Lab3 Pagetable

21307110014

1 打印页表

1.1 实现

主体代码是在vm.c中增加的`vmprint`以及用来递归的`_vmprint`辅助函数：

```
void _vmprint(pagetable_t pagetable, int level, uint64 va) {
  // there are 2^9 = 512 PTEs in a page table.
  for (int i = 0; i < 512; i++) {
    pte_t pte = pagetable[i];
    if (!(pte & PTE_V)) continue;
    uint64 child = PTE2PA(pte);
    for (int l = 2; l > level; l--) {
      printf("||   ");
    }
    if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
      // non-leaf
      printf("||idx: %d: pa: %p, flags: ----\n", i, child);
      _vmprint((pagetable_t)child, level - 1, va | ((uint64)i << PXSHIFT(level)));
    } else {
      // leaf
      char flags[5];
      flags[0] = (pte & PTE_R) ? 'r' : '-';
      flags[1] = (pte & PTE_W) ? 'w' : '-';
      flags[2] = (pte & PTE_X) ? 'x' : '-';
      flags[3] = (pte & PTE_U) ? 'u' : '-';
      flags[4] = '\0';
      printf("||idx: %d: va: %p -> pa: %p, flags: %s\n", i, va | ((uint64)i <<
PXSHIFT(level)), child, flags);
    }
  }
}

// print pagetable
void vmprint(pagetable_t pagetable) {
  printf("page table %p\n", pagetable);
  _vmprint(pagetable, 2, 0);
}
```

其中遍历页表参考`freewalk`，而需要打印的虚拟地址`va`是在遍历的同时维护（由每一个level的index组成）。

然后在def.h中加上声明，并在exec.c的`exec`里加上`if (p->pid == 1) vmprint(p->pagetable);`。

1.2 运行结果

boot时打印页表如下：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
page table 0x0000000087f24000
||idx: 0: pa: 0x0000000087f20000, flags: ----
||   ||idx: 0: pa: 0x0000000087f1f000, flags: ----
||   ||   ||idx: 0: va: 0x0000000000000000 -> pa: 0x0000000087f21000, flags: rwxu
||   ||   ||idx: 1: va: 0x0000000000001000 -> pa: 0x0000000087f1e000, flags: rwx-
||   ||   ||idx: 2: va: 0x0000000000002000 -> pa: 0x0000000087f1d000, flags: rwxu
||idx: 255: pa: 0x0000000087f23000, flags: ----
||   ||idx: 511: pa: 0x0000000087f22000, flags: ----
||   ||   ||idx: 510: va: 0x0000003fffffe000 -> pa: 0x0000000087f76000, flags: rw--
||   ||   ||idx: 511: va: 0x0000003ffffff000 -> pa: 0x0000000080007000, flags: r-x-
init: starting sh
$
```

2 独立内核页表

2.1 实现

- In `vm.h`

1. 修改`kvmpa`中`pte = walk(kernel_pagetable, va, 0);`为`pte = walk(myproc()->k_pagetable, va, 0);`
2. 增加`p_kvminit`函数对独立内核页表进行初始化（其中`va`和`pa`是`kstack`的地址映射）：

```
void p_kvminit(pagetable_t pagetable, uint64 va, uint64 pa) {
  if (pagetable == 0) return;
  mappages(pagetable, UART0, PGSIZE, UART0, PTE_R | PTE_W);
  mappages(pagetable, VIRTIO0, PGSIZE, VIRTIO0, PTE_R | PTE_W);
  mappages(pagetable, CLINT, 0x10000, CLINT, PTE_R | PTE_W);
  mappages(pagetable, PLIC, 0x400000, PLIC, PTE_R | PTE_W);
  mappages(pagetable, KERNBASE, (uint64)etext - KERNBASE, KERNBASE, PTE_R | PTE_X);
  mappages(pagetable, (uint64)etext, PHYSTOP - (uint64)etext, (uint64)etext, PTE_R |
PTE_W);
  mappages(pagetable, TRAMPOLINE, PGSIZE, (uint64)trampoline, PTE_R | PTE_X);
  mappages(pagetable, va, PGSIZE, pa, PTE_R | PTE_W);
}
```

3. 增加`p_kvmfree`函数对独立内核页表进行释放：

```
void p_kvmfree(pagetable_t pagetable, uint64 kstack) {
  if (kstack) uvmunmap(pagetable, kstack, 1, 1);
  uvmunmap(pagetable, UART0, 1, 0);
  uvmunmap(pagetable, VIRTIO0, 1, 0);
  uvmunmap(pagetable, CLINT, 0x10000 / PGSIZE, 0);
  uvmunmap(pagetable, PLIC, 0x400000 / PGSIZE, 0);
  uvmunmap(pagetable, KERNBASE, (PHYSTOP - KERNBASE) / PGSIZE, 0);
  uvmunmap(pagetable, TRAMPOLINE, 1, 0);
  uvmfree(pagetable, 0);
}
```

- In `def.h`

4. 增加第3步和第4步函数的声明，以便在`proc.c`中调用：

```
void            p_kvminit(pagetable_t, uint64, uint64);
void            p_kvmfree(pagetable_t, uint64);
```

- In `proc.h`

5. 在`struct proc`里加上成员变量`pagetable_t k_pagetable;`

- In `proc.c`

6. 在`allocproc`中加入对`p->k_pagetable`的初始化：

```
  p->k_pagetable = uvmcreate();
  if (p->k_pagetable == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  uint64 pa = (uint64)kalloc();
  if(pa == 0) {
    panic("kalloc");
  }
  uint64 va = KSTACK(0);
  p_kvminit(p->k_pagetable, va, pa);
  p->kstack = va;
```

7. 在`freeproc`中加入对`p->k_pagetable`的释放：

```
  p_kvmfree(p->k_pagetable, p->kstack);
  p->k_pagetable = 0;
```

8. 在`scheduler`中进行页表的切换：

```
// at beginning of file
extern pagetable_t kernel_pagetable;
//...
        // in scheduler
        w_satp(MAKE_SATP(p->k_pagetable));
        sfence_vma();

        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        w_satp(MAKE_SATP(kernel_pagetable));
        sfence_vma();
```

2.2 运行结果

```
$ kvmtest
kvmtest: start
test_pagetable: 1
kvmtest: OK
$
```

```
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

3 问答题

3.1 SV39标准下虚拟地址0xFFFFFFE789ABCDEF如何被转化为物理地址

可以参考如下`walk`函数：

```
// Return the address of the PTE in page table pagetable
// that corresponds to virtual address va.  If alloc!=0,
// create any required page-table pages.
//
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
pte_t *walk(pagetable_t pagetable, uint64 va, int alloc) {
  if (va >= MAXVA) panic("walk");

  for (int level = 2; level > 0; level--) {
```

```
    pte_t *pte = &pagetable[PX(level, va)];
    if (*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      if (!alloc || (pagetable = (pde_t *)kalloc()) == 0) return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

页表分210三级，拿到一个虚拟地址，我们会从高到低依次用`PX(level, va)`取出它在当前一级内的索引（页表项指针`pte`），然后`PTE2PA(*pte)`得到其下一级页表（如果存在）。最终物理地址为`PA2 << 30 | PA1 << 21 | PA0 << 12 | OFFSET`。

解析给定地址可以得到：

级别2索引: 0x19E （位30-38）
级别1索引: 0x04D （位21-29）
级别0索引: 0x0BC （位12-20）
页内偏移: 0xDEF （位0-11）

3.2 为什么SV39标准下虚拟地址的L2, L1, L0 均为9位

该标准下每个页表项`pte`大小为8字节（uint64），而页大小为4096字节（4KB）。由于一个页目录的大小必须与页的大小等大，我们可以由此计算出一个页目录能记录$\frac{4096}{8} = 512 = 2^9$个页表项，而每级都是在一个页目录上索引，因此每级索引均为9位。