xv6-rust 设计文档

2022全国大学生计算机系统能力大赛操作系统设计赛

剑指OS

彭泽杰、 刘辰、 董慧勇

一、设计简介

本项目是在去年比赛实现的 xv6-rust 基础之上进行二次开发，主要有两个方面：一是增加新的功能，二是修改完善现有代码的一些 bug，具体如下所示：

- 实现进程间的通信功能：
  - 信号量
  - 消息队列
  - 共享内存
  - 管道
- 实现线程协程：
  - 内核级线程
  - 用户级线程(协程)
- 简单封装运行库
- 功能改进：
  - 页面延迟分配
  - 优先级调度
- 修改重构：
  - 重写进程结构体及其锁的形式，改用大粒度锁
  - 修改原有管道的 bug
  - 修改内存管理的 bug

以上便是初赛阶段已完成的一些内容，还有一些目前正在做待完成

- 模拟中断实现信号机制
- 重新组织进程结构体，采用红黑树并实现 CFS 调度算法
- 写时复制、mmap 等系统调用。

二、设计实现

下面便详细说说以上的一些功能的设计实现，首先是进程间的通信功能

1 进程间的通信

进程间要进行通信，从通信的内容来讲大致分为两类，一种是信号往来，比如说信号、信号量机制，可以用来通知某些时间的发生，某些共享资源的使用情况。

另外一种是数据往来，数据的往来需要较多的空间，所以一般有两种方式：

- **一是直接通过内存来传递数据消息**，比如说共享内存，管道，消息队列，本质上都是通过“一块内存”来传递消息，只不过访问这一块内存的方式有所不同罢了。
- 二是直接不同进程读写相同的文件来进行通信，这里的文件指的是普通磁盘上的文件，而非广义上的"一切皆文件"，只不过好像读写文件太过普遍的原因，一般说进程间的通信并未加上这种方式。

下面便详细讲述我们小组实现的几种进程间的通信方式

1.1 共享内存

共享内存应该是不同进程之间通信最直接，最快速的一种通信方式了，不同进程共享一块内存，对这块内存进行读写操作便可实现通信。

每个进程都有自己的页表，通过页表将虚拟内存映射到实际的物理内存。共享内存指的是不同进程的虚拟内存通过各自的页表映射到相同的物理内存。

而要共享物理内存实际意思为：将不同进程各自的虚拟内存映射到相同的物理内存，具体来讲就是分配一块空闲的物理内存 P，然后 A 进程找一块空闲的虚拟内存 V_a，添加页表项使得 V_a 映射到 P，B 进程找一块空闲的虚拟内存 V_b 也映射到 P，这样便实现了 A B 进程共享物理内存 P。

这里涉及了几个关键问题：

- **每个进程的空闲虚拟内存如何找寻**，原本的 $xv6$ 里面没有虚拟内存管理，所以用作共享内存的这部分虚拟内存需要我们自己实现管理
- **空闲的物理内存分配**，这一部分倒是不用我们操心，原本 $xv6$ 物理内存管理实现了物理内存分配和释放，向外提供了 $kalloc$ 和 $free$ 接口。
- **物理内存到虚拟内存的映射关系**，实际上就是填写相应的页表项，这部分 $xv6$ 也提供了相应的接口 $mappages$

共享区管理

所以其实这里需要我们做得便是从整个虚拟地址空间中划分一个区域作为共享区，然后对这块区域做相应的管理，实现如下：

定义两个常量，将内核下面的 $128M$ 当作共享区：

```
pub const MAP_END: usize = KERNEL_BASE - PGSIZE;
pub const MAP_START: usize = MAP_END - 128*1024*1024;
```

每次映射的共享内存大小最大为 4M，如果一页大小为 4K，则每次最多映射 1024 页

```
pub const SHARE_MEM_AREA_SIZE: usize = 4*1024*1024;
pub const SHARE_MEM_MAP_PAGES: usize = SHARE_MEM_AREA_SIZE/(PGSIZE);
```

然后便是对这 128M 的区域做相应的管理，**对于内存的管理，一般两种方式，位图和空闲链表**，空闲链表的方法应该更好，不过鉴于当前 xv6 的内存管理方式，先使用位图进行管理，后续对于底层内存管理的方式重构之后再使用链表的形式。

使用 1bit 表征一页的使用情况，所以位图大小为：

```
pub const SHARE_MEM_BIT_MAP_SIZE: usize = SHARE_MEM_MAP_PAGES/8+1;
```

位图实际上就是特定大小的字节数组，定义位图数据结构如下：

```
pub struct BitMap{
    bitmap: [u8; SHARE_MEM_BIT_MAP_SIZE]
}
```

每个进程都有自己管理共享区的位图结构体，所以在进程结构体中添加该信息：

```
pub struct task_struct{
    //...
    pub sharemem_bitmap: *mut BitMap;
    //...
}
```

接下来便是一些关于位图的一些操作：

```
pub fn page_to_addr(page: usize) -> usize //将共享区中的第page页转换为虚拟地址
pub fn addr_to_page(addr: usize) -> usize //将共享区中的虚拟地址转换为第xx页

pub fn get_bit(&self, page: usize) -> usize //获取第page页的使用情况
pub fn set_bit(&self, page: usize, bit: usize) //设置第page页的使用情况为bit

//根据位图寻找连续npages页都空闲的区域，返回其地址
pub fn get_unmapped_addr(&mut self, npages: usize) -> usize
```

共享内存实现

我们具体实现共享内存的方式为：**设计一个共享内存结构体 `ShareMem`，使用它来记录映射信息比如映射的物理地址，映射了多少页，该内存的属性等等。不同进程想要通过共享内存通信，就要先获取相同的 `ShareMem`，获取到里面的物理地址，然后从自己的共享区找一块空闲虚拟内存映射到该物理地址指向的物理内存，如此便实现了不同进程的虚拟内存映射到相同的物理内存，实现内存共享。**

`ShareMem` 定义如下：

```
struct ShareMem{
    id: usize,
    used: bool,
    //vaddr: usize,
    paddr: usize,
    npages: usize,
    flags: usize,
    links: usize,
    name: [u8; MAX_NAME_LEN],

    lock: Spinlock<()>
}
```

各元素意义如下：

- `id` 为该 `ShareMem` 的唯一标识，通过它来表示不同进程获取的是否同一 `ShareMem`
- `used` 该结构体是否使用
- `paddr` 记录映射的物理地址
- `npages`：连续映射了多少页
- `flags`：该片内存的属性
- `links`：该片内存被映射了多少次

- name：该 ShareMem 的名字，主要用来找到相应的 ShareMem
- lock：共享内存是共享资源，避免竞争需要用个锁来保护

ShareMem 有三重要接口：

```
pub fn free(&mut self)
```

free 函数用来释放该结构体，主要有两件事：释放物理内存，将该结构体的各属性字段"清零"

```
pub fn map(&mut self, shmaddr: usize, flags: usize) -> Option<usize>{
    if shmaddr == 0 {   //如果shmaddr==0
        vaddr = get_unmapped_addr();  //则表示内核(我们)自个儿在共享区里面找一块空闲的虚拟内存

        if self.paddr == 0{   //如果该ShareMem物理地址为空，则说明第一次映射
            paddr = kalloc();   //调用物理内存分配函数分配一块物理内存
            pagetable.map(vaddr, paddr, PGSIZE, perm)  //虚拟内存映射到物理内存
        }
    } else {
        //待完善
    }
    self.links += 1;  //该共享内存的映射次数加1

    Some(self.vaddr)
}

pub fn unmap(&mut self) {
    //取消该进程的共享内存的映射
    //1、清除相应的页表项
    //2、相应的位图清零
    //3、links--
}
```

上述即为 `ShareMem` 的三个重要接口，对外开放 map 和 unmap 函数，也是后面要添加的系统调用，**map 和 unmap 一般说来只是建立和取消映射关系，并不实际进行分配和释放映射的物理内存。**

一个 ShareMem 管理一个共享内存的映射，如果有多个进程在不同的共享内存上通信，则需要多个 ShareMem 来管理，为此再顶一个结构体 ShareMemManager：

```
pub struct ShareMemManager{
    shares: [ShareMem; SHARE_MEM_TYPE_NR],
    lock: Spinlock<()>
}
```

ShareMemManager 实为 ShareMem 数组，我们使用 ShareMemManager 来实现 ShareMem 的分配和回收，因此有三重要接口：

```
pub fn alloc(&mut self, name: [u8; NAME_LEN], size: usize) -> Option<usize>
```

该函数用来分配一个 ShareMem 结构体，实际操作就是遍历 shares 数组，根据 used 字段找一个空闲 ShareMem，然后根据 name 和 size 字段初始化该 ShareMem 结构体

```
pub fn get(&mut self, name: [u8; NAME_LEN], size: usize, flags: uszie) ->
Option<usize>
```

该函数用来获取 ShareMem 结构体，此函数是 alloc 的封装，如果 flags 指示的是新建(分配)一个 ShareMem 的话就直接调用 alloc 来分配一个空闲的，这一般是第一次映射。一般情况下，根据 name 字段寻找现有的 ShareMem，然后返回其 ID 值

```
pub fn put(&mut self, id: usize)
```

放下 ID 值为 id 的 ShareMem，主要调用 ShareMem.free 函数来释放物理内存，各属性字段清零

添加系统调用

对于共享内存，对用户提供了 4 各系统调用：

- get 获取一个 ShareMem，返回其 ID
- map 实现共享区的虚拟内存到物理内存的映射，返回起始虚拟地址
- unmap 取消上述映射关系
- put 回收该 ShareMem，释放物理内存

1.2 管道

管道是一种特殊文件，本质上是一块内存区域，一端只允许写，一端只允许读，数据的流向只能是写端到读端，感觉就像管道一般，所以将这种文件命名为管道文件。原有 xv6 有匿名管道的基本实现，但是有 bug，我们所做的工作便是修改原有 bug，并在其基础上实现有名管道。

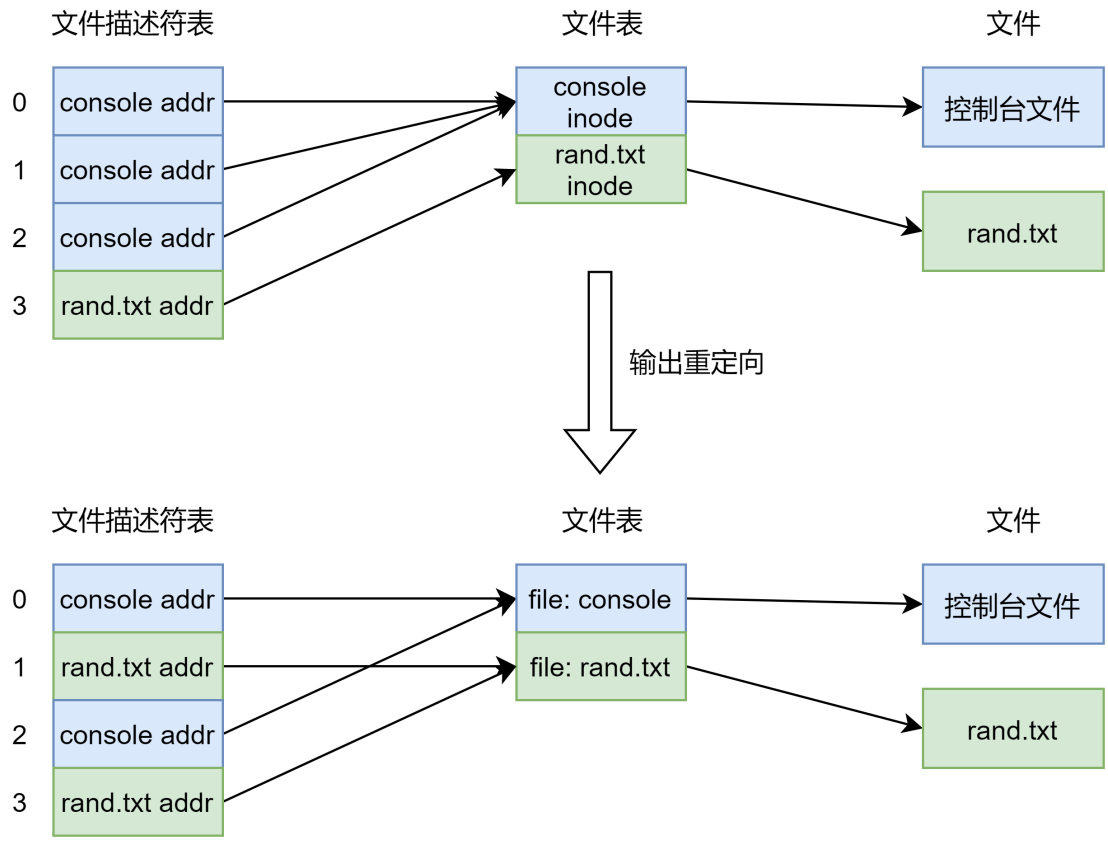
重写匿名管道

匿名管道和有名管道的本质都是一块内存区域，只不过对其读写的方式不同：

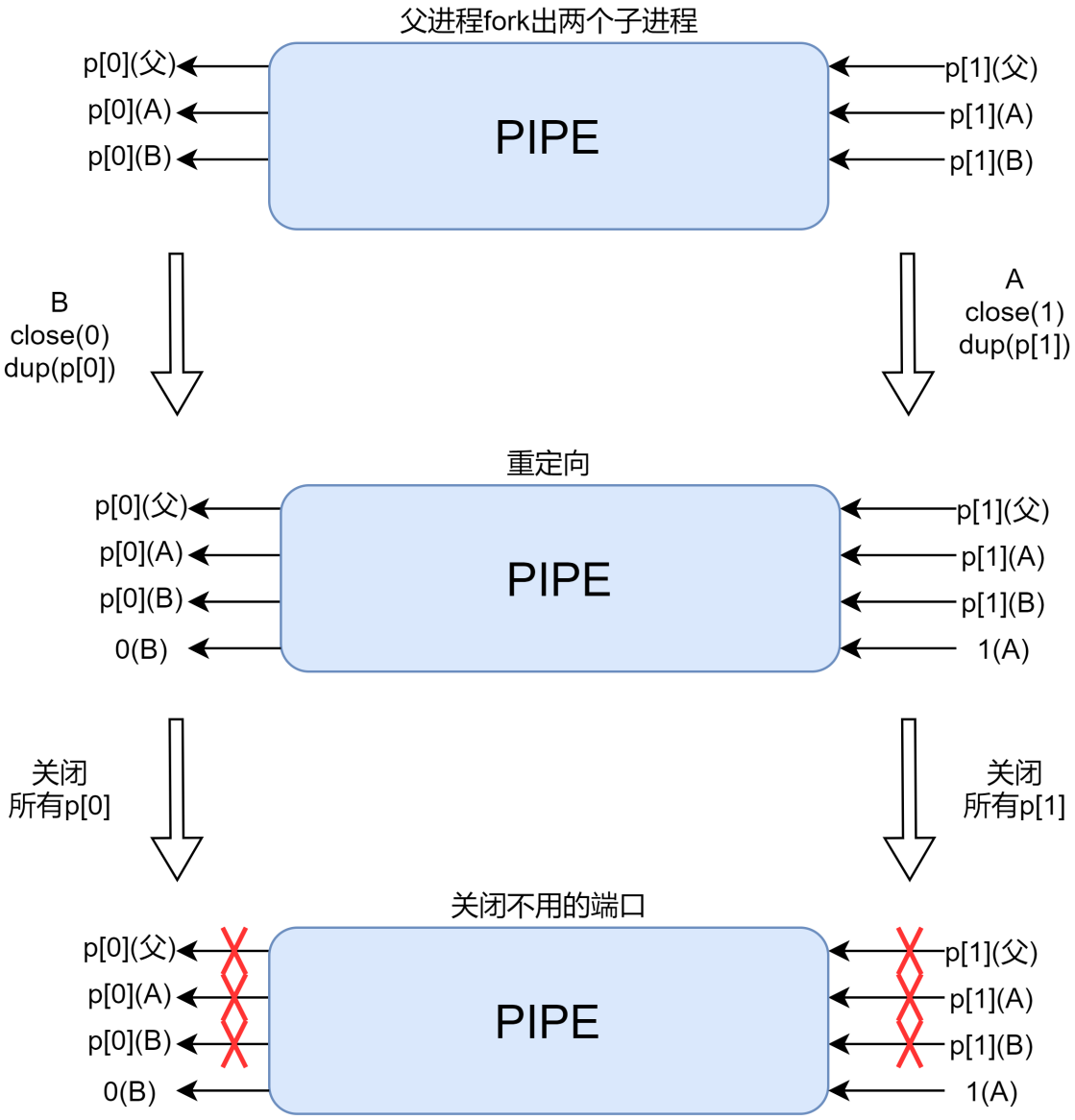
对于匿名管道，基本步骤是：

1. 为其两端分配两个文件结构体，并关联到进程的两个文件描述符
2. 读写时一般需要重定向：先关闭 0、1 标准输入输出，使用 dup 使上述的两文件结构体关联到 0、1 文件描述符，然后父子进程便可以通过标准输入输出来通信

重定向示意图如下：



通过匿名管道进行读写的示意图如下：



关于匿名管道的实现基本沿用原有项目，只是对原来的数据结构设计进行了简化，原来是管道数据结构包裹在 Spinlock 里面，而我们小组当时调试时总是有竞争锁的问题出现，于是直接简化，将管道与锁分离，与管道的各属性一起组成 Pipe 结构体：

```
pub struct Pipe {
    data: [u8; PIPE_SIZE],
    nread: usize,
    nwrite: usize,
    read_open: bool,
    write_open: bool,
    pub pipe_lock: Spinlock<()>
}
```

对于 Pipe 相应的函数基本与原有项目的逻辑相同，只是修改了其中的一些 bug，这里不再详细说明，下面介绍有名管道的实现。

实现有名管道

而我们现在就是要在这基础之上实现有名管道 `fifo`，如果按照“标准”来实现的话，`fifo` 是一种特殊文件，能够安装到目录里面，`ls` 命令能够识别打印信息。我们可以定义一个结构体 `fifo`，它是 `pipe` 结构体的封装，有两读写方法，`fifowrite`、`fiforead`，分别是 `pipewrite`、`piperead` 的封装。再改写一下文件系统部分，让其能够识别 `FIFO`，那么读写有名管道的函数调用顺序可以为 $write \rightarrow filewrite \rightarrow fifowrite \rightarrow pipewrite$ 。

不过要将 $fifo$ 当作文件安装在文件系统中，需要对底层的文件系统进行改写，私以为理论上逻辑上是没什么问题的，但是这种我没去实验，待后续完善。为了简单我直接将 `fifowrite`、`fiforead` 对用户开放为系统调用，所以调用方式为直接为 $fifowrite \rightarrow pipewrite$ 。

因此如何实现有名管道也就明了了，定义的 `Fifo_t` 结构体如下：

```
pub struct Fifo_t {
    pub pipe: Option<*mut Pipe>,  //复用匿名管道Pipe
    pub name: [u8; NAME_LEN],     //名字
    pub used: bool,               //该结构体是否使用
    pub ID:   usize               //ID值
}
```

其相应的方法也就是 `Pipe` 的封装：

```
pub fn read(& self, addr: usize, len: usize) -> Result<usize, &'static str> {
    let pipe = unsafe{&mut *self.pipe.unwrap()};
    pipe.read(addr, len)
    //...
}

pub fn write(&self, addr: usize, len: usize) -> Result<usize, &'static str> {
    let pipe = unsafe{&mut *self.pipe.unwrap()};
    pipe.write(addr, len)
    //...
}

pub fn close(&self){
    let pipe = unsafe{&mut *self.pipe.unwrap()};
    pipe.close(true);
    pipe.close(false);
    drop(self);
    //...
}
```

对于 `fifo` 结构体需要进行管理，分配回收等等，所以定义了如下 `FifoTable` 结构体：

```
pub struct FifoTable{
    fifos: [Fifo_t; N_FIFOS],
    fifos_lock: Spinlock<()>,
}
```

本质上就是一个 `fifo` 数组，只不过多配了一把锁，因为 `fifos` 是共享资源，为避免竞争需要用锁来保护。

`FifoTable` 重要的方法如下：

```
//分配一个空闲的fifo结构体，
pub fn alloc(&mut self, s: [u8; NAME_LEN]) -> Option<usize>
//获取一个现有的fifo结构体
pub fn get(&mut self, name: [u8; NAME_LEN]) -> Option<usize>
//放下回收一个fifo结构体
pub fn put(&mut self, id: usize) -> Option<usize>
```

添加系统调用

有名管道对用户提供了 5 个系统调用：

```
//调用alloc创建一个fifo
pub fn sys_mkfifo(&self) -> SysResult
//调用get获取一个fifo
pub fn sys_fifo_get(&self) -> SysResult
//调用put回收一个fifo
pub fn sys_fifo_put(&self) -> SysResult
//调用fifo read进行读取
pub fn sys_fifo_read(&self) -> SysResult
//调用fifo_write进行写
pub fn sys_fifo_write(&self) -> SysResult
```

1.3 消息队列

共享内存是通过直接读写内存进行通信，管道是通过对这片内存进行包装，像外提供读写接口来通信，而消息队列呢？与管道一样，是对用于数据往来的内存进行包装，只不过实现的方式有所不同。

数据结构设计

`消息 队列`，从名字上来看是要用队列来实现，基本思想为：**发送为将消息包装成结点，插入到消息队列，接收方从消息队列中取出消息结点获取消息。**

因此设计了两种数据结构：`msgque` 和 `msg`

```
//消息结构体
struct msg {
    data: [u8; MSG_LEN],     //实际的数据
    dataID: usize,           //数据长度
    flags: usize,            //属性

    next: *mut msg           //下一个msg
}
```

这是定义的消息结构体，也是消息队列中的一个结点，相应的方法有：

```
pub fn new() //新建一个msg
pub fn free()  //释放这个消息，释放内存
```

```
//消息队列结构体
struct msgque {
    pmsgHead: *mut msg,    //头节点
    pmsgTail: *mut msg,    //尾结点
    id: usize,             //该消息队列结构体ID
    name: [u8; 16],        //名字
    used: bool,            //是否使用
    lock: Spinlock<()>,    //锁
}
```

通过消息队列进行通信的进程都要获取相同的 `msgque`，使用 `id` 值来区分，头节点尾结点尾分别为 `msg` 队列的头尾，其他属性基本同前面的 `IPC` 控制结构体，不再赘述，`msgque` 较为重要的方法有：

```
pub fn write(&mut self, addr: usize, len: usize){
    pmsg = msg::new();   //新建一个msg
    memcopy(pmsg->data, addr, len);  //将数据从用户拷贝到内核
    self.insert(pmsg);  //将pmsg插入到msg队列
}

pub fn read(&mut self, addr: usize, len: usize){
    pmsg = self.pmsgHead;   //从头节点获取msg指针
    memcopy(addr, pmsg->data, len)  //将数据复制到用户态
    self.remove(pmsg);      //移除结点
}
```

一个 `msgque` 只能管理一个通信，多个通信需要多个 `msgque`，同前面两种 `IPC`，`msgque` 也设计成一个集合 `MsgQueManager`：

```
pub struct MsgQueManager {
    msg_queues: [msgque; N_MSG_QUEUES],
    lock: Spinlock<()>
}
```

本质上就是一个 `msgque` 数组，分配一把锁来控制 `msgque` 的分配，获取，回收，因此 `MsgQueManager` 的方法如下：

```
alloc() //分配一个空闲msgque
get()   //获取一个现有的msgque
put()   //回收msgque
```

添加系统调用

以上便是消息队列的设计，对用户提供了 5 个系统调用：

```
sys_mq_alloc //封装MsgQueManager的alloc函数
sys_mq_get
sys_mq_put
sys_mq_send
sys_mq_recv
```

1.4 信号量

信号量是一种不同进程/线程之间的一种同步手段，其内有一个计数器可以用来表示资源的个数，对外有 down，up 两种操作，分别可以表示获取和释放资源，当资源数小于 0 时执行 down 操作便会阻塞，阻塞到其他进程调用 up 释放资源。

数据结构设计

semaphore 本身就是一个数儿，所以有如下设计：

```
pub struct semaphore{
    cnt: i32,
    sem_lock: Spinlock<()>,
}
```

cnt 表示计数器，lock 来确保原子加减操作，相应的方法有：

```
semaphore_down()   //原子减
semaphore_up()     //原子加
```

这里对于 semaphore 本身用 Spinlock 实现的，后续考虑使用 rust 自带原语实现

对于 semaphore 我们在其上再做封装为 sem_t 类型：

```
pub struct sem_t{
    sem: semaphore,
    used: bool,
    id: i32,
}
```

增加了 used 和 id 属性，用作 sem_t 结构体的分配和回收，sem_t 的方法伪码如下：

```
sem_init()  //初始化 sem_t

sem_down() {
    while semaphore.cnt <= 0{   //如果计数值小于0
        sleep(semaphore, semaphore.lock)      //休眠
    }
    semaphore_down()   //原子减
}

sem_up() {
    semaphore_up();
    if semaphore.cnt > 0 {
        wakeup(semaphore)    //唤醒休眠在该semaphore上的进程
    }
}
```

xv6 本身的休眠和唤醒机制比较简单，每个进程结构体都有个休眠对象的属性：`chan: usize`，本质为休眠对象的地址，休眠操作就是将 chan 设置为休眠对象的地址，设置状态为 SLEEPING 然后让出 CPU，而唤醒操作 wakeup(chan) 就是遍历任务结构体，如果休眠对象的地址为相应的 chan，则将该进程的状态设置为 RUNNABLE，等待被调度。

所以我们要实现信号量其实就很简单，对于 down，up 操作只需要简单判断当前计数是否大于 0 即可，具体伪码如上所示。

另外需要注意的就是 down 操作判断 cnt 是否小于 0 时需要用 while 判断，因为 xv6 的唤醒机制是遍历整个任务结构体表，唤醒操作是统一唤醒，比如当前 A B C 三个进程都被唤醒想要获取 semaphore，但是只有最先被调度的 A 能够获取到，那么 B C 是获取不到的，需要用 while 再次进行判断才能保证逻辑正确，如果用 if 判断的话，那么 B C 也能获取 semaphore，则会出现错误。

目前我们所设计的信号量其实还是内核级的信号量，不能像标准的那样直接再用户态下声明一个 sem_t 即可使用，这里我们还是使用老办法设计一个 manager 对 sem_t 进行管理分配和回收

```
pub struct SemTable{
    sems: [sem_t; N_SEM],
    st_lock: Spinlock<()>,
}
```

相应的接口函数也就不言而喻了，跟前面的一模一样：

```
alloc()
get()
put()
```

添加系统调用

对外提供了 4 个系统调用

```
sys_sem_get   //封装semTable.get和semTable.alloc 获取一个 sem_t
sys_sem_put   //放下回收sem_t
sys_sem_up    //up操作
sys_sem_down  //down操作
```

2 线程、协程

进程是资源分配的单位，线程是调度的单位，这句话我们耳熟能详。根据进程的调度执行的过程，我们知道如果一个线程想要被单独调度而不是附属进程作为一个函数执行，它得有以下资源：

- **内核被调用者保存寄存器组成的任务切换上下文，这是由 swtch 函数进行任务切换而必须的，本质上就是个函数跳转操作，只不过这里是我们自己手写汇编实现，保存的寄存器也要多一点**
- **中断、异常处理时的上下文，每个线程在执行期间都可能遭遇中断异常等实践，停下手头工作转而去处理这些紧急事件时需要保存原任务的上下文**
- **代码执行需要的栈资源，包括在内核运行时的内核栈，在用户态下执行的用户栈**

这就是线程能够正常工作所需要的所有资源，其他资源比如说页表，堆，等等都不是必要的，与附属进程共享，所以说线程粒度较小，速度较快。

线程的实现方式多种多样，有内核级线程，也有用户级线程也是现在常说的协程，我们小组两者都做了简易实现。

2.1 内核级线程

在 Linux 里面没有明确的进程线程之分，都是使用同一结构体 task_struct 结构体表示，这里我们沿用这一思想，对于线程仍然使用 task_struct 结构体(原 proc 结构体，这里改了个名)，向其中添加了一项属性字段：

数据结构设计

```
pub struct task_struct{
    //...
    pub ustack: usize;   //线程的用户栈
}
```

不像进程的用户栈是在 exec 函数里面分配好了，释放的时候根据页表释放就行了，线程的用户栈是在进程的堆里面分配的，需要记录下来，线程执行完成的时候才能找到位置然后释放。

方法函数实现

对于线程提供了两个核心函数 clone 和 join，分别对标进程的 fork 和 wait，功能也是相似，只不过对于资源的处理有些许区别，伪码如下：

```
clone(ustackAddr, funcAddr, argAddr){
    task = get_task_struct();   //获取一个空闲的任务结构体
    task.init_context();   //初始化这个任务结构体的切换上下文
    task.init_trapframe(); //分配和初始化异常处理上下文

    task.pagetable = parent.pagetable; //线程直接使用父进程的页表
    //处理pid,parent,open_file,cwd,size,name等属性字段

    task.trapframe.epc = funcAddr;  //返回用户态的PC值指向func函数
    task.trapframe.sp  = ustackAddr + PGSIZE - 1;  //栈顶指针指向用户栈栈顶
    task.trapframe.a0  = argAddr;   //根据调用约定a0里面存放着第一个参数

    task.state = RUNNABLE;   //该线程准备好了，可以为其分配CPU了
}

join(ustackAddr){
    task.free_trapframe();    //释放trapframe
    task.pagetable = 0;       //将页表指针清零但不释放，释放是父进程的事

    *ustackAddr = task.ustack;  //将用户栈指针传出去
    //处理其他属性字段

    task.state = UNUSED;    //该任务结构体空闲了
}
```

添加系统调用

上述就是内核级线程的 内核部分主要实现，下面说说用户调用的接口，对用户提供了两个接口

内核接口

```
sys_clone(funcAddr, argAddr, ustackAddr)  //clone函数的封装
sys_join(ustackAddr)    //join的封装
```

用户接口

```
int clone(void (*start_routine)(void *), void* arg, void *stack);
int join(void*);

int thread_create(void (*start_routine)(void*), void *arg){
```

```
    void *stack = malloc(PGSIZE);    //分配用户栈
    int ret = clone(start_routine, arg, stack);  //clone系统调用
    return ret;
}

int thread_join(){
    long stack;
    int ret = join((void*)&stack);  //join系统调用
    free((void*)stack);    //释放用户栈
    return ret;
}
```

2.2 用户级线程

接下来继续说明用户级线程的实现，在用户态下实现线程，仿照内核，主要有一下几个点：

- 协程结构体设计
- 调度
- 上下文操作

一个一个来说明，先看上下文操作，这是最基础的部分

上下文

这里的上下文就相当于函数切换的上下文，同内核里面任务切换上下文，定义如下：

```
typedef struct {
    long ra;
    long sp;

    long s0;
    long s1;
    long s2;
    long s3;
    long s4;
    long s5;
    long s6;
    long s7;
    long s8;
    long s9;
    long s10;
    long s11;
} ucontext;
```

相应的操作，仿照 ucontext 类

```
getcontext(ucontext* ctx) //获取当前的上下文，保存到ctx处
setcontext(ucontext* ctx) //恢复ctx指向的上下文
swapcontext(ucontext *o, ucontext *n) //保存当前的上下文到o 恢复n指向的上下文
```

这三个函数的具体操作实际上就是 ld sd 指令，不再赘述。ucontext 类里面还有一个 makecontext 函数，这里我就没仿照实现了，直接在创建线程里面做了相应处理

协程

结构体设计如下：

```
typedef void (*Func)(void*);
enum ThreadState {FREE, RUNNABLE, RUNNING};

typedef struct uthread_t {
    ucontext ctx;    //上下文
    Func func;       //要执行的函数地址
    void *arg;       //函数参数地址
    enum ThreadState state;  //协程状态
    char *stack;     //协程的用户栈
}uthread_t;
```

相应操作的伪代码如下：

```
int uthread_create(Func func, void *arg){
    uthread = get_uthread();    //获取一个空闲的uthread结构体
    uthread.func = func;
    uthread.arg = arg;

    uctx = uthread.ctx;
    uctx.ra = (long)func;     //返回地址为func地址
    void *stack = malloc(PGSIZE);  //分配协程使用的栈
    uctx.sp = (long)stack + PGSIZE - 1;  //栈顶指针设置为上述分配的栈的栈顶

    uthread.state = RUNNABLE;   //该协程准备好了
}
```

调度

对于调度设计了一个调度类：

```
typedef struct scheduler {
    ucontext ctx;               //调度器的上下文
    int running_thread;         //当前运行的协程id
    uthread_t threads[MAX_THREADS];  //该调度器管理的协程
} scheduler;
```

调度器拥有一个上下文，调度器实际上相当于主线程，由它来选择相应的协程执行，执行完之后又回到调度器。相应的操作如下：

```
void scheduler_init(scheduler* schedule){    //调度器初始化
    for(int i = 0; i < MAX_THREADS; i++){
        sched->threads[i].state = FREE;
    }
    schedule.running_thread = -1;
}
void uthread_yield(scheduler *schedule){   //主动让出CPU
    int id = schedule->running_thread;   //获取当前协程
    uthread_t *t = &(schedule->threads[id]);
    t->state = RUNNABLE;         //状态置为RUNNABLE
    swapcontext(&t->ctx, &schedule->ctx);  //切换到调度器
```

```
}
void uthread_exit(scheduler *schedule){  //协程退出
    int id = schedule->running_thread;   //获取当前协程
    uthread_t *t = &(schedule->threads[id]);

    void *stack = t.stack;   //获取栈地址
    free(stack);       //释放栈

    t->state = FREE;         //结构体只空
    setcontext(&schedule->ctx);   //设置为调度器上下文回到调度器
}

void runScheduler(scheduler *schedule){
    while(not all finished){
        uthread = seek_one_Runnable();   //寻找一个RUNNABLE协程
        schedule->running_thread = uthread->id;  //正在执行的协程id更新
        swapcontext(&schedule->ctx, &uthread->ctx);  //切换!
        schedule->running_thread = -1;   //没有在执行的
    }
}
```

上述便是用户级线程的实现，主要就是模拟内核里面调度，上下文的操作。

3 其他功能实现

3.1 封装运行库

运行库，顾名思义用户程序运行需要的环境组成的一个库，可以看作是标准库的超集，其中一个功能便是给 main 函数提供环境。

main 函数也是函数，根据函数调用约定，如果 P 调用 Q，P 要提供 Q 需要的参数，P 还要要处理 Q 执行的结果。那谁来为 main 函数提供准备参数呢？谁又来处理 main 函数的执行结果呢？答案就是这里的运行库。

原有的 xv6 没有封装运行库，所以 exec 函数执行完之后，直接中断退出到用户态开始执行用户程序，用户程序执行完之后不能像普通函数那样 return 返回，只能调用 exit 函数进行退出。封装了运行库之后，使得 main 函数更像普通函数，可以使用 return 语句。具体实现如下：

```
.globl main
.globl exit

.section .text
.globl _start
_start:

    jal ra, main
    jal x0, exit
```

编写如上所示的汇编代码，指定 _start 标识，然后和用户程序编译在一起：

```
CRT = $(ULIB) $(USER)/start.o $(UTHREADLIB)

#$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
$(LD) $(LDFLAGS) -N -Ttext 0 -o $@ $^
```

这样以来，用户程序的入口点从指定的 main，变为默认的 _start，这样的话执行 main 函数之前会先执行 _start 汇编程序。

这里首先就直接跳到 main 函数执行，之所以没有为其准备参数是因为在 exec 函数里面已经准备好了，a0 a1 寄存器里面就是现在 main 函数需要的参数，不需要做任何改变。

main 函数执行完成使用 return x 退出之后，a0 里面存放的是返回状态码，也是 exit 函数需要的参数，同样的我们不需要做任何改变，直接调用 exit 函数。

这就是运行库的作用之一，为 main 函数准备环境和善后收尾。

3.2 改写任务结构体

实现前面的功能时老是会出现锁的问题，就想着先使用大粒度锁来简化操作，后续开发再使用小粒度锁提高性能。

对于内核任务管理锁的方面做如下处理：使用一把锁 ptable_lock 来管理整个任务结构体表，用来保护任务结构体的分配，回收，状态变化等等。再使用一把锁 wait_lock 来确保 wait 函数不会错过休眠。

xv6 中的休眠唤醒机制如下所示：

```
sleep(objAddr, obj_lock);
```

休眠在某个对象上是指将该进程的 chan 属性字段设置为 objAddr，然后设置状态为 SLEEPING，调用 swtch 函数进行任务切换。

之所以需要 obj_lock 是为了避免错过唤醒，原因如下：

首先需要一个 obj_lock 来保护资源在不同进程之间的竞争，另外如果没有 obj_lock 的话，那么 唤醒进程 调用 wakeup 函数是可能在 休眠进程 调用 sleep 之前的。

举个例子，唤醒进程使用 A 来表示，休眠进程使用 B 来表示，B 调用了 sleep 函数休眠，需要 A 调用 wakeup 来唤醒。

如果没有 obj_lock，B 直接 sleep(obj) 休眠在 obj 上，如果 B 因为某些事件比如说中断延后执行 sleep，由于没有 obj_lock 的保护，那么进程 A 可以调用 wakeup 进行唤醒，但毫无作用。B 进程回来之后调用 sleep 进行休眠，但是 A 已经执行过 wakeup 了，这就是错了唤醒，所以必须使用 obj_lock 来保证不会错过唤醒。

但是必须得把 obj_lock 当作参数传进 sleep 函数，原因如下：

1. **我们不能带锁休眠，不然会死锁，所以在实际将 state = SLEEPING; swtch() 之前，需要释放锁。**
2. **但是如果在 state = SLEEPING; swtch() 之前释放锁，完全有可能出现前面所说的情况，在这之前 B 进程因为某些事件延后执行，就比如中断。这时候 A 进程取得 obj_lock，执行 wakeup 函数，那么 B 进程就会错过唤醒。**

所以综上，**sleep 函数需要在 state = SLEEPING ; swtch 之前释放掉 obj_lock 来保证不会带锁休眠导致死锁，又需要在 state = SLEEPING ;swtch 之后释放调用来保证不会错过唤醒。**

两者明显冲突，**因此需要另外一个来保证两者是原子操作就行，这里就使用了 ptable_lock**，所以 sleep 函数中的操作为：

```
sleep(obj, obj_lock){
    acquire(ptable_lock);
    release(obj_lock);

    state = SLEEPING;
    swtch();

    release(ptable_lock);
    acquire(obj_lock);
}
```

如此便解决了带锁休眠，错过唤醒的两个问题，不过还有一个小问题：**wait 函数中休眠的对象锁就是 ptable_lock，冲突了，为此设计了一个 wait_lock 来供 wait 函数使用。**

经过上述改造，对于进程的调度，状态改变锁的使用简单许多，虽然性能有些许下降，但是便于后面的功能实现和测试，待完善以后再使用小粒度锁提高性能。

3.3 修改页表内存释放 bug

原有项目代码对于页表占用的内存释放有 bug，并未完全释放，这里修改，释放进程的函数主要有两个

```
uvm_free(size){ //根据用户程序大小释放内存
    for page in 0..size {
        free(page)
    }
}
free_pagetable(pgt){ //释放页表占用的内存
    //如果不是叶子页表，即页表项指向的内存不是物理内存还是页表
    if pgt.isleaf() == 0 {
        for pte in pgt {   //遍历页表项
            free_pagetable(pte as pgt)  //递归释放子页表
        }
    }
    free(pgt)  //释放了子页表，释放自己
}
```

3.4 延迟分配

缺页异常有三种情况：

1. **读写页表项不存在的内存，可以利用这个来实现延迟分配**
2. **读写内存没有相应的权限，比如对只读内存进行写操作，可以用来实现写时复制**
3. **读写页表项 P 位为 0 的内存，表明相应的数据不在主存，而在外存磁盘上，可以用来实现页面置换算法**

这里我们先实现了页面延迟分配。

需要分配内存的地方就是 malloc，malloc 有个函数 morecore，morecore 又调用 sbrk 系统调用向内核申请空间。

sbrk 系统调用会调用 growproc 来进行物理内存分配，然后使用 map 建立映射关系，这里我们需要做的就是 sbrk 里面并不实际调用 growproc 来增长物理内存，只将用户程序的大小增加相应的值然后返回。

待到实际向这片区域进行读写操作时，就会触发缺页异常，这时候再从 stval 中读取到引发缺页异常的地址，对这个地址向下取整，然后为其分配实际的物理内存再建立映射关系。

3.5 进程优先级调度

对于进程的调度算法，在原先的基础上实现了优先级调度，目前的实现思想和方式比较简单，**就是在结构体里面添加 priority 优先级字段，每次调度器 scheduler 执行的时候，如果当前进程还没有执行完成，那么优先级按照一定算法降低，最简单的就是减去一个固定值。然后 scheduler 每次都从就绪队列中选取一个优先级最高的进程为其分配 CPU。**

由于原有的进程结构体的组织方式用的是数组，每次都要遍历，效率其实不高，这在后续中我们会改成使用链表或者树结构，实现红黑树以及 CFS 完全公平调度算法。

4 待/正开发的功能

4.1 信号机制

Signal 机制也是进程间通信的一部分，它是用软件对中断的模拟。这是一般情况下的意义，只不过现在信号的含义比较广泛，它可能与外部中断有关，比如说键盘按下 Ctrl + C 触发 SIGINT 信号。另外信号机制在英文中有时也称为 software interrupt，x86 架构下的 INT x 指令也是这个称呼，但这其实是两种不同的概念。

这里我们打算实现有软件来模拟中断的信号机制，数据结构设计如下：

数据结构设计

```
struct proc {
    //...
    uint sig_pending;    //等待处理得信号
    uint sig_mask;       //屏蔽得信号
    void *sig_handlers;  //处理信号的函数的位置
    //...
}
```

主体部分就是在进程结构体 proc 中添加以上三个属性字段，各含义如上所示。

```
struct sigaction {
  void (*sa_handler)(int);   //sig_handler
  uint sigmask;              //是否屏蔽
};
```

这个结构体用作注册信号使用，具体含义如上所示

下面来讲述信号的处理过程以及上述结构涉及的一些方法

信号处理过程

注册信号

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
{
    struct proc *p = myproc();   //获取当前进程
    //将当前的 sigaction 传到用户态的old位置处
    copyout(p->pagetable, (uint64)&oldact->sa_handler, (void *)&p-
>sig_handlers[signum], sizeof(p->sig_handlers[signum]));
    copyout(p->pagetable, (uint64)&oldact->sigmask, (void *)&p-
>handlers_mask[signum], sizeof(p->handlers_mask[signum]));
    //将用户态的sigaction结构体信息复制到内核，即将处理该信号的函数地址赋到
sig_handlers[signum]
    copyin(p->pagetable, (void *)&p->sig_handlers[signum], (uint64)&act-
>sa_handler, sizeof(act->sa_handler));
    copyin(p->pagetable, (void *)&p->handlers_mask[signum], (uint64)&act-
>sigmask, sizeof(act->sigmask));
}
```

发送信号

发送信号是 kill 函数来实现，这里我们改进原 xv6 里面的 kill 函数

```
int kill(int pid, int signum){
    struct proc *p = getProcByPid();
    //...
    p->sig_pending = p->sig_pending | (1 << signum);
    //...
}
```

如上所示，发送信号是相当简单，将进程号为 pid 的进程的 sig_pending 属性字段与 (1<<signum) 做或操作即可。

处理信号

处理信号就相当于中断处理，这期间必不可少的操作就是上下文的保存与恢复，模拟中断处理过程，我们的信号处理过程大致为：保存上下文->处理信号->恢复上下文 这三个过程。

但是信号处理与实际的中断处理还是有所不同，中断的处理过程一定是在内核当中实现，也就是说中断处理函数是个内核函数，需要在内核态执行，但是信号处理函数不一定，信号处理函数有些是内核函数，比如说 kill 一个进程发送的 SIGKILL 信号，这个信号的处理是可以在内核中直接完成的。但是对于一些用户注册的信号，其处理函数是在用户态，虽然内核权力大可以执行用户态的代码，存取用户态的数据，但是各司其职，其处理过程还是应当在用户态完成，所以这里就会有上下文保存恢复，特权级转移，我们的伪码实现如下所示：

```
void handle_signal(){
    struct proc *p = myproc();
    for(int i = 0; i < 32; i++){    //遍历所有信号
        //如果当前信号未处理并且未屏蔽
        if(p->sig_pending & (1 << i) && !p->sig_mask & (1 << i)){
            if(i == SIGKILL | SIGSTOP | ...){ //如果内核可以直接处理
                handle_in_kernel();
            }else{

                p->trapframe->sp -= sizeof(struct trapframe);  //在用户栈分配用户态
的trapframe空间
                p->signal_trapframe = p->trapframe->sp;    //将这个地址记录在案
```

```
                //将当前的trapframe填充到用户栈
                copyout(p->pagetable, p->signal_trapframe, p->trapframe,
sizeof(struct trapframe));

                //信号处理退出函数的大小
                long size = (long)&sigret_end - (long)&sigret_start;
                p->trapframe->sp -= size;  //在栈中为信息处理函数分配空间
                //将信息处理函数复制到用户栈
                copyout(p->pagetable, p->trapframe->sp, (void*)&sigret_start,
size);

                p->trapframe->a0 = i;   //第一个参数为signum
                p->trapframe->ra = p->trapframe->sp;  //信号处理函数执行完之后执行这个
函数sigret

                p->trapframe->epc = (long)p->sig_handlers[i];   //返回用户态时的地
址，执行信号处理函数

            }
        }

        p->sig_pending &= ~(1 << i);  //该信号“在处理了”
    }
}

void sigret(void){
    struct proc *p = myproc();
    //将保存在用户态的trapframe复制到该进程的trapframe
    copyin(p->pagetable, (char *)t->trapframe, (uint64)t->usertrap_backup,
sizeof(struct trapframe));
    //..
    p->trapframe->sp += sizeof(struct trapframe);  //栈指针向上移动一个trapframe大小
}
```

上述便是处理信号处理的核心过程，最重要的就是保存上下文到用户栈，然后更新 epc，ra 寄存器来改变执行流的操作。这部分会在后续完善。

4.2 写时复制

写时复制是利用页表项的读写位来实现的，主要用在 fork 程序当中，在原本的 fork 程序当中，子进程是实实在在的复制了一份父进程的页表和用户态的所有数据(代码和数据)。但是写时复制只会复制父进程的页表，不会复制数据，只有当一个进程在写的时候才会实际分配物理内存复制一份数据。因此，按照这个思想，写实复制实现的思路如下所示：

```
int uvmcopy(pagetable_t old, pagetable_t new, uint64 sz){
    for(int i = 0; i < sz; i++){
        pte = getPte(i);   //获取地址i所在页的pte，不存在的话就分配
        pa = PTE2PA(pte);  //取出里面存放的物理地址

        pte.W = 1;  //旧页表项禁止写操作
        pte.COW = 1; //将COW位置1，没有这个位，我们将页表项中的保留未使用的位当作COW位
        flags = pte.flags;  //获取页表项的flags;
        //将地址i映射到pa，标识为flags，同样禁止写，此函数会自动创建页表项。
        mappages(new, i, PGSIZE, pa, flags);
    }
}
```

此函数便是改进后复制用户数据的函数，可以看出并未实际复制数据，只是复制了一份页表，并且将叶子页表项的 W 位清零表示禁止写操作。

所以如果有进程对相应内存进行写操作，那么就会触发缺页异常，因此接下来需要做得就是在异常处理函数处理该异常，为触发异常的进程分配内存，复制一份数据，修改页表项为可写，伪码如下：

```
trap(){
    struct proc *p = myproc();
    if(r_scause() == 15){   //表示StorePageFault
        va = r_stval(); //获取触发异常的虚拟地址
        va = PG_ROUND_DOWN(va);  //向下取整到PGSIZE整数倍

        pte = walk(p->pagetable, va);  //获取该地址va所在页的页表项
        if(pte.COW == 1){
            mem = kalloc();  //分配物理内存
            pa = getPaddr(p->pagetable, va);  //根据页表获取va的物理地址
            memmove(mem, pa, PGSIZE);  //复制一份数据

            flags = getFlags(pte);   //获取flags
            flags |= PTE_W;      //增加可写
            flags &= ~PTE_COW;  //取消COW位

            mappages(p->pagetable, va, PGSIZE, mem, flags);  //重新将va映射到mem
        }
    }
}
```

上述便是 COW 写时复制的基本实现过程，核心就是不复制数据，只复制页表，等到出错触发缺页异常的时候才真正复制一份数据出来并重新映射修改页表项使其可写。

复制相当于之前的分配操作，我们已经完成，但有分配就有回收，还需要对回收操作进行改动，因为在写时复制下，多个进程共享内存，当一个进程退出时会释放内存，但是因为共享，不能真正的释放内存，解决方法如下：

```
char pa_ref[NR_PG];   //设计一个数组存放每个数据页的引用次数
copy_uvm(){       //复制数据的时候
    add_ref(pa);  //增加该页的引用次数
}
kree(pa){     //释放该页
    sub_ref(pa)   //该页引用次数减1
    if(pa_ref[pa] == 0){  //只有该页的引用次数为 0 的时候才真正的释放
        free(pa);
    }
}
```

核心思想就是增加一个引用计数，只有当物理页的引用次数为 0 的时候才真正的进行释放操作。

以上便是写时复制的实现思路，这在原本的 xv6 上是很好实现的，因为分配和释放物理内存的单位都是页，不过现有项目底层实现了伙伴系统和 Slab 分配器，实现方式会有所不同，但核心思想不变，这在后面完善实现。

4.3 mmap 系统调用

mmap 系统调用的功能很多，我们准备实现其基本零拷贝读写文件的功能，**mmap 与共享内存挂钩，可以看作是 共享内存 + bread**，我们首先在共享区申请一块共享内存，然后调用文件系统底层的读取快缓冲函数将数据读到该共享内存。如此便实现了零拷贝读写文件的功能，在取消映射的时候将数据同步到磁盘。

4.4 CFS 调度算法

现下只是实现了简单的优先级调度，后续会改进进程结构体的组织方式，使用红黑树实现，然后在其上实现 CFS 调度算法。将 task_struct 维护成一个红黑树，排序规则为 虚拟时间大小，最左侧为虚拟时间最小的节点，每次取最左侧的节点分配 CPU，是为“完全公平”。一些数据结构设计如下所示：

红黑树

红黑树是一种自平衡的二叉树，最左的叶子节点永远是 key 最小的节点。红黑树通过插入（更新）和删除时的操作保证这些原子操作之间红黑树永远是平衡的。

运行时间

$$运行时间t_n = 调度周期T \times 进程权重w/运行队列中全部进程的权重之和S$$

调度周期/调度延迟 是内核里面中的固有概念(不是固定值),他表示了一段时间,并且在这段时间内,所有的可被调度程序都应该至少被运行一次.

但是在 CFS 中没有固定的时间片了,我们的方法是先规定好调度延时,然后进程再根据比例去瓜分一个调度延时(周期)的时间.

最小运行时间

为了避免过度频繁的抢占发生，我们设置每个 Task（进程）的最小运行时间（或称运行时间粒度），在这个时间内，这个进程的 CPU 资源是不可被抢占的。除非进程主动让出 CPU 或者执行了阻塞的系统调用，一般而言进程都可以执行最少执行完这个时间。

虚拟运行时间

$$虚拟运行时间 = 真实运行时间 \times NICE\_0\_LOAD/进程的权重$$

其中 nice 值为 0 的权重 NICE_0_LOAD=1024，nice 值每差1，权重大约差 1.25 倍。这里的1.25计算依据来源于 nice 值差 1，运行时间相差 10% 这样的设计

权重越大的值虚拟运行时间越小，可能使得**高优先级进程运行15ms=低优先级运行5ms**

在内核里面我们将调度队列上的进程按照 vruntime 排列成红黑树，这样选取最小 vruntime 的进程就变为了简单地选出最左边的叶子节点了，一次来达到“完全公平”调度。

三、问题与解决方法

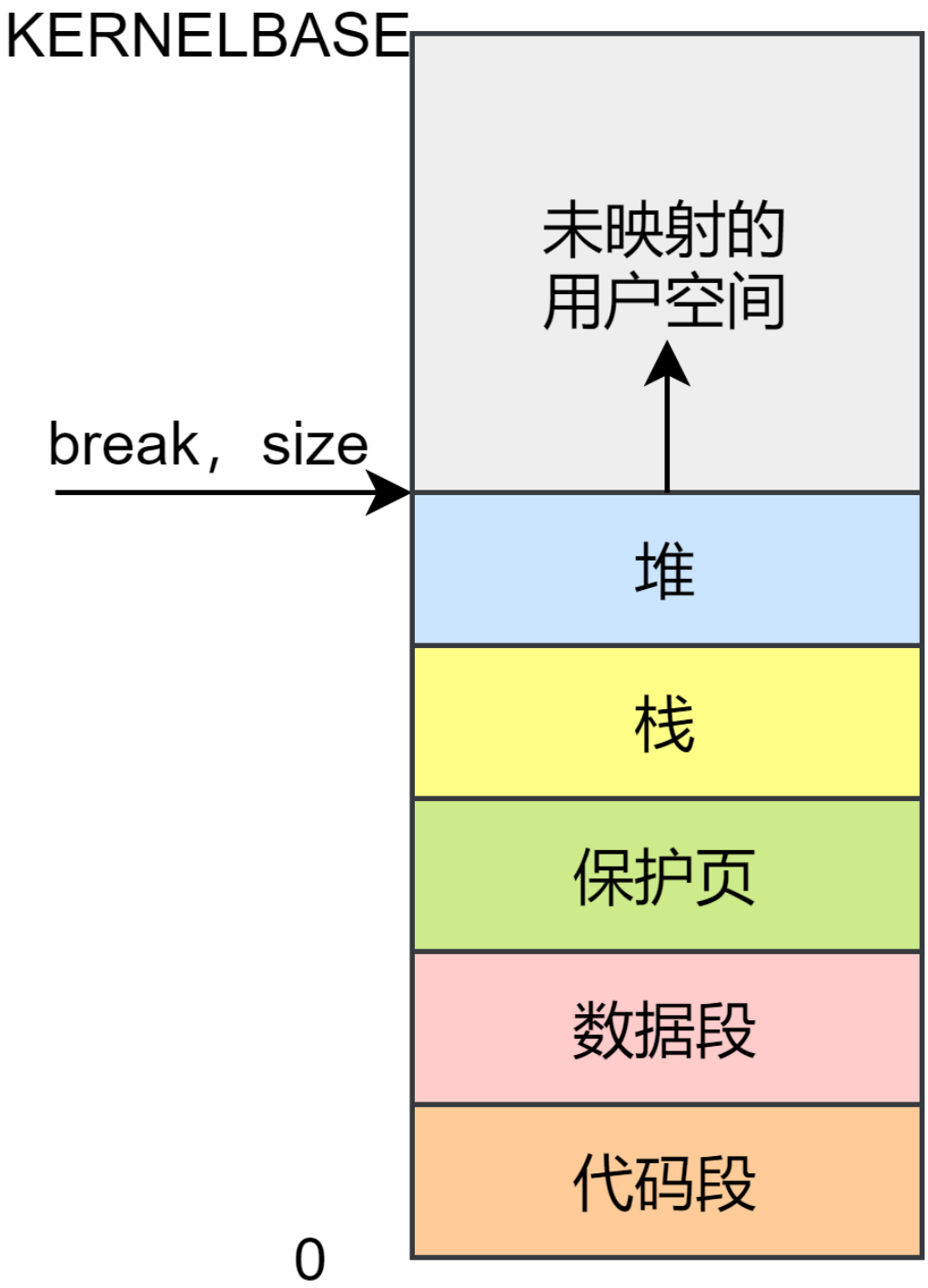
1 rust语言和riscv架构

此次项目是在 riscv 架构下使用 rust 语言开发，对于 rust 语言和 riscv 架构之前没怎么接触过，不是很了解。rust 语言有着自己的特性，比如所有权，生命周期等概念，rust安全性很高，在编译时期就将许多错误指出来，所以编写代码时经常需要与编译器斗智斗勇。

架构方面，以前较为熟悉的是 x86 架构，此次的接触 riscv 架构，使用精简指令集，对于汇编的写法，函数调用的约定，中断处理的过程，特权级的分离转移等等都有所不同。经常因为 x86 的定势思维而写错代码。

2 用户栈溢出问题

xv6 里面的内存布局与现在的 Linux 有所不同，栈直接位于数据段上面，并且只分配了一页，如下图所示：



所以用户栈的大小其实比较小，最开始在实现协程时，为其分配协程栈时直接定义成了 `char` 数组，大小为 `4096` 字节，局部变量都是存在栈中的，所以这会导致栈溢出。

最开始没有注意到这个问题，因为对于线程协程的实现，对于上下文的操作涉及到精密的寄存器操作，定向思维觉得是上下文操作更可能出错，便没有注意到栈溢出的问题。

后来反复打印消息和使用 `gdb` 进行调试，发现定义一个协程可以语句可以执行，但只要一执行后面的语句马上出错，这让我意识到应该是用户栈出了问题。然后仔细排查，得出声明的协程类中栈应该设置为一个 `char *` 的指针即可，实际的内存从堆里面分配。

3 trapframe 覆盖问题

这个问题也还是定向思维的问题，我之前一直了解的是 x86 下的 xv6，x86 架构下，trapframe 没有单独分配空间，使用的是内核栈空间。切换进程最重要的操作就是更换内核栈，然后恢复任务切换上下文，再然后弹出恢复存放在内核栈里面的 trapframe 恢复到用户态。

而且 x86 架构下的中断机制通过 IDT，中断向量，中断描述符等机制实现，跳转到的中断服务程序地址由中断描述符中记录的地址决定。再者 x86 下对于特权级转移时 内核栈 用户栈 是通过 TSS 中记录的 esp0 来实现的的。

但 riscv 架构下的 xv6 不同，trapframe 的虚拟地址固定，中断跳转退出的地址都固定。每个进程要退回用户态时都从同一个地方获取 trapframe，然后恢复上下文。

在实现内核级线程的时候，线程有自己单独的 trapframe，但是它的地址不再是预定义好的 TRAPONLINE，所以退出的时候也不能从 TRAPONLINE 获取 trapframe 的地址。

这个 bug 我找了很多天，一直没找到，后面仔细研究了 riscv 的中断机制，然后细细阅读代码找到了这个原因。然后便给出了上述章节 2.1 给出的解决方案，每次读取的地址变为 $TRAPONLINE + threadID * sizeof(struct\ \ trapframe)$，便解决了问题，具体方案见前。

4 锁的问题

xv6 底层锁的本身设计比较简单，但是使用起来比较困难，特别是如果想要提高性能，将锁的粒度降下来更是困难。在开发当中出现了很多与锁相关的问题，但细细研究其实出问题的地方并非是锁，比如上述的 trapframe 覆盖，栈溢出等问题，由于栈中的数据和 trapframe 中保存到的上下文出了问题，这些都涉及到了实际的指令方面的问题，会导致莫名的错误，然后又是由于多 CPU，总总原因最后会导致取锁问题，然后 panic。

这种莫名问题最是难以调试，而且由于调度时机等问题，每次调试的结果都不一样，为了简化操作，缩小出问题的范围，便将锁的粒度给降下来，比如说进程方面，只是用了一个 ptable_lock，wait_lock 来保护，重构了管道使其数据与锁分离等等，来减小因为锁出问题的概率，然后缩小实际出错的范围，便于调试。

5 其他问题

其他问题大多就是写代码的时候由于粗心大意，用错函数，比如说某些函数的参数较多，参数顺序弄错，就如 mappages(pagetable，va，size，pa，perm)，其中虚拟地址 va 和 pa 是分开的，中间隔了一个 size，而我以为 va 和 pa 两个参数在一起，导致错误发生，这类错误有的还是很好排查，有的不太起眼便不太容易。

另外还有就是由于不太清楚 rust、makefile、qemu 等等一些配置文件，有时稍有不经意的改动，便会出错，比如说当时安装 tmux 因占用端口的原因导致不能远程连接 debug，后来仔细分析报错信息和阅读一些配置脚本文件，将错误改了过来。

四、未来展望

目前我们组只实现了最基本的功能，还有许多地方都未完善，将在后面完善，提高安全稳定健壮性，具体如下所示

1 进程间通信

改写内存管理方式，将共享区使用链表来管理

改写文件系统，使得有名管道像文件一样可以安装到文件系统中

信号量机制目前只实现了内核级的，并不能像已有库里面那样直接在用户程序里面声明使用，另外对于计数器的原子加减是通过锁来实现的，后续可以利用 `rust` 特性来实现。

消息队列以及上述所有的通信方式都还需要在生产者消费者模型上，以及多个进程通信上完善

另外后续也会尝试实现较为复杂的信号机制，将外部中断，软件中断，实际的信号等抽象出来

2 线程协程

对于内核级线程，后续可以尝试将进程线程稍微分离开来，并不是简单粗暴的线程直接使用进程的结构体，只不过在某些资源方面"置空"处理，将线程需要独占的资源分离出来，用另外的数据结构表示，再实现相应的方法，如此性能可能进一步提高

对于用户级线程，后续会完善 `ucontext` 类，对于用户级线程的调度器也还需要完善。

3 缺页异常

利用缺页异常可以实现延迟分配，写时复制，页面置换，目前已经实现了延迟分配，后续可以实现写时复制和页面置换。

写时复制的核心思想是：调用 `fork` 赋值用户态数据时并不进行实际的复制操作，只复制相应的页表，在复制的过程中将父子进程的页表项的读写为设置为只读，如此父子进程的某一方需要对内存进行写操作时便会触发缺页异常，这时我们才在缺页异常处理函数中为其分配实际的物理内存修改页表项重新建立映射关系

目前写时复制已经实现了这一步，不过写时复制应该还要增加页面引用计数功能，这在原先的物理内存分配接口 `kalloc，free` 是比较方便操作的，不过现有项目底层的物理内存分配器为伙伴系统和 `Slab` 分配器结合体，具体如何操作有待后续思考完成

4 调度算法

关于调度算法目前只是在原有的时间片轮转法基础上为进程添加优先级属性，`scheduler` 每次挑选进程时总是挑选优先级高的进程，以此来实现了最简单的优先级调度算法。

使用数组来组织任务结构体效率较低，后续准备使用链表或者红黑树来组织任务结构体，然后实现 `CFS` 调度算法。

不只是任务结构体的组织方式，休眠唤醒机制也可以使用链表来实现，不然每次遍历任务结构体数组，效率太低。

五、代码结构

项目的代码结构如下所示：

```
├──allocator
├──bin
├──docs
|   └──static
├──fs-lib
|   ├──src
```

```
|   └──target
|
├──kernel
|   ├──.cargo
|   └──src
|       ├──asm
|       ├──console
|       ├──define
|       ├──driver
|       ├──fs
|       ├──ipc
|       ├──interrupt
|       ├──lock
|       ├──logo
|       ├──memory
|       |   └──mapping
|       ├──net
|       ├──process
|       ├──register
|       ├──syscall
|       └──test
├──mkfs
|   ├──src
|   └──target
├──user
|   ├──.cargo
|   └──src
|       ├──include
|       ├──uthread
|       └──bin
└──utils
```

`allocator` 单独作为一个 `crate` 实现 `Buddy System` ，在 kernel 中引入

`bin` 用来存放用户程序的二进制可执行程序

`docs` 用来存放文档

`fs-lib` 用来为 `mkfs` 实现一些基础的文件系统的数据结构和方法

`kernel` 用来存放操作系统内核打代码

- `asm` 用来存放汇编代码
- `console` 用来存放输入输出相关代码，包括`uart` 和 `console` 的实现
- `define` 主要用来存放常量的定义
- `driver` 存放设备驱动代码
- `fs` 存放文件系统的实现
- `ipc` 存放进程间通信的各种实现
- `interrupt` 存放 `trap` 的处理函数
- `lock` 实现了 `spinlock` 和 `sleeplock`
- `logo` 为本项目的 logo
- `memory` 存放内存分配以及地址映射等相关代码实现
- `net` 存放网络栈的实现代码
- `process` 存放进程与调度相关的代码
- `register` 主要用来存放各种寄存器的相关代码

- `syscall` 用来存放系统调用的实现
- `test` 主要存放一些用于在内核测试的代码

`mkfs` 用来存放构建文件镜像的实现代码

`user` 用来存放用户程序，其中包括函数库，系统调用和用户程序的实现

- `uthread`，实现了协程库和 `ucontext` 库
- 其下还有上述实现(二)的各种测试程序

`utils` 主要用于存放一些帮助程序运行的工具代码

六、收获

- 使用 Rust 改写 xv6，更加深刻地理解了操作系统与硬件是如何协同工作的
- 熟悉了 Rust 语言在系统级应用上的开发
- 增强了调试程序的能力
- 提高了自己开发较大型工程的能力

七、参考实现

- xv6-riscv
- rCore-Tutorial-v3
- xbook2