

PA3 实验报告

14307130078 张博洋

一、实验进度

完成了 PA3 所有内容（实际上我已经完成 PA4 所有内容）。

二、部分选答题

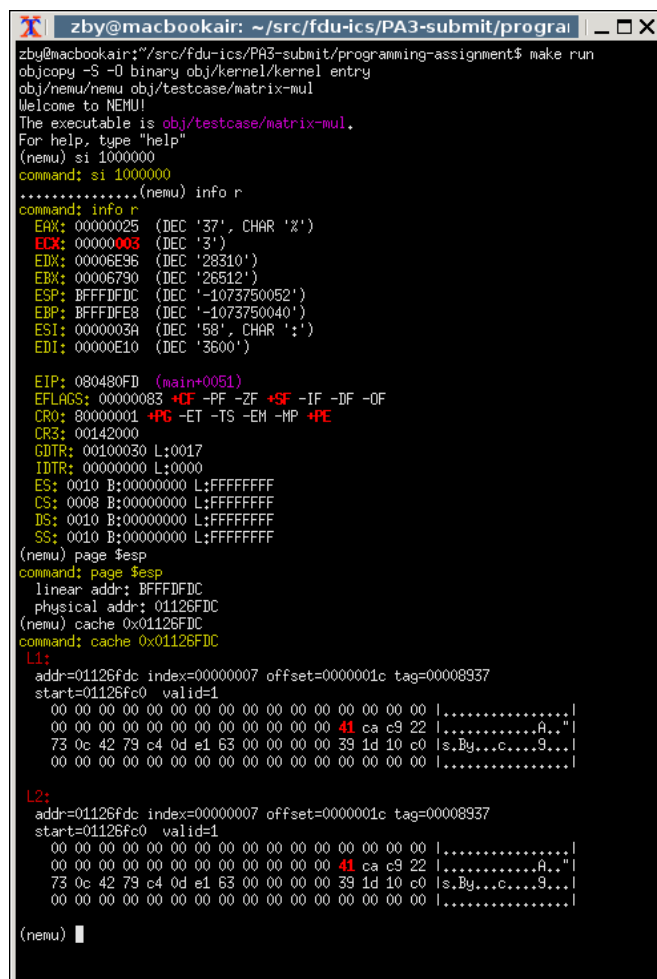
1. 理解 DRAM 的工作方式

如果编译器分配变量的时候没有对齐，那么这个变量所处的位置很有可能跨越两个行的边界，或者跨越 BURST 的边界。在这两种情况下，原本一次就能完成的读写操作，需要两次才能完成，使得读写的效率变低。

2. 简易调试器 (3)

实现的 cache 调试器命令截图如下：

```
si 1000000 //执行 1000000 条指令
info r // 查看寄存器内容
page $esp // 由于开启了页转换，先转换为物理地址
cache 0x01126fdc // 查找物理地址 0x01126fdc 对应的 cache 内容
```



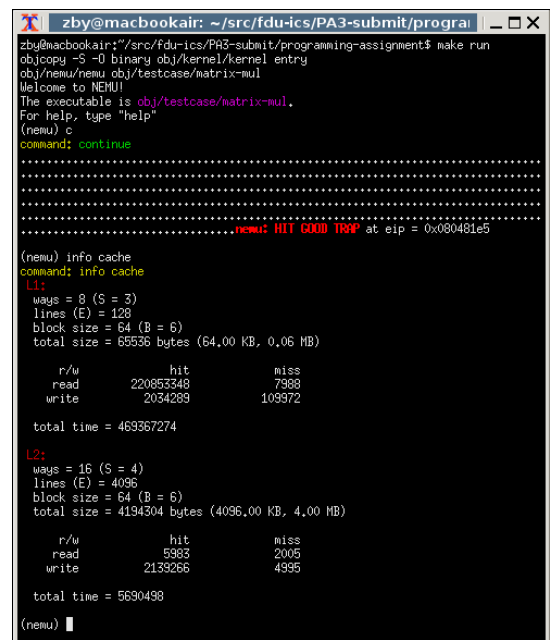
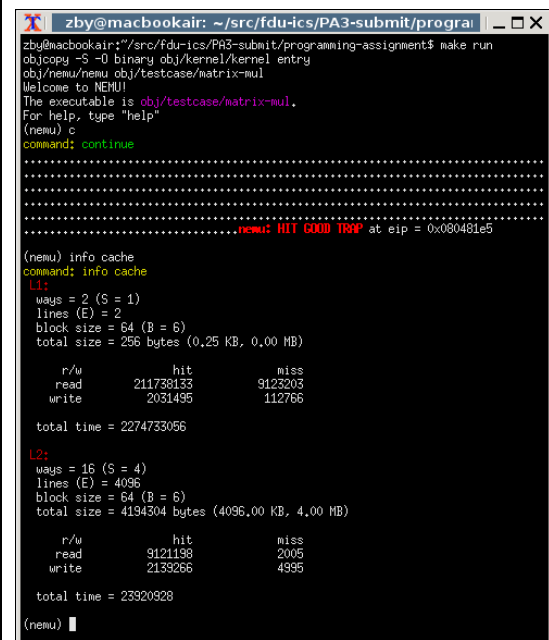
```
zby@macbookair: ~/src/fdu-ics/PA3-submit/program
zby@macbookair:~/src/fdu-ics/PA3-submit/program$ make run
objcopy -S -O binary obj/kernel/kernel entry
obj/nemu/nemu obj/testcase/matrix-mul
Welcome to NEMU!
The executable is obj/testcase/matrix-mul.
For help, type "help"
(nemu) si 1000000
command: si 1000000
.....(nemu) info r
command: info r
EAX: 00000025 (DEC '37', CHAR '%')
ECX: 00000003 (DEC '3')
EDX: 00006E96 (DEC '28310')
EBX: 00006790 (DEC '26512')
ESP: BFFFFDFC (DEC '-1073750052')
EBP: BFFFFDF8 (DEC '-1073750040')
ESI: 0000003A (DEC '58', CHAR ':')
EDI: 00000E10 (DEC '3600')

EIP: 080480FD (main+0051)
EFLAGS: 00000083 +CF -PF -ZF +SF -IF -DF -OF
CR0: 80000001 +PG -ET -TS -EM -MP +PE
CR3: 00142000
GDTR: 00100030 L:0017
IDTR: 00000000 L:0000
ES: 0010 B:00000000 L:FFFFFFFF
CS: 0008 B:00000000 L:FFFFFFFF
DS: 0010 B:00000000 L:FFFFFFFF
SS: 0010 B:00000000 L:FFFFFFFF
(nemu) page $esp
command: page $esp
linear addr: BFFFFDFC
physical addr: 01126FDC
(nemu) cache 0x01126FDC
command: cache 0x01126FDC
L1:
addr=01126fdc index=00000007 offset=0000001c tag=00008937
start=01126fc0 valid=1
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00 00 00 00 00 00 00 00 00 00 00 00 00 41 ca c9 22 |.....A..|
73 0c 42 79 c4 0d e1 63 00 00 00 00 39 1d 10 c0 |s.By...c...9...|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
L2:
addr=01126fdc index=00000007 offset=0000001c tag=00008937
start=01126fc0 valid=1
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00 00 00 00 00 00 00 00 00 00 00 00 00 41 ca c9 22 |.....A..|
73 0c 42 79 c4 0d e1 63 00 00 00 00 39 1d 10 c0 |s.By...c...9...|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
(nemu) █
```

可以看出，栈顶附近的内存既在 L1 Cache 中，也在 L2 Cache 中。

3. 观察 cache 的作用

我在 NEMU 里实现了 info cache 指令用来显示 cache 的统计信息，可以显示各个缓存的命中次数、未命中次数和程序的“运行时间”。

<p>L1: 8 组，每组 128 行，块大小 64 字节 L1 缓存总大小: 64KB</p> 	<p>L1: 2 组，每组 2 行，块大小 64 字节 L1 缓存总大小: 256B</p> 
---	--

可以看出，对于 L1 Cache，将总大小由 64KB 减小为 256B 后，L1 命中次数 (hit) 略微减小，但是 L1 未命中次数 (miss) 显著增大。因此，程序的“运行时间”增大了约 5 倍。

4. Icache 和 Dcache

我没有实现指令缓存。但是我思考了将数据缓存和指令缓存分开实现时可能遇到的问题：有一些程序可能会在执行过程中修改自身的指令（自修改程序）。如果将数据缓存与指令缓存分离，那么新的指令只会被写入数据缓存，而指令缓存存储的仍然是旧的指令。这样一来就会发生不一致问题。解决的办法是，在写入数据时，检查指令缓存中的内容，并让对应的缓存行失效。这样一来便可以解决这种不一致问题。

5. 为什么要采用这种有重叠的寻址方式？

如果段寄存器不是 16 位而是 4 位，那就意味着每个段的基地址只能是 64K 的倍数，因此即使计算机装配了 1MB 内存，所有可能的段的基地址也只有 16 个。这样一来，基地址取值范围非常小，几乎不能随意设置基地址，不方便使用。

6. GDT 能有多大？

从段选择子来看：INDEX 域占用 12 个位，因此最多可索引 $2^{12}=8196$ 个段描述符。

从 GDT 的 LIMIT 来看：LIMIT 最大为 0xFFFF，这意味着 GDT 的最大大小为 0x10000B=64KB，由于每一个段描述符占用 8B，因此最多可存 8K 个段描述符。

7. 为什么是线性地址？

如果 GDT 中存放的是虚拟地址，那么意味着访问 GDT 本身就需要进行段级地址转换，这会造成一种无尽的递归：段级地址转换需要访问 GDT，访问 GDT 需要段级地址转换.....这种递归会无穷尽地下去，因此 GDT 中不能存放虚拟地址。

8. 段式存储管理的缺点

段式存储管理的缺点在于，系统中段的数量只有为数不多的几个，并且一个段对应的是一整段物理内存，而且程序使用的内存数量不可以超过物理内存数量。这样一来实现多任务操作系统很困难。假设操作系统把 B 程序的数据段紧挨着放在了 A 程序的数据段之后，如果 A 程序想要增大数据段的大小（动态分配一些内存），那么就必须对 B 程序进行搬移，因此代价很大，实现起来也很麻烦。

9. 页式存储管理的优点

页式存储管理的优点是，页式映射比段式映射更灵活，程序使用的内存数量可以超过物理内存的数量（可以把暂时不用的内存页交换到磁盘上）。如果程序想要动态分配内存，只需要找到一些物理页面，将它们映射到所需位置上即可。在这个过程中，并不需要物理页面连续，就可以为程序分配连续的虚拟内存。

10. 被"淘汰"的段式存储管理

测试样例 struct.c 编译后有一条指令 `lea 0x4(%esp), %edx` 是用来计算结构体分量的地址的，由于 `%esp` 寄存器中存储的栈指针是相对堆栈段基地址的偏移量，因此实际上存入 `%edx` 的是该分量相对于堆栈段基地址的偏移量。而在随后的代码中，GCC 直接使用 `mov` 指令，用数据段来索引对应的内容。在平坦模式下，这没有什么问题，但是在修改了堆栈段的基地址之后，这样便相当于少加了一个基地址，因此程序出现了错误。

11. 一些问题

(1) 为什么表项中的基地址信息只有 20 位，而不是 32 位？

由于物理页一定是按照页大小对齐。由于 x86 页大小为 $4KB=2^{12}B$ ，因此可以省略掉 12 位。

(2) 物理地址是必须的吗？能否使用虚拟地址或线性地址？

不可以，原因同“GDT 不可以使用虚拟地址”。

(3) 为什么不采用一级页表？或者说采用一级页表会有什么缺点？

如果使用一级页表，那么页表项将达到 $2^{20}=1M$ 个，占用内存为 4MB。这再早期内存较小的计算机上是不可接受的。

12. 空指针真的是"空"的吗？

空指针之所以为空，是因为线性地址 0 处没有映射。如果将线性地址 0 所在的虚拟页找到一个物理页与它建立映射，那么是完全可以使用线性地址 0 的。

13. 在扁平模式下如何进行保护？

虽然从段级保护层面看，程序可以随意访问整个地址空间的内容，但是由于线性地址还需要做页级地址转换才会变为物理地址，而在这个转换中页级保护会保证程序不能随意访问它不该访问的内容。

14. 团结力量大

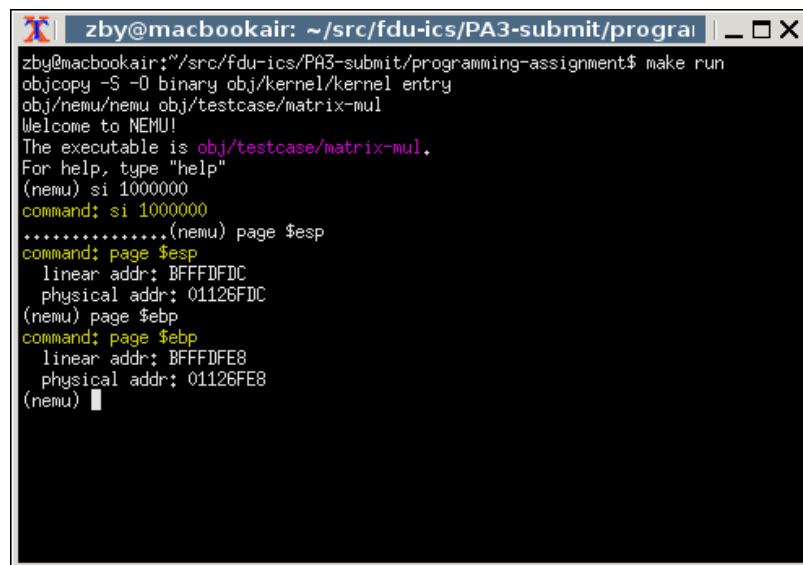
链接选项 `Ttext` 指定了内核最终被映射到哪里（即内核代码认为它会在哪里运行），`KOFFSET` 指示内核所在的物理地址（即 NEMU 把内核加载到哪里）与最终运行的虚拟地址直接的偏移量，如果三者任一出现错误，那么内核的启动过程就会存在使用错误地址的情况，进而导致内核运行失败。

15. 不连续的物理页面

我的 loader 代码支持把程序加载到不连续的物理页面中。这只需要把数据的复制方式由直接整段复制改为按页为单位复制数据即可。

16. 简易调试器 (4)

我实现了 `page` 命令，截图如下：



```
zby@macbookair: ~/src/fdu-ics/PA3-submit/program
zby@macbookair:~/src/fdu-ics/PA3-submit/programming-assignment$ make run
objcopy -S -O binary obj/kernel/kernel entry
obj/nemu/nemu obj/testcase/matrix-mul
Welcome to NEMU!
The executable is obj/testcase/matrix-mul.
For help, type "help"
(nemu) si 1000000
command: si 1000000
.....(nemu) page $esp
command: page $esp
linear addr: BFFFFDFDC
physical addr: 01126FDC
(nemu) page $ebp
command: page $ebp
linear addr: BFFFFFE8
physical addr: 01126FE8
(nemu) █
```

可以看出线性地址 `0xBFFFFDFDC` 对应物理地址 `0x01126FDC` 等。

17. 有本事就把我找出来！

这段代码的问题在于，`pframe_addr` 是无符号类型，而 `for` 循环的循环条件是 `pframe_addr >= 0`，这个式子显然成立（无符号数当然大于等于 0）。在循环的最后，在对 `pframe_addr` 做减法运算时，会发生一次溢出，导致循环永远不会结束。

18. 暗藏杀机的分页机制

(1) 为什么在开启分页之前不能使用全局变量，但却可以使用局部变量？

不能使用全局变量是因为全局变量被分配在 `bss` 或 `data` 节中，而在建立页表、开启分页前，`bss` 或 `data` 节对应地址出根本没有内存与它对应。可以使用局部变量是因为，局部变量存储在栈上，是根据栈指针来寻址的，只要设置好栈指针（在 `start.S` 中设置好了），让栈指针指向的位置向下有一段空间，就可以正常使用局部变量了。

(2) 仍然使用了一些全局变量，但却没有造成错误，这又是为什么？

不能直接使用全局变量是因为，此时访问全局变量是使用建立映射后的地址去访问，而此时映射还没有建立起来，而如果仔细小心地用 `va_to_pa()` 宏把

映射后的地址转换为真正的物理地址，就可以使用全局变量了。

(3) 为什么调用位于高地址的 `init_page()` 却不会发生错误？

因为一般的函数调用都是通过“`call rel32`”指令完成的，而此指令是“相对跳转”，由于建立映射前（即内核运行在低地址）和映射后（即内核运行在高地址）这两种状态下，各个函数之间相对的位置并不会发生变化。“相对跳转”并不会造成任何问题。

(4) 在 `init_page()` 的循环中有这样两行代码，尝试注释掉其中一行，重新编译 `kernel` 并运行，你会看到发生了错误。请解释这个错误具体是怎么发生的。

注释掉第一行：

结合 NEMU 的错误信息和 log 可以看到，在刚开启分页后（即刚刚执行完 `mov %eax, %cr0`（即置 CR0 的 PG 位），由于低地址部分没有建立映射，当前 EIP 指向的位置的内容消失掉了，NEMU 找不到下一条指令于是崩溃。

注释掉第二行：

结合 NEMU 的错误信息和 log 可以看到，在 `kernel/src/main.c` 文件的 `init()` 函数中，最后有个 `jmp` 指令用来让内核跳到高地址上去执行。但是由于高地址处根本没有建立映射，于是 NEMU 找不到下一条指令就崩溃了。

(5) 在 `init()` 函数中，我们通过内联汇编把 `%esp` 加上 `KOFFSET` 转化成相应的虚拟地址。尝试把这行内联汇编注释掉，重新编译 `kernel` 并运行，你会看到发生了错误。请解释这个错误具体是怎么发生的。

注释掉这句话后，栈指针在内核跳到高地址去执行之后，仍然指向低地址的内存区域。虽然在内核的页表中，高地址（即 `0xc0000000`）和低地址（即 `0`）起始的区域都映射到了实际的物理内存，但是在用户程序的页表中，低地址区域没有这样的映射，只有高地址区域映射到了实际的物理内存。因此，在 `loader()` 函数加载完用户程序，把 CR3 切换到了用户页表之后，内核的栈就从虚拟内存中消失了，因此内核代码在访问栈上数据时就崩溃了。

(6) 在 `init()` 函数中，我们通过内联汇编间接跳转到 `init_cond()` 函数。尝试把这行内联汇编注释掉，改成通过一般的函数调用来跳转到 `init_cond()` 函数，重新编译 `kernel` 并运行，你会看到发生了错误。请解释这个错误具体是怎么发生的。

改成一般的函数调用后，反映到汇编层面就是使用 `call rel32` 指令去跳转到 `init_cond()`，由于 `call rel32` 指令是相对跳转，这样内核仍然在低地址区域运行，根本没跳到应该去的高地址处。因此因为之前的全局变量问题而崩溃。

(7) 在 `init_mm()` 函数中，有一处代码用于拷贝 `0xc0000000` 以上的内核映射，尝试注释这处代码，重新编译 `kernel` 并运行，你会看到发生了错误。请解释这个错误具体是怎么发生的。

`init_mm()` 函数的作用是，把内核的页目录的高地址部分，拷贝到用户程序的页目录的高地址部分，即在用户程序的页表中的高地址部分建立与内核页表相同的映射。这样做，一是为了内核在切换到用户页表后，内核可以继续运行，不至于因为找不到内核代码而崩溃；二是为了在用户程序中，能够让用户程序陷入内核代码中（如果整个地址空间没有内核代码，CPU 在陷入内核的时候就找不到内核代码了）。如果注释掉这处代码，用户页表的高地址部分将是一片未映射的区域，会导致在 `loader()` 函数中切换页表时，CPU 由于找不到下一条指令而崩溃。

19. 实现 TLB

我实现了一个额外的 `info tlb` 命令，可以用来查看当前 TLB 中的内容。

```
zby@macbookair: ~/src/fdu-ics/PA3-submit/programing-assignment$ make run
objcopy -S -O binary obj/kernel/kernel entry
obj/nemu/nemu obj/testcase/matrix-mul
Welcome to NEMU!
The executable is obj/testcase/matrix-mul.
For help, type "help"
(nemu) c
command: continue
.....
.....
.....
.....
.....nemu: HIT GOOD TRAP at eip = 0x080481e5
(nemu) info tlb
command: info tlb
TLB:
   0: Y C0100000 => 00100000      1: Y C7FFE000  => 07FFE000
   2: Y C7FFF000  => 07FFF000      3: Y C0101000  => 00101000
   4: Y BFFFD000  => 01126000      5: Y 08048000  => 01000000
   6: Y 08066000  => 0101E000      7: Y 08049000  => 01001000
   8: Y 08052000  => 0100A000      9: Y 08053000  => 0100B000
  10: Y 08054000  => 0100C000     11: Y 08055000  => 0100D000
  12: Y 08056000  => 0100E000     13: Y 08057000  => 0100F000
  14: Y 08058000  => 01010000     15: Y 08059000  => 01011000
  16: Y 0805A000  => 01012000     17: Y 0805B000  => 01013000
  18: Y 0805C000  => 01014000     19: Y 0805D000  => 01015000
  20: Y 08067000  => 0101F000     21: Y 0804A000  => 01002000
  22: Y 0805E000  => 01016000     23: Y 08068000  => 01020000
  24: Y 0804B000  => 01003000     25: Y 0805F000  => 01017000
  26: Y 08069000  => 01021000     27: Y 0804C000  => 01004000
  28: Y 08060000  => 01018000     29: Y 0806A000  => 01022000
  30: Y 0804D000  => 01005000     31: Y 08061000  => 01019000
  32: Y 0806B000  => 01023000     33: Y 0804E000  => 01006000
  34: Y 08062000  => 0101A000     35: Y 0806C000  => 01024000
  36: Y 0804F000  => 01007000     37: Y 08063000  => 0101B000
  38: Y 0806D000  => 01025000     39: Y 08050000  => 01008000
  40: Y 08064000  => 0101C000     41: Y 0806E000  => 01026000
  42: Y 08051000  => 01009000     43: Y 08065000  => 0101D000
  44: Y 0806F000  => 01027000     45: Y 08070000  => 01028000
  46: N 00000000  => 00000000     47: N 00000000  => 00000000
  48: N 00000000  => 00000000     49: N 00000000  => 00000000
  50: N 00000000  => 00000000     51: N 00000000  => 00000000
  52: N 00000000  => 00000000     53: N 00000000  => 00000000
  54: N 00000000  => 00000000     55: N 00000000  => 00000000
  56: N 00000000  => 00000000     57: N 00000000  => 00000000
  58: N 00000000  => 00000000     59: N 00000000  => 00000000
  60: N 00000000  => 00000000     61: N 00000000  => 00000000
  62: N 00000000  => 00000000     63: N 00000000  => 00000000
(nemu)
```

图中，绿色的项是有效的项，而黄色的项是无效的项（还没用到的项）。可以看出，对于 `matrix-mul` 来说，64 项的 TLB 都没有用满，因此 TLB 命中率极高（除了第一次访问没有命中，剩下的每次访问 TLB 都命中了）。

三、必答题

结合 `kernel` 的框架代码理解分页机制

