OS-lab4 report

刘泓尊 2018011446 计84

Chapter4 实现的内容

1. 增加了动态内存分配器
2. 支持虚拟内存抽象，实现地址空间的页式管理
3. 实现基于地址空间的分时多任务OS
4. 将lab2, 3的内容兼容到lab4，删除了lab3关于运行时间上限的规定
5. 实现了mmap, munmap系统调用

编程作业：

1. mmap的实现

   在`TaskManager`里增加了`map_virtual_pages`接口，首先检测addr按页对齐(4096Byte), 并且len 在 0 ~ 1GB之间。且`(port & !0x7) == 0 && port & 0x7 != 0`. 之后将port翻译为`MapPermission`类型，创建[addr, addr+len)按页对齐的`MapArea`. 之后检查map_area的vpn_range中不存在已经被映射的页。如果一切正常，就将map_area压入本进程的memory_set中，返回实际map的空间大小。

2. munmap的实现

   在`TaskManager`里增加了`unmap_virtual_pages`接口，首先检测addr按页对齐(4096Byte), 并且len 在 0 ~ 1GB之间。之后检测[addr, addr+len) 对应的vpn_range中存在未被映射的页的情况。如果一切正常，就在当前进程的memory_set中unmap对应的vpn_range, 返回实际unmap的空间大小。需要注意用户unmap的地址区间也要进行权限检查。

测试结果截图

```
[rustsbi] Platform: QEMU (Version 0.2.0)
[rustsbi] misa: RV64ACDFIMSU
[rustsbi] mideleg: 0x222
[rustsbi] medeleg: 0xb1ab
[rustsbi-dtb] Hart count: cluster0 with 1 cores
[rustsbi] Kernel entry: 0x80200000
Hello world from user mode program!
Test hello_world OK!
3^10000=5079
3^20000string from data section
strinstring from stack section
strin
Test write1 OK!
Test set_priority OK!
get_time OK! 79
current time_msec = 80
Test 04_1 OK!
Test 04_4 test OK!
Test 04_5 ummap OK!
Test 04_6 ummap2 OK!
=8202
3^30000=8824
3^40000=5750
3^50000=3824
3^60000=8516
3^70000=2510
3^80000=9379
3^90000=2621
3^100000=2749
Test power OK!
time_msec = 181 after sleeping 100 ticks, delta = 101ms!
Test sleep1 passed!
Test sleep OK!
```

问答作业

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页表项组成如下，包括保留位，三级物理页号，以及若干标志位。

63 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

   riscv文档中给出的标志位具有如下含义：

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

   RWX 3个标志位给出了读写执行的权限，同时还指示了是否是叶子节点。可写的页面也必须是可读的。

   U标志位指示是否可以被U态访问。

   V标志位指示该页表项是否有效。

G标志位指示是全局映射(global mapping), global mapping是指存在于所有地址空间的映射。非叶子PTE标记为G说明其后代节点都是全局的。

RSW位共2bit, 是保留给OS用的，由OS负责解释。

D，A标志位指示写、读有效。在叶子节点上一般都是1， 简单的硬件实现会把在叶子节点遇到D(写时)，A(访问时)为0的情况抛出page fault. 复杂的硬件设计会在这种情况下将其置1，同时检查权限位和V位。这个操作必须是原子的，并且同步到存储层次结构中。

D, A, U标志位在非叶子节点都是0.

2. **缺页**

    1. **请问哪些异常可能是缺页导致的?**

        page fault共有3个，按优先级由高到低排列：

        - Instruction page fault (12)
        - Store/AMO page fault(15)
        - Load page fault (13)

        在没有X权限的页 取指 会抛出Instruction page fault；在没有R权限的页 执行Load操作会抛出Load page fault；在没有W权限的页 进行 store 或 AMO 操作 会抛出 Store/AMO page fault。V位为0或U态访问了非U态页面，也会根据当前指令抛出对应的page fault.

    2. **发生缺页时，描述相关的重要寄存器的值。**

        `sepc` : 保存发生Trap时的pc(当然S态处理Trap时也可以修改它来实现APP切换). 用于Trap处理结束之后返回到sepc.

        `stvec` : Trap入口向量。发生异常的时候CPU将pc设置成stvec 的BASE << 2, 我们的OS中BASE就是 __alltraps

        `scause` : 记录Trap的原因(Trap代码)。最高位是0时表示异常，是1时表示中断。

        `stval` : 给出 Trap 附加信息。当取指、load、store发生misaligned 或 access fault 或 page fault的时候，stval会被写入发生错误的**虚拟地址。**

        `status` : S特权级最重要的 CSR, 可以从很多方面控制 S 特权级的CPU行为和执行状态. 其 SPP 等字段给出 Trap 发生之前 CPU 处在哪个特权级（S/U）等信息.

        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。

    3. **这样做有哪些好处?**

        Lazy策略体现了按需分配，在发生缺页时分配物理内存，可以避免加载一些没有被访问的地址，带来潜在的性能提升；同时也会节省空间，比如用户申请了1G，但实际上只用了1M，所以没必要在一开始就分配给程序1G。

        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)?**

            页面大小 4096Byte, 10G 就是 2621440 个物理页面。每个页表项64bit = 8Byte, 所以大约需要 20MB 的页表项。在理想情况下，大约占用 5120 个物理页面存放这些页表项，这需要上一级页表项 共40 KB。再到第一级页表，大约又需要几Byte. 如果只考虑数量级，需要操作的页表大约需要 20MB 的物理内

存。

- **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。**

  在程序申请内存的时候，可以在内核中维护一个数据结构保存请求的逻辑地址空间，但是不实际分配物理内存。在处理缺页异常的时候，如果这个空间之前请求过，那么就分配实际内存，并建立页表映射；在加载程序的时候，也是先保存.text段的信息，在发生缺页时再给程序段分配物理内存。

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。

4. **此时页面失效如何表现在页表项(PTE)上?**

   swap出去的页面在 对应进程 的 叶子节点页表项 上 把**V置0**。再次访问时就会触发缺页异常，以执行换入操作。

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。

1. **如何更换页表?**

   修改 `satp` 的PPN段和ASID段即可。PPN指向了第一级页表基址，ASID作为地址空间标识符标识了不同进程的地址空间。注意在这之后调用 `sfence.vma` 刷新TLB。

2. **单页表情况下，如何控制用户态无法访问内核页面?**

   将内核页面对应的页表项的 U位 置0 即可。

3. **单页表有何优势?**

   避免了频繁切换页表，使得TLB总是失效，也就是单页表的时间开销应该更短。此外也节省空间。

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?**

   双页表情况下，在U态trap到S态处理异常和中断的时候，以及处理完成后S态返回U态之前，都需要切换页表，可以在 `trap.S` 中找到对应实现。切换进程的时候也需要更换页表，但是已经由 `trap.S` 中的 `__alltraps` 和 `__restore` 负责了，先切换到内核页表，再切换回新进程页表。

   单页表情况下，切换进程的时候需要更换页表。也就是从S态返回到U态时，加载新进程上下文前切换页表。