Lab3-page-tables

Speed up systems calls

实验目的

- 通过优化系统调用，减少内核切换，提高 xv6 操作系统的性能
- 了解映射只读页，存储当前进程的 PID，并通过共享页面加速其他可能的系统调用的机制

实验步骤

1. 原有的`user/ulib.c`中已经定义好了`ugetpid()`函数,通过`USYSCALL`映射来获取进程的`PID`:

```
#ifdef LAB_PGTBL
int
ugetpid(void)
{
  struct usyscall *u = (struct usyscall *)USYSCALL;
  return u->pid;
}
#endif
```

2. `kernel/memlayout.h`中定义的虚拟地址`USYSCALL`，映射了一个只读页，存储了一个`struct usyscall`,并将器初始化为存储当前进程的`PID`

```
#define TRAMPOLINE (MAXVA - PGSIZE)

#define TRAPFRAME (TRAMPOLINE - PGSIZE)
#ifdef LAB_PGTBL
#define USYSCALL (TRAPFRAME - PGSIZE)

struct usyscall {
  int pid;  // Process ID
};
#endif
```

`TRAMPOLINE`，指向内核的跳板代码页面。跳板代码用于在从用户态切换到内核态时保存和恢复 CPU 寄存器。`MAXVA`表示最大的虚拟地址，`PGSIZE`表示页面大小。因此，`TRAMPOLINE`位于虚拟地址空间的最高页面。

`TRAPFRAME`，指向保存当前进程陷阱帧（trapframe）的页面。陷阱帧包含处理器的寄存器状态，当发生陷阱（如系统调用或中断）时，处理器的状态会被保存到陷阱帧中。`TRAPFRAME`位于

`TRAMPOLINE`的下一个页面。

`USYSCALL`，指向用户态系统调用共享页面。在这个页面中存储了`struct usyscall`结构，用于在用户空间和内核空间之间共享系统调用相关的信息，如当前进程的PID。`USYSCALL`位于`TRAPFRAME`的下一个页面。

`struct usyscall`结构体用于存储系统调用相关的信息，即当前进程的PID。

3. 在kernel/proc.c/proc_pagetable()中添加映射：

```
pagetable_t
proc_pagetable(struct proc *p)
{
  pagetable_t pagetable;

  // An empty page table.  创建空页表
  pagetable = uvmcreate();
  if(pagetable == 0)
    return 0;

  // map the trampoline code (for system call return)
  // at the highest user virtual address.
  // only the supervisor uses it, on the way
  // to/from user space, so not PTE_U.  映射跳板代码页
  if(mappages(pagetable, TRAMPOLINE, PGSIZE,
              (uint64)trampoline, PTE_R | PTE_X) < 0){
    uvmfree(pagetable, 0);
    return 0;
  }

  // map the trapframe page just below the trampoline page, for
  // trampoline.S.  映射陷阱帧页
  if(mappages(pagetable, TRAPFRAME, PGSIZE,
              (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }

  // 添加映射 映射用户系统调用页
  if (mappages(pagetable, USYSCALL, PGSIZE, (uint64)(p->usys), PTE_R | PTE_U) < 0)
{
        // 若映射失败，恢复上述页
        uvmunmap(pagetable, TRAMPOLINE, 1, 0);
        uvmunmap(pagetable, TRAPFRAME, 1, 0);
        uvmfree(pagetable, 0);
        return 0;

}
  return pagetable;
}
```

```
// 同时释放时也要解除这样的映射
void proc_freepagetable(pagetable_t pagetable, uint64 sz){
uvmunmap(pagetable, TRAMPOLINE, 1, 0);
uvmunmap(pagetable, TRAPFRAME, 1, 0);
uvmunmap(pagetable, USYSCALL, 1, 0);
uvmfree(pagetable, sz);
}
```

需要在`kernel/proc.h`的`struct proc`中定义物理存储空间的指针：

```
struct proc {
 struct spinlock lock;

 // p->lock must be held when using these:
 enum procstate state; // Process state
 void *chan;           // If non-zero, sleeping on chan
 int killed;           // If non-zero, have been killed
 int xstate;           // Exit status to be returned to parent's wait
 int pid;              // Process ID

 // wait_lock must be held when using this:
 struct proc *parent; // Parent process

 // these are private to the process, so p->lock need not be held.
 uint64 kstack;               // Virtual address of kernel stack
 uint64 sz;                   // Size of process memory (bytes)
 pagetable_t pagetable;       // User page table
 struct trapframe *trapframe; // data page for trampoline.S
 struct context context;      // swtch() here to run process
 struct file *ofile[NOFILE];  // Open files
 struct inode *cwd;           // Current directory
 char name[16];               // Process name (debugging)
 // 添加物理空间指针
 struct usyscall *usys; // 物理存储空间
};
```

4. 在`allocproc()`函数中分配一个新页面，并将其映射到`USYSCALL`地址,还要将`PID`移动至共享页中

```
static struct proc*
allocproc(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = allocpid();
  p->state = USED;

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  // 将PID移动到共享页中 begin

  if ((p->usys = (struct usyscall *)kalloc()) == 0) {
        freeproc(p);
        release(&p->lock);
        return 0;
  }
  memmove(p->usys, &p->pid, sizeof(int));// 将pid移动至共享页中
// end

// An empty user page table.
 p->pagetable = proc_pagetable(p);
 if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
}

// Set up new context to start executing at forkret,
 // which returns to user space.
 memset(&p->context, 0, sizeof(p->context));
 p->context.ra = (uint64)forkret;
 p->context.sp = p->kstack + PGSIZE;

return p;
```

```
}
```

5. 在freeproc()中补充释放该页面的代码

```
static void
freeproc(struct proc *p)
{
// p->usys begin
if (p->usys) {
  kfree((void *)p->usys);
}
p->usys = 0;

// end

if(p->trapframe)
kfree((void*)p->trapframe);
p->trapframe = 0;
if(p->pagetable)
proc_freepagetable(p->pagetable, p->sz);
p->pagetable = 0;
p->sz = 0;
p->pid = 0;
p->parent = 0;
p->name[0] = 0;
p->chan = 0;
p->killed = 0;
p->xstate = 0;
p->state = UNUSED;
}
```

6. 运行 `pgtbltest` 测试

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
```

运行make grade

```
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
```

实验中遇到的困难和解决办法

运行报错`kernel/proc.c:136:9: error: ‘struct proc’ has no member named ‘usys’`

原因是在`struct proc`中没有定义该成员。在上述步骤3中，在`kernel/proc.h`的`struct proc`中定义物理存储空间的指针即可解决该问题

实验心得

在实现过程中，最大的挑战是确保在进程创建和销毁时正确管理内存分配和释放。这需要我们在`allocproc()`和`freeproc()`函数中精细地操作内存，防止内存泄漏或非法访问。当我第一次尝试映射用户系统调用共享页时，遇到了结构体成员未定义的错误，这让我意识到对数据结构的完整性检查和初始化的重要性。通过这次实验，我不仅学会了具体的技术实现，更重要的是掌握了在面对复杂系统时，如何逐步定位问题和解决问题的能力。

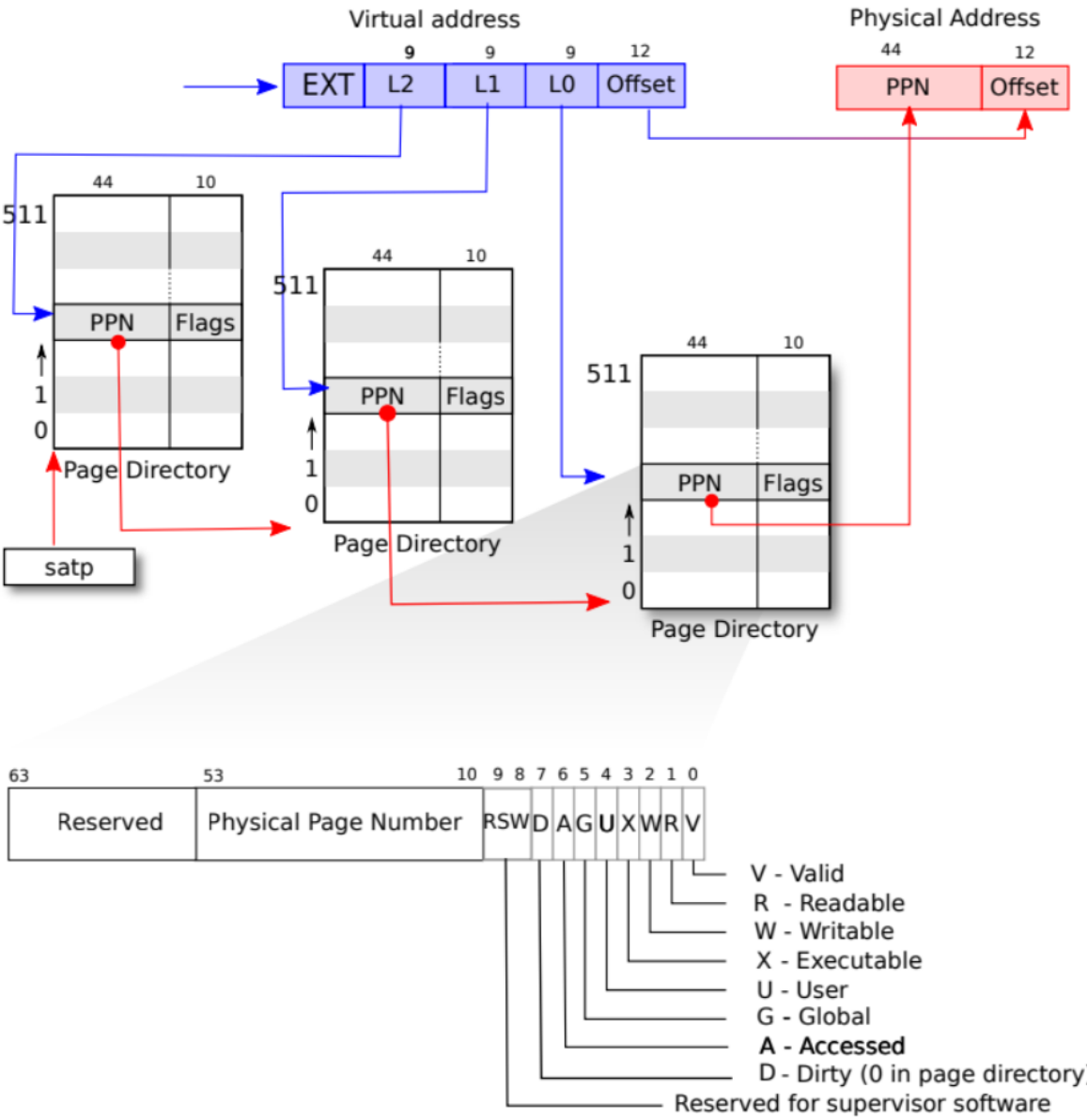
Print a page table

实验目的

- 熟悉 RISC-V 页表的结构和实现
- 掌握页表项（PTE）的解析和输出
- 实现页表的调试功能
- 理解XV6的三级页表结构

实验步骤

1. 理解XV6的三级页表结构



- 一级页表（`L1 Page Table`）：也称为页全局目录（Page Global Directory，PGD），存放二级页表（L2 Page Table）的基址。
- 二级页表（`L2 Page Table`）：也称为页目录（Page Directory），存放三级页表（L3 Page Table）的基址。
- 三级页表（`L3 Page Table`）：存放实际的页表项，用于映射虚拟地址到物理地址。

每个页表有512个页表项，每个页表项包含了一个44位的物理页码和一些标志位，标志位如下：

- PTE_V (Valid Bit)：表示该页表项是否有效，即是否有映射关系。
- PTE_R (Read Bit)：表示是否可读。
- PTE_W (Write Bit)：表示是否可写。

- PTE_X (Execute Bit)：表示是否可执行。

标志位组合情况：

- **PTE_V = 0**：
  - 虚拟地址无效，无需查找下一级页表，直接认为该地址无效。
- **PTE_V = 1, PTE_R = 0, PTE_W = 0, PTE_X = 0**：
  - 虚拟地址有效，但没有读、写或执行权限。
  - 这通常表示该地址是一级页表映射。
- **PTE_V = 1, PTE_R = 1 或 PTE_W = 1 或 PTE_X = 1**：
  - 虚拟地址有效，并且有读、写或执行权限中的至少一种。
  - 这通常表示需要继续查找下一级页表，即二级页表。

2. 在 `kernel/vm.c` 中定义 `vmprint()` 函数，该函数接收一个 `pagetable_t` 类型的参数，并以指定格式打印页表的内容。

```
void vmprint(pagetable_t pgtbl, int level)
{
    // there are 2^9 = 512 PTEs in a page table.
    for (int i = 0; i < 512; i++) {

        pte_t pte = pgtbl[i];
        if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
            // this PTE points to a lower-level page table.
            uint64 child = PTE2PA(pte);
            for (int j = 0; j < level; j++) {
                printf("..");
            }
            printf("%d: pte %p pa %p\n", i, pte, child);
            vmprint((pagetable_t)child, level + 1);
        }
        else if (pte & PTE_V) {
            uint64 child = PTE2PA(pte);
            for (int j = 0; j < level; j++) {
                printf("..");
            }
            printf("%d: pte %p pa %p\n", i, pte, child);
        }
    }
}
```

3. **修改 `kernel/defs.h`，声明 `vmprint` 函数原型**：确保在其他文件中可以调用该函数。

```
void vmprint(pagetable_t, int);
```

4. **在 `exec.c` 中调用 `vmprint()`**：在 `if(p->pid==1)` 条件下，调用 `vmprint(p->pagetable)` 打印第一个进程的页表。

```
  // Commit to the user image.
  oldpagetable = p->pagetable;
  p->pagetable = pagetable;
  p->sz = sz;
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer
  proc_freepagetable(oldpagetable, oldsz);

 // 添加vmprint()调用
  if (p->pid == 1) {
    printf("page table %p\n", p->pagetable);
    vmprint(p->pagetable, 1);
  }

  return argc; // this ends up in a0, the first argument to main(argc, argv)

 bad:
  if(pagetable)
    proc_freepagetable(pagetable, sz);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;  }
```

5. 测试

   启动qemu即可看到自动调用了vmprint()函数

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6b000
..0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
....0: pte 0x0000000021fd9801 pa 0x0000000087f66000
......0: pte 0x0000000021fda01b pa 0x0000000087f68000
......1: pte 0x0000000021fd9417 pa 0x0000000087f65000
......2: pte 0x0000000021fd9007 pa 0x0000000087f64000
......3: pte 0x0000000021fd8c17 pa 0x0000000087f63000
..255: pte 0x0000000021fda801 pa 0x0000000087f6a000
....511: pte 0x0000000021fda401 pa 0x0000000087f69000
......509: pte 0x0000000021fdcc13 pa 0x0000000087f73000
......510: pte 0x0000000021fdd007 pa 0x0000000087f74000
......511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
$
```

运行make grade

```
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (1.0s)
```

实验中遇到的困难和解决办法

vmprint递归函数遇到第三级页表时停止递归，如何判断第三级页表和低级页表是个难点

判断代码如下：

```
if ((pte & PTE_V) && (pte & (PTE_R | PTE_W | PTE_X)) == 0) {
    // this PTE points to a lower-level page table.
    uint64 child = PTE2PA(pte);
    for (int j = 0; j < level; j++) {
        printf("..");
    }
    printf("%d: pte %p pa %p\n", i, pte, child);
    vmprint((pagetable_t)child, level + 1);
} else if (pte & PTE_V) {
    uint64 child = PTE2PA(pte);
    for (int j = 0; j < level; j++) {
        printf("..");
    }
    printf("%d: pte %p pa %p\n", i, pte, child);
}
```

- **判断低级页表的条件**:
  - `(pte & PTE_V)`：首先检查 PTE 是否有效。
  - `(pte & (PTE_R | PTE_W | PTE_X)) == 0`：然后检查 PTE 是否没有读、写或执行权限。如果没有任何权限，则表示该 PTE 指向一个低级页表。
- **判断三级页表的条件**:
  - `else if (pte & PTE_V)`：如果 PTE 有效，并且有读、写或执行权限中的至少一个，则表示该 PTE 指向一个实际的物理页。

实验心得

在完成 `vmprint()` 函数的过程中，我最深刻的体会是对 RISC-V 页表结构的理解和掌握。起初，我对页表的多级结构和各个页表项（PTE）中的标志位感到十分困惑。通过不断查阅资料和调试代码，我逐渐理解了这些标志位的实际作用。特别是在实现 `vmprint()` 函数时，如何区分不同级别的页表项是一个关键点，需要通过检查有效位（PTE_V）和权限位（PTE_R、PTE_W、PTE_X），来判断页表项是指向下一级页表还是指向实际的物理页。

Detect which pages have been accessed

实验目的

- 在xv6操作系统中添加一个新功能，能够检测并报告哪些内存页被访问过
- 实现一个名为 `pgaccess()` 的系统调用，该调用将检查给定范围内的内存页，并将访问结果报告给用户空间。

实验步骤

1. 阅读`user/pgtlbtest.c`中的`pgaccess_test()`函数，了解`pgaccess`的使用方法。
2. 在`kernel/riscv.h`中定义PTE_A（访问位）

```
#define PTE_A (1L << 6) // bit of access
```

3. 在`kernel/sysproc.c`中实现`sys_pgaccess()`函数。

```c
#define PGACCESS_MAX_PAGE 32
int sys_pgaccess(void)
{
    // lab pgtbl: your code here.
    uint64 va, buf;
    int pgnum;

    // 解析参数
    argaddr(0, &va);
    argint(1, &pgnum);
    argaddr(2, &buf);

    if (pgnum > PGACCESS_MAX_PAGE)
        pgnum = PGACCESS_MAX_PAGE;

    struct proc *p = myproc();
    if (!p) {
        return -1;
    }

    pagetable_t pgtbl = p->pagetable;
    if (!pgtbl) {
        return -1;
    }

    uint64 mask = 0; // 位掩码
    for (int i = 0; i < pgnum; i++) {
        pte_t *pte = walk(pgtbl, va + i * PGSIZE, 0); // 访问PTE，检查
        if (*pte & PTE_A) {
            *pte &= (~PTE_A); // 复位
            mask |= (1 << i); // 标注第i个页是否被访问过
        }
    }

    // 复制到用户栈区
    copyout(p->pagetable, buf, (char *)&mask, sizeof(mask));

    return 0;
}
```

`sys_pgaccess` 用于检测和报告指定范围内内存页访问情况的系统调用实现。接收三个参数：起始虚拟地址、页数以及存储结果的用户地址。函数通过检查页表项的访问位，记录哪些页被访问过，并将结果以位掩码的形式存储在用户提供的缓冲区中。同时，函数会重置访问位，以便下次调用时能够检测新的访问情况。

4. 测试

    1. 运行 `pgtbltest`

```
$ pgtbltest
ugetpid_test starting
ugetpid_test: OK
pgaccess_test starting
pgaccess_test: OK
pgtbltest: all tests succeeded
```

2. 运行 `make grade`

```
== Test pgtbltest ==
$ make qemu-gdb
(3.6s)
== Test   pgtbltest: ugetpid ==
  pgtbltest: ugetpid: OK
== Test   pgtbltest: pgaccess ==
  pgtbltest: pgaccess: OK
```

实验中的注意点

调用 `walk()` 函数时可能找不到对应的页表项，返回空值。因此需要在每次调用 `walk()` 后检查返回值，如果为空则继续下一页的检查。并且记得要复位 `PTE_A` ，以便下一次调用 `pgaccess()` 时可以检测新的访问情况。代码如下：

```
pte_t *pte = walk(pgtbl, va + i * PGSIZE, 0);
if (pte && (*pte & PTE_A)) {
    *pte &= (~PTE_A);
    mask |= (1 << i);
}
```

实验心得

在这次实验中，最令我感触深刻的是对页表访问位（PTE_A）的处理。在实现 `sys_pgaccess` 函数时，我一开始忽略了复位访问位这一操作，导致每次检查的结果都是一样的，无法反映最新的页面访问情况。这个问题让我意识到，操作系统中每一个微小的位操作都会对整体功能产生重要影响。通过反复查询信息，我终于理解了PTE_A的作用，并正确实现了其复位操作。

总结

本实验旨在通过优化系统调用、实现页表打印、检测页面访问等实际操作，加深对页表机制的理解和应用。具体目标包括优化系统调用性能、理解和实现RISC-V页表结构、实现系统调用来检测页面访问等。

- **Speed up systems calls**
  - **目的**：
    - 通过优化系统调用，减少内核切换，提高Xv6操作系统的性能。
    - 了解映射只读页，存储当前进程的PID，并通过共享页面加速其他系统调用的机制。
  - **心得**： 通过优化系统调用，减少了内核与用户空间切换的开销，提高了系统性能。过程中遇到内存管理和数据结构初始化的问题，通过逐步调试和检查解决了这些问题，提升了对系统调用和内存管理的理解。

- **Print a page table**
  - **目的**：
    - 熟悉RISC-V页表的结构和实现。
    - 掌握页表项（PTE）的解析和输出，理解XV6的三级页表结构。
  - **心得**： 实现 `vmprint()` 函数过程中，理解了RISC-V页表结构和标志位的作用。通过递归解析和打印页表项，掌握了页表的多级结构和页表项的具体含义。这一过程增强了对页表和内存管理的理解。

- **Detect which pages have been accessed**
  - **目的**：
    - 在Xv6操作系统中添加一个新功能，检测并报告哪些内存页被访问过。
    - 实现 `pgaccess()` 系统调用，检查给定范围内的内存页，并将访问结果报告给用户空间。
  - **心得**： 在实现 `sys_pgaccess` 函数时，遇到了访问位复位的问题。通过仔细理解和实现访问位的处理，成功检测并报告了页面访问情况。这个实验让我理解了页表的实际应用和细节处理的重要性。