PRJ 4

> 本文档仅记录写代码过程，中间的代码未必与最终版代码一致

TASK 1

1. 背景知识

起初看着各个变量的命名&各种宏，真的是一头雾水……先从网上查一下Linux的内存页表映射机制。

在32位下的情况，只有三级页表：PGD，PMD，PTE；

在64位情况下，会有四级页表：PGD，PUD，PMD，PTE。此处只关注32位的情况。

（1）Linux虚拟内存三级页表

Linux虚拟内存三级管理由以下三级组成：

- PGD: Page Global Directory (页目录)
- PMD: Page Middle Directory (页目录)
- PTE: Page Table Entry (页表项)

每一级有以下三个关键描述宏：

- SHIFT
- SIZE
- MASK

如页的对应描述为：

```
/* PAGE_SHIFT determines the page size  asm/page.h */
#define PAGE_SHIFT      12
#define PAGE_SIZE       (_AC(1,UL) << PAGE_SHIFT)
#define PAGE_MASK       (~(PAGE_SIZE-1))
```

数据结构定义如下：

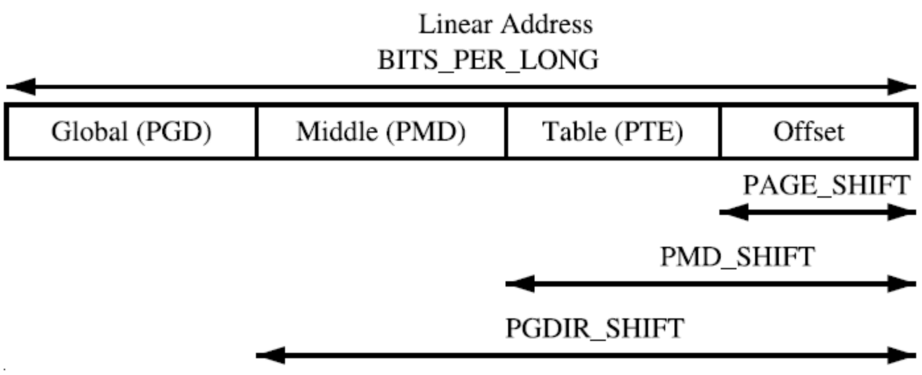
```
/* asm/page.h */
typedef unsigned long pteval_t;

```

```c
typedef pteval_t pte_t;
typedef unsigned long pmd_t;
typedef unsigned long pgd_t[2];
typedef unsigned long pgprot_t;

#define pte_val(x)      (x)
#define pmd_val(x)      (x)
#define pgd_val(x)  ((x)[0])
#define pgprot_val(x)   (x)

#define __pte(x)        (x)
#define __pmd(x)        (x)
#define __pgprot(x)     (x)
```

(2) Page Directory (PGD and PMD)

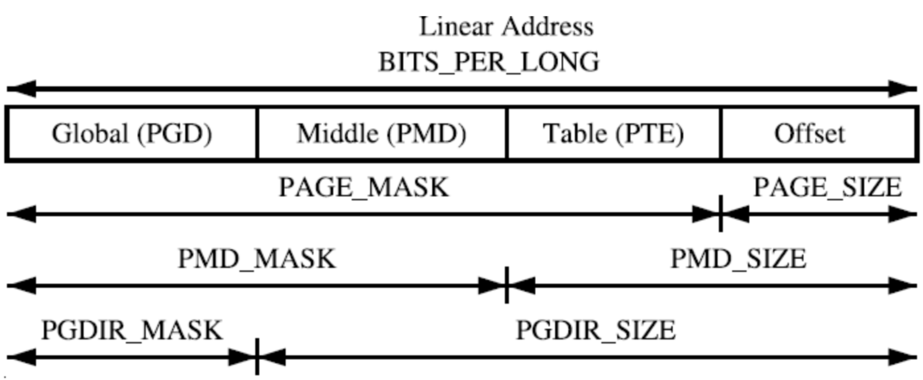
每个进程有它自己的PGD(Page Global Directory)，它是一个物理页，并包含一个pgd_t数组。

虚拟地址SHIFT宏图:



Linear Address Bit Size Macros

虚拟地址MASK和SIZE宏图:



Linear Address Size and Mask Macros

(3) Page Table Entry

PTEs, PMDs和PGDs分别由pte_t, pmd_t 和pgd_t来描述。为了存储保护位，pgprot_t被定义，它拥有相关的flags并经常被存储在page table entry低位(lower bits)，其具体的存储方式依赖于CPU架构。

每个pte_t指向一个物理页的地址，并且所有的地址都是页对齐的。因此在32位地址中有PAGE_SHIFT(12)位是空闲的，它可以为PTE的状态位。

PTE的保护和状态位如下图所示：

Bit	Function
_PAGE_PRESENT	Page is resident in memory and not swapped out
_PAGE_PROTNONE	Page is resident but not accessable
_PAGE_RW	Set if the page may be written to
_PAGE_USER	Set if the page is accessible from user space
_PAGE_DIRTY	Set if the page is written to
_PAGE_ACCESSED	Set if the page is accessed

Page Table Entry Protection and Status Bits

(4) 如何通过3级页表访问物理内存

为了通过PGD、PMD和PTE访问物理内存，其相关宏在asm/pgtable.h中定义。

- pgd_offset

根据当前虚拟地址和当前进程的mm_struct获取pgd项的宏定义如下：

```
/* to find an entry in a page-table-directory */
#define pgd_index(addr)      ((addr) >> PGDIR_SHIFT)  //获得在pgd表中的索引
#define pgd_offset(mm, addr)    ((mm)->pgd + pgd_index(addr)) //获得pmd表的起始地址

/* to find an entry in a kernel page-table-directory */
#define pgd_offset_k(addr)  pgd_offset(&init_mm, addr)
```

- pmd_offset

根据通过pgd_offset获取的pgd 项和虚拟地址，获取相关的pmd项(即pte表的起始地址)

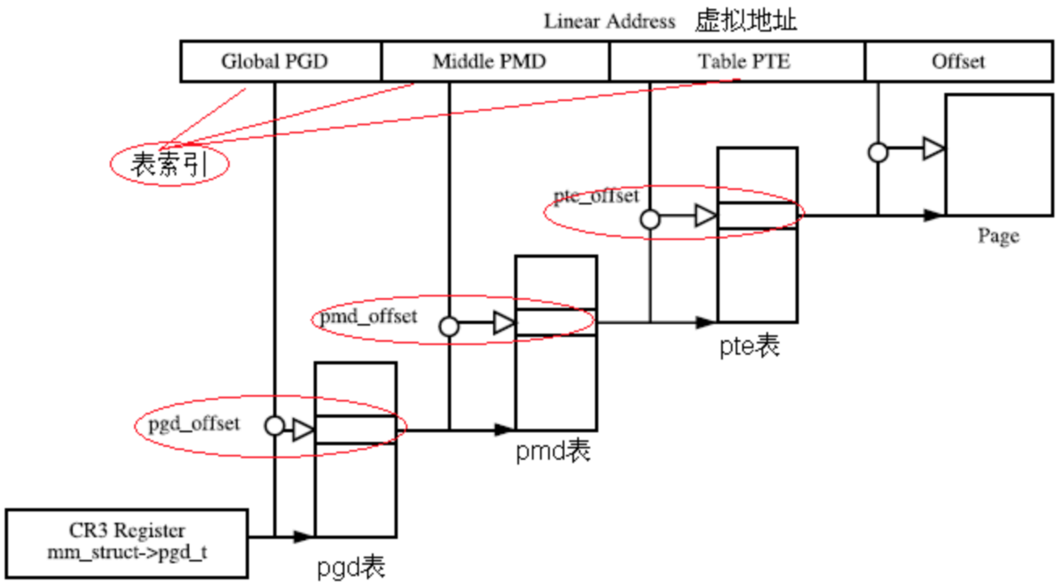
```
/* Find an entry in the second-level page table.. */
#define pmd_offset(dir, addr)   ((pmd_t *)(dir))   //即为pgd项的值

```

- pte_offset

根据通过pmd_offset获取的pmd项和虚拟地址，获取相关的pte项(即物理页的起始地址)

```
#ifndef CONFIG_HIGHPTE
#define __pte_map(pmd)      pmd_page_vaddr(*(pmd))
#define __pte_unmap(pte)    do { } while (0)
#else
#define __pte_map(pmd)      (pte_t *)kmap_atomic(pmd_page(*(pmd)))
#define __pte_unmap(pte)    kunmap_atomic(pte)
#endif

#define pte_index(addr)     (((addr) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))

#define pte_offset_kernel(pmd,addr) (pmd_page_vaddr(*(pmd)) + pte_index(addr))

#define pte_offset_map(pmd,addr)    (__pte_map(pmd) + pte_index(addr))
#define pte_unmap(pte)          __pte_unmap(pte)

#define pte_pfn(pte)        (pte_val(pte) >> PAGE_SHIFT)
#define pfn_pte(pfn,prot)   __pte(__pfn_to_phys(pfn) | pgprot_val(prot))

#define pte_page(pte)       pfn_to_page(pte_pfn(pte))
#define mk_pte(page,prot)   pfn_pte(page_to_pfn(page), prot)

#define set_pte_ext(ptep,pte,ext) cpu_set_pte_ext(ptep,pte,ext)
#define pte_clear(mm,addr,ptep) set_pte_ext(ptep, __pte(0), 0)
```

其示意图如下图所示：



Page Table Layout

（5）用户虚拟地址&内核虚拟地址&物理地址

[(8条消息) 用户虚拟地址转化成物理地址，物理地址转换成内核虚拟地址，内核虚拟地址转换成物理地址，虚拟地址和对应页的关系*p0inter*的博客-*CSDN*博客虚拟地址对应的页号]()

2. Debug记录

（1）关于boot_block→...→main的传参

开始就报错：

```
No ethernet found.

virtio read: device 0 block # 0, count 2 ... 2 blocks read: OK
=> loadbootm
It's a bootloader...
qemu-system-riscv64: virtio: bogus descriptor or out of resources
```

gdb跟踪得知一跳转到main就会出错：

```
virtio read: device 0 block # 0, count 2 ... 2 blocks
 read: OK
=> loadbootm
It's a bootloader...
qemu-system-riscv64: virtio: bogus descriptor or out
of resources
```

```
Continuing.

Thread 1 hit Breakpoint 2, main ()
    at ./arch/riscv/boot/bootblock.S:50
50              j kernel
(gdb) s
```

再更细粒度地跟踪：

```
Continuing.

Thread 1 hit Breakpoint 4, _start ()
    at ./arch/riscv/kernel/head.S:32
32          la tp, m_pid0_pcb
(gdb) s
33          la sp, M_KERNEL_SP    // set stack pointer
to the given addr
(gdb) p $tp
$1 = (void *) 0xffffffc050204e98 <m_pid0_pcb>
(gdb) n
_start () at ./arch/riscv/kernel/head.S:34
34          call main
(gdb) p $sp
$2 = (void *) 0x50501000
(gdb) 
```

奇怪的是 `init_task_info` 被执行了两次，同时下一次执行就会报错

```
            Type: Hard Disk
            Capacity: 0.0 MB = 0.0 GB (68 x 512)
... is now current device
** No partition table - virtio 0 **
No ethernet found.
No ethernet found.

virtio read: device 0 block # 0, count 2 ... 2 blocks
 read: OK
=> loadbootm
It's a bootloader...
qemu-system-riscv64: virtio: bogus descriptor or out
of resources
```

```
(gdb) s
196             smp_init();
(gdb) n
197             lock_kernel();
(gdb)
203             init_task_info(app_info_loc, app_inf
o_size);
(gdb)
200             init_jmptab();
(gdb)
203             init_task_info(app_info_loc, app_inf
o_size);
(gdb) n
```

同时gdb跟踪时发现参数都被优化掉了，理想状态下执行的flow：

$$bootbloader \rightarrow \_boot \rightarrow boot\_kernel \rightarrow \_start \rightarrow main$$

在逐层调用的过程中信息早就传丢了：

```
Breakpoint 2, main (app_info_loc=0,
    app_info_size=0,
    seq_end_loc=331776,
    seq_start_loc=0)
    at ./init/main.c:205
205         cpu_id = get_current_cpu_id(
);
```

再有，取消临时映射时保持错：

```
Breakpoint 1, disable_tmp_map ()
    at ./init/main.c:33
33          uint64_t va = 0x50200000;
(gdb) s
34          PTE *pgdir = pa2kva(PGDIR_PA);
(gdb) n
35          uint64_t vpn2 =
(gdb) p pgdir
$1 = (PTE *) 0xffffffc051000000
(gdb) s
37          uint64_t vpn1 = (vpn2 << PPN_BITS) ^
(gdb) p $sp
$2 = (void *) 0x50500fa0
(gdb) p $pc
$3 = (void (*)()) 0xffffffc0502026b0 <disable_tmp_map+40>
(gdb) p $ra
$4 = (void (*)()) 0xffffffc0502026a0 <disable_tmp_map+24>
(gdb) s
38                          (va >> (NORMAL_PAGE_SHIFT + PPN_BI
(gdb)
37          uint64_t vpn1 = (vpn2 << PPN_BITS) ^
(gdb)
39          PTE *pmd = (PTE *)pgdir[vpn2];
(gdb)
41          pmd[vpn1] = 0;
(gdb) p pmd
$5 = (PTE *) 0x14400801
```

意识到应该从表项中取出实地址后再传递给pmd。

（2）未理清虚实地址

在bzero出错：

```
Breakpoint 2 at 0xffffffc050202506: file ./arch/ri
scv/kernel/boot.c, line 72.
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0x0000000000000000 in ?? ()
(gdb) p $pc
$1 = (void (*)()) 0x0
(gdb) p $ra
$2 = (void (*)()) 0xffffffc050207216 <bzero+34>
```

断点打在clear_pgdir（代码中唯一调用该函数的母函数）：

```
Breakpoint 1, clear_pgdir (
    pgdir_addr=<error reading variable: Cannot acc
ess memory at address 0x50500f28>)
    at ./arch/riscv/include/pgtable.h:122
122         bzero((void*)pgdir_addr, NORMAL_PAGE_S
IZE);
(gdb) p $ra
$1 = (void (*)()) 0xffffffc0502042f6 <alloc_page_h
elper+202>
(gdb)
```

猜测 是在alloc_page_helper处没处理好虚实地址映射，原本处理如下：

```
    if (pgd[vpn2] == 0) {
        // 分配一个新的三级页目录，注意需要转化为实地址！
        set_pfn(&pgd[vpn2], kva2pa(allocPage(1)) >>
NORMAL_PAGE_SHIFT);
        set_attribute(&pgd[vpn2], _PAGE_PRESENT | _PAGE_USER);
        clear_pgdir(get_pa(pgd[vpn2]));
    }
```

意识到 `get_pa` 后还需要转换为虚拟地址。

（3）取消临时映射后的奇怪bug

改为在boot_kernel处向main传参：

但bios_sdread时还是会出错

```
Breakpoint 1, main (
    app_info_loc=64148,
    app_info_size=168,
    seq_end_loc=331776,
    seq_start_loc=1344283048)
    at ./init/main.c:205
205         cpu_id = get_current_cpu_id();
(gdb) p buffer
$1 = '\000' <repeats 1023 times>
(gdb) x buffer
0xffffffc05020acb0 <buffer>:    0x00000000
(gdb)
```

发现参数被截断：

```
(gdb) x buffer
0xffffffc05020acb0 <buffer>:    0x00000000
(gdb) p start_src
No symbol "start_src" in current context.
(gdb) p start_sec
$1 = 125
(gdb) s
bios_sdread (mem_address=1344318640,
    num_of_blocks=1, block_id=125)
    at include/os/kernel.h:54
54          return call_jmptab(SD_READ, (long
)mem_address, (long)num_of_blocks, \
(gdb) p/x mem_address
$2 = 0x5020acb0
(gdb) p $satp
$3 = -9223372036854444032
(gdb) p/x $satp
$4 = 0x8000000000051000
(gdb) ptype/o mem_address
type = unsigned int
(gdb)
```

应该在取消临时映射前读取任务信息还是取消临时映射后？取消映射后将无法访问参数？

```
Breakpoint 1, main (app_info_loc=64140,
    app_info_size=168, seq_end_loc=331776,
    seq_start_loc=1344283048)
    at ./init/main.c:205
205         cpu_id = get_current_cpu_id();
(gdb) n
207         if(cpu_id==0){
(gdb)
209             disable_tmp_map();
(gdb) p $app_info_loc
$1 = void
(gdb) p app_info_loc
$2 = 64140
(gdb) n
211             smp_init();
(gdb) p app_info_loc
Cannot access memory at address 0x50500fec
(gdb)
```

不如直接在main里读taskinfo的信息，省的出事……

似乎在取消映射后将导致main中数据都无法正常读取：

```
(y or n) y
Reading symbols from build/main...
(gdb) b main
Breakpoint 1 at 0x502030a2: main.
(2 locations)
(gdb) d 1
(gdb) b alloc_page_helper
Breakpoint 2 at 0x5020422e: alloc_
page_helper. (2 locations)
(gdb) c
Continuing.

Breakpoint 2, alloc_page_helper
    (
    va=<error reading variable: Ca
nnot access memory at address 0x50
500f68>,
    pgdir=<error reading variable:
 Cannot access memory at address 0
x50500f60>)
    at ./kernel/mm/mm.c:51
51          va &= VA_MASK;
```

那就在做完所有初始化工作之后再取消映射（取消映射的原因是避免用户程序也用到该页表导致错误，故只要在切换到用户程序之前取消该映射即可）。

---

gdb跟踪：

```
73          screen_move_cursor(current_running[cpu_id]->cursor_x, current_running[cpu_id]->cursor_y);
(gdb)
74          set_satp(SATP_MODE_SV39, current_running[cpu_id]->pid, current_running[cpu_id]->pgdir >> NORMAL_PAGE_SHIFT);
(gdb)
Remote connection closed
(gdb)
```

查看函数原型，意识到要使用页表的实地址：

```
static inline void set_satp(
    unsigned mode, unsigned asid, unsigned long ppn)
```

修改后：

```
zero  : 0000000000000000  ra  v: 000000000000>0gp   : 000000005f771d98  tp   : 00000000000
  t1   : 0000000000000000  t2   : 00000000000
  s1   : 0000000000000000  a0   : 00000000000
  a2   : 0000000000000000  a3   : 00000000000
  a5   : 0000000000000000  a6   : 00000000000
  s2   : 0000000000000000  s3   : 00000000000
  s5   : 0000000000000000  s6   : 00000000000
  s8   : 0000000000000000  s9   : 00000000000
  s11  : 0000000000000000  t3   : 00000000000
zero  : 0000000000000000  ra  v: 000000000000000004sp   : 0000000000000000
> gp   : 000000005f771d98  tp   : 000000000000000(  t0   : 0000000000000000
  t1   : 0000000000000000  t2   : 0000000000000000 s0/fp : 0000000000000000
  s1   : 0000000000000000  a0   : 0000000000000000  a1   : 0000000000000000
  a2   : 0000000000000000  a3   : 0000000000000000  a4   : 0000000000000000
  a5   : 0000000000000000  a6   : 0000000000000000  a7   : 0000000000000000
  s2   : 0000000000000000  s3   : 0000000000000000  s4   : 0000000000000000
  s5   : 0000000000000000  s6   : 0000000000000000  s7   : 0000000000000000
  s8   : 0000000000000000  s9   : 0000000000000000  s10  : 0000000000000000
  s11  : 0000000000000000  t3   : 0000000000000000  t4   : 0000000000000000
  t5   : 0000000000000000  t6   : 0000000000000000

 sstatus: 0x40020 sbadaddr: 0x0 scause: 12

 sepc: 0x0

 tval: 0x0 cause: 0xc

Assertion failed at handle_other in ./kernel/irq/irq.c:78
```

```
#define EXC_INST_PAGE_FAULT 12
```

注意区别实地址、用户虚地址、内核虚地址，表项与上USER_MODE后，报错变为 `EXC_STORE_PAGE_FAULT`；

```
zero  : 0000000000000000  ra  v: 0000000000001000004sp  0
> gp   : 000000005f771d98  tp   : 0000000000000000  t0 0
  t1   : 0000000000000000  t2   : 0000000000000000 s0/f0
  s1   : 0000000000000000  a0   : 0000000000000000  a1 0
  a2   : 0000000000000000  a3   : 0000000000000000  a4 0
  a5   : 0000000000000000  a6   : 0000000000000000  a7 0
  s2   : 0000000000000000  s3   : 0000000000000000  s4 0
  s5   : 0000000000000000  s6   : 0000000000000000  s7 0
  s8   : 0000000000000000  s9   : 0000000000000000  s100
  s11  : 0000000000000000  t3   : 0000000000000000  t4 0
  t5   : 0000000000000000  t6   : 0000000000000000
 sstatus: 0x40020 sbadaddr: 0xf0000fff8 scause: 15
 sepc: 0x10002
 tval: 0xf0000fff8 cause: 0xf
QEMU: Terminated at handle_other in ./kernel/irq/irq.c:8
```

gdb跟踪，意识到存储在pcb中的user_sp不应该为内核虚地址，应该为用户虚地址，而传参时我们才需要使用内核虚地址。

```
(gdb) p $tp
$3 = (void *) 0xffffffc05020b498 <pcb>
(gdb) p $sepc
$4 = -273533624284
(gdb) p/x $sepc
$5 = 0xffffffc050202024
(gdb) n
ret_from_exception ()
    at ./arch/riscv/kernel/entry.S:218
218        sret
(gdb) p/x $sepc
$6 = 0x10000
(gdb) p/x $sp
$7 = 0xffffffc052006000
(gdb) p/x $sp
```

但是还是报错，意识到在使用alloc_page_helper时应该修改为如下：

```
        uint64_t user_sp_kva = (reg_t)alloc_page_helper(USER_STACK_ADDR-PAGE_SIZE, pcb[index].pgdir)+PAGE_SIZE;
```

由于栈是向低地址生长的，这样才能保证页表存在对应项。

虽然还是报错，但咱至少换位置了不是......

```
zero  : 0000000000000000  ra  v: 000000000001000004sp   0
> gp  : 000000005f771d98  tp   : 0000000000000000  t0  0
  t1  : 0000000000011f60  t2   : 0000000000000000 s0/fp0
  s1  : 0000000000000000  a0   : 0000000000000000  a1  0
  a2  : 0000000000000000  a3   : 0000000000000000  a4  0
  a5  : 0000000000000000  a6   : 0000000000000000  a7  0
  s2  : 0000000000000000  s3   : 0000000000000000  s4  0
  s5  : 0000000000000000  s6   : 0000000000000000  s7  0
  s8  : 0000000000000000  s9   : 0000000000000000  s10 0
  s11 : 0000000000000000  t3   : 0000000000000000  t4  0
  t5  : 0000000000000000  t6   : 0000000000000000
 sstatus: 0x40020 sbadaddr: 0x11e80 scause: 15
 sepc: 0x10018
 tval: 0x11e80 cause: 0xf
Assertion failed at handle_other in ./kernel/irq/irq.c:78
```

查看反汇编代码：

```
0000000000010000 <_start>:
   10000:   1141                    addi    sp,sp,-16
   10002:   e422                    sd  s0,8(sp)
   10004:   e006                    sd  ra,0(sp)
   10006:   0800                    addi    s0,sp,16
   10008:   00002297            auipc   t0,0x2
   1000c:   e7828293            addi    t0,t0,-392 # 11e80 <_edata>
   10010:   00002317            auipc   t1,0x2
   10014:   f5030313            addi    t1,t1,-176 # 11f60 <__BSS_END__>

0000000000010018 <do_clear>:
   10018:   0002a023            sw  zero,0(t0)
   1001c:   0291                    addi    t0,t0,4
   1001e:   fe535de3            bge t1,t0,10018 <do_clear>
   10022:   080000ef            jal ra,100a2 <main>
   10026:   4885                    li  a7,1
```

原本load task时是这样处理：

```c
uint64_t map_task(char *taskname, uintptr_t pgdir){
    int i;
    uint64_t entry_addr;
    for(i=0;i<TASK_MAXNUM;i++){
        if(strcmp(taskname, tasks[i].taskname)==0){
            entry_addr = pa2kva(TASK_MEM_BASE + TASK_SIZE * i);
            uintptr_t va = alloc_page_helper(USER_ENTRYPOINT, pgdir);
            // 将任务拷贝到分配的虚地址
            memcpy(va, entry_addr, tasks[i].task_size);
```

```
            // // 将虚地址和实地址做映射，存于用户的页表中
            // map_page(USER_ENTRYPOINT, entry_addr, pgdir);
            // // 清空bss段
            // bzero(va+task[i].task_size, task[i].p_memsz - task[i].task_size);
            return USER_ENTRYPOINT;  // 返回用户虚地址
        }
```

意识到这样未考虑到memsz大于一个PAGE的情况，如上述的 `11f60 <__BSS_END__>` ，其相较于起始地址 `0x10000` ，实际上是位于下一页了。

单核可以跑了，照理说起双核原理也是一样的，但莫名其妙两个核都卡在了抢锁上：

```
    at ./kernel/locking/lock.c:35
35          while(atomic_swap(LOCKED, &lock->status)==LOCKED);
(gdb) c
Continuing.
^C
Thread 2 received signal SIGINT, Interrupt.
[Switching to Thread 1.2]
0xffffffc0502045dc in spin_lock_acquire (
    lock=0xffffffc05020b508 <klock>)
    at ./kernel/locking/lock.c:35
35          while(atomic_swap(LOCKED, &lock->status)==LOCKED);
(gdb) thread 1
[Switching to thread 1 (Thread 1.1)]
#0  0xffffffc0502045dc in spin_lock_acquire (
    lock=0xffffffc05020b508 <klock>)
    at ./kernel/locking/lock.c:35
35          while(atomic_swap(LOCKED, &lock->status)==LOCKED);
(gdb) p klock
$4 = {status = LOCKED}
(gdb)
```

gdb跟踪：

```
Thread 1 hit Breakpoint 1, ret_from_exception ()
    at ./arch/riscv/kernel/entry.S:216
216         call unlock_kernel
(gdb) thread 2
[Switching to thread 2 (Thread 1.2)]
#0  0xffffffc0502045dc in spin_lock_acquire (
    lock=<error reading variable: Cannot access memory at address 0x50501fc8>)
    at ./kernel/locking/lock.c:35
35            while(atomic_swap(LOCKED, &lock->status)==LOCKED);
<nt_running
$1 = {
  0xffffffc05020b520 <pcb>,
  0xffffffc050207d70 <s_pid0_pcb>}
```

意识到在持有锁的情况下触发了异常，导致后续重复抢锁：

```
(gdb) thread 2
[Switching to thread 2 (Thread 1.2)]
#0  lock_kernel () at ./kernel/smp/smp.c:23
23            spin_lock_acquire(&klock);
(gdb) p $sie
$9 = 0
(gdb) p $sstatus
$10 = 288
(gdb) p/x $sstatus
$11 = 0x120
(gdb) p $ra
$12 = (void (*)()) 0xffffffc050202168 <exception_handler_entry+112>
(gdb) p $stval
$13 = 1347428328
(gdb) p/x $stval
$14 = 0x50501fe8
(gdb) p/x $scause
$15 = 0xd
(gdb)
```

gdb断点打在例外入口：

```
    0xffffffc0502030fa <call_bios+20>:
        sd  a3,-64(s0)
(gdb) p $scause
$2 = -9223372036854775803
(gdb) p/x $scause
$3 = 0x8000000000000005
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, exception_handler_entry ()
    at ./arch/riscv/kernel/entry.S:224
224         SAVE_CONTEXT
(gdb) p $ra
$4 = (void (*)()) 0xffffffc0502030ac <main+256>
(gdb) 
```

查看当前对应指令：

```
ffffffc0502030a0:   4705                    li  a4,1
ffffffc0502030a2:   c7f8                    sw  a4,76(a5)
ffffffc0502030a4:   8e0ff0ef            jal ra,ffffffc050202184 <setup_exception>
ffffffc0502030a8:   524020ef            jal ra,ffffffc0502055cc <get_ticks>
ffffffc0502030ac:   872a                    mv  a4,a0
ffffffc0502030ae:   6785                    lui a5,0x1
ffffffc0502030b0:   38878793            addi    a5,a5,904 # 1388 <boot_stack_top_base-0x50200c78>
ffffffc0502030b4:   97ba                    add a5,a5,a4
```

发现是在bios_set_timer处触发的例外。

变量开在栈上，但是栈地址不对：

```
(gdb)
212             smp_init();
(gdb) s
smp_init ()
```

```
    at ./kernel/smp/smp.c:11
11          spin_lock_init(&klock);
(gdb)
spin_lock_init (
    lock=<error reading variable: Cannot access memory at address 0x50500fc8>)
    at ./kernel/locking/lock.c:23
23          lock->status = UNLOCKED;
(gdb) p &klock
$1 = (spin_lock_t *) 0xffffffc05020b5a0 <klock>
(gdb) p $sp
$2 = (void *) 0x50500fc0
(gdb)
```

但是这么写会报错：

```
  /* TODO: [p1-task2] setup C environment */
  la tp, m_pid0_pcb
  la sp, M_KERNEL_SP    // set stack pointer to the given addr
  call main
// 从核只需设置初始tp和sp
s_start:
  la tp, s_pid0_pcb
  la sp, S_KERNEL_SP
```

> [Error: offset too large · Issue #I4QAZ3 · unicornx/riscv-operating-system-mooc - Gitee.com]()

但是这样可行？

```
la tp, m_pid0_pcb
```

不对得用 `li` ……（我就是人类高质量sb）

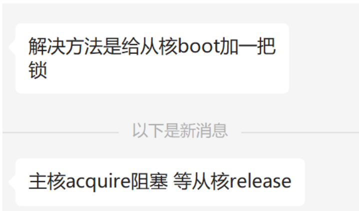
（4）双核时关闭临时映射的时机

双核跑起来后ps一下发现只有主核在跑，gdb跟踪一下：

```
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
0x000000005ff92a10 in ?? ()
(gdb) thread 2
[Switching to thread 2 (Thread 1.2)]
#0  _boot () at ./arch/riscv/kernel/start.S:20
20          csrw CSR_SIE, zero
(gdb) n
Cannot access memory at address 0x50202000
(gdb)
```

发现由于主核已经关闭了地址映射，导致从核初始化有问题。难道让从核关闭地址映射？那样单核就不行了......

再回忆一下关闭地址映射是为了啥：避免用户程序复制内核页表后出错，所以我们只需要保证用户页表初始化前地址映射关闭了即可，如何保证从核初始化完毕后才起用户页表？询问助教：



3. Design Review的Q&A

（1）页表的基本info

- 页表的数据结构：
- 页表项的数据结构：

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

图 P4-5: Sv39 页表项

- 内核页表实地址&页表大小？

```
#define PGDIR_PA 0x51000000lu
```

一个二级页表：

38　30	29　21	20	0
VPN[2]	VPN[1]		page offset
9	9	9	12

地址空间32位，页表项4B（64bit），对一级页表：

$$4GB/4KB*4Byte=4MB$$

worst case：再加一级页目录：

$$4MB/4KB*4Byte=4KB$$

共计$4MB+4KB$，但实际取决于用了多少页。

（2）用户页表初始化的flow

- 给每个PCB绑定一个页表；
- 每次分配前：
  - 清空页表
  - share内核页表

> 但在创建线程时共享页表......

（3）载入用户程序?

- 主核初始化时读入固定位置

> 我是统一放在 `TASK_MEM_BASE   0x57000000` 之后

- 初始化时为每个PCB绑定内核栈，复用即可；
- 用户栈需每次重新分配内核虚地址，因为会clear用户页表；

> 除非额外记录用户栈的用户虚地址所对应的三级页表的各个表项

- taskinfo新设立成员变量mem_sz,根据mem_sz分配页，并在_start里做bss段清空；
- 根据name从已经载入的程序中copy

（4）页缺失处理?

- 使用alloc_page_helper为触发例外的虚地址 `stval` 分配页表（不区分load or store，统一置A,D）
- 刷新tlb

(5）页替换算法

还没写，拟采用FIFO。

> 推荐维护计数器

(6）创建线程

- 共用的：代码段&数据段、页表

  > 其中代码段的共用是通过指定entry+共享页表实现的；
  > 数据段的共享：全局数据通过相对于pc的偏移得到的；

- 私有的：用户&内核栈

> 内核栈也可共用

> 问题：
>
> - task3中：
>
>   1. 在 QEMU 上调试本任务时，当 SD 卡读写的范围超过镜像大小时将会报错。建议在本任务中制作完成镜像后，在后方 padding 一些空间以方便 QEMU 上 SD 卡的读写。可以采用命令：
>
>   ```
>   dd if=/dev/zero of=image oflag=append conv=notrunc bs=512MB count=2
>   ```
>
>   该命令表示在镜像 image 后方 padding 两块大小为 512MB 的空间即 1GB，为了方便，建议大家将该命令写入到 Makefile 中。
>
>   加在哪？
>
> - 如果想实现其他换页算法，如何获知页被访问的次数&最近一次访问的时间？

TASK 2 动态页表和按需调页

1. 思路分析

此task很直观，主要任务是处理缺页例外，在发生 `inst_page_fault`，`load_page_fault`，`store_page_fault` 时检查是否未建立映射，并分配物理页

2. Debug记录

(1）用户栈传参处理有误

处理页缺失时卡死：

```
(gdb) c
Continuing.

```

```
Breakpoint 1, handle_page_fault (
    regs=0xffffffc052004ee0,
    stval=18446743800207572968, scause=13)
    at ./kernel/irq/irq.c:86
86            alloc_page_helper(stval, current_running[cpu_id]->pgdir);
(gdb) p/x stval
$3 = 0xffffffc05202ffe8
(gdb) c
Continuing.

Breakpoint 1, handle_page_fault (
    regs=0xffffffc052004ee0,
    stval=18446743800207572968, scause=13)
    at ./kernel/irq/irq.c:86
86            alloc_page_helper(stval, current_running[cpu_id]->pgdir);
(gdb) p/x stval
$4 = 0xffffffc05202ffe8
```

发现即使分配了也在原地一直触发例外：

```

Breakpoint 1, handle_page_fault (
    regs=0xffffffc052004ee0,
    stval=18446743800207572968, scause=13)
    at ./kernel/irq/irq.c:86
86            alloc_page_helper(stval, current_running[cpu_id]->pgdir);
(gdb) s
alloc_page_helper (
    va=18446743800207572968,
    pgdir=18446743800207396864)
    at ./kernel/mm/mm.c:52
52            va &= VA_MASK;
(gdb) n
53            uint64_t vpn2 =
(gdb)
55            uint64_t vpn1 = (vpn2 << PPN_BITS) ^
(gdb)
56                            (va >> (NORMAL_PAGE_SHIFT + PPN_BITS));
(gdb)
55            uint64_t vpn1 = (vpn2 << PPN_BITS) ^
(gdb)
57            uint64_t vpn0 = (vpn2 << (PPN_BITS + PPN_BITS)) ^
(gdb)
58                            (vpn1 << PPN_BITS) ^
(gdb)
```

```
57          uint64_t vpn0 = (vpn2 << (PPN_BITS + PPN_BITS)) ^
(gdb)
59                          (va >> NORMAL_PAGE_SHIFT);
(gdb)
57          uint64_t vpn0 = (vpn2 << (PPN_BITS + PPN_BITS)) ^
(gdb)
60          PTE *pgd = (PTE*)pgdir;
(gdb)
61          if (pgd[vpn2] == 0) {
(gdb) p pgd[vpn2]
$1 = 339739649
(gdb) n
67          PTE *pmd = (uintptr_t *)pa2kva((get_pa(pgd[vpn2])));
(gdb)
68          if(pmd[vpn1] == 0){
(gdb) p pmd[vpn1]
$2 = 343933135
(gdb) n
74          PTE *pte = (PTE *)pa2kva(get_pa(pmd[vpn1]));
(gdb)
75          if(pte[vpn0] == 0){
(gdb) p pte[vpn0]
$3 = 0
(gdb) n
77              ptr_t pa = kva2pa(allocPage(1));
(gdb)
78              set_pfn(&pte[vpn0], pa >> NORMAL_PAGE_SHIFT);
(gdb) p pa
$4 = 1375928320
(gdb) p/x pa
$5 = 0x52030000
(gdb) n
80                  &pte[vpn0], _PAGE_PRESENT | _PAGE_READ | _PAGE_WRITE |
(gdb)
79              set_attribute(
(gdb)
83          return pa2kva(get_pa(pte[vpn0]));
(gdb) s
get_pa (entry=343982303)
    at ./arch/riscv/include/pgtable.h:94
94          return (uint64_t) ((entry>>_PAGE_PFN_SHIFT) << NORMAL_PAGE_SHIFT);
(gdb) p $a0
$6 = 343982303
(gdb) p/x $a0
$7 = 0x1480c0df
(gdb) s
95      }
```

```
(gdb)
pa2kva (pa=1375928320)
    at ./arch/riscv/include/pgtable.h:86
86          return pa + KVA_OFFSET ;
(gdb)
88      }
(gdb)
alloc_page_helper (va=276253835240,
    pgdir=18446743800207396864)
    at ./kernel/mm/mm.c:84
84      }
(gdb) p $a0
$8 = -273501978624
(gdb) p/x $a0
$9 = 0xffffffc052030000
(gdb) p pte[vpn0]
$10 = 343982303
(gdb) n
handle_page_fault (
    regs=0xffffffc052004ee0,
    stval=18446743800207572968, scause=13)
    at ./kernel/irq/irq.c:87
87          local_flush_tlb_all();
(gdb)
88      }
(gdb) c
Continuing.

Breakpoint 1, handle_page_fault (
    regs=0xffffffc052004ee0,
    stval=18446743800207572968, scause=13)
    at ./kernel/irq/irq.c:86
86          alloc_page_helper(stval, current_running[cpu_id]-
>pgdir);
(gdb) s
alloc_page_helper (
    va=18446743800207572968,
    pgdir=18446743800207396864)
    at ./kernel/mm/mm.c:52
52          va &= VA_MASK;
(gdb) n
53          uint64_t vpn2 =
(gdb)
55          uint64_t vpn1 = (vpn2 << PPN_BITS) ^
(gdb)
56                          (va >> (NORMAL_PAGE_SHIFT +
PPN_BITS));
(gdb)
55          uint64_t vpn1 = (vpn2 << PPN_BITS) ^
```

```
(gdb)
57          uint64_t vpn0 = (vpn2 << (PPN_BITS + PPN_BITS)) ^
(gdb)
58                          (vpn1 << PPN_BITS) ^
(gdb)
57          uint64_t vpn0 = (vpn2 << (PPN_BITS + PPN_BITS)) ^
(gdb)
59                          (va >> NORMAL_PAGE_SHIFT);
(gdb)
57          uint64_t vpn0 = (vpn2 << (PPN_BITS + PPN_BITS)) ^
(gdb)
60          PTE *pgd = (PTE*)pgdir;
(gdb)
61          if (pgd[vpn2] == 0) {
(gdb)
67          PTE *pmd = (uintptr_t *)pa2kva((get_pa(pgd[vpn2])));
(gdb)
68          if(pmd[vpn1] == 0){
(gdb)
74          PTE *pte = (PTE *)pa2kva(get_pa(pmd[vpn1]));
(gdb) p pte[vpn0]
Cannot access memory at address 0x178
(gdb) n
75          if(pte[vpn0] == 0){
(gdb) p pte[vpn0]
$11 = 343982303
(gdb) n
83          return pa2kva(get_pa(pte[vpn0]));
(gdb)
```

gdb查看此时触发例外的pc，查看对应的反汇编代码：

```
  {
      mem1 = atol(argv[i]);
 100c4:   fec42783          lw  a5,-20(s0)
 100c8:   078e                  slli    a5,a5,0x3
 100ca:   fc043703          ld  a4,-64(s0)
 100ce:   97ba                  add a5,a5,a4
 100d0:   639c                  ld  a5,0(a5)
 100d2:   853e                  mv  a0,a5
 100d4:   792000ef          jal ra,10866 <atol>
```

发现还是在取参数的时候，可是这个时候怎么会触发例外呢？

回忆参数的处理：

```c
for(int i=argc-1; i>=0; i--){
            int len = strlen(argv[i])+1;    //要拷贝'\0'
            user_sp_kva -=len;
            pcb[index].user_sp -=len;
            argv_ptr[i] = (char*)user_sp_kva;
            strcpy((char*)user_sp_kva, argv[i]);
        }
```

意识到在用户态使用这些参数，则存在argv指针数组里的应该是用户态虚地址，而非内核态虚地址，否则将导致取数访问错误地址。同理argv_ptr也应该使用内核态虚地址。

妥了：

```
  0x10800000, 1282426202
0x80200000, 1695834973
0xa0000320, 447732900
Success!









------------------- COMMAND -------------------
> root@UCAS_OS: exec rw 0x10800000 0x80200000 0xa0000320
Info: excute rw successfully, pid = 2
> root@UCAS_OS:
```

TASK 3 换页

1. 思路分析

(1) 思路1

- 链表结点中存的是什么？
    - uva
    - pa
    - 在磁盘中的位置（如果被换出的话）
    - 无需维护status，由其所在的链表自然可得
    - 其对应的页表信息

- 要维护几个list?
  - `free_list`：个数maximum=我们所认为的用户可能用的最多page数（需要给定因为我们没有实现malloc）；
  - `in_use_list`：个数maximum=内核地址所允许的上限;
    - 每次分配插入表尾;
    - 替换从表头拿，放到 `swap_out_list`；
  - swap_out_list：被换出的uva对应结点放在这里面，便于查找;
    - 每次替换回来需将之放回 `in_use_list`；
- 如何通过uva or pa得到下标?
  - 老实查找（听起来不太聪明的样子......）

> 这个处理方式可以较好地复用之前链表的API，缺点是索引不便，一个实地址可能对应多个node，页替换的时候需要格外小心。

（2）思路2（写废了）

- Trigger：既然可使用的总物理页框是已知的，且若刚开始就给每个物理页框分配一个info结构体，由于pa是连续的，还可根据pa快速地索引info结点;
- info结点中存储的内容:
  - `MAX_TASK_NUM` 个元素的数组，存放uva;
  - `pgdir_mask`，理论上只要 `MAX_TASK_NUM` 个即可，对应位为1表示pa在该页表中存在映射关系;
    - 回收时还需要看 `pgdir_mask` 是否为0才可回收对应的物理页框

      > Note：共享内存、多线程的case
  - `on_disk_sec`，若被换出，其在磁盘中的位置
- 链表形式：双向循环静态链表;
  - 静态链表（数组）的头三个元素：空闲链表头结点、被换出的元素列表的头结点、处于内存中的页框的链表头结点;
  - 为啥要双向?
    - 较易实现FIFO算法：若每次皆从表头插入新结点，则表尾就是下一次该被swap出的结点，双向可快速定位到表尾;
  - 为啥要静态?
    - 便于根据pa直接获取index

（3）测试用例设计

- 限定用户可使用的物理页框数；
- 测试用例中访问页数多于该值，以触发换页；
- 设计history结构体记录每次随机写的写地址和数据；
- 后续check时对照history比较数据是否正确；

TASK 4 多线程

1. 思路分析

P1做过了，唯一有变化的就是页表要使用父进程的，其他倒没啥。

2. Debug记录

（1）用户栈虚拟地址设置有误

copy参数时触发例外：

```
Breakpoint 1, do_pthread_create (
    thread=0xf0000ff58, start_routine=0x1041e,
    arg=0x61) at ./kernel/sched/sched.c:203
203         int index = search_free_pcb();
(gdb) n
204         if(index==-1)   // 进程数已满，返回
(gdb)
207         pcb[index].pgdir = current_running[cpu_id]->pgdir;
(gdb)
208         pcb[index].kernel_sp = ROUND(pcb[index].kernel_sp,
PAGE_SIZE);
(gdb)
209         pcb[index].user_sp = USER_STACK_ADDR;
(gdb)
210         uint64_t user_sp_kva =
(reg_t)alloc_page_helper(USER_STACK_ADDR-PAGE_SIZE,
pcb[index].pgdir)+PAGE_SIZE;
(gdb)
211         pcb[index].pid = tasknum + 1; // pid 0 is for kernel
(gdb)
212         pcb[index].status = TASK_READY;
(gdb)
213         pcb[index].cursor_x = 0;
(gdb)
214         pcb[index].cursor_y = 0;
(gdb)
```

```
215         pcb[index].cpu_mask = current_running[cpu_id]->cpu_mask;    // 继承父进程的mask
(gdb)
218         user_sp_kva -= sizeof(char*);
(gdb)
219         pcb[index].user_sp -= sizeof(char*);
(gdb)
221         int len = strlen((char*)arg)+1;    //要拷贝'\0'
(gdb)
^C
Program received signal SIGINT, Interrupt.
0xffffffc05020469a in spin_lock_acquire (
    lock=0xffffffc05020b648 <klock>)
    at ./kernel/locking/lock.c:35
35          while(atomic_swap(LOCKED, &lock->status)==LOCKED);
(gdb) p $stval
$1 = 97
(gdb) p $scause
$2 = 13
(gdb) p/x $stval
$3 = 0x61
(gdb) p $sstatus
$4 = 262400
(gdb) p/x $sstatus
$5 = 0x40100
```

但是查看sstatus已经置位。注意到此时的 `stval` 有点诡异，意识到此时传参不再是使用char*数组，查看测试用例：

```
    pthread_create(&recv, recv_thread, (void*)(unsigned long)id);


void recv_thread(void *arg)
{
    char id = (unsigned long) arg;
    ..
}
```

只要简单的放在寄存器里，无需搬到栈上。修改后报错：

```
[U-BOOT] ERROR: truly_illegal_insn       : 00000004
exception code: 2 , Illegal instruction , epc 11bc0 , ra 11bc0
### ERROR ### Please RESET the board ###
```

gdb查看出错原因：

```
Program received signal SIGINT, Interrupt.
0x000000005ffcabc8 in ?? ()
(gdb) p $ra
$1 = (void (*)()) 0x5ffcabc8
(gdb) p $stval
$2 = 0
(gdb) p $scause
$3 = -9223372036854775803
(gdb) p/x $scause
$4 = 0x8000000000000005
(gdb) p/x $sepc
$6 = 0x1091e
```

由于线程共用页表，则开栈时不能用同样的栈地址，原先实现如下：

```
    pcb[index].kernel_sp = ROUND(pcb[index].kernel_sp, PAGE_SIZE);

    pcb[index].user_sp = USER_STACK_ADDR;
    uint64_t user_sp_kva = (reg_t)alloc_page_helper(USER_STACK_ADDR-PAGE_SIZE, pcb[index].pgdir)+PAGE_SIZE;
```

现修改如下：

```
    pcb[index].pgdir = current_running[cpu_id]->pgdir;
    pcb[index].kernel_sp = ROUND(pcb[index].kernel_sp, PAGE_SIZE);

    pcb[index].user_sp = USER_STACK_ADDR + index*PAGE_SIZE;
    uint64_t user_sp_kva = (reg_t)alloc_page_helper(pcb[index].user_sp-PAGE_SIZE, pcb[index].pgdir)+PAGE_SIZE;
```

TASK 5 共享内存

1. 思路分析

- 需要实现给出虚地址、实地址，并在给出的页表内建立映射的功能（这样用户栈也可以复用了嘿嘿嘿），参考xv6中 `mappages` 的实现。
- 取消共享页面？ → 取消映射，即把pa置零；

2. Debug记录

(1) 错把内核虚地址分配给用户

起起来后没反应，gdb跟踪：

```
sys_shmpageget (
    key=42)
    at tiny_libc/syscall.c:231
231     }
(gdb)
main (argc=1,
    argv=0xf00010ff8)
    at test/test_project4/consensus.c:55
55          if (vars == NULL) {
(gdb)
62          sys_shmpagedt((void*)vars);
(gdb)
sys_shmpagedt (
    addr=0xffffffc052030000)
    at tiny_libc/syscall.c:236
236         invoke_syscall(SYSCALL_SHM_DT, addr, 0, 0, 0, 0);
(gdb)
invoke_syscall (
    sysno=57,
    arg0=-273501978624, arg1=0,
    arg2=0,
    arg3=0,
    arg4=0)
    at tiny_libc/syscall.c:13
13          asm volatile(
(gdb)
27          return ret_value;
(gdb)
28      }
(gdb)
sys_shmpagedt (
    addr=0xffffffc052030000)
    at tiny_libc/syscall.c:237
237     }
(gdb)
main (argc=1,
    argv=0xf00010ff8)
    at test/test_project4/consensus.c:65
65          vars->magic_number = MAGIC;
(gdb)
^C
Program received signal SIGINT, Interrupt.
```

```
0x000000005ff92a80 in ?? ()
<p $scause
$2 = 8
(gdb) p $sepc
$3 = 67606
(gdb) p/x $sepc
$4 = 0x10816
(gdb) p $a7
$5 = 2
```

明明都对，却总是报错？忘记flush！

修改后还是会报错……意识到此时处于用户态，而此时返回给用户的是内核虚地址，访问会报错，应该使用一个未使用的用户态地址，分配后：

```
(2) I am selected at round 1
(12) I am selected at round 1
(4) I am selected at round 1
(14) I am selected at round 1
(6) I am selected at round 1
(7) I am selected at round 1
(9) I am selected at round 1
(13) I am selected at round 1
```

开始乱来……意识到此时应该使用该用户虚地址所对应的物理地址做mapping，而不是简单地使用 `kva2pa(shm_pages[i].uva)`：

```
    for(i=0; i< SHM_PAGE_NUM; i++){
        if(shm_pages[i].usage == USING && shm_pages[i].key == key){
            if(map_page_helper(shm_pages[i].uva, kva2pa(shm_pages[i].uva),
                        current_running[cpu_id]->pgdir)==0)
                return 0;
            local_flush_tlb_all();
            shm_pages[i].user_num++;
            return shm_pages[i].uva;
        }
    }
```

(2) 我也不知道这叫什么bug好

> 隔了三天反过来整理debug记录，我居然不知道我在说啥……反正最终代码不是像下面这样的……

惊奇地发现创建了好几个进程：

```

Breakpoint 1, do_getpid ()
    at ./kernel/sched/sched.c:325
325         return current_running[cpu_id]->pid;
<[cpu_id]->pid
$6 = 10
(gdb) c
Continuing.

Breakpoint 1, do_getpid ()
    at ./kernel/sched/sched.c:325
325         return current_running[cpu_id]->pid;
<[cpu_id]->pid
$7 = 12
<rrent_running
$8 = {
  0xffffffc05020c2b0 <pcb+1232>,
  0xffffffc050208428 <s_pid0_pcb>}
(gdb)
```

再查看共享页面的使用情况：

```
(gdb) p shm_pages
$9 = {{key = 42,
    usage = USING,
    uva = 64424644608, kva = 18446743800207581184, user_num = 1},
```

发现只有一个user，猜测是实现有误导致多次分配新页面。gdb跟踪发现barrier也被初始化多次：

```
Continuing.

Breakpoint 1, do_barrier_init (key=42,
    goal=9)
    at ./kernel/locking/lock.c:102
102         for(int i=0; i<BARRIER_NUM; i++){
<rent_running
$1 = {
  0xffffffc05020bec0 <pcb+224>,
  0xffffffc050208428 <s_pid0_pcb>}
(gdb) c
Continuing.

Breakpoint 1, do_barrier_init (key=42,
    goal=9)
    at ./kernel/locking/lock.c:102
102         for(int i=0; i<BARRIER_NUM; i++){
<rent_running
```

```
$2 = {
  0xffffffc05020bfa0 <pcb+448>,
  0xffffffc050208428 <s_pid0_pcb>}
(gdb)
```

gdb跟踪查看每次获取的共享页面虚地址：

```
Reading symbols from ./build/main...
Remote debugging using :1234
0x0000000000010000 in ?? ()
add symbol table from file "build/main" at
<hout paging--
        .text_addr = 0x50202000
(y or n) y
Reading symbols from build/main...
<ild/consensus
add symbol table from file "./build/consensus"
(y or n) y
Reading symbols from ./build/consensus...
<consensus.c:66
Breakpoint 1 at 0x1022e: file test/test_project4/consensus.c, line
66.
(gdb) c
Continuing.

Breakpoint 1, main (
    argc=14, argv=0xd)
    at test/test_project4/consensus.c:66
66          vars = (consensus_vars_t*) sys_shmpageget(SHMP_KEY);
(gdb) n
133             sys_sleep(2);
(gdb) p vars
$1 = (consensus_vars_t *) 0xf0000fff0
(gdb) c
Continuing.

Breakpoint 1, main (
    argc=1,
    argv=0xf00010ff8)
    at test/test_project4/consensus.c:66
66          vars = (consensus_vars_t*) sys_shmpageget(SHMP_KEY);
(gdb) cn
Undefined command: "cn".  Try "help".
(gdb) n
67          sys_move_cursor(1, print_location);
(gdb) p vars
$2 = (consensus_vars_t *) 0xf00021000
(gdb) c
Continuing.
```

```

Breakpoint 1, main (
    argc=2,
    argv=0xf00011ff0)
    at test/test_project4/consensus.c:66
66          vars = (consensus_vars_t*) sys_shmpageget(SHMP_KEY);
(gdb) n
67          sys_move_cursor(1, print_location);
(gdb) p vars
$3 = (consensus_vars_t *) 0x0
(gdb)
```

原本map_page实现如下:

```
    // 若pa等于0，即取消映射操作
    if(pa==0){
        pte[vpn0] = 0;
        return 1;
    }
    // 将对应实地址置为pa
    if(pte[vpn0]==0){
        set_pfn(&pte[vpn0], pa >> NORMAL_PAGE_SHIFT);
        set_attribute(
            &pte[vpn0], _PAGE_PRESENT | _PAGE_READ | _PAGE_WRITE |
                                _PAGE_EXEC | _PAGE_ACCESSED |
_PAGE_DIRTY | _PAGE_USER);
        return 1;
    }
```

意识到未考虑设置的pa已经等于给定表项的情况，也应该返回1，修改如下:

```
    // 若pa等于0，即取消映射操作
    if(pa==0){
        pte[vpn0] = 0;
        return 1;
    }
    // 将对应实地址置为pa
    else if(pte[vpn0]==0){
        set_pfn(&pte[vpn0], pa >> NORMAL_PAGE_SHIFT);
        set_attribute(
            &pte[vpn0], _PAGE_PRESENT | _PAGE_READ | _PAGE_WRITE |
                                _PAGE_EXEC | _PAGE_ACCESSED |
_PAGE_DIRTY | _PAGE_USER);
        return 1;
    }
    // 已经建立映射
    else if(else if(get_pa(pte[vpn0])==
((pa>>NORMAL_PAGE_SHIFT+_PAGE_PFN_SHIFT)<<NORMAL_PAGE_SHIFT)))
```

```
        return 1;
```

每次返回0后寻找下一个空闲虚页：

```
    for(i=0; i< SHM_PAGE_NUM; i++){
        if(shm_pages[i].usage == USING && shm_pages[i].key == key){
            while(map_page_helper(shm_pages[i].uva, kva2pa(shm_pages[i].kva),
                        current_running[cpu_id]->pgdir)==0){
                // 选取新的空闲虚页
                shm_pages[i].uva = free_user_va;
                free_user_va += PAGE_SIZE;
            }
            local_flush_tlb_all();
            shm_pages[i].user_num++;
            return shm_pages[i].uva;
        }
    }
```

实现目标效果：

```
(2) exit now
(3) I am selected at round 5
(4) I am selected at round 6
(5) I am selected at round 1
(6) I am selected at round 2
(7) I am selected at round 7
(8) I am selected at round 4
(9) I am selected at round 3
(10) I am selected at round 8
```

TASK 6 COW机制

1. 思路分析

(1) 实现

- 封装系统调用，允许用户设置哪些页面需要实施该机制；
- 将对应页面设置为只读：拉低 `_PAGE_EXEC` 和 `_PAGE_WRITE` ，下次触发例外时为之分配新的页面

（2）测试用例设计

- 初始化时对各个页面实施写时复制机制；
- 随机对若干页面进行修改，并打印信息：
  - [N]:表示 `not modified`；
  - [M]:表示 `modified`；
  - modify前后对应的物理地址；
- 复用task 3的history结构体，用于后续做比较，可以看到被modified页面都可找到被修改的数据，而未被modified的页面diff处为空。

效果示意：

```
----------------Format:[_] pa1-pa2-------------------
[M]0x52040000-0x5203f000 [M]0x52041000-0x52040000 [N]0x52041000-0x52041000
[N]0x52041000-0x52041000 [N]0x52041000-0x52041000 [N]0x52041000-0x52041000
[M]0x52042000-0x52041000 [N]0x52042000-0x52042000 [M]0x52043000-0x52042000
----------------Format: cur-data-------------------
Diff: 530-1652531652
Diff: 566-1305077752
Diff:
Diff:
Diff:
Diff:
Diff: 318-577189408
Diff:
Diff: 327-1853722797 645-1443039267 729-2111108423 955-878394481






------------------- COMMAND -------------------
> root@UCAS_OS: exec cow &
Info: excute cow successfully, pid = 2
> root@UCAS_OS:
```