Lab 4

计 83 李天勤 2018080106

实验目的

- ch4
- 更新 sys_write 的范围检查，改为基于页表的检查方法。
- 实现 mmap 和 munmap 两个自定义系统调用，并通过 C测例中chapter4对应的所有测例。

实验结果

Expected Result from testcases:

```
ch4_mmap0.cpp Test 04_1 OK!
ch4_mmap1.cpp 程序触发访存异常，被杀死。不输出 error 就算过
ch4_mmap2.cpp 程序触发访存异常，被杀死。不输出 error 就算过。
ch4_mmap3.cpp 对于错误的 mmap 返回 -1，最终输出 Test 04_4 test OK!
ch4_unmap.cpp  输出 Test 04_5 ummap OK!
ch4_unmap2.cpp 输出 Test 04_6 ummap2 OK!
```

After running

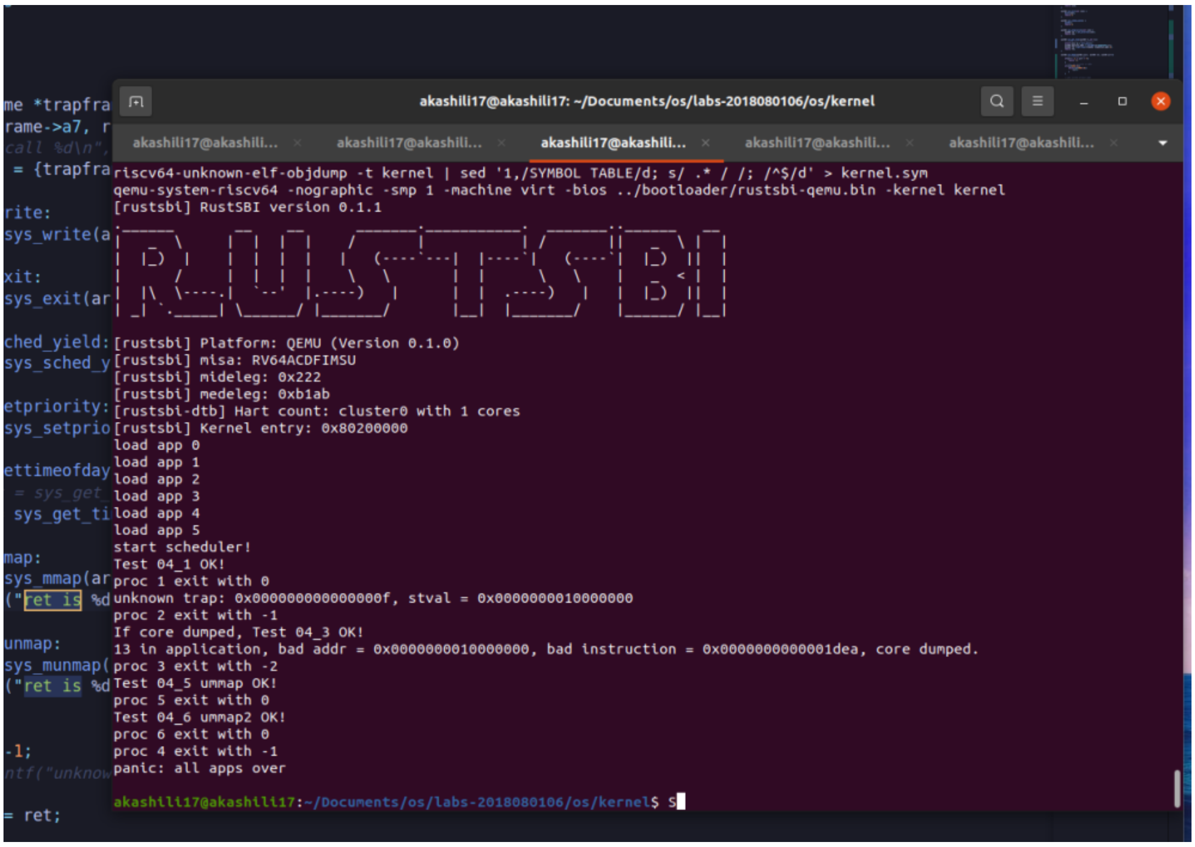
```
~/path/labs-2018080106/os/user$ make clean
~/path/labs-2018080106/os/user$ make all CHAPTER=4
~/path/labs-2018080106/os/user$ cd ../kernel
~/path/labs-2018080106/os/kernel$ make clean
~/path/labs-2018080106/os/kernel$ make run
```

I get the result

```
load app 0
load app 1
load app 2
load app 3
load app 4
load app 5
load app 6
load app 7
load app 8
load app 9
load app 10
load app 11
load app 12
start scheduler!
proc 1 exit with 1234
3^10000=5079
3^20000=8202
3^30000=8824
3^40000=5750
3^50000=3824
3^60000=string from data section
```

```
strinstring from stack section
strin
Test write1 OK!
proc 4 exit with 0
Test set_priority OK!
proc 5 exit with 0
get_time OK! 636
current time_msec = 639ret is 4096
Test 04_1 OK!
proc 8 exit with 0
ret is 4096
unknown trap: 0x000000000000000f, stval = 0x0000000010000000
proc 9 exit with -1
ret is 4096
ret is -1
ret is 4096
proc 11 exit with -1

Test 04_5 ummap OK!
proc 12 exit with 0
ret is 4096
ret is -1
ret is -1
Test 04_6 ummap2 OK!
proc 13 exit with 0
Hello world from user mode program!
Test hello_world OK!
proc 2 exit with 0
8516
3^70000=2510
3^800ret is 4096
If core dumped, Test 04_3 OK!
13 in application, bad addr = 0x0000000010000000, bad instruction =
0x0000000000001dea, core dumped.
proc 10 exit with -2
00=9379
3^90000=2621
3^100000=2749
Test power OK!
proc 3 exit with 0
time_msec = 744 after sleeping 100 ticks, delta = 105ms!Test sleep1 passed!
proc 7 exit with 0
Test sleep OK
proc 6 exit with 0
panic: all apps over
```

To make it clearer, I just ran test cases for chapter 4.

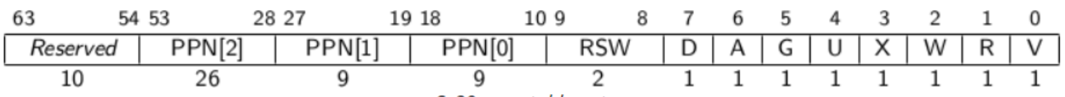


```
riscv64-unknown-elf-objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
qemu-system-riscv64 -nographic -smp 1 -machine virt -bios ../bootloader/rustsbi-qemu.bin -kernel kernel
[rustsbi] RustSBI version 0.1.1
.______       __    __      _______.___________.  _______..______   __
|   _  \     |  |  |  |    /       |           | /       ||   _  \ |  |
|  |_)  |    |  |  |  |   |   (----`---|  |----`|   (----`|  |_)  ||  |
|      /     |  |  |  |    \   \       |  |      \   \    |   _  < |  |
|  |\  \----.|  `--'  |.----)   |      |  |  .----)   |   |  |_)  ||  |
| _| `._____| \______/ |_______/       |__|  |_______/    |______/ |__|

[rustsbi] Platform: QEMU (Version 0.1.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
load app 0
load app 1
load app 2
load app 3
load app 4
load app 5
start scheduler!
Test 04_1 OK!
proc 1 exit with 0
unknown trap: 0x000000000000000f, stval = 0x0000000010000000
proc 2 exit with -1
If core dumped, Test 04_3 OK!
13 in application, bad addr = 0x0000000010000000, bad instruction = 0x0000000000001dea, core dumped.
proc 3 exit with -2
Test 04_5 ummap OK!
proc 5 exit with 0
Test 04_6 ummap2 OK!
proc 6 exit with 0
proc 4 exit with -1
panic: all apps over

akashili17@akashili17:~/Documents/os/labs-2018080106/os/kernel$ $
```

** commented out `printf("ret is...`

实验问答

1. 请列举 SV39 页表页表项的组成，结合课堂内容，描述其中的标志位有何作用 / 潜在作用？



Sv39 page table entry.

   As shown in the picture above, every SV39 PTE (page table entry) has 64 bits. The highest 10 weights are reserved, 53-28 are for PPN2, 27-19 are for PPN1, 19-10 are for PPN0, bits9-9 are reserved for Supervisor mode, and 7 to 0 are for flags. Their functionality include

   - Dirty: indicates that page has been written to.
   - Accessed: indicates that it has been accessed
   - Global: Global address mapping
   - User: indicates that can be accessed by User mode
   - X(ex)cutable: indicates that it can be executed
   - Writable: indicates that it can be written to
   - Readable: indicates that it can be read

2. 缺页

   这次的实验没有涉及到缺页有点遗憾，主要是缺页难以测试，而且更多的是一种优化，不符合这次实验的核心理念，所以这里补两道小题。

   缺页指的是进程访问页面时页面不在页表中或在页表中无效的现象，此时 MMU 将会返回一个中断，告知 os 进程内存访问出了问题。os 选择填补页表并重新执行异常指令或者杀死进程。

   - 请问哪些异常可能是缺页导致的？

     【解答】Instruction/Load/Store Page Fault.

   - 发生缺页时，描述相关的重要寄存器的值（lab2中描述过的可以简单点）。

【解答】`mcause` and `mtval` and values can represent reading, writing, storing, and memory location exceptions

缺页有很多可能原因，其中之一是 Lazy 策略，也就是直到内存页面被访问才实际进行页表操作。比如，一个程序被执行时，进程的代码段理论上需要从磁盘加载到内存。但是 os 并不会马上这样做，而是会保存 .text 段在磁盘的位置信息，在这些代码第一次被执行时才完成从磁盘的加载操作。

- 这样做有哪些好处？

  【解答】Due to the high cost of disk access, operations that delay and avoid disk access are likely to improve efficiency. For example, when a program is executed, not all its code is used immediately, and not all its related data are accessed immediately. Therefore, only the code segments executed immediately are loaded, the rest are delayed, saving disk space

此外 COW(Copy On Write) 也是常见的容易导致缺页的 Lazy 策略，这个之后再说。其实，我们的 mmap 也可以采取 Lazy 策略，比如：一个用户进程先后申请了 10G 的内存空间，然后用了其中 1M 就直接退出了。按照现在的做法，我们显然亏大了，进行了很多没有意义的页表操作。

- 请问处理 10G 连续的内存页面，需要操作的页表实际大致占用多少内存(给出数量级即可)？

  【解答】One of the smallest pages of sv39 is 4 KiB, and the large pages are 2 MiB and 1 GiB. the permissions are consistent, 10 large pages can be used to manage the 10GiB space, and the corresponding page table overhead is only 80 B. If you want to subdivide the permissions according to the minimum page, the number of PTEs required is 8 B each, where the main part takes about 20 MIB

- 请简单思考如何才能在现有框架基础上实现 Lazy 策略，缺页时又如何处理？描述合理即可，不需要考虑实现。

  【解答】 When SYS_MMAP is called, or when the program is loaded, it records which virtual addresses are mapped, but does not operate the page table. After the program accesses the unmapped address, a page fault exception is triggered, and the mapping record is checked in the exception handling function.

  If the accessed virtual address has been mapped already or should have been loaded before, the actual operation page table allocates memory for it, and then returns to the user program

缺页的另一个常见原因是 swap 策略，也就是内存页面可能被换到磁盘上了，导致对应页面失效。

- 此时页面失效如何表现在页表项(PTE)上？

  【解答】If the XWR of a page is not all zero, but V is zero, it means that the page is meaningful and not in memory, which in turn means that it is in external storage.

3. 双页表与单页表

   为了防范侧信道攻击，我们的 os 使用了双页表。但是传统的设计一直是单页表的，也就是说，用户线程和对应的内核线程共用同一张页表，只不过内核对应的地址只允许在内核态访问。(备注：这里的单/双的说法仅为自创的通俗说法，并无这个名词概念，详情见KPTI)

   - 如何更换页表？

     【解答】Because `satp` CSR manages the address of the root of the page table, it modifies `satp` when replacing the page table, and then clears the TLB to avoid the previous cached page table entries interfering with the new page table

   - 单页表情况下，如何控制用户态无法访问内核页面？（tips:看看上一题最后一问）

     【解答】If the U flag of kernel page is set to 0, MMU will check the current privilege level during translation. If it is in U mode, MMU will report `page fault exception`

- 单页表有何优势？（回答合理即可）

  【解答】The thread switch is relatively less, and the privilege level switch of system call is frequent. A thread of single page table only uses one page table, so it is more efficient not to switch page table during the switching of privileges'

- 双页表实现下，何时需要更换页表？假设你写一个单页表操作系统，你会选择合适更换页表（回答合理即可）？

  【解答】The double page table should be replaced when switching at privilege level or thread switching

  For single page table system, change the page table when switching the process