?
     * Bug here, but no effect for base is 0 always
     */
    gdtr_base = ((gdt_entry >> 16) & 0xFFFF) + (((gdt_entry >> 32) &
0xFF) << 16) + (((gdt_entry >> 56) & 0xFF) << 24);
    linear_addr = addr + gdtr_base;
    printk(KERN_INFO "[*] step 2: base addr: %lx linear addr %lx\n",
gdtr_base, linear_addr);

//===================================================================
=======
    // Find the pgd of that process (which is equal to CR3)
    pgd = get_pgd(pid);

//===================================================================
=======
    // First level: PML4E
    /*
     * LinearAddr[47:39] 9 bits
     * value * 8 correspond to the entry size (that is, 64 bits)
     * attention here the CR3 store the virtual address because PAE
used
     */
    temp = pgd + GET_FIRST_INDEX(linear_addr) * 8;
    printk(KERN_INFO "[*] step 3: pgd [%lx] entry addr: [%lx]:
%lx\n", pgd, temp, read_user(temp, 1));

//===================================================================
=======
    // Second level
    /*
     * LinearAddr[38:30] 9 bits
     * value * 8 correspond to the entry size (that is, 64 bits)
     * physical address since
     */
    second_page = read_user(temp, 1) & (0xfffffffff000);    // the
000 indicates 12 bit for page property
    temp = second_page + GET_SECOND_INDEX(linear_addr) * 8;
    printk(KERN_INFO "[*] step 4: second level: [%lx] entry addr:
[%lx]: %lx\n", second_page, temp, read_user(temp, 0));

//===================================================================
=======
    // Third level
    /*
     * LinearAddr[29:21] 9 bits
     * value * 8 correspond to the entry size (that is, 64 bits)
     */
    third_page = read_user(temp, 0) & (0xfffffffff000);
    temp = third_page + GET_THIRD_INDEX(linear_addr) * 8;
    printk(KERN_INFO "[*] step 5: third level: [%lx] entry addr:
[%lx]: %lx\n", third_page, temp, read_user(temp, 0));

//==================================================================
=======
    // Fourth level
    /*
     * LinearAddr[20:12] 9 bits
     * value * 8 correspond to the entry size (that is, 64 bits)
     */
    fourth_page = read_user(temp, 0) & (0xfffffffff000);
    temp = fourth_page + GET_FOURTH_INDEX(linear_addr) * 8;
    printk(KERN_INFO "[*] step 6: fourth level: [%lx] entry addr:
[%lx]: %lx\n", fourth_page, temp, read_user(temp, 0));

//==================================================================
=======
    // Physical level
    /*
     * LinearAddr[11:0] 12 bits offset
     */
    physical_frame = read_user(temp, 0) & (0xfffffffff000);
    temp = physical_frame + (linear_addr & 0xfffLL);
    printk(KERN_INFO "[*] step 7: physical page frame: [%lx]
physical addr: [%lx]: %lx\n", physical_frame, temp, read_user(temp,
0));

//==================================================================
=======
    // done
```

- Makefile部分

```
obj-m += mapping.o
KERNELBUILD := /lib/modules/$(shell uname -r)/build
default:
        make -C $(KERNELBUILD) M=$(shell pwd) modules
clean:
        rm -rf *.o *.ko *.mod.c .*.cmd *.markers *.order *.symvers
.temp_versions
```

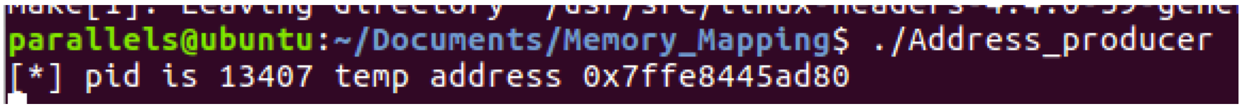
用户态程序编写

- 编写用户态程序用于得到进程的pid和局部变量的逻辑地址

```c
#include <stdio.h>
int main()
{
    long temp = 0x12345678aabbccdd;
    printf("[*] pid is %d temp address %p \n", getpid(), &temp);
    while (1);
    return 0;
}
```

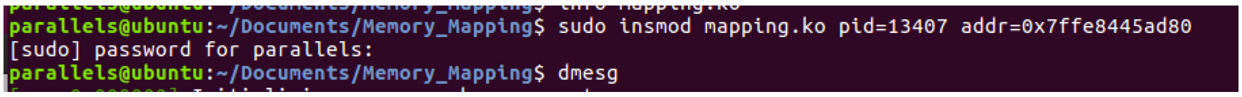
结果分析

- 进程的 pid 与变量地址
    - 可得 pid = 13407, adds = 0x7ffe8445ad80



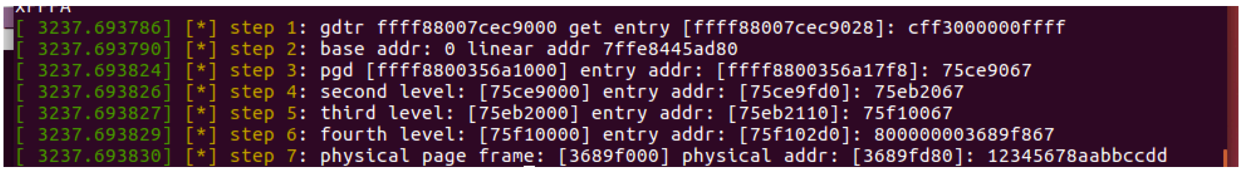
```
parallels@ubuntu:~/Documents/Memory_Mapping$ ./Address_producer
[*] pid is 13407 temp address 0x7ffe8445ad80
```

- 装载内核模块
    - 保证模块已经正确编译



```
parallels@ubuntu:~/Documents/Memory_Mapping$ sudo insmod mapping.ko pid=13407 addr=0x7ffe8445ad80
[sudo] password for parallels:
parallels@ubuntu:~/Documents/Memory_Mapping$ dmesg
```

- 查看结果
    - 我们可以看到地址转换过程中的每一步，同时最后变量的值0x12345678aabbccdd和预期完全一致，故实验成功



```
[ 3237.693786] [*] step 1: gdtr ffff88007cec9000 get entry [ffff88007cec9028]: cff3000000ffff
[ 3237.693790] [*] step 2: base addr: 0 linear addr 7ffe8445ad80
[ 3237.693824] [*] step 3: pgd [ffff8800356a1000] entry addr: [ffff8800356a17f8]: 75ce9067
[ 3237.693826] [*] step 4: second level: [75ce9000] entry addr: [75ce9fd0]: 75eb2067
[ 3237.693827] [*] step 5: third level: [75eb2000] entry addr: [75eb2110]: 75f10067
[ 3237.693829] [*] step 6: fourth level: [75f10000] entry addr: [75f102d0]: 800000003689f867
[ 3237.693830] [*] step 7: physical page frame: [3689f000] physical addr: [3689fd80]: 12345678aabbccdd
```