练习0：填写已有实验

本实验的代码合并要用到lab6的相关代码

```
proc.c/alloc_proc()
{
    ...
         proc->rq = NULL;
        list_init(&(proc->run_link));
        proc->time_slice = 0;
        proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc-
>lab6_run_pool.parent = NULL;
        proc->lab6_stride = 0;
        proc->lab6_priority = 0;

        proc->filesp = NULL;
}
```

```
default_sched_stride.c/proc_stride_comp_f(void *a, void *b){
     struct proc_struct *p = le2proc(a, lab6_run_pool);
     struct proc_struct *q = le2proc(b, lab6_run_pool);
     int32_t c = p->lab6_stride - q->lab6_stride;
     if (c > 0) return 1;
     else if (c == 0) return 0;
     else return -1;
}

stride_init(struct run_queue *rq) {//开始初始化运行队列，并初始化当前的运行队，然后设置当
前运行队列内进程数目为0
     list_init(&(rq->run_list));//初始化调度器类
     rq->lab6_run_pool = NULL;//对斜堆进行初始化，表示有限队列为空
     rq->proc_num = 0;//设置运行队列为空
}

stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {//将进程 proc 添加
到运行队列 rq 中。这个函数根据宏定义 USE_SKEW_HEAP 的状态，使用不同的方式将进程添加到运行队列
中
#if USE_SKEW_HEAP
     rq->lab6_run_pool =
          skew_heap_insert(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);//将进程 proc 插入到斜堆 rq->lab6_run_pool 中，按照给定的比较函数
proc_stride_comp_f 进行排序
#else//如果宏定义 USE_SKEW_HEAP 未被定义，表明使用链表作为调度算法的数据结构
     assert(list_empty(&(proc->run_link)));//确保进程的 run_link 指针为空，以确保进程
不在其他运行队列中
     list_add_before(&(rq->run_list), &(proc->run_link));//将进程 proc 添加到运行队
列 rq 的末尾
#endif
     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {//检查
进程的时间片（time slice）是否符合预期。
          proc->time_slice = rq->max_time_slice;//将该进程剩余时间置为时间片大小
```

```
     }
     proc->rq = rq;//将进程的 rq 指针指向当前运行队列 rq，表示该进程被加入到该运行队列中
     rq->proc_num ++;//增加运行队列 rq 中进程数量的计数器
}

stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {//将进程 proc 从运行队列 rq 中移除
#if USE_SKEW_HEAP
     rq->lab6_run_pool =
          skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool), proc_stride_comp_f);//删除斜堆中的指定进程
#else
     assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
     list_del_init(&(proc->run_link));
#endif
     proc->rq = NULL;
     rq->proc_num --;//维护就绪队列中的进程总数
}

stride_pick_next(struct run_queue *rq) {//从运行队列 rq 中选择下一个要执行的进程,根据stride算法，只需要选择stride值最小的进程，即斜堆的根节点对应的进程即可
#if USE_SKEW_HEAP
     if (rq->lab6_run_pool == NULL) return NULL;
     struct proc_struct *p = le2proc(rq->lab6_run_pool, lab6_run_pool);//选择 stride 值最小的进程
#else
     list_entry_t *le = list_next(&(rq->run_list));

     if (le == &rq->run_list)
          return NULL;

     struct proc_struct *p = le2proc(le, run_link);
     le = list_next(le);
     while (le != &rq->run_list)
     {
          struct proc_struct *q = le2proc(le, run_link);
          if ((int32_t)(p->lab6_stride - q->lab6_stride) > 0)
               p = q;
          le = list_next(le);
     }

#endif
     if (p->lab6_priority == 0)//优先级为 0
          p->lab6_stride += BIG_STRIDE;//步长设置为最大值
     else p->lab6_stride += BIG_STRIDE / p->lab6_priority;//步长设置为优先级的倒数，更新该进程的 stride 值
     return p;
}

stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {//时钟中断
     if (proc->time_slice > 0) {//到达时间片
          proc->time_slice --;//执行进程的时间片 time_slice 减一
     }
     if (proc->time_slice == 0) {//时间片为 0
```

```
            proc->need_resched = 1;//设置此进程成员变量 need_resched 标识为 1，进程需要调
度
        }
}
```

练习1: 完成读文件操作的实现（需要编码）

```
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
offset, size_t *alenp, bool write) {
    if ((blkoff = offset % SFS_BLKSIZE) != 0) {
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }
        alen += size;
        if (nblks == 0) {
            goto out;
        }
        buf += size;
        blkno ++;
        nblks --;
    }

    size = SFS_BLKSIZE;

    while (nblks != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
            goto out;
        }
        alen += size;
        buf += size;
        blkno ++;
        nblks --;
    }

    if ((size = endpos % SFS_BLKSIZE) != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
            goto out;
        }
        alen += size;
    }
}
```

- **对齐处理**

```
if ((blkoff = offset % SFS_BLKSIZE) != 0) {//判断offset是否对其，bloff为块内偏移量
    size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
    //size是需要读写的长度
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {//获取映射块
        goto out;
    }
    if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {//读写对应块
        goto out;
    }
    alen += size;//更新，如果没有块就跳出
    if (nblks == 0) {
        goto out;
    }
    buf += size;
    blkno ++;
    nblks --;
}
```

- **对齐块读写**

```
size = SFS_BLKSIZE;

while (nblks != 0) {//处理对齐的块，循环读写每个块
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
        goto out;
    }
    alen += size; //更新alen，移动缓冲区指针和块号，继续处理下一个块
    buf += size;
    blkno ++;
    nblks --;
}
```

- **不对齐部分处理**

```
if ((size = endpos % SFS_BLKSIZE) != 0) {//文件末尾的不足一个块的部分
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
        goto out;
    }
    alen += size;
}
```

练习2: 完成基于文件系统的执行程序机制的实现（需要编码）

> 改写`proc.c`中的`load_icode`函数和其他相关函数，实现基于文件系统的执行程序机制。执行：`make qemu`。如 果能看看到`sh`用户程序的执行界面，则基本成功了。如果在`sh`用户界面上可以执行"ls","hello"等其他放 置在`sfs`文件系统中的其他执行程序，则可以认为本实验基本成功。

`load_icode`函数

(1) 为当前进程创建一个新的内存管理结构（mm）。

(2) 通过`setup_pgdir(mm)`创建一个新的页目录,并将`mm->pgdir`设置为内核虚拟地址。

(3) 读取`ELF`头部信息，并检查`ELF`头部的魔数是否正确验从而验证文件格式是否正确。

(4) 遍历ELF文件的程序头表，并对每个程序段进行处理。具体的处理包括了内存映射、读取程序内容到内存、设置页面权限等操作。

- `load_icode_read`读取可执行文件的程序头表，并对每个程序段进行必要的检查，以确保程序段的合法性和正确性。
- 设置`vm_flags`和`perm`,这两个变量用于表示页面的权限和标志，根据程序段的标志位`（p_flags)`来设置。
- 调用`mm_map`函数，创建新的虚拟内存区域，并映射到对应的物理内存，将程序段映射到当前进程的地址空间中。
- `pgdir_alloc_page`在进程的页表中分配一个物理页面，并将ELF文件中的内容读取并复制到新分配的页中。
- `while`循环对程序段进行清零填充操作，确保页面中未被程序段内容覆盖的部分进行清零填充。

(5) 使用`sysfile_close(fd);`关闭文件。

(6) 调用`mm_map`函数为用户进程设置用户栈，并将参数放入用户栈中。用户栈的位置在用户虚空间的顶端，大小为 256 个页。至此, 进程内的内存管理 `vma` 和 `mm` 数据结构已经建立完成。

(7) 设置当前进程的`mm`、`cr3`:将`mm->pgdir`赋值到`cr3`寄存器中， 即更新了用户进程的虚拟内存空间。

(8) 在用户栈中设置`uargc`和`uargv`： `uargc`和`uargv`被定义为指向整数和指向指针的指针，用于在用户栈中存储参数个数和参数列表的地址。

(9) 为用户栈设置陷阱帧。

(10) 如果上述步骤失败，则应清理环境。

```
static int
load_icode(int fd, int argc, char **kargv) {
    assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);

    if (current->mm != NULL) {//检查当前进程的内存管理结构mm是否为空
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
//------------------------------创建一个新的内存管理结构mm-------------------------
--
    struct mm_struct *mm;//创建一个新的内存管理结构mm
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {//创建新的页目录pgdir
```

```
        goto bad_pgdir_cleanup_mm;
    }

    struct Page *page;

    struct elfhdr __elf, *elf = &__elf;
    if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {//读取
ELF文件的头部信息
        goto bad_elf_cleanup_pgdir;
    }

    if (elf->e_magic != ELF_MAGIC) {//验证文件格式是否正确
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    struct proghdr __ph, *ph = &__ph;//proghdr结构体存储可执行文件的程序头表中的每个程序
段的信息
    uint32_t vm_flags, perm, phnum;
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {//遍历ELF文件的程序头表
        off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;//计算当前程序
头的偏移量
//----------读取可执行文件的程序头表，并对每个程序段进行必要的检查，以确保程序段的合法性和正确
性---------------
        if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0)
{//读取可执行文件中的程序头表
            goto bad_cleanup_mmap;
        }
        if (ph->p_type != ELF_PT_LOAD) {//检查程序段的类型是否为ELF_PT_LOAD
        //如果不是，则跳过当前程序段的处理
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {//检查程序段的文件大小是否大于内存大小
        //如果是，则说明文件大小超出了内存大小，这是一个错误情况
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {
            // continue ;
            // do nothing here since static variables may not occupy any space
        }
//------------------设置vm_flags和perm-----------------
        vm_flags = 0, perm = PTE_U | PTE_V;//设置vm_flags和perm
        //这两个变量用于表示页面的权限和标志，根据程序段的标志位（p_flags）来设置
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        // modify the perm bits here for RISC-V
        if (vm_flags & VM_READ) perm |= PTE_R;
        if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
        if (vm_flags & VM_EXEC) perm |= PTE_X;

//-----------调用mm_map函数，创建新的虚拟内存区域，并映射到对应的物理内存，将程序段映射到当前
进程的地址空间中------
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
```

```
            goto bad_cleanup_mmap;
        }
        off_t offset = ph->p_offset;
        size_t off, size;
        uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);

        ret = -E_NO_MEM;

        end = ph->p_va + ph->p_filesz;
        while (start < end) {
            //在进程的页表中分配一个物理页面，并将其映射到虚拟地址la处
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                ret = -E_NO_MEM;
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {//如果结束地址小于la，说明当前页面不需要完全填充，需要调整size
的大小。
                size -= la - end;
            }
            /*load_icode_read函数将程序段的内容从文件中读取到内存中。
            load_icode_read函数用于从文件fd中的偏移量offset处读取size字节的数据，然后将数
据复制到page2kva(page) + off的内存地址处。这样就实现了将程序段的内容从文件中读取到内存中的操
作*/
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset))
!= 0) {
                goto bad_cleanup_mmap;
            }
            start += size, offset += size;
        }
        end = ph->p_va + ph->p_memsz;//更新结束地址为程序段的虚拟地址加上内存大小

        if (start < la) {//如果起始地址小于当前页面的起始地址，说明当前页面不需要完全填充，
需要进行清零填充操作
            /* ph->p_memsz == ph->p_filesz */
            if (start == end) {//如果起始地址等于结束地址，说明当前页面不需要填充任何内
容，直接跳过
                continue ;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;//计算偏移量off,需要填充
的大小size
            if (end < la) {//如果结束地址小于当前页面的结束地址，需要调整size的大小
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);//使用memset函数将页面中未被程序段
内容覆盖的部分进行清零填充
            start += size;
            assert((end < la && start == end) || (end >= la && start == la));
        }
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                ret = -E_NO_MEM;
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
```

```
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
        }
    }
    sysfile_close(fd);//关闭fd

//---------------------------调用mm_map函数设置用户栈，并将参数放入用户栈中---------------
---------------
    vm_flags = VM_READ | VM_WRITE | VM_STACK;//设置用户栈的虚拟内存标志
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
!= 0) {//将用户栈映射到虚拟地址USTACKTOP - USTACKSIZE处
        goto bad_cleanup_mmap;
    }
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);//在
用户栈的顶部分配一个物理页面，并将其映射到虚拟地址USTACKTOP - PGSIZE处，大小为 256 个页，即
1MB
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);

    mm_count_inc(mm);//增加进程的引用计数
    current->mm = mm;//将当前进程的mm指向新的进程的内存管理结构
    current->cr3 = PADDR(mm->pgdir);//将当前进程的页表基址设置为新的进程的页表基址
    lcr3(PADDR(mm->pgdir));//切换到新页表

//------------------------------在用户栈中设置uargc和uargv-------------------------
--------
    //setup uargc, uargv
    uint32_t argv_size=0, i;
    for (i = 0; i < argc; i ++) {//计算参数总大小
        argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }

    //计算用户栈的栈顶位置，即参数所占空间的大小加上一个整型变量的大小
    uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
    //定义指向参数的指针数组的指针
    char** uargv=(char **)(stacktop  - argc * sizeof(char *));

    argv_size = 0;
    for (i = 0; i < argc; i ++) {
        //将每个参数复制到用户栈上，并将对应的指针存储到指针数组中
        uargv[i] = strcpy((char *)(stacktop + argv_size ), kargv[i]);
        argv_size +=  strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }

    stacktop = (uintptr_t)uargv - sizeof(int);//计算用户栈的栈顶位置，即指向指针数组的
指针所在位置减去一个整型变量的大小
    *(int *)stacktop = argc;//将参数数量存储到用户栈的栈顶位置。
//--------------------------------为用户栈设置陷阱帧--------------------------------
    struct trapframe *tf = current->tf;//获取当前进程的陷阱帧指针
    // Keep sstatus
    uintptr_t sstatus = tf->status;//保存陷阱帧中的状态寄存器
```

```
    memset(tf, 0, sizeof(struct trapframe));//清空陷阱帧

    //设置用户栈指针 (sp)
    tf->gpr.sp = USTACKTOP;
    //设置程序入口地址 (epc)
    tf->epc = elf->e_entry;
    //将状态寄存器中的SPIE和SPP位清零，以切换到用户模式
    tf->status = read_csr(sstatus) & ~(SSTATUS_SPIE | SSTATUS_SPP);

    ret = 0;
out:
    return ret;
//----------清理环境---------
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

`alloc_proc`函数

- `proc->rq = NULL;`：`rq`指向调度队列的指针，初始化为`NULL`。
- `list_init(&(proc->run_link));`：`run_link`是一个链表节点，用于将进程连接到调度队列中。使用`list_init`将其初始化为一个空链表。
- `proc->time_slice = 0;`：进程的时间片初始化为`0`
- `proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc->lab6_run_pool.parent = NULL;`：初始化`lab6`中的一个二叉堆，将左子节点、右子节点和父节点初始化为`NULL`。
- `proc->lab6_stride = 0;`：`Lab6`中使用的步长初始化为`0`。
- `proc->lab6_priority = 0;`：Lab6中使用算法的优先级初始化为`0`。
- `proc->filesp = NULL;`：指向文件描述符表的指针初始化为`NULL`，表示该进程还没有打开任何文件。

```
alloc_proc(void) {
    // 为proc分配一个大小为struct proc_struct的内存块，并将其地址赋给proc
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    // 如果proc不为NULL，说明内存分配成功
    if (proc != NULL) {
        // 初始化proc结构体的各个字段
        proc->state = PROC_UNINIT;  // 进程状态初始化为PROC_UNINIT
        proc->pid = -1;  // 进程ID初始化为-1
        proc->runs = 0;  // 运行次数初始化为0
        proc->kstack = 0;  // 内核栈指针初始化为0
        proc->need_resched = 0;  // 是否需要调度初始化为0
        proc->parent = NULL;  // 父进程指针初始化为NULL
        proc->mm = NULL;  // 内存管理结构体指针初始化为NULL
        // 对proc的context字段进行初始化，使用memset将其内存清零
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;  // 中断帧指针初始化为NULL
```

```
        proc->cr3 = boot_cr3;  // 页目录表物理地址初始化为boot_cr3
        proc->flags = 0;  // 进程标志初始化为0
        // 对proc的name字段进行初始化，使用memset将其内存清零
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0;  // 等待状态初始化为0
        proc->cptr = proc->optr = proc->yptr = NULL;  // 子进程、兄弟进程和父进程指针
初始化为NULL
        proc->rq = NULL;  // 运行队列指针初始化为NULL
        // 对proc的run_link字段进行初始化，使用list_init将其初始化为一个空链表
        list_init(&(proc->run_link));
        proc->time_slice = 0;  // 时间片初始化为0
        // Lab6中使用的相关字段初始化为NULL
        proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc-
>lab6_run_pool.parent = NULL;
        proc->lab6_stride = 0;  // Lab6中使用的步长初始化为0
        proc->lab6_priority = 0;  // Lab6中使用的优先级初始化为0
        proc->filesp = NULL;  // 文件描述符表指针初始化为NULL
    }
    // 返回proc指针，如果proc为NULL，表示内存分配失败
    return proc;
}
```

`proc_run`函数

添加`flush_tlb`函数，用于刷新TLB。当操作系统进行进程切换时，页表也需要相应地切换，以确保正确的虚拟地址映射到正确的物理地址。`flush_tlb()`清空TLB中的旧映射，以便它可以重新加载新的页表映射，从而确保新的虚拟地址能够正确映射到新的物理地址。

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            lcr3(next->cr3);
            flush_tlb();
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);

    }
}
```

`do_fork`函数

使用`copy_files()`复制当前进程的文件描述符表到新进程，如果复制失败，跳转到`bad_fork_cleanup_kstack`标签处，释放新进程的文件描述符表。

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
  ...
```

```
    if (copy_files(clone_flags, proc) != 0) { // 复制当前进程的文件描述符表到新进程，如果复制失败，跳转到bad_fork_cleanup_kstack标签处
        goto bad_fork_cleanup_kstack;
    }

    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_fs;
    }
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        set_links(proc);
        hash_proc(proc);
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

    ret = proc->pid;

fork_out:
    return ret;

bad_fork_cleanup_fs:  //如果复制内存映像失败，释放新进程的文件描述符表
    put_files(proc);
bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

make qemu:

```
Breakpoint
user sh is running!!!
Hello world!!.
I am process 3.
hello pass.
Hello, I am process 4.
Back in process 4, iteration 0.
Back in process 4, iteration 1.
Back in process 4, iteration 2.
Back in process 4, iteration 3.
Back in process 4, iteration 4.
All done in process 4.
yield pass.
sleep 1 x 100 slices.
sleep 2 x 100 slices.
sleep 3 x 100 slices.
sleep 4 x 100 slices.
sleep 5 x 100 slices.
sleep 6 x 100 slices.
sleep 7 x 100 slices.
sleep 8 x 100 slices.
sleep 9 x 100 slices.
sleep 10 x 100 slices.
use 10000 msecs.
sleep pass.
$
```

make grade:

扩展练习 Challenge1：完成基于“UNIX的PIPE机制”的设计方案

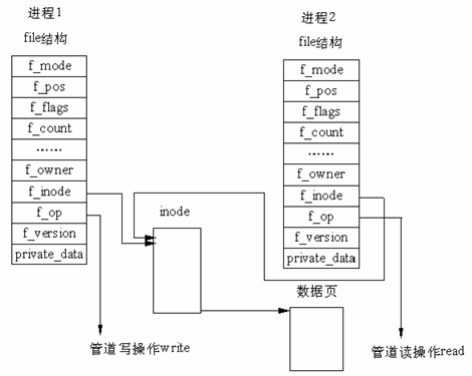
如果要在ucore里加入UNIX的管道（Pipe)机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个(或多个）具体的C语言struct定义。在网络上查找相关的Linux资料和实现，请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。）

数据结构

管道机制：管道是由内核管理的一个缓冲区，相当于我们放入内存中的一个纸条。

- 管道的一端连接一个进程的输出。这个进程会向管道中放入信息。管道的另一端连接一个进程的输入，这个进程取出被放入管道的信息。
- 一个缓冲区不需要很大一般为4K大小，它被设计成为**环形的数据结构**，以便管道可以被循环利用。
- 当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。
- 当管道被放满信息的时候，尝试放入信息的进程会等待，直到另一端的进程取出信息。
- 当两个进程都终结的时候，管道也自动消失

在Linux中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的file结构和VFS的索引节点inode。**通过将两个file结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现的。**



接口与操作

- `pipe_read`：从管道中读取数据。
  - 检查管道，是否没有数据或内存被锁定。如果是且无阻塞，立即返回错误。
  - 使用互斥锁锁定管道 。复制物理内存中的字节而读出数据。解锁管道。
  - 如果没有成功读取到数据，当前进程就进入睡眠状态，直到有新的事件唤醒它。
- `pipe_write`：向管道中写入数据。
  - 检查 VFS 索引节点中的信息，如果内存不够或者内存被读程序锁定，且非阻塞，立即返回错误信息。
  - 锁定内存。将字节复制到 VFS 索引节点指向的物理内存而写入数据。写入之后，解锁内存，所有休眠在索引节点的读取进程会被唤醒。
  - 如果没有成功写入，写入进程就休眠在 VFS 索引节点的等待队列中，接下来，内核将调用调度程序，而调度程序会选择其他进程运行。

```
//从管道中读取数据
static ssize_t pipe_read(struct kiocb *iocb, struct iov_iter *to)
{
    //锁定管道-->复制物理内存中的字节而读出数据，数据保存在iocb中,字节数保存在ret中-->解锁管道
    //唤醒等待队列的进程
    //如果没有数据或有人在write且非阻塞，立即返回错误信息


    //如果成功读取到数据 (ret > 0)，将读取等待队列 pipe->read_wait 添加到管道的等待队列 pipe->wait 中
    //如果有人在write，将错误等待队列pipe->error_wait加入等待集合中。
    //如果没有成功读取到数据 (ret <= 0)，就调用 schedule() 函数让当前进程进入睡眠状态，直到有新的事件唤醒它。

    //从等待集合中移除pipe->error_wait和pipe->read_wait。

}
//向管道中写入数据。
static ssize_t pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
```

```
    //锁定管道-->将字节复制到 VFS 索引节点指向的物理内存而写入数据-->解锁管道
    //唤醒等待队列的进程
    //如果管道的内存不够或者管道被read操作锁定且非阻塞，立即返回错误信息

    //如果成功写入数据 (ret > 0)，将读取等待队列 pipe->write_wait 添加到管道的等待队列
pipe->wait 中
    //如果有人在reade，将错误等待队列pipe->error_wait加入等待集合中。
    //如果没有成功读取到数据 (ret <= 0)，就调用 schedule() 函数让当前进程进入睡眠状态，
直到有新的事件唤醒它。

    //从等待集合中移除pipe->error_wait和pipe->read_wait。


}
```

同步互斥问题处理

内核使用了**锁、等待队列和信号量**

读写操作都需要互斥锁保证读写操作的原子性。

等待队列保证了不满足条件的进程可以在等待队列里休眠，直到满足条件被唤醒后，再执行相关操作。

信号量用来进行等待和唤醒的操作，可以实现线程或进程之间的协调，确保它们以指定的顺序执行。

扩展练习 Challenge2：完成基于“UNIX的软连接和硬连接机制”的设计方案

> 如果要在 `ucore` 里加入 `UNIX` 的软连接和硬连接机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个（或多个）具体的 `C` 语言 `struct` 定义。在网络上查找相关的 `Linux` 资料和实现，请在实验报告中给出设计实现 `UNIX` 的软连接和硬连接机制 的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。

软连接和硬连接允许文件在文件系统中的多个位置共享相同的物理存储空间或者引用相同的文件。对于实现这两种链接机制，需要考虑以下设计方面：

1. Inode 结构的扩展：

在 UNIX 文件系统中，每个文件都有一个相应的 inode（索引节点），用于存储文件的元数据信息。为了支持链接机制，需要对 inode 结构进行扩展，以存储链接相关的信息。

2. 硬链接和软链接的区别：

- **硬链接**：
    - 硬链接是文件系统中同一个文件的多个名称。多个文件名都指向同一个 inode，并且在文件系统中没有区别对待它们。每个硬链接都有一个独立的目录项，删除一个硬链接并不会影响其他链接。
    - 在设计中，需要考虑对 inode 结构的扩展，以记录连接到该 inode 的硬链接数量。
- **软链接**：

- 软链接是一个特殊的文件，它包含指向另一个文件或目录的路径名。软链接类似于快捷方式，在打开时被解析为实际文件或目录。
- 在设计中，需要为软链接引入新的数据结构以存储软链接的目标路径信息。

扩展的 Inode 结构和链接结构：

- **Inode 扩展结构**（`inode_extension`）：
  - 包含链接计数器（`nlinks`），记录连接到该 inode 的硬链接数量。
  - 包含指向软链接结构的指针，用于存储软链接的目标路径信息。
- **软链接结构**（`symlink`）：
  - 包含一个字符数组来存储软链接的目标路径。

接口和操作：

- **创建链接**：
  - 创建硬链接：增加链接计数器，将新链接与现有 inode 关联。
  - 创建软链接：存储软链接信息，并创建一个指向目标路径的符号链接。
- **删除链接**：
  - 删除硬链接：减少链接计数器，如果计数器为零则释放 inode。
  - 删除软链接：从文件系统中删除软链接。
- **读取链接信息**：
  - 读取硬链接：获取链接计数器的值。
  - 读取软链接：读取软链接的目标路径。
- **修改链接计数器**：
  - 对连接计数器进行递增和递减操作需要确保同步访问，避免并发修改导致的问题。

同步与异常处理：

- **同步机制**：
  - 引入信号量、互斥锁等同步机制，确保对链接操作的原子性和线程安全性。
- **异常情况处理**：
  - 处理链接目标不存在、软链接路径无效或其他异常情况，以确保链接操作的可靠性和安全性。

数据结构

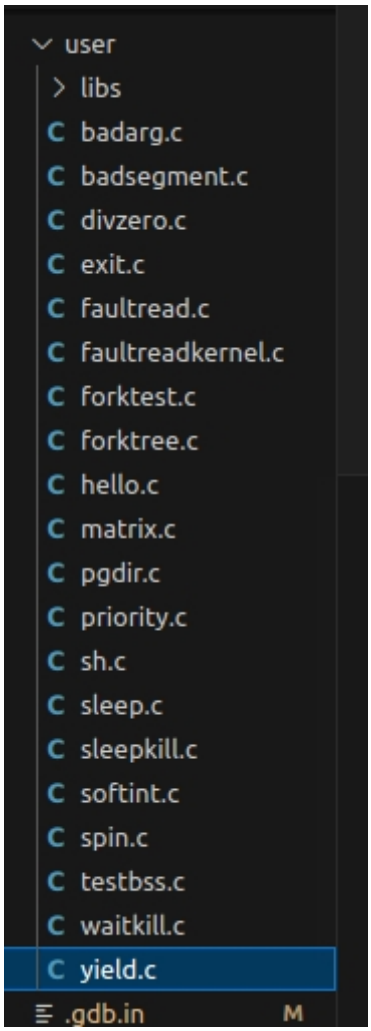
```
#define MAX_PATH_LENGTH 256
#define MAX_INODE_EXTENSIONS 1000  // 假设最多有1000个扩展inode结构

// 软链接结构
struct symlink {
    char target_path[MAX_PATH_LENGTH];  // 软链接的目标路径
};

// 扩展的Inode结构
struct inode_extension {
    int nlinks;  // 被链接计数
    struct symlink *symlink;  // 指向软链接结构的指针
};

// Inode结构
```

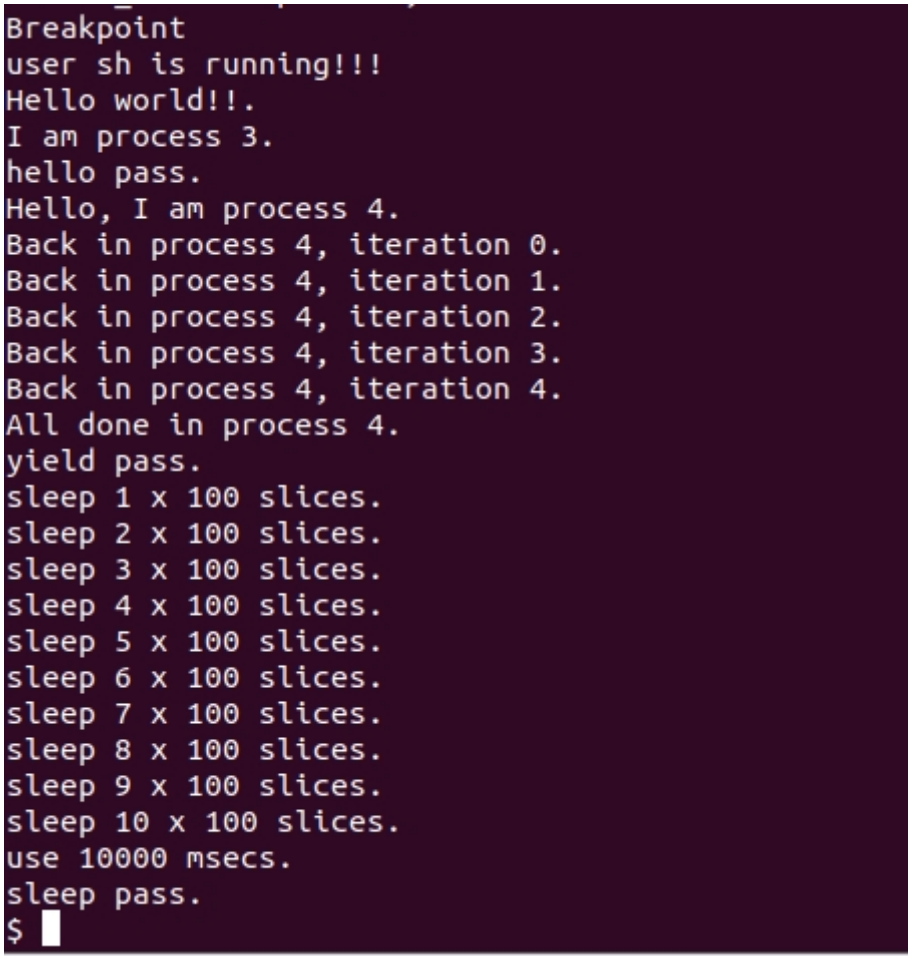
```
struct inode {
    // 其他 inode 相关信息...

    // 连接计数器，用于记录硬链接数
    int link_count;

    // 指向扩展信息的指针
    struct inode_extension *extension;

    // 锁，用于同步访问和修改 inode 结构
    struct semaphore inode_lock;

    // 其他属性...
};
```

接口代码

```
//创建硬链接
int create_hard_link(const char* existing_path, const char* new_link_path) {
    // 通过 existing_path 找到相应的 inode

    // 分配一个新的目录项，指向相同的 inode

    // 增加链接计数器

    return 0; // 返回成功或失败的状态码
}

//创建软连接
int create_soft_link(const char* target_path, const char* link_path) {
    // 创建一个新的 inode

    // 分配一个新的目录项，指向新的 inode

    // 存储软链接信息到 inode 的扩展结构中

    return 0; // 返回成功或失败的状态码
}

//删除链接
int unlink(const char* path) {
    // 通过 path 找到相应的 inode

    // 如果是硬链接，则减少链接计数器
    // 如果是软链接，则释放相关资源

    // 删除对应的目录项

    return 0; // 返回成功或失败的状态码
}

//读取链接的目标路径
ssize_t readlink(const char* path, char* buf, size_t bufsize) {
    // 通过 path 找到相应的 inode
```

```
    // 如果是软链接，读取目标路径信息到 buf 中

    return 0; // 返回读取的字节数或错误码
}
```





知识点

文件系统的初始化流程

kern\init\init.c中的kern_init函数，增加了对fs_init函数的调用。fs_init函数就是文件系统初始化的总控函数。它的调用函数如下图所示。

下面对上面这些函数做一些解释

首先要知道，信号量结构体semaphore_t，包括：

- 信号量value。它的作用是控制访问的次序以及允许访问的数量。通常将 value 初始化为资源可用的数量， 每次线程或进程访问该资源时将 value 减一，离开该资源时将 value 加一。 当 value 为 0 时，其他线程或进程需要等待进程离开资源后才能访问。
- 等待该信号量的线程链表wait_queue。

接着可以看到kern_init调用了下面三个函数，这三个初始化函数联合在一起，协同完成了整个虚拟文件系统、SFS文件系统和文件系统对应的设备（键盘、串口、磁盘）的初始化工作。

- 虚拟文件系统初始化函数vfs_init：建立了一个device list双向链表vdev_list，为后续具体设备（键盘、串口、磁盘）以文件的形式呈现建立查找访问通道。
    - 用sem_init()：初始化信号量bootfs_sem，它的信号量为1，表示只能允许一个线程或进程使用，并且它的等待链表为空。
    - 用vfs_devlist_init()：
        - 初始化一个空的双向链表vdev_list
        - 初始化信号量vdev_list_sem，信号量为1
- 与文件相关的设备初始化函数dev_init：
    - 经过一些函数的嵌套调用，最后由disk0/stdin/stdout_device_init完成对具体设备的初始化，把它们抽象成一个设备文件，并建立对应的inode数据结构，最后把它们链入到vdev_list中。
- Simple FS文件系统的初始化函数sfs_init
    - 完成对Simple FS的初始化工作，经过一些函数的嵌套调用，最后由vfs_mount函数（实际调用mountfunc函数）和sfs_do_mount函数把此实例文件系统挂在虚拟文件系统中，从而让ucore的其他部分能够通过访问虚拟文件系统的接口来进一步访问到SFS实例文件系统。

ucore的文件系统构架组成

- 通用文件系统访问接口层：这一层访问接口让应用程序能够通过一个简单的接口获得 ucore 内核的文件系统服务。
- 文件系统抽象层：
  - 作用：向上提供一个一致的接口给内核其他部分（文件系统相关的系统调用实现模块和其他内核功能模块）访问。向下把不同文件系统的对外共性接口提取出来，放进同样的抽象函数指针列表中，这样屏蔽了不同文件系统的实现细节。
  - 构成：
    - file接口：用结构体file表示，保存进程在内核中直接访问的文件相关信息（文件的当前状态、可读可写、访问文件的当前位置、该文件对应的内存inode指针等）。
    - dir接口：用结构体files_struct表示，保存进程访问文件的数据（进程访问文件的数据、进程打开文件的数组、一个信号量以确保fs_struct的互斥访问）
      - 当创建一个进程后，该进程的 files_struct 将会被初始化或复制父进程的 files_struct。
      - 当用户进程打开一个文件时，将从 fd_array 数组中取得一个空闲 file 项，然后会把此 file 的inode指针设置为此文件的 inode 的起始地址。
    - inode接口：用结构体inode表示，保存不同文件系统的特定索引节点信息，以避免进程直接访问具体文件系统。这个结构体包含了：
      - 设备文件系统与SFS文件系统的内存inode信息
      - 此inode所属文件系统类型
      - 此inode的引用计数
      - 打开此inode对应文件的个数
      - 抽象的文件系统指针
      - 抽象的inode函数in_ops，可以说对常规文件、目录、设备文件所有操作都可以通过这个抽象函数实现。用户只需要调用这个函数，就能执行前面的所有操作，而不需要知道每种访问操作的具体实现。
      - ps:每个 `inode` 也占用一个 block
- Simple FS 文件系统层：一个基于索引方式的简单文件系统实例。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数。向下访问外设接口
  - 这里该文件系统以block （4K，与内存 page 大小相等）为基本单位。
  - sfs的布局如下：

    - <./3.png" alt="image-20231223004524368" style="zoom: 33%;" />
        - 第0个块为超级块，用结构体sfs_super表示，保存文件系统的所有关键参数，当计算机被启动或文件系统被首次接触时，超级块的内容就会被装入内存。（魔数、已使用和未使用的块数等）这里的魔数其实是特定文件格式的内存特征，以判断磁盘镜像是否是合法的 SFS img。
        - 第1个块为根目录索引节点
        - 第2个块为空闲块映射，它的每一个比特位表示对应的块是否空闲。
        - 磁盘索引节点，用sfs_disk_inode表示，用一级或者二级索引，记录文件内容的存储位置。保存在硬盘中，在进程需要时才被读入到内存中。
        - 内存索引节点，用sfs_inode表示，包含了磁盘索引节点的信息，在打开一个文件后才创建的，如果关机则相关信息都会消失。
            - sfs_bmap_load_nolock ()： 根据目录的inode和inode中的逻辑块索引，找到磁盘块的编号。
            - sfs_bmap_truncate_nolock ()：释放文件末尾的磁盘块。当一个文件或目录被删除时，sfs 会循环调用该函数直到释放所有的数据页。
            - sfs_dirent_read_nolock ()：从包含该文件项的磁盘块中读取该文件项
            - sfs_dirent_search_nolock()：搜索指定文件名的文件项，返回该文件项的索引和文件的磁盘块号。
- 外设接口层：向上提供 device 访问接口屏蔽不同硬件细节。向下实现访问各种具体设备驱动的接口，比如 disk 设备接口/串口设备接口/键盘设备接口等。

设备

- `vfs_dev_t` 数据结构：把 `device` 和 `inode` 联通起来.让文件系统通过一个链接 `vfs_dev_t` 结构的双向链表找到 `device` 对应的 `inode` 数据结构，一个 `inode` 节点的成员变量 `in_type` 的值是 `0x1234`，则此 `inode` 的成员变量 `in_info` 将成为一个 `device` 结构。这样 `inode` 就和一个设备建立了联系，这个 `inode` 就是一个设备文件。
- `vdev_list` 双向链表：设备链表，通过访问 `vdev_list` 链表，可以找到 `ucore` 能够访问的所有设备文件。
- `stdin` 设备：时钟中断，每次时钟中断检查控制台是否输入新的字符，及时把它放在 `stdin` 缓冲区里。
- `stdout` 设备：只需要支持写操作，调用 `cputchar()` 把字符打印到控制台。
- `disk0` 设备：封装了一下 `ramdisk` 的接口，每次读取或者写入若干个 `block`。

`OPEN` 系统调用的执行过程

- 通用文件访问接口层：进入内核态，执行 `sysfile_open()` 函数。把位于用户空间的字符串 `__path` 拷贝到内核空间中的字符串 `path` 中，然后调用了 `file_open`， `file_open` 调用了 `vfs_open`, 使用了 `VFS` 的接口, 进入到文件系统抽象层的处理流程完成进一步的打开文件操作中。
- 文件系统抽象层
    - 调用 `file_open` 函数，给这个即将打开的文件分配一个 `file` 数据结构的变量。
    - 调用 `vfs_open` 函数来找到 `path` 指出的文件所对应的基于 `inode` 数据结构的 `VFS` 索引节点 `node`。 `vfs_open` 函数需要完成两件事情：通过 `vfs_lookup` 找到 `path` 对应文件的 `inode`；调用 `vop_open` 函数打开文件。
    - `vfs_lookup` 函数首先调用 `get_device` 函数，并进一步调用 `vfs_get_bootfs` 函数（其实调用了）来找 到根目录“/”对应的 `inode`。通过调用 `vop_lookup` 函数来查找到根目录“/”下对应文件 `sfs_filetest1` 的索引节点，如果找到就返回此索引节点。

- SFS 文件系统层
  - sfs_lookup 有三个参数：node，path，node_store。其中 node 是根目录“/”所对应的 inode 节点； path 是文件 sfs_filetest1 的绝对路径/sfs_filetest1，而 node_store 是经过查找获得的 sfs_filetest1 所对应的 inode 节点。 sfs_lookup 函数以“/”为分割符，从左至右逐一分解 path 获得各个子目录和最终文件对应的 inode 节点。
  - sfs_lookup_once 调用 sfs_dirent_search_nolock 函数来查找与路径名匹配的目录项，如果找到目录项，则根据目录项中记录的 inode 所处的数据块索引值找到路径名对应的 SFS 磁盘 inode， 并读入 SFS 磁盘 inode 对的内容，创建 SFS 内存 inode。

Read 系统调用执行过程

- 通用文件访问接口层
  - 先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数：read->sys_read->syscall，从而引起 系统调用进入到内核态。
  - 到达内核态后，通过中断处理例程，需要调用 sys_read 内核函数，并进一步调用 sysfile_read 内核函数，进入到文件系统抽象层处理流程完成进一步读文件的操作。
- 文件系统抽象层
  - 检查错误，即检查读取长度是否为 0 和文件是否可读。
  - 分配 buffer 空间，即调用 kmalloc 函数分配 4096 字节的 buffer 空间。
  - 读文件过程
    - 实际读文件：循环读取文件，每次读取 buffer 大小。然后调用 file_read 函数将文件内容读取到 buffer 中，alen 为实际大小。调用 copy_to_user 函数将读到的内容拷贝到用户的内存空间中，调整各变量以进行下一次循环读取，直至指定长度读取完成。最后函数调用层层返回至用户程序，用户程序收到了读到的文件内容。
    - file_read 函数：首先调用 fd2file 函数找到对应的 file 结构，并检查是否可读。调用 filemap_acquire 函数使打开这个文件的计数加 1。调用 vop_read 函数将文件内容读到 iob 中。 调整文件指针偏移量 pos 的值，使其向后移动实际读到的字节数 iobuf_used(iob)。最后调用 filemap_release 函数使打开这个文件的计数减 1，若打开计数为 0，则释放 file。
- SFS 文件系统层
  - sfs_read 函数：先找到 inode 对应 sfs 和 sin，然后调用 sfs_io_nolock 函数进行读取文件操作，最后调用 iobuf_skip 函数调整 iobuf 的指针。
  - sfs_io_nolock 函数：先计算一些辅助变量，并处理一些特殊情况（比如越界），然后有 sfs_buf_op = sfs_rbuf， sfs_block_op = sfs_rblock，设置读取的函数操作。接着进行实际操作，先处理起始的没有对齐到块的部分，再以块为单位循环处理中间的部分，最后处理末尾剩余的部分。每部分中都调用 sfs_bmap_load_nolock 函数得到 blkno 对应的 inode 编号，并调用 sfs_rbuf 或 sfs_rblock 函数读取数据，调整相关变量。完成后如果 offset + alen > din->fileinfo.size，则调整文件大小为 offset + alen 并设置 dirty 变量。
  - sfs_bmap_load_nolock 函数：调用 sfs_bmap_get_nolock 来完成相应的操作。 sfs_rbuf 和 sfs_rblock 函数最终都调用 sfs_rwblock_nolock 函数完成操作，而 sfs_rwblock_nolock 函数调用 dop_io->disk0_io->disk0_read_blks_nolock->ide_read_secs 完成对磁盘的操作。