Project 4 Virtual Memory 设计文档

中国科学院大学
王嵩岳
2020 年 12 月 7 日

1. 内存管理设计

1.1 虚拟内存地址设计

本次实验我采用 Sv39 页表规范，使用 39 位虚地址。内核全地址映射采用二级页表设计（一页 2MB），用户页表采用三级页表设计（一页 4KB）。

1.1.1 内核虚地址空间

我们使用的系统中，用户可用的物理内存地址空间是 0x5000_0000~0x6000_0000，内核需要做内存管理，要把全地址空间映射过去。我采用线性映射的方式。具体 layout 如下图（本仓库 README.md 的截图）。

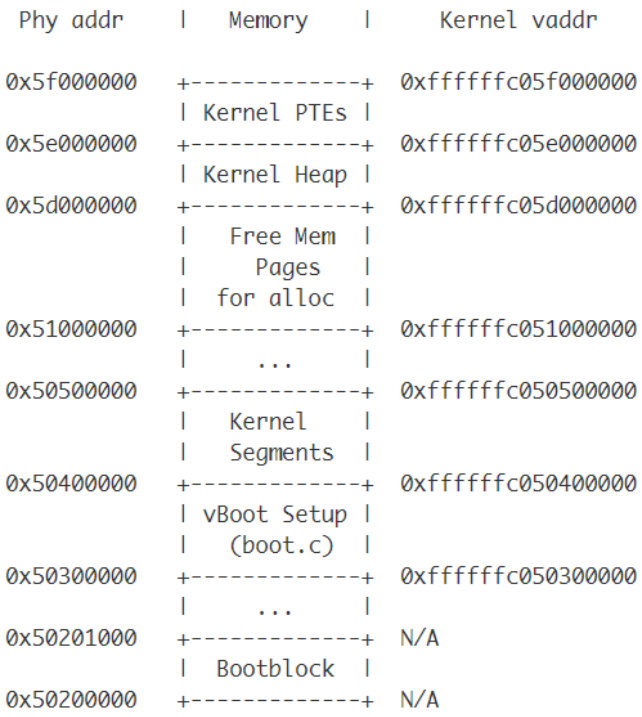

Figure 1 内核地址空间 layout

1.1.2 用户虚地址空间

每个用户进程都有自己的页目录。对处理器在 U 态的每个用户进程，它能看到的地址空间如下（本仓库 README.md 的截图）。

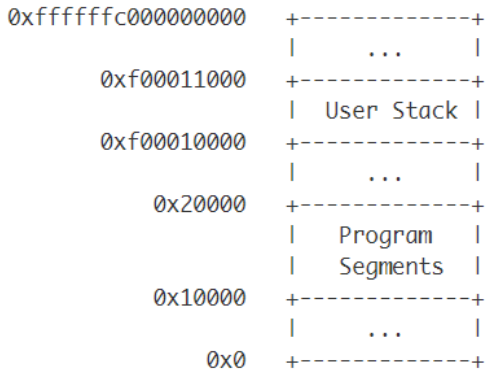

Figure 2 用户进程用户态地址空间

1.2 内核进程的启动

内核的启动主要分为以下几步：

主处理器核：

（1） bootloader 引导拷贝镜像至内存相应位置，并跳转至内核启动处理程序（boot.c）的预处理程序 head.S。

（2） 在 head.S 中，为 boot.c 的启动设置程序栈，并做相应的处理器状态设置。

（3） 在 boot.c 中，设置内核页表映射，通过 load_elf 方法解析内核进程的 ELF 文件，并摆放至内存相应位置（解析各段得到虚地址），设置映射完毕后，设置 CSR[SATP]并开启虚存。

（4） 跳转至内核进程预备程序 start.S，为内核进程设置 gp 指针，程序栈。

（5） 进入内核进程后，发出 IPI。

从处理器核：

（1）在 bootloader 中开启 S 态软件中断，等待 IPI 发出。

（2）IPI 发出后，跳转至 head.S 做 boot.c 的初始化。

（3）在 boot.c 中，仅做设置 CSR[SATP]并开启虚存。

（4）跳转至内核进程预备程序 start.S，为内核进程设置 gp 指针，程序栈。

内核页目录的设置：

由于编译内核为 ELF 文件时，已经将入口地址设置为 0xffffffc050400000，且我们在 load_elf 前已经设置好了相应的内存映射，故在 load_elf 时直接将各段写入相应虚地址即可。

内核的页表项都不设置 U 位，这样可以保证处理器处于 U 态时访问不到内核的程序（指令缺页）。

内核页目录我设置在物理地址 0x5e000000 处，所在的这一页为一级页目录。接下来的 512 页为二级页表项（采用两级页表）。

1.3 用户进程的启动

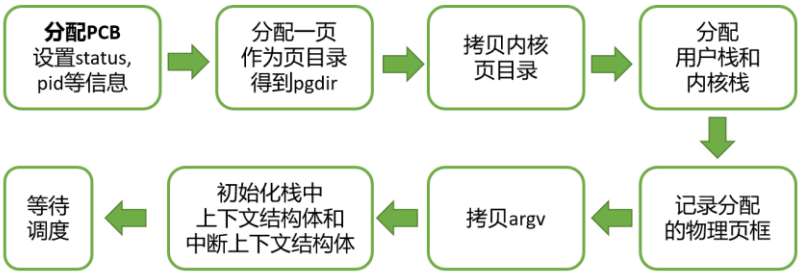

Figure 3 用户进程的启动流程

关键在于加载用户程序 ELF 文件（已经由 elf2char 转换成了字符串，存放在 user_programs.c 中）至物理内存，以及页表项的设置。

为保证进程间地址空间隔离，每个进程都有自己的页目录。我们首先通过 alloc_page_helper 为其分配一页作为页目录。然后我们把内核的页目录从 0xffffffc05e000000 处拷贝过来（使用 kmemcpy），这样自然用户进程在内核态时也能访问内核地址空间，而在用户态则不行（因为 U 位未设置）。

加载时，每个程序段仍会解析出一个虚地址，这时我们使用 alloc_page_helper（提供一个虚地址，在该进程的页目录中分配一个物理页框并返回此页框的内核虚地址）为其分配一页，并加载至对应位置。

最后要为用户进程分配一个用户栈和内核栈。用户栈的虚地址固定为 0xf00001000，而内核栈虚地址与先前一样分配即可，因为内核态可以访问全部地址空间。

Argv 的拷贝是跨地址空间的拷贝（从一个进程的用户栈拷贝至另一个进程的用户栈）。这里我的做法是：将用户栈的高 0xc0 字节作为 argv 数组存放的区域。然后根据 ABI 和函数调用规范，我们在中断上下文中交给新进程的 a0 寄存器的值赋值为 argc，a1 寄存器给 argv 也就是参数数组的地址（用户栈虚地址 0xf00001000-0xc0）。

而对 argv 数组中的内容，由于我们打开了 CSR[SSTATUS]的 SUM 位，因此内核态仍然可以访问用户的地址空间（用户栈）。因此我们可以从用户栈中使用 kmemcpy 方法将内容拷贝过来。

```
// Copy new argv
uintptr_t new_argv = USER_STACK_ADDR - 0xc0;
uint64_t *kargv = user_sp;
for (int j = 0; j < argc; j++){
    *(kargv + j) = (uint64_t)new_argv + 0x10 * (j + 1);
    kmemcpy((uint64_t)user_sp + 0x10 * (j + 1), argv[j], 0x10 * (j + 1));
}
```

Figure 4 拷贝函数参数

2. 缺页处理设计

缺页处理导致的异常主要有三种：指令缺页、Load 缺页和 Store 缺页。对指令缺页，这是无法解决的。而对 Load 缺页和 Store 缺页，我的一般处理流程如下。

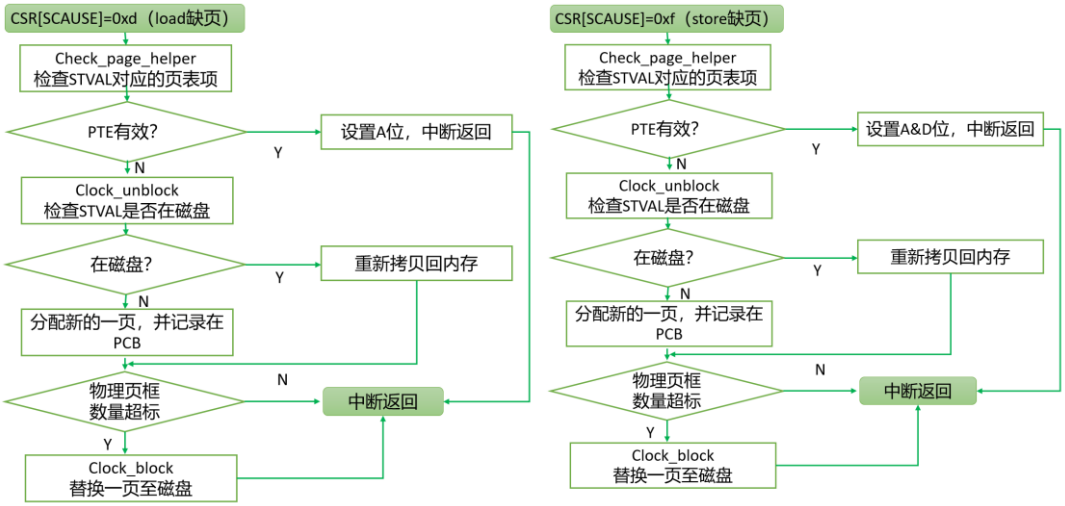

Figure 5 缺页处理流程

由于我想通过 A 位和 D 位判定此页近期是否被访问或修改过，以此使用 clock 算法进行页替换，因此在 alloc_page_helper 中我没有设置 A 位和 D 位。那么对每一页来说，第一次访问必定造成缺页，此时因为页表项有效，直接置位 A 或 D 位即可。

由于需要实现换页机制，因此我为进程分配的每一页非指令页和非页表页都设置了一个 PAGE_CONTROL_BLOCK，它被存放在 PCB 的 plist 域，通过双向链表组织，它的数据结构如下：

```
typedef struct {
    uintptr_t pa;
    uintptr_t va;
    int atsd;
    int block;
    uintptr_t pte;
    list_node_t list;
}page_t;
```

Figure 6 物理页框控制块（PageCB）结构体

其中，va 是用户看到的虚地址，pa 是物理实地址，atsd 位标志此页是否在 SD 卡中，block 指示此页在 SD 卡的哪个扇区，PTE 是此物理页框的 PTE 的物理地址，list 域是链表指针域。每当进程 alloc 一个新页框时，就从堆中申请一块空间作为此页的 PAGE_CONTROL_BLOCK，更新信息后加入 pcb 的 plist 链表中。当换页、重新拷贝时更新之，待进程退出后，释放 plist 中指示的所有页。

如果页表项无效，说明内存中此页不存在。我首先检查此页是否在磁盘中，若在磁盘中则拷贝回内存的 pa 位置。若不在，则另分配一页。然后检查此进程持有的物理页框数是否超标（我们限制进程除页表项和各段之外，最多持有 4 个物理页框），若超标则使用 clock 算法将其中一页换到 SD 卡上。

3. C-Core 设计（做 C-Core 的同学需要写）

由于 RISCV64 的 Sv39 页表规范中，PTE 有 A 位和 D 位，因此我采用 Clock 算法进行页替换。如第二节所述，页替换不是由单独的进程完成，而是访存出现缺页时，由于持有物理页框数过多触发，由内核进行替换。具体流程如下图。

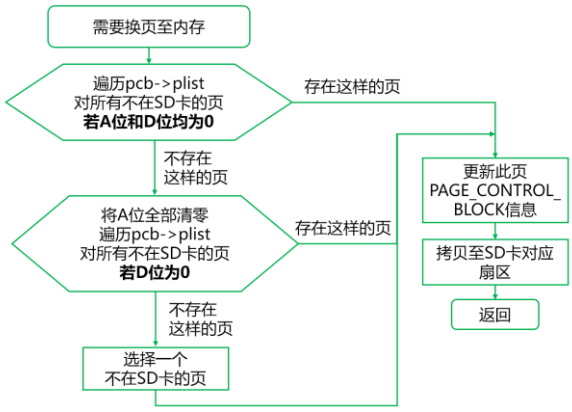

Figure 7 页替换流程（Clock 算法）

我使用自编写的 swap.c 作为测试程序。文件路径在./test/test.c，ELF 文件名为 swap。测试的前提是，我限制每个进程的栈以及后续分配的页框数最多为 4（除去两个栈，实际上后续按需调页最多保留 2 页）。而对栈我们不执行替换，因为栈频繁被访问，来回换的“乒乓”效应开销过大，所以我们直接标记栈页的状态为在 SD 卡中，这样遍历时就不会替换它了。

测试程序中，一共会访问三个不同的页地址，它们都是按需调页所得。

PFN	Name	vaddr
1	PFN1	0x1000
2	PFN2	0x1000000
3	PFN3	0x2000000

Figure 8 测试程序访问的地址

首先，测试程序会尝试访问 PFN1 的起始地址，这会造成指令缺页并分配一个物理页框，映射到这个虚地址，并且新设置的 PFN1 页表项 A 位置高（实际其实走了两次中断，第一次设置页表项，第二次设置 A 位）。

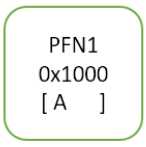

Figure 9 Swap 进程第一轮访存后拥有的物理页框（不计栈）

然后，测试程序会读写 PFN2 的起始地址，并向 0x1000000 处写入 MAGIC 数（一个宏，方便后续校验内存内容）。那么 PFN2 对应的页表项的 A 位和 D 位都为高。

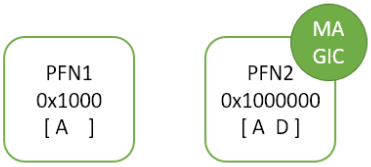

Figure 10 Swap 进程第二轮访存后拥有的物理页框（不计栈）

然后，测试程序会尝试写 PFN3，向 0x2000000 写入 MAGIC 数（一个宏，方便后续校验内存内容），此时会分配一个新的物理页框，并映射到 PFN3 的虚地址。但是此时进程持有 5 个物理页框，此时需要替换一个页框到 SD 卡中。

根据我实现的 Clock 算法，由于 PFN1 只访问过但没写过（A=1, D=0），因此它将被替换到 SD 卡中。

Figure 11 Swap 进程第三轮访存后拥有的物理页框（不计栈）

接下来，我们将尝试向 PFN1 段的物理地址 0x1008 写入数据。此时由于 PFN1 的内容在 SD 卡中，所以会将其拷贝回来，再选择一个页框替换出去。但是注意此时 PFN2 和 PFN3 都是 A=1, D=1，因此替换哪一个取决于在链表中的顺序，但肯定会换出去一个。

Figure 12 Swap 进程第四轮访存后拥有的物理页框（不计栈）

最终，我们要验证之前向 PFN2 和 PFN3 中写的 MAGIC 数还存在。因为此时 PFN2 和 PFN3 中必有一个在 SD 卡中，所以我们都进行验证，此时会把它从 SD 卡再次调回物理内存。期间可能有多次拷贝。若 MAGIC 数仍在，证明存储和拷贝过程是正确的。

为了验证拷贝的过程，我在每次发起内存和 SD 卡之间的数据交换时，都打印拷贝的信息，以此验证执行的行为。最终程序呈现的结果也是正确的，与理论分析一致。

```
void disk2mem(int block, int num, uintptr_t pa, uintptr_t va)
{
    sbi_sd_read(pa, num, block);
    prints("[swap] from SD block %d to MEM 0x%lx (v0x%lx) (#blocks %d)\n",
           block, pa, va, num);
    screen_reflush();
}

void mem2disk(int block, int num, uintptr_t pa, uintptr_t va)
{
    sbi_sd_write(pa, num, block);
    prints("[swap] from MEM 0x%lx (v0x%lx) to SD block %d (#blocks %d)\n",
           pa, va, block, num);
    screen_reflush();
}
```

Figure 13 拷贝扇区的提示信息

```
blocks read error!!ting. Please note swap infomation.
 > 1: Read from (pf1) 0x1000
 > 2: Read from (pf2) 0x1000000
 > 3: Write to  (pf2) 0x1000000
 > 4: Write to  (pf3) 0x2000000 (swap)
 [swap] from MEM 0x5101b000 (v0x1000) to SD block 10000 (#blocks 8)
 > 5: Write to  (pf1) 0x1008 (swap)
 [swap] from SD block 10000 to MEM 0x5101b000 (v0x1000) (#blocks 8)
 [swap] fromeMEMr0x5101f000 (v0x2000000) to SD block 10008 (#blocks 8)
 > 6: Check we write on step 4 (maybe swap)
 [swap] from SD block 10008 to MEM 0x5101f000 (v0x2000000) (#blocks 8)
 [swap] fromeMEMr0x5101d000 (v0x1000000) to SD block 10016 (#blocks 8)
 [swap] from SD block 10016 to MEM 0x5101d000 (v0x1000000) (#blocks 8)
 [swap] fromeMEMr0x5101f000 (v0x2000000) to SD block 10024 (#blocks 8)
 > Data raw success!
 End testing.

.....................................................................
 [root@UCAS_OS / #] exec swap
 > Successfully exec in pid 2.
```

Figure 14 QEMU 上 swap 验证截图（忽略 uboot 信息）

同时，页替换也可以通过测试程序 rw 进行验证，当读写三个不同页框的地址时，势必会替换一页到 SD 卡中。

```
argc = 4, argv at 0xf0000ff40
argv[1] = 0x2, at 0xf0000ff60
argv[2] = 0x5000, at 0xf0000ff70
argv[3] = 0x10000000, at 0xf0000ff80
--------------------
test 1: 0x2, write 1224589679
test 2: 0x5000, write 606672462
[swap] from MEM 0x5101c000 (v0x5000) to SD block 10000 (#blocks 8)
test 3: 0x10000000, write 57709377
Success!
```

```
[root@UCAS_OS / #] exec rw 0x2 0x5000 0x10000000
> Successfully exec in pid 2.
[root@UCAS_OS / #] blocks write error!
```

Figure 15 QEMU 上 rw 的测试页替换

参考文献

[1] **The RISC-V Instruction Set Manual** Volume II: Privileged Architecture Version 1.10