第三章

Lab1报告

在仔细思考了应该在哪一部分加入这一功能后，我们选择了在Task_Manager部分加入该功能。因为Task_Manager中的Task_Control_Block(TCB)结构本来就维护了每个task的信息，在其中额外加入并维护get_task_info所需要返回的信息的想法是十分自然的。

进一步地，我们将所需要实现的功能分解成了两个部分：第一，是要让Task_Manager具有获取task_info的能力；第二，是要让Task_Manager具有更新task_info的能力。

首先，修改TCB，在其中加上我们所需要记录的task_info。

> 在每个任务开始时，先记录`start_time`；每次调用`syscall`时，在进入具体的中断处理程序之前，先更新`task_info`信息，使对应的`syscall_times[syscall_id]+=1`；`task_status`这里没有特意维护，因为调用`get_task_info`时状态始终为`RUNNING`。

调用get_task_info时，读取当前执行任务的TCB中所保存的task_info，并用当前时间减去start_time作为所经过的时间。

第三章问答题

1.

1. 正确进入 U 态后，程序的特征还应有：使用 S 态特权指令，访问 S 态寄存器后会报错。 请同学们可以自行测试这些内容 (运行 Rust 三个 bad 测例 (ch2b_bad_*.rs) ， 注意在编译时至少需要指定 `LOG=ERROR` 才能观察到内核的报错信息) ， 描述程序出错行为，同时注意注明你使用的 sbi 及其版本。

RUSTSBI version 0.2.2, adapting to RISV-V SBI v1.0.0

ch2b_bad_address.rs: [ERROR] [kernel] PageFault in application, core dumped. 即访问错误地址

ch2b_bad_instructions.rs: [Error] [kernel] IllegalInstruction in application, core dumped. 即执行非法指令

ch2b_bad_register.rs: [Error] [kernel] IllegalInstruction in application, core dumped. 即执行非法指令

2.

2. 深入理解 trap.S 中两个函数 `__alltraps` 和 `__restore` 的作用，并回答如下问题:
   1. L40：刚进入 `__restore` 时， `a0` 代表了什么值。请指出 `__restore` 的两种使用情景。
   2. L43-L48：这几行汇编代码特殊处理了哪些寄存器？这些寄存器的的值对于进入用户态有何意义？请分别解释。

```
ld t0, 32*8(sp)
ld t1, 33*8(sp)
ld t2, 2*8(sp)
csrw sstatus, t0
csrw sepc, t1
csrw sscratch, t2
```

3. L50-L56：为何跳过了 `x2` 和 `x4`？

```
ld x1, 1*8(sp)
ld x3, 3*8(sp)
.set n, 5
.rept 27
   LOAD_GP %n
   .set n, n+1
.endr
```

4. L60：该指令之后，`sp` 和 `sscratch` 中的值分别有什么意义？

```
csrrw sp, sscratch, sp
```

5. `__restore`：中发生状态切换在哪一条指令？为何该指令执行之后会进入用户态？
6. L13：该指令之后，`sp` 和 `sscratch` 中的值分别有什么意义？

```
csrrw sp, sscratch, sp
```

7. 从 U 态进入 S 态是哪一条指令发生的？

(1)

__ restore的两种使用场景是：

1. **从内核态返回到用户态**：在处理完中断或异常后，系统需要从内核态切换回用户态，继续执行用户程序。
2. **上下文切换**：在多任务操作系统中，`__restore` 可能用于在进程或线程之间进行上下文切换。

a0中的值是要切换到的用户线程的TrapContext的地址，或指向内核栈。

(2)

1. **sstatus**：保存了程序的状态信息。在这里，`sstatus`被从内核栈中加载到t0，然后写入sstatus寄存器。这样做的目的是恢复陷阱发生前的程序状态。在RISC-V中，sstatus寄存器包含了一些重要的状态位，如全局中断使能位、当前的特权级别等。恢复sstatus寄存器可以确保程序在用户态下正确执行。
2. **sepc (Exception Program Counter)**：

   `sepc` 寄存器存储了发生异常或中断时的程序计数器（PC）值。在处理完异常或中断并准备返回用户态时，`sepc` 寄存器的值用于确定从哪里恢复程序的执行。它确保程序能够从中断点（或适当的异常处理点）继续执行。

3. **sscratch (Scratch Register)**：

   `sscratch` 是一个用于临时存储的寄存器，通常用于上下文切换过程中。在中断或异常处理过程中，`sscratch` 可能被用来临时保存一个重要的值（如用户栈的指针），这在从内核态切换回用户态时是必要的。

(3)

跳过x2和x4的加载是因为这两个寄存器在这个上下文中有特殊的用途，不能简单地通过加载内核栈中的值来恢复。

(4)

首先明确csrrw指令的含义： csrrw r1, r2, r3即为把 r2 写进 r1 ，把 r3 写进 r2 。在这里，CSRRW指令会原子性的交换CSR和寄存器中的值。

同时，调用该指令前，sscratch的值为内核态栈指针，sp为用户态栈指针。

因此，执行该指令后，sscratch中为 sp的旧值，即内核模式被激活之前的用户栈指针。sp的值为sscratch 的旧值，即用户态的栈指针。这时，sp 被更新为指向内核栈。

(5)

_restore函数中的状态切换发生在最后一条指令sret，它是RISC-V架构中的一条特殊指令，用于从特权级别更高的模式（如内核态）返回到特权级别更低的模式（如用户态）。因为它改变了PC寄存器的值，即改变了程序执行的位置。

(6)

同(5)

(7)

上题中所述的L13指令标志着从U态进入S态。

因为它将当前栈指针(sp)由指向用户态栈改为了指向内核态栈，这是为了保证操作系统的安全性与隔离性，在内核空间中更为安全地执行处理中断等操作。

第四章

Lab2报告

虚拟内存导致sys_get_time和sys_task_info函数失效，在中断处理函数中添加查表将虚拟内存转为物理内存，迁移其余内容直接搬运即可

实现mmap和munmap：检查边界，然后找到vpn并使用维护的memory_set进行插入和删除即可。

第四章问答题

1.

SV39的页表项一共有64位，其中 [63:54] 这10位为保留位，增强其扩展性； [53:10] 这 44 位是物理页号，用于指示物理内存中页的位置；最低的 8 位 [7:0] 则是标志位，用于标识页面的权限（只读等）或状态（脏页等）等信息。

2.

2. 缺页

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?
- 发生缺页时，描述相关重要寄存器的值，上次实验描述过的可以简略。

缺页有两个常见的原因，其一是 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?

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。

- 处理 10G 连续的内存页面，对应的 SV39 页表大致占用多少内存 (估算数量级即可)?
- 请简单思考如何才能实现 Lazy 策略，缺页时又如何处理？描述合理即可，不需要考虑实现。

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?

(1)

1. lazy策略，即直到内存页面被访问才实际进行页表操作。在首次被访问时会触发缺页

2. 对页的访问权限不够，例如实现COW时，当某个进程试图写共享页面时，会因为权限为“只读”而触发权限不够导致的缺页错误。

3. 物理页面被暂时换出到了磁盘中的交换区，这种情况下需要把被换出的页面再读回。

4.没有完成虚拟地址与物理地址之间的映射建立，导致虽然页面在物理内存中依然触发缺页错误。

(2) 缺页重要寄存器的值

程序计数器（Program Counter, PC）：存储了发生缺页时的指令地址。这使得操作系统可以确定哪个指令导致了缺页。

缺页地址寄存器（Page Fault Address Register）：在许多架构中，有一个专门的寄存器（如 CR2 在 x86 架构中）用于存储引起缺页的虚拟地址。

状态寄存器（Status Register）：包含了缺页时的处理器状态信息，如当前的特权级别等。

错误码寄存器（Error Code Register）：在某些架构中，缺页中断可能伴随一个错误码，提供了导致缺页的具体原因，如是否由于写操作、用户/内核模式的访问、不存在的页等。

其他寄存器：发生缺页时，其他各种寄存器（如通用寄存器）可能包含了重要的上下文信息，操作系统需要保存和恢复这些寄存器以确保进程可以在缺页处理完成后继续正确执行。

(3) Lazy策略的好处?

1. 减少程序启动延迟：在程序启动时不用一次性把所有代码都加载到物理内存中，而是等到实际访问到再进行读取，这样可以让程序启动时更快。
2. 优化IO操作：减少了不必要的IO操作，提高IO资源利用率。
3. 提高内存利用率：执行时才加载代码，意味着不必要的代码不会被加载进内存空间，这提高了内存的利用效率。

(4) 处理 10G 连续的内存页面，对应的 SV39 页表大致占用多少内存？

1. **计算总共需要多少个 4KB 页面**： 10GB / 4KB = 10 * 1024 * 1024KB / 4KB = 2,621,440 个页面。
2. **L3 页表项**:

   每个 L3 页表可以映射 512 个 4KB 页面（因为每个页表有 512 个页表项，每个页表项映射一个 4KB 页面）。

   所以，总共需要的 L3 页表数量 = 2,621,440 / 512 ≈ 5121.25。

   每个 L3 页表占用 512 * 8 字节 = 4KB。

   因此，L3 页表总共占用 ≈ 5122 * 4KB ≈ 20.5MB。
3. **L2 页表项**:

   每个 L2 页表可以映射 512 * 512 个 4KB 页面。

   总共需要的 L2 页表数量 = 2,621,440 / (512 * 512) ≈ 10。

   每个 L2 页表占用 4KB。

   因此，L2 页表总共占用 ≈ 10 * 4KB ≈ 40KB。
4. **L1 页表项**:

   每个 L1 页表可以映射 512 * 512 * 512 个 4KB 页面，远大于 10GB 所需。

   因此，只需要一个 L1 页表。

   L1 页表占用 4KB。

综上所述，SV39 页表总共大约占用20MB空间

(5) 请简单思考如何才能实现 Lazy 策略，缺页时又如何处理？描述合理即可，不需要考虑实现。

不考虑局部性的情况下, 使用lazy策略时, 根据发生缺页异常的类型, 去memory_set的各个段寻找 [缺页的虚拟地址在当前逻辑段 && 逻辑段权限达到缺页的要求], 如果寻找成功, 则将页面加载到物理内存并映射。

(6) swap策略导致的页面失效如何体现在页表项上?

1. 有效位置零：被换出内存的页面有效位会置零。
2. 权限改变：被换出内存的页面的权限位（可读等）会被修改，以防止进程错误地访问不在内存的页面。
3. PPN可能的变化：由于被置换出内存，PPN也就失去了意义，因此可能会被修改。

3. 双页表和单页表

(1)在单页表情况下，如何更换页表?

①进入内核态：当程序执行系统调用或触发中断时，CPU 会切换到内核态。

②使用相同页表：用户空间和内核空间共用一张页表，页表中既包含用户空间的映射，也包含内核空间的映射，但内核空间的地址仅在内核态可访问。

(2)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)

内核对应的地址只允许在内核态访问，将PTE条目中的权限位设为用户态不可读，每次试图访问时都需要检查权限。

(3)单页表有何优势？（回答合理即可)

由于用户与内核公用一张页表，故用户态和内核态切换时不用更换页表，在切换状态时TLB等不会立刻失效，加快中断处理速度。同时也可以节省页表所占空间。

(4)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，你会选择何时更换页表（回答合理即可）？

何时更换：从内核态回到用户态，或是从用户态切换到内核态时

更换页表时机选择：①进程切换时 ②返回用户态时