Lab3 实验报告

游子诺

2019 年 4 月 25 日

1 实验思考题

Thinking 1

为什么我们在构造空闲进程链表时必须使用特定的插入的顺序？(顺序或者逆序)

这样做使空闲进程块的访问变得规律，例如通过逆序遍历 envs 数组 + 头部插入 free_list 链表，链表中最终的进程块呈 envs 递增，当第一次使用 env_alloc 函数时，返回的一定是 env[0] 进程块。

Thinking 2

思考 env.c/mkenvid 函数和 envid2env 函数:

- 请你谈谈对 mkenvid 函数中生成 id 的运算的理解，为什么这么做？
- 为什么 envid2env 中需要判断 e->env_id != envid 的情况？如果没有这步判断会发生什么情况？

mkenvid 运算

$$id = (++count) << (1 + log2env)|idx$$

由此可以看出 id 实际由两部分所组成，第一部分是根据进程块结构体在进程块数组的相对位置而得到的 idx，但是仅有此并不能创造唯一的进程，因为同一个进程块通过销毁复用可以被多个进程所使用，因此还需要在 32 位 id 的高位（idx 无法达到的高位）或上进程创建的序号，程序利用了 C 语言中的 static 变量统计次数，最终两个部分组成了 id。

envid2env 条件判断 和 mkenvid 运算得到的 id 一样,对于通过 e = envs[ENVX(envid)] 而得到的进程块，造成 env_id != envid 的情况就是高位序号的不同，在没有这一步的情况下，当给出一个处在 FREE 状态的进程块 id，理应的查找结果是-E_NO_ENV，但

实际会返回一个不存在的进程（进程从没有被创建过或者已经被终止），造成一系列的错误。

Thinking 3

结合 include/mmu.h 中的地址空间布局，思考 env_setup_vm 函数：

- 我们在初始化新进程的地址空间时为什么不把整个地址空间的 pgdir 都清零，而是复制内核的 boot_pgdir 作为一部分模板？(提示:mips 虚拟空间布局)
- UTOP 和 ULIM 的含义分别是什么，在 UTOP 到 ULIM 的区域与其他用户区相比有什么最大的区别？
- 在 step4 中我们为什么要让 pgdir[PDX(UVPT)]=env_cr3?(提示: 结合系统自映射机制)
- 谈谈自己对进程中物理地址和虚拟地址的理解.

- 本操作系统采用 2G+2G 的内存空间布局方式，在 UTOP 及其以上的地址空间主要是操作系统内核的空间（当然还包括 VPT、UVPT、ENVS 等空间），一方面是允许进程对一些共用资源如 envs、pages 等的访问，另一方面是当进程陷入内核态时，无需更换页表，仅需赋予能够进入内核内存区的权限就转换为了内核态，提升了切换的效率。
- UTOP 以下是用户区域，用户区域可以任意地进行访问和读写，ULIM 以上是内核区域，[UTOP,ULIM] 的区域属于用户和内核的公共区域，是内核特意将某些数据暴露给用户进程方便其读取知晓内容的区域，这部分的数据就是 env、page 和 page table，但权限为只读而不能写。
- 由于自映射机制，pgdir 这张页目录其实是 1024 张二级页表中的一页（专门映射页目录的 4M 空间），在这行这条语句之前，env_cr3 刚赋予了作为页目录新页的物理地址，利用 PDX(UVPT) 得到了 1024 张页表所对应的 4M 内存空间在页目录上的索引，而索引所对应的二级页表的物理地址其实就应该是 pgdir 的地址。
- 物理地址完全是由操作系统维护的，用户进程访问内存靠的就是 4G 的虚拟地址空间，由 MMU 部件通过 CP0 寄存器中 pgdir 的指引转换为物理地址进行访问，当页面有效时则正常访问，当页面无效、越界等情况时，则陷入内核态，由系统内核分配额外的新物理页、kill 进程等操作。因此，只有操作系统才能同时感受到虚拟地址和物理地址，而用户进程只能感受到虚拟地址，硬件只能感受到物理地址，正是这样的桥梁使得软硬件“低耦合”地工作，各司其职。

Thinking 4

思考 user_data 这个参数的作用。没有这个参数可不可以？为什么？（如果你能说明哪些应用场景中可能会应用这种设计就更好了。可以举一个实际的库中的例子）

根据实际编程实践，user_data 这个参数其实就是 env 结构体的未解析的数据形式，没有这个参数不可行，当调用 load_icode_mapper 时，分配了物理页装载了执行文件数据后，必须要把物理页和虚拟地址建立映射才能够供以后的程序使用，而每套进程都有一套不同的虚拟地址，因此 user_data 其实就是指明要和哪个进程建立映射。

Thinking 5

结合 load_icode_mapper 的参数以及二进制镜像的大小，考虑该函数可能会面临哪几种复制的情况？你是否都考虑到了？

这部分我认为代码上给的注释提示是不完整的，特别是没有考虑将要加载到的虚拟页位置是否已经存在物理页的情况。

Offset 部分

- offset 虚拟页已对应物理页：利用 page_lookup 函数找出对应物理页，在此物理页上使用 bcopy 进行局部拷贝。
- offset 虚拟页未对应物理页：使用 page_alloc 申请新的物理页，并在此物理也上使用 bcopy 进行局部拷贝，并 page_insert。

Bin_size 部分

- 未装载的 bin_size 部分小于一页：同 offset 部分一样，考察是否已有对应的物理页，如有，找到并加载；若无，创建、加载、插入。
- 未装载的 bin_size 部分大于一页：创建新的一页并整页拷贝，并 page_insert。

空白 Segment 部分

- 未装载的 seg 部分小于一页：同 offset 部分一样，考察是否已有对应的物理页，如有，找到并加载；若无，创建、加载、插入。
- 未装载的 seg 部分大于一页：创建新的空页，并 page_insert。

Thinking 6

思考上面这一段话，并根据自己在 lab2 中的理解，回答：

- 我们这里出现的”指令位置”的概念，你认为该概念是针对虚拟空间，还是物理内存所定义的呢？
- 你觉得 entry_point 其值对于每个进程是否一样？该如何理解这种统一或不同？

- 虚拟空间
- 我认为常规进程的 entry_point 应该是相同的，因为当链接器将 elf 文件连接成可执行文件时，会依据 linker_script 把对应段加载至对应的位置，程序的入口点在 text 段起始位置（一般链接器会在 main 函数前自动加上 _start 汇编进程初始化）。

  当然，我认为不排除的是，如果在汇编时将 text 段的位置进行自定义，并非是规定的 text 起始地址，那么此时 main 函数入口位置就发生了变化，这样 entry_point 也就发生了变化，因此最终 entry_point 的确认要以 elf_loader 读取到的内容为准。

Thinking 7

思考一下，要保存的进程上下文中的 env_tf.pc 的值应该设置为多少？为什么要这样设置？

pc 值应该设置为 epc 的值，因为如果进入调用了 env_run 函数，说明已经由正常进程陷入了内核状态。pc 值是用于保存进程被挂起时的运行的位置的，在陷入内核状态后，硬件会自动地将陷入时的地址保存在 epc 中，而 R3000 处理器采用 j+rfe 恢复正常状态，不能 eret，因而要将 epc 值赋给 pc，以便其成功跳转。

Thinking 8

思考 TIMESTACK 的含义，并找出相关语句与证明来回答以下关于 TIMESTACK 的问题：

- 请给出一个你认为合适的 TIMESTACK 的定义.
- 请为你的定义在实验中找出合适的代码段作为证据 (请对代码段进行分析)
- 思考 TIMESTACK 和第 18 行的 KERNEL_SP 的含义有何不同?

- TIMESTACK 的定义：TIMESTACK 是一个“快照栈”，是内核的调度程序和中断异常处理程序约定的存放**陷入内核前处理器现场**的空间。

- 下述代码段是程序在陷入内核并由异常向量索引至对应的处理程序后，每个异常处理程序都会执行的 SAVE_ALL 汇编指令，这其中有一个 macro 定义的 get_sp，其中具体地将 sp 在某些异常情况下赋予了 0x82000000，而根据 mmu.h 中的宏定义，这个恰好为 TIMESTACK，因此，TIMESTACK 实际就是保存现场的地方。

```
.macro SAVE_ALL
....
get_sp
sw $1 TF_REG1(sp)
sw $2 TF_REG2(sp)
sw $3 TF_REG($sp)
sw $4 TF_REG(%sp)
....

.macro get_sp
....
li sp, 0x82000000
....
.endm
```

- KERNEL_SP 是在某些状况下赋予 sp，根据 get_sp 中分支跳转，当产生中断的原因代码非零（即内部异常）时，现场会被保存在 KERNEL_SP 栈而非 TIMESTACK 栈中，这两个栈虽然都是用作保存现场，不过使用场景不同（根据目前情况猜测，可能是因为和**外部处理程序签订的地址协约**不同。

Thinking 9

阅读 kclock_asm.S 文件并说出每行汇编代码的作用

```
.set push 保存所有的汇编器设置
mfc0, t0, CP0_STATUS  读取CP0的SR寄存器的值到t0
or t0, set|clr 设置set位，并为清除做准备
xor t0, clr  置零清除位（在clr中用1表示）
mtc0, t0, CP0_STATUS 设置CP0的SR寄存器
.set pop 恢复所有的汇编器设置

LEAF(set_timer)     定义set_timer的枝叶函数
li t0, 0x01 赋值
sb t0, 0xb5000100 启动时钟，并且是第100时间单位的时钟中断
```

```
sw sp, KERNEL_SP     将sp栈保存在KERNEL_SP内存空间中
setup_c0_status STATUS_CU0|0x1001 0 设置函数
jr ra     返回调用
END(set_timer)    函数结束
```

Thinking 10

阅读相关代码，思考操作系统是怎么根据时钟周期切换进程的。

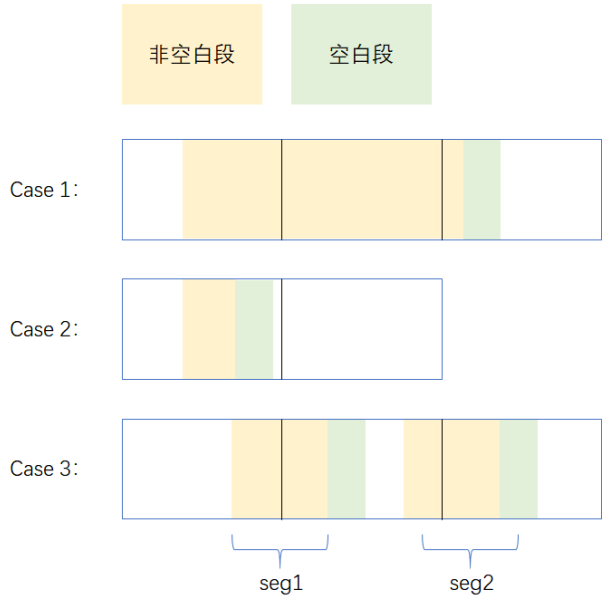
操作系统在时钟中断后，中断服务程序根据异常向量跳转至 sched_yield 程序进行进程切换的检查。

本实验中采用时间片进行调度，进程的优先级就是时间片，数字越大的优先级代表能够“禁得住更多次的中断”，因而能够运行更长的时间。当时间片用尽后，sched_yield 会调用 env_run 函数，当前进程会被停止并备份，切换到其他进行进行运行。

对于进程切换的维护，此实验中用了两条调度链表（充当就绪链表和备份链表），当进程首次被创建时会加入到 1 号链表中。在切换进程时，换下的进程会从插入到备份链表的尾部，而从就绪链表中选出一个新进程并规划新的时间片，如果就绪链表为空，那么就将就绪链表和备份链表交换。

2 实验难点图示

在 lab3 中，实现上比较困难的是 load_icode_mapper() 函数，在此我例举了一些情况（不完整，但都有一定的针对性），来体现在实现中的一些坑。



case 1

- Offset 部分需要留意并加载。
- 最后一部分非空白的 segment 加载时留意 bcopy 的范围。
- 空白的 segment 部分需要留意并加载，并注意 bzero 的范围。

case 2

- 考虑加载器是否适用只有 offset 的非空白 segment 情况。
- 考虑加载空白 segment 时是否需要 alloc 新的一页。

case 3

- 当 seg2 已被加载而准备加载 seg1 时，考虑 seg1 的加载（部分非空白 seg+ 空白 seg）需要是否 alloc 新的一页，考虑 seg1 置零空白 seg 部分时 bzero 的范围。
- 当 seg1 已被加载而准备加载 seg2 时，seg2 的加载（偏移部分）需要考虑是否 alloc 新的一页。

避免加载问题二几个核心思想

- 理智地申请新页：原来的虚页地址很可能已经有物理页对应，因此不要盲目地申请新页而覆盖了旧页。
- 不多不少刚刚好：bzero(),bcopy() 的使用范围一定要划定清楚，不能嫌麻烦用 BY2PG 增大分为，也不能算错偏移量减小范围。

3 体会与感想

在本次 lab 中，我有以下两点很深的体会：

虚拟地址的巧妙性 在进入 lab3 之前，我对 lab2 实现的虚拟内存管理感到非常困惑，特别是如下两个问题：为什么每个进程的都要有内核部分？进程之间的切换是如何完完整整地实现的？在本次实验中，对于这些问题我也一个一个地有了比较清晰的答案：

首先是操作系统所采用的“2G+2G”布局设置，“临时代理班长”身份的引入使程序陷入内核态运行时由进程切换变换为上下文保存，因而性能开销小了很多，为更高频次地进入内核态实现功能（如时间片轮转、系统调用等）提供了可能。当然，在实现这个布局的时候还是有很多细节和边界情况，例如在虚拟内存拷贝时，内存的划分并不仅仅是根据内核和用户态”非黑即白“的划分，比如介于两者之间的 UENV、UPAGE 等中间内存段的处理就需要仔细思考。

而后是进程切换的令人惊叹之处，对于一个进程而言，如果在某时刻给其拍一张完整的快照，那么这张快照其实就是由虚拟内存空间 +CPU 环境两部分组成。在这里，我惊叹于虚拟内存管理技术在进程切换时的巧妙，仅需简单地向和 MMU 约定的内存空间 mCONTEXT 替换新的 pgdir 地址，有了虚拟内存“根地图”，MMU 就什么都找得到了；而对于 CPU 环境的备份和恢复，虽然压栈入栈这种大体的思想很清楚，但是在细节上，则也会深深体会到精妙之处，例如在备份现场时，从保存现场的 TIMESTACK 中提取数据并移植到 TF 结构体中“尽可能保存了中断进程的活性”，防止在中断和备份之间出现临界的差错；在使用 lw 和 sw 恢复和备份时，k0 和 k1 内核寄存器“大家先走，我断后”的牺牲精神才使得其他寄存器能够完整地保留下来。

C 和汇编的合作 我们的系统源代码中，既有 C 语言又有汇编，作为高级语言，C 语言使我专注于功能的实现而无需过多关注细节上的实现，而汇编以其高度的灵活自由性和低可阅读性，常常让我产生抵触。不过在本章中，我渐渐体会到 C 和汇编两者相辅相成，不可或缺的关系。

C 语言专注于**在安全而稳定的情况下实现复杂的功能**，而汇编则擅长**在临界状态下实现核心的细节处理**，例如在内核的陷入、进程的恢复和备份等临界的”节骨眼“上，**每个寄存器和内存空间的值都需要精确地控制**，稍有偏差进程就变味了，那这个时候只好程序员”一句一句“地命令 CPU 该怎么行动，以安全顺利地切换到 C 语言能够工作的安全环境中去。

4 指导书残留难点与反馈

4.1 UPAGE, UENV 等空间的权限

通过资料查阅，用户空间和内核空间之间一段双方都可读的空间：对于操作系统，其不能用这段空间虚拟地址写，只能从内核段空间写；对于用户，这些数据是操作系统故意暴露的只读数据不具有写权限，但是在 lab1-“内存管理”中，pmap.cd 的 mips_vm_init() 中有如下一段：

```
boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
```

根据宏定义，PTE_R 有效时等同于访问者具有写权限，在 env_setup_init() 时每个进程都直接复制了 boot_pgdir 的 UTOP 以上的内容，也就是具有了对 UENVS 和 UPAGES 的写权限，这样权限不就错误了吗？

4.2 load_icode_mapper() 函数的定义与使用

在这部分里，基于严谨性的考虑，我认为指导书和代码中给予的提示还不够完整。我之前已在实验思考题提到过关于 mapper() 更细致的考虑，我认为目前实验中没有考虑的部分主要是**是否总是需要新建页来加载数据而忽略检查已存在的物理页**。

在这里，我例举一个这种逻辑的漏洞：如果一个进程有两个 segment：seg1, seg2，且两个段的大小均为 1kb（也就是说合计都小于一页 4kb 大小），如果两个段的虚拟地址是 0x0 和 0x800（存在于同一个虚拟页上），那么按照这样的逻辑当 seg1 加载完后，seg2 加载时会直接 page_alloc 新页，而后依据 offset 填充，并用 page_insert() 覆盖掉 seg1 填充时建立的虚实映射，这样一来，等价于 seg1 直接被“吃“了。

按照不考虑此情况的方式写，之所以在测试时没有出现问题的原因，是因为题目要求我们加载的 A 和 B 打印程序**只有一个段，而且不含有空白的 segment**。此前我在上课时与老师讨论此问题，当时得到的解答是对于每个 segment 的地址划分，编译链接器有责任避免两个不同的 segment 的部分数据能够存在于同一页上。然而通过 readelf 指令解析了更多可执行文件后，我发现程序 segment 段之间的排列是很紧凑的，存在很多“多个 segment 在一个虚拟页上”的情况（其实按照实际情况来考虑，如果每个进程都要浪费 0 4kb 的内存空间，合并起来浪费的量还是很惊人的）。

因此，对于 load_icode_mapper(),，我建议在指导书上添加上述内容的引导，在代码完成阶段增添 page_lookup() 等函数的提示，在测试阶段增加能够测试该情况的样例。

4.3 env_run() 的使用情景

作为一个 DDL 型选手，做完一周的 lab3-1 后，lab3-2 的核心部分当然就搁在下一周完成啦 。不过，在对 env_check() 进行阅读时，我却发现根本没有对 exercise-

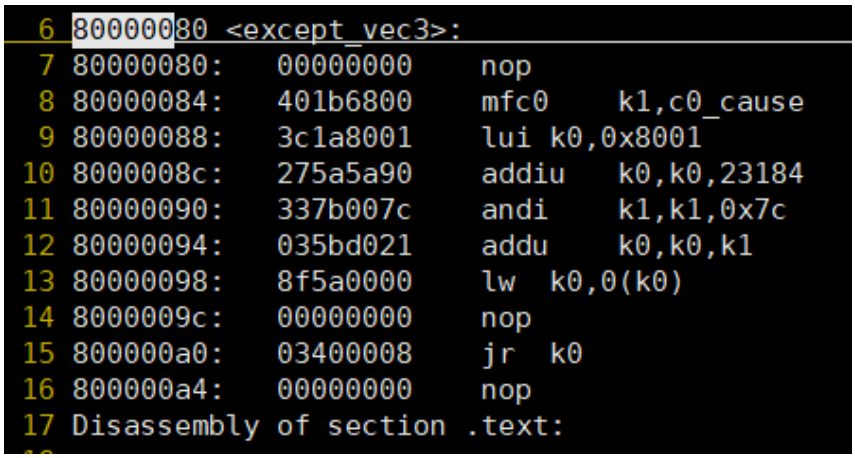


图 1: 异常入口汇编代码

3.6 及以后的内容进行评测，如何检验这一部分的正确性引起了我的兴趣，而运行进程的函数 env_run() 显然就是一个绝佳的途径，在原来的程序中，init.c 会调用 create_env_priority() 进而进行一个进程的初始化和内容的装载,我尝试着在 create_env_priority 函数中在创建完成后直接调用 env_run() 函数以启动这一个新的进程，却发现程序会由于 pc=0 而不断地进入 tlb_exception。

对于这个问题出现的背景，在此还要做更多的补充：首先，我完成了 lab3-2 的异常向量建立和链接器设置，仅剩下没有写 env_sched.c 文件；其次，问题应该就是出在直接调用 env_run() 这种方式，因为如果在 env_sched.c 中随机取 envs 数组中 RUNNABLE 的进程运行，程序就完全正确；最后，问题再细化的话，其实是 env_run() 函数中的 env_tf_pop() 汇编函数里 jr k1 这条指令。

在掌握 **objdump 核心科技**，通过 gxemul 的调试，我最终发现直接调用 env_run() 失败的问题，在于 trap_init() 的问题：jr k1 一定会触发 tlb 的缺页异常，而触发异常后会进入 0x80000080 开头的异常独立头，在这里将调用 cp0 中的 cause 寄存器，根据异常号对应到异常向量中，从而跳转至对应的异常处理程序，但是由于 create_env_prioirty() 在 init.c 中其实是先于 trap_init() 执行，因此异常向量根本就没有准备好，取出来的异常处理程序地址是错误的 0x0。因此，为了避免更多人踩坑，我建议指导书：

- 在 lab0 阶段就增加有关 MIPS-4KC-??? 一系列工具的使用介绍（因为 objdump、readelf 这些指令本就是有的，但无法读取交叉编译形成的文件，这点要提醒同学们注意）。
- 增加指导书的内容，指导同学们如何在 lab3-1 自主地评测 exercise-3.6 及以后的练习正确性。