lab8实验报告

练习0: 填写已有实验

本实验依赖实验2/3/4/5/6/7。请把你做的实验2/3/4/5/6/7的代码填入本实验中代码中有“LAB2”/“LAB3”/“LAB4”/“LAB5”/“LAB6” /“LAB7”的注释相应部分。

经测试，只用填写我们已经做过的LAB2~LAB5实验并进行改进，并采用已有的LAB6进程调度函数代码（需要更改文件名），再完成本次实验即可成功make grade。

对前面的LAB进行补充：

LAB4中proc_run函数：

```
// in process/proc.c
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);
    }
}
```

该函数调用Risc-V汇编中的sfence.vma指令。具体的做法是：如果操作系统修改了页表，那么TLB也需要跟随刷新。S 模式添加 sfence.vma 通知处理器，软件可能已经修改了页表，于是处理器可以相应地刷新转换缓存，由此维护了TLB一致性。

LAB6相关

sched.c文件中的调度初始化函数sched_init(void)调用了default_sched_class变量，而该变量在LAB8中给出的对应函数在LAB6需要我们补全，关注项目代码框架，我们可以使用框架已给出的调度算法。

因此将schedule目录下文件名进行修改：即对应RR算法的被禁用文件重新启用，default_sched_c 改为 default_sched_c.c， 再把default_sched_stride.c 改为 default_sched_stride，阻止其调用需要补全的LAB6代码。这样程序就能调用默认给出的RR调度算法（已经给出），而非LAB6需要补全的stride算法（虽然stride算法的思想也不算十分复杂）。

```
// in schedule/sched.c
void
sched_init(void) {
    list_init(&timer_list);

    sched_class = &default_sched_class;

    rq = &__rq;
    rq->max_time_slice = MAX_TIME_SLICE;
    sched_class->init(rq);

    cprintf("sched class: %s\n", sched_class->name);
}

// in /schedule/default_sched_c(.c)
struct sched_class default_sched_class = {
    .name = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};
```

此外，alloc_proc函数也需要我们再次进行进行初始化添加：

```
        proc->rq = NULL;       //当前的进程在队列中的指针
        list_init(&(proc->run_link));  // 运行队列的指针
        proc->time_slice = 0; // 该进程剩余的时间片，只对当前进程有效
        proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc->lab6_run_po
        proc->lab6_stride = 0;  // 该进程的调度步进值，在 LAB6 使用
        proc->lab6_priority = 0;  // 该进程的调度优先级，在 LAB6 使用
        proc->filesp = NULL;  // 初始化 PCB 下的 fs(进程相关的文件信息)

```

do_fork函数的工作已经给出，我们调用相关函数：

```
    if (copy_files(clone_flags, proc) != 0)
    { // for LAB8
        goto bad_fork_cleanup_kstack;
    }
```

读文件的流程。

根据实验代码，一个独写文件的流程，大概是经过：

- 在一个进程调用了读文件的接口。
- 通过该接口不断递归调用到内核调用抽象层的接口，抽象层是屏蔽了文件系统的具体实现，向操作系统提供了一个统一的函数调用。
- 从抽象层进入到我们这个操作系统的真正的文件系统。
- 文件系统调用硬盘io接口。

在内核初始化的时候，init.c中多了一个fs_init()操作，它干了三件事：

- 初始化vfs，包括给根文件系统bootfs的信号量置为1，同时初始化vfs的设备列表，它的对应的信号量也置为1.
- 设备初始化，主要是将这次实验用到的stdin、stdout和磁盘disk0初始化，用到了一个init_device的宏来完成。
- 初始化sys，将disk0挂载，使其可以被访问和操作。

练习1: 完成读文件操作的实现（需要编码）

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，填写在kern/fs/sfs/sfs_inode.c中 的sfs_io_nolock()函数，实现读文件中数据的代码。

由于本次没有额外的问题，所以我们直接结合代码的关键部分进行分析。我们看到主要处理的就是首部未对齐部分，对齐部分，尾部不完整部分，都是通过sfs_bmap_load_nolock函数获取文件索引编号，然后调用sfs_buf_op完成实际的文件读写操作。

```
assert(din->type != SFS_TYPE_DIR);
```

这里使用 `assert` 断言，确保文件类型不是目录类型，应该是qemu不支持对目录的读写操作。

```
off_t endpos = offset + *alenp, blkoff;
*alenp = 0;
```

我们计算文件的结束位置，并将 `alenp` 归0，后面会用来保存实际读写的长度。

```
if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
    return -E_INVAL;
}
```

之后进行边界检查，以确保偏移位置在合法范围内。

```
if (offset == endpos) {
    return 0;
}
if (endpos > SFS_MAX_FILE_SIZE) {
    endpos = SFS_MAX_FILE_SIZE;
}
if (!write) {
    if (offset >= din->size) {
        return 0;
    }
    if (endpos > din->size) {
        endpos = din->size;
    }
}
```

并处理一些特殊情况，比如偏移位置等于结束位置时，直接返回；对于写操作，如果偏移位置大于等于文件大小，则不需要进行读写操作。

```
int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t blkno, off
int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno, uint32_t nblk
if (write) {
    sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
}
else {
    sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
}
```

在进行操作之前根据读写类型选择相应的操作函数。

以下实现了对文件内容的读写，通过使用不同的函数处理未对齐的块、对齐的块以及最后一个不完整的块。

首先处理未对齐的块

```
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;
    buf += size;

    if (nblks == 0) {
        goto out;
    }

    blkno++;
    nblks--;
}
```

当文件开始位置的有不对齐的情况时，如果偏移位置不是块的起始位置，那么需要从当前位置开始读写，`blkoff` 记录了当前块内的偏移量。如果有多个块需要读写，先处理当前块，然后将偏移量更新到下一个块的起始位置，减少待处理块的数量。

```
if (nblks > 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_block_op(sfs, buf, ino, nblks)) != 0) {
        goto out;
    }

    alen += nblks * SFS_BLKSIZE;
    buf += nblks * SFS_BLKSIZE;
    blkno += nblks;
    nblks -= nblks;
```

```
13 }
```

这部分处理的是文件中对齐的块，直接调用 `sfs_block_op` 函数读写一系列完整的块。

```
1 if ((size = endpos % SFS_BLKSIZE) != 0) {
2     if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
3         goto out;
4     }
5     if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
6         goto out;
7     }
8     alen += size;
9 }
```

对于文件末尾位置的不完整块。计算最后一个块内需要读写的大小，然后调用 `sfs_buf_op` 函数进行读写。

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。

```
1     struct elfhdr __elf, *elf = &__elf;
2     if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
3         goto bad_elf_cleanup_pgdir;
4     }
5
6     if (elf->e_magic != ELF_MAGIC) {
7         ret = -E_INVAL_ELF;
8         goto bad_elf_cleanup_pgdir;
9     }
```

相较于lab5，这里做的改动就是在于读取ELF文件的方式，在没有文件系统时所有的文件都被加载到内存，但此时有了文件系统就要通过文件描述符查找文件，而lab5中就是通过获取ELF在内存中的位置，根据ELF的格式进行解析，而在lab8中则是通过ELF文件的文件描述符调用load_icode_read（）函数来进行解析程序。

并在最后判断了它的magic有没有“砸舞台”（是否是对应的文件格式）。

分段读取二进制文件的过程也需要修改为根据文件描述符进行读取，之前是根据二进制文件指针位置进行计算偏移量并进行读取。这里需要分别对于每段程序的头部和程序的内容分别进行读取。

```
    // (3.4)
    phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;
    if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0)
    {
        goto bad_cleanup_mmap;
    }
    // 省略中间部分
    //(3.6.1) copy TEXT/DATA section of bianry program
    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;
            }
            // memcpy(page2kva(page) + off, from, size);
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset))
            {
                goto bad_cleanup_mmap;
            }
            start += size, offset += size;
        }
```

涉及到lab5中传入参数binary的地方也需要进行一定量的修改。

由于函数的传入参数变为了参数个数和对应的内核栈中参数的指针kargv，因此，我们需要修改对应传入参数的地方，并在设置中断帧之前（第6步之前），计算我们需要多少在哪里设置中断帧的栈顶，并在用户栈中存入参数uargv。

```
   //为用户空间设置trapeframe
    uint32_t argv_size = 0, i;
    //确定传入给应用程序的参数具体应当占用多少空间
    for (i = 0; i < argc; i++)
    {
        argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
        // +1表示字符串结尾的'\0'
```

```
    }

    uintptr_t stacktop = USTACKTOP - (argv_size / sizeof(long) + 1) * sizeof(lon
    // 用户栈顶减去所有参数加起来的长度，与4字节对齐找到真正存放字符串参数的栈的位置
    char **uargv = (char **)(stacktop - argc * sizeof(char *));

    //找到用户参数存放位置后，再进行按位置复制
    argv_size = 0;
    for (i = 0; i < argc; i++)
    {
        uargv[i] = strcpy((char *)(stacktop + argv_size), kargv[i]);
        //复制参数

        argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
    }

    stacktop = (uintptr_t)uargv - sizeof(int); //栈顶再移动一个int位置存放参数个数
    *(int *)stacktop = argc; //存放参数个数
```

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。）

要在 Ucore 中加入类似 UNIX 管道（Pipe）的机制，需要定义以下数据结构和接口：

1. 数据结构：
   - `pipe_t` 结构：表示管道的数据结构，包含管道缓冲区、读写指针等信息。

```
struct pipe_t {
    char buffer[PIPE_BUFFER_SIZE];
    int read_index;
    int write_index;
    semaphore_t mutex; // 用于同步访问管道数据结构的互斥锁
    semaphore_t full;  // 用于同步管道是否满的信号量
    semaphore_t empty; // 用于同步管道是否空的信号量
};

```

需要考虑 `PIPE_BUFFER_SIZE` 控制管道缓冲区的大小，并添加信号量来处理同步和互斥。

2. 接口：
   - `pipe_create()`：创建一个新的管道并返回其句柄。
   - `pipe_read(pipe_t *pipe, void *buffer, size_t size)`：从管道中读取数据到指定缓冲区，返回实际读取的字节数。
   - `pipe_write(pipe_t *pipe, const void *buffer, size_t size)`：将数据写入管道，返回实际写入的字节数。
   - `pipe_close(pipe_t *pipe)`：关闭管道。

```
int pipe_create(pipe_t **pipe);
int pipe_read(pipe_t *pipe, void *buffer, size_t size);
int pipe_write(pipe_t *pipe, const void *buffer, size_t size);
int pipe_close(pipe_t *pipe);

```

这些接口应当能够处理管道的创建、读取、写入和关闭等基本操作。在实现时，需要考虑对管道数据结构的同步和互斥，可以利用信号量等机制。

3. 同步和互斥问题：
   - 使用信号量：可以使用信号量来控制对管道数据结构的访问，包括读写指针的更新以及对缓冲区的访问。
   - 互斥锁：在管道数据结构中使用互斥锁，确保对数据结构的访问是互斥的，防止多个线程同时修改管道数据结构。
   - 信号量用于同步：使用信号量来进行读写之间的同步，确保在数据可用时读取线程能够正常执行，而在管道满时写入线程能够等待。

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。）

要在 Ucore 中实现 UNIX 的软连接和硬连接机制，需要定义以下数据结构和接口：

- 数据结构设计：

1. inode 结构：

```
struct inode {
    // ... 其他属性
    uint32_t i_number;         // inode 编号
    uint16_t i_links_count;    // 连接数（硬连接数）
    // ... 其他属性
};
```

2. 文件描述符表：

```
struct file_descriptor {
    struct inode *inode; // 指向 inode 的指针
    // ... 其他属性
};
```

3. 目录项：

```
struct dirent {
    char name[MAX_FILE_NAME_LEN]; // 文件名
    uint32_t i_number;            // 文件对应的 inode 编号
    // ... 其他属性
};
```

- 接口设计：

4. 创建硬连接：

```
int create_hard_link(const char *existing_path, const char *new_link_path);
```

5. 创建软连接：

```
int create_soft_link(const char *target_path, const char *link_path);
```

6. 删除连接：

```
int unlink(const char *path);
```

7. 读取目录项：

```
1 struct dirent *readdir(const char *dir_path);
```

- 设计概要方案：

1. 硬连接（Hard Link）实现：
   - 当创建硬连接时，增加目标 inode 的链接数（`i_links_count`）。
   - 当删除硬连接时，减少目标 inode 的链接数，如果链接数为 0，则释放 inode。
2. 软连接（Soft Link）实现：
   - 创建软连接时，创建一个新的 inode，该 inode 存储链接信息，并将目标路径保存在 inode 中。
   - 读取软连接时，按照软连接指向的路径读取文件。
3. 目录项管理：
   - 在目录中增加目录项，记录文件名和对应的 inode 编号。
4. 同步互斥处理：
   - 使用互斥锁来保护对 inode、文件描述符表、目录项等数据结构的访问。
5. 错误处理：
   - 考虑可能的错误场景，例如创建不存在文件的硬连接、软连接指向不存在路径等情况。
6. 适应 Ucore 特性：
   - 根据 Ucore 提供的内存管理机制，进行动态内存分配。