OSlab8

组员：

2110729 张弛
2111451 朱景博
2111566 米奕霖

练习0: 经典合并代码

本实验的代码合并环节略显繁杂，由于跳过了 `lab6&lab7` 直接完成本实验，故需要预填充（虽然 `lab8` 在实现上与前两个实验并无直接关联，但奈何试错多次发现完成对 `lab6` 的相关代码合并是 `make grade` 成功的必要条件...）的部分不仅包括已完成的系列实验，还包括没有硬性要求的 `lab6` 相关代码。

具体涉及到的部分如下（需要改动的仅有 `proc.c` 中的相关函数，具体细节见下文 `Ex2` 部分，其他皆照搬 copy 即可）：

```
proc.c
pmm.c
vmm.c
default_scehed_stride.c
check_sync.c (不必要)
monitor.c（不必要）
```

此处仅对 `schedule/default_scehed_stride.c` 中的填充，也就是 `Stride Scheduling` 调度算法的实现进行详细说明：

```
/* default_scehed_stride.c */

#include <defs.h>
#include <list.h>
#include <proc.h>
#include <assert.h>
#include <default_sched.h>

#define USE_SKEW_HEAP 1

/* You should define the BigStride constant here*/
/* LAB6: YOUR CODE */
#define BIG_STRIDE    0x7FFFFFFF /* ??? */

/* The compare function for two skew_heap_node_t's and the
 * corresponding procs*/
static int
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;
}

static void
stride_init(struct run_queue *rq) {
     /* LAB6: YOUR CODE */
     list_init(&(rq->run_list));
     rq->lab6_run_pool = NULL;
     rq->proc_num = 0;
}

static void
stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
#if USE_SKEW_HEAP
     rq->lab6_run_pool =
          skew_heap_insert(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);
#else
     assert(list_empty(&(proc->run_link)));
     list_add_before(&(rq->run_list), &(proc->run_link));
#endif
     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
          proc->time_slice = rq->max_time_slice;
     }
     proc->rq = rq;
     rq->proc_num ++;
}

static void
stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
#if USE_SKEW_HEAP
     rq->lab6_run_pool =
          skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool),
proc_stride_comp_f);
#else
     assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
     list_del_init(&(proc->run_link));
#endif
     rq->proc_num --;
}

static struct proc_struct *
stride_pick_next(struct run_queue *rq) {
     /* LAB6: YOUR CODE */
#if USE_SKEW_HEAP
     if (rq->lab6_run_pool == NULL) return NULL;
     struct proc_struct *p = le2proc(rq->lab6_run_pool, lab6_run_pool);
#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)
          p->lab6_stride += BIG_STRIDE;
     else p->lab6_stride += BIG_STRIDE / p->lab6_priority;
     return p;
}

static void
stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
     if (proc->time_slice > 0) {
          proc->time_slice --;
     }
     if (proc->time_slice == 0) {
          proc->need_resched = 1;
     }
}

struct sched_class default_sched_class = {
     .name = "stride_scheduler",
     .init = stride_init,
     .enqueue = stride_enqueue,
     .dequeue = stride_dequeue,
     .pick_next = stride_pick_next,
     .proc_tick = stride_proc_tick,
};
```

首先是 `stride_init` 函数，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。

```
//stride_init：进行调度算法初始化的函数，在本 stride 调度算法的实现中使用了斜堆来实现优先队列，因此需要对相应的成员变量进行初始化
static void stride_init(struct run_queue *rq) {
     /* LAB6: YOUR CODE */
     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
}
```

然后是入队函数 `stride_enqueue`，此处主要是初始化刚进入运行队列的进程 proc 的 stride 属性，然后比较队头元素与当前进程的步数大小，选择步数最小的运行，即将其插入放入运行队列中去，这里并未放置在队列头部。最后初始化时间片，然后将运行队列进程数目加一。

```
static void stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
```

```
     /* LAB6: YOUR CODE */
#if USE_SKEW_HEAP
     rq->lab6_run_pool = skew_heap_insert(rq->lab6_run_pool, &(proc-
>lab6_run_pool), proc_stride_comp_f);//将新的进程插入到表示就绪队列的斜堆中，该函数的返回
结果是斜堆的新的根
#else
     assert(list_empty(&(proc->run_link)));
     list_add_before(&(rq->run_list), &(proc->run_link));
#endif
     if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
          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;//更新进程的就绪队列
     rq->proc_num ++;//维护就绪队列中进程的数量加一
}
```

其中条件编译涉及到的 `USE_SKEW_HEAP` 定义为 1，因此 `#else` 与 `#endif` 之间的代码将会被忽略。

接着是出队函数 `stride_dequeue`，即完成将一个进程从队列中移除的功能，这里使用了优先队列。最后运行队列数目减一。

```
//stride_dequeue: 将指定进程从就绪队列中删除，只需要将该进程从斜堆中删除掉即可
static void stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
#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
     rq->proc_num --;//维护就绪队列中的进程总数
}
```

里面的代码比较简单，只有一个主要函数 `skew_heap_remove`。该函数实现过程如下：

```
static inline skew_heap_entry_t *skew_heap_remove(skew_heap_entry_t *a,
skew_heap_entry_t *b, compare_f comp)
{
     skew_heap_entry_t *p   = b->parent;
     skew_heap_entry_t *rep = skew_heap_merge(b->left, b->right, comp);
     if (rep) rep->parent = p;

     if (p)
     {
          if (p->left == b)
               p->left = rep;
          else p->right = rep;
          return a;
     }
     else return rep;
}
```

接下来就是进程的选择调度函数 `stride_pick_next`。观察代码，它的核心是先扫描整个运行队列，返回其中 stride 值最小的对应进程，然后更新对应进程的 stride 值，将步长设置为优先级的倒数，如果为 0 则设置为最大的步长。

```
//stride_pick_next: 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
static struct proc_struct *stride_pick_next(struct run_queue *rq) {
     /* LAB6: YOUR CODE */
#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;
}
```

最后是时间片函数 `stride_proc_tick`，主要工作是检测当前进程是否已用完分配的时间片。如果时间片用完，应该正确设置进程结构的相关标记来引起进程切换。这里和之前实现的 `Round Robin` 调度算法一样。

```
//stride_proc_tick: 每次时钟中断需要调用的函数，仅在进行时间中断的ISR中调用
static void stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
     /* LAB6: YOUR CODE */
     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）

> 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。

从 ucore 的角度来看，其文件系统架构包含四类主要的数据结构, 它们分别是：

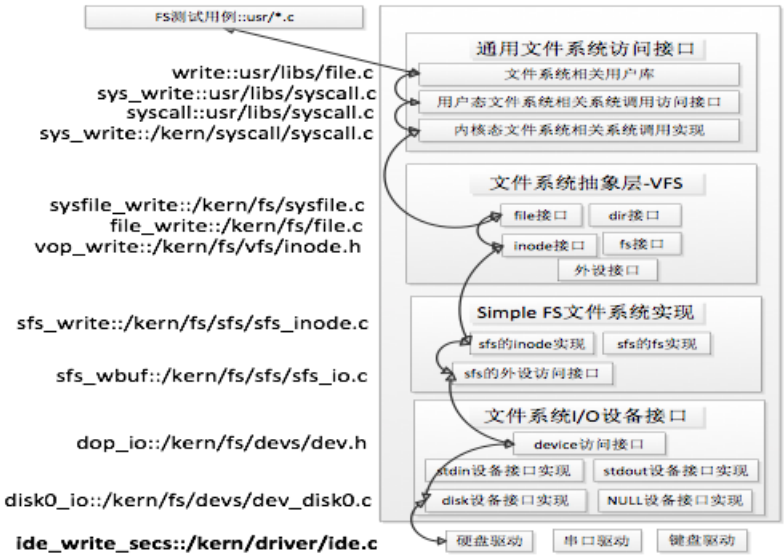
- ***超级块（SuperBlock）***，它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个OS空间。
- ***索引节点（inode）***：它主要从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置。它的作用范围是整个OS空间。
- ***目录项（dentry）***：它主要从文件系统的文件路径的角度描述了文件路径中的特定目录。它的作用范围是整个 OS 空间。
- ***文件（file）***，它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

文件系统，会将磁盘上的文件（程序）读取到内存里面来，在用户空间里面变成进程去进一步执行或其他操作。通过一系列系统调用完成这个过程。

同样根据实验指导书，我们可以了解到 ucore 的文件系统架构主要由四部分组成：

- ***通用文件系统访问接口层***：该层提供了一个从用户空间到文件系统的标准访问接口。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***：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。向下提供一个抽象函数指针列表和数据结构来屏蔽不同文件系统的实现细节。
- ***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
- ***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。

此处我们可以通过下图来理解上述四个部分的关系：



接下来分析下打开一个文件的详细处理的流程。

例如某一个应用程序需要操作文件（增删读写等），首先需要通过文件系统的通用文件系统访问接口层给用户空间提供的访问接口进入文件系统内部，接着由文件系统抽象层把访问请求转发给某一具体文件系统(比如 Simple FS 文件系统)，然后再由具体文件系统把应用程序的访问请求转化为对磁盘上的 block 的处理请求，并通过外设接口层交给磁盘驱动例程来完成具体的磁盘操作。

对应到 ucore 中，具体过程如下：

- 以打开文件为例，首先用户会在进程中调用 `safe_open()` 函数，然后依次调用如下函数 `open->sys_open->syscall`，从而引发系统调用然后进入内核态，然后会由 `sys_open` 内核函数处理系统调用，进一步调用到内核函数 `sysfile_open`，然后将字符串 "/test/testfile" 拷贝到内核空间中的字符串 path 中，并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。
- 在文件系统抽象层，系统会分配一个 file 数据结构的变量，这个变量其实是 `current->fs_struct->filemap[]` 中的一个空元素，即还没有被用来打开过文件，但是分配完了之后还不能找到对应对应的文件结点。所以系统在该层调用了 `vfs_open` 函数通过调用 `vfs_lookup` 找到 path 对应文件的 inode，然后调用`vop_open` 函数打开文件。然后层层返回，通过执行语句 `file->node=node;`，就把当前进程的 `current->fs_struct->filemap[fd]`（即 file 所指变量）的成员变量 node 指针指向了代表文件的索引节点 node。这时返回 fd。最后完成打开文件的操作。
- 在上一步中，调用了 SFS 文件系统层的 `vfs_lookup` 函数去寻找 node，这里在 `sfs_inode.c` 中易知 `.vop_lookup = sfs_lookup`。

```
/* sfs_lookup */
static int
sfs_lookup(struct inode *node, char *path, struct inode**node_store) {
    struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
    assert(*path != '\0' && *path != '/');    //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。
    vop_ref_inc(node);
    struct sfs_inode *sin = vop_info(node, sfs_inode);
    if (sin->din->type != SFS_TYPE_DIR) {
        vop_ref_dec(node);
        return -E_NOTDIR;
    }
    struct inode *subnode;
    int ret = sfs_lookup_once(sfs, sin, path, &subnode, NULL);  //循环进一步调用sfs_lookup_once查找以“test”子目录下的文件“testfile1”所对应的inode节点。

    vop_ref_dec(node);
    if (ret != 0) {
        return ret;
    }
    *node_store = subnode;  //当无法分解path后，就意味着找到了需要对应的inode节点，就可顺利返回了。
    return 0;
}
```

- 观察 `sfs_lookup` 函数传入的三个参数，其中 node 是根目录 `/` 所对应的 inode 节点；path 是文件的绝对路径（例如 `/test/file`），而 node_store 是经过查找获得的 file 所对应的 inode 节点。 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。在本例中是分解出 test 子目录，并调用 `sfs_lookup_once` 函数获得 test 子目录对应的 inode 节点 subnode，然后循环进一步调用 `sfs_lookup_once` 查找以 test 子目录下的文件 testfile1 所对应的 inode 节点。当无法分解 path 后，就意味着找到了 testfile1 对应的 inode 节点，就可以顺利返回了。
- 再进一步观察 `sfs_lookup_once` 函数，它调用 `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。

```
static int
sfs_lookup_once(struct sfs_fs *sfs, struct sfs_inode *sin, const char *name,
struct inode **node_store, int *slot)
{
    int ret;
    uint32_t ino;
    lock_sin(sin);
    {   // find the NO. of disk block and logical index of file entry
        ret = sfs_dirent_search_nolock(sfs, sin, name, &ino, slot, NULL);
    }
    unlock_sin(sin);
    if (ret == 0) {
        // load the content of inode with the the NO. of disk block
        ret = sfs_load_inode(sfs, node_store, ino);
    }
    return ret;
 }
```

至此已对文件打开的处理流程有了大致的了解，可以完成对 `sfs_io_nolock` 的填写了，填充部分与相应解析如下：

```
static int
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（Unaligned Part）**：

```
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 --;
}
```

  - 如果 `offset` 不是块对齐的，需要先处理不对齐的部分。
  - `blkoff` 记录了在当前块内的偏移量。
  - `size` 记录了需要读写的长度，可能是当前块的剩余部分，也可能是直到文件末尾的部分。
  - 调用 `sfs_bmap_load_nolock` 获取块映射，并通过 `sfs_buf_op` 读写对应的块。
  - 更新 `alen`，如果没有块了，直接跳出。
- **对齐块读写（Aligned Block I/O）**：

```
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 --;
}
```

  - 处理对齐的块，循环读写每个块。
  - 调用 `sfs_bmap_load_nolock` 获取块映射，通过 `sfs_block_op` 读写对应的块。
  - 更新 `alen`，移动缓冲区指针和块号，继续处理下一个块。
- **最后不对齐部分处理（Trailing Unaligned Part）**：

```
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;
}
```

- 处理最后不对齐的部分，即文件末尾的不足一个块的部分。
- 调用 `sfs_bmap_load_nolock` 获取块映射，并通过 `sfs_buf_op` 读写对应的块。
- 更新 `alen`。

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。

此处直接在 lab5 的基础上完成适配文件系统的几处修改即可，具体涉及到了`proc.c`中的如下函数：

```
alloc_proc
proc_run
do_fork
load_icode
```

各模块修改与对应解析如下：

`alloc_proc`

```
static struct proc_struct *alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {

        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0;
        proc->cptr = proc->optr = proc->yptr = NULL;

        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;
    }
    return proc;
}
```

与 lab5 相比增加了 `lab6&lab8` 相关字段的初始化，解析如下：

- `proc->rq = NULL;`：
  - `rq` 为指向调度队列（runqueue）的指针，这里将其初始化为 `NULL`，表示该进程还没有被放入任何调度队列。
- `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。`list_init` 用于初始化链表节点，将其指针域设置为空，表示链表为空。
- `proc->time_slice = 0;`：
  - `time_slice` 表示进程的时间片，这里初始化为 `0`，表示该进程的时间片初始值为 0。
- `proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc->lab6_run_pool.parent = NULL;`：
  - 这一段代码是在初始化一个二叉堆（即 lab6 调度算法中涉及到的数据结构），将左子节点、右子节点和父节点初始化为 `NULL`。
- `proc->lab6_stride = 0;`：
  - `lab6_stride` 表示 Stride Scheduling 算法中的步幅值，初始化为 `0`。
- `proc->lab6_priority = 0;`：
  - `lab6_priority` 表示 Stride Scheduling 算法中的优先级，初始化为 `0`。
- `proc->filesp = NULL;`：
  - `filesp` 为指向文件描述符表的指针，这里将其初始化为 `NULL`，表示该进程还没有打开任何文件。

添加完毕上述字段的初始化后，参数在栈中的布局如下所示：

```
| High Address |
----------------
|   Argument   |
|      n       |
----------------
|     ...      |
----------------
|   Argument   |
|      1       |
----------------
|    padding   |
----------------
|   null ptr   |
----------------
|  Ptr Arg n   |
----------------
```

```
|     ...      |
----------------
|  Ptr  Arg 1  |
----------------
|  Arg  Count  | <-- user esp
----------------
| Low  Address |
```

proc_run

```
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(); //flush the tlb before switch_to()
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

添加 `flush_tlb` 语句，以在切换页目录表后对 TLB 刷新。从理论角度这是必要的，因为页目录表的切换可能导致 TLB 中的缓存的映射关系不再有效。通过刷新 TLB，可以确保新的进程的地址空间映射关系得到正确加载，避免出现地址转换错误。

do_fork

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    ...
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;
    assert(current->wait_state == 0);

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }

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

添加 `copy_files` 语句来处理文件描述符的复制。在其实现中，当 `copy_files` 函数返回非零值时，表示文件描述符的复制出现了错误。此时，通过 `goto bad_fork_cleanup_kstack;` 将控制流转移到 `bad_fork_cleanup_kstack` 标签，执行相应的清理工作，确保在出现错误的情况下，不会留下未释放的资源，并返回错误码，以便上层调用处理异常。

`load_icode`

```
static int
load_icode(int fd, int argc, char **kargv) {
    assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        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) {
        goto bad_elf_cleanup_pgdir;
    }

    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }
    struct proghdr __ph, *ph = &__ph;
    uint32_t vm_flags, perm, phnum;
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
        off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;
        if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0)
{
            goto bad_cleanup_mmap;
        }
```

```
        if (ph->p_type != ELF_PT_LOAD) {
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {
            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
        }
        vm_flags = 0, perm = PTE_U | PTE_V;
        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;
        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) {
            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;
            }
            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;

        if (start < la) {
            /* ph->p_memsz == ph->p_filesz */
            if (start == end) {
                continue ;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;
            if (end < la) {
                size -= la - end;
            }
```

```
            memset(page2kva(page) + off, 0, size);
            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);

    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
!= 0) {
        goto bad_cleanup_mmap;
    }
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
    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);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //setup argc, argv
    uint32_t argv_size=0, i;
    for (i = 0; i < argc; i ++) {
        argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }

    uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
    char** uargv=(char **)(stacktop  - argc * sizeof(char *));

    argv_size = 0;
    for (i = 0; i < argc; i ++) {
        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);
    *(int *)stacktop = argc;

    struct trapframe *tf = current->tf;
    // Keep sstatus
    uintptr_t sstatus = tf->status;
```

```
    memset(tf, 0, sizeof(struct trapframe));

    // Set user stack pointer (sp)
    tf->gpr.sp = USTACKTOP;
    // Set entry point of user program (epc)
    tf->epc = elf->e_entry;
    // Set status register
    tf->status = read_csr(sstatus) & ~(SSTATUS_SPIE | SSTATUS_SPP);

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

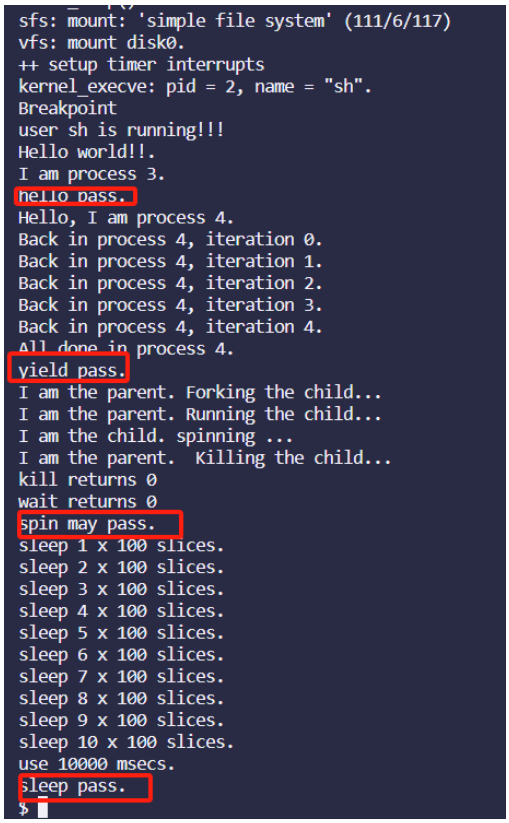
`load_icoed` 函数主要用于从磁盘上（此处即同之前实现的主要区别，不是从内存直接读取）加载用户程序的二进制文件（ELF格式）到当前进程的内存中，并设置进程的上下文，包括内存映射、栈设置、参数传递等。具体的分步解析如下：

1. **创建新的 mm（内存管理结构）**：调用 `mm_create` 创建一个新的 `mm_struct` 结构。
2. **设置新的页目录**：调用 `setup_pgdir` 在新的 `mm` 结构中设置页目录，并将其赋给 `mm->pgdir`。
3. **读取 ELF 头部信息**：使用 `load_icode_read` 从文件中读取 ELF 头部信息，包括程序入口地址等。
4. **验证 ELF 头部**：检查 ELF 头部的魔数是否正确，确保文件是合法的 ELF 文件。
5. **循环处理程序头部（Program Header）**：遍历 ELF 文件中的程序头部，处理每个程序头部。
   a. **判断程序头部类型**：如果是 `ELF_PT_LOAD` 类型，表示这是一个可加载的段。
   b. **根据类型处理**：
      - 计算 `vm_flags` 和 `perm` 标志：这里涉及虚拟内存区域的权限标志。
      - 使用 `mm_map` 创建新的虚拟内存区域，并映射到对应的物理内存。
      - 使用 `pgdir_alloc_page` 分配物理页面，并将 ELF 文件中的内容复制到这些页面上。
6. **关闭文件**：使用 `sysfile_close` 关闭文件。
7. **建立用户栈**：使用 `mm_map` 在用户栈的位置建立一个新的虚拟内存区域，即用户栈。同时，分配了一些页面用于用户栈。
8. **设置当前进程的 mm、cr3 寄存器**：将新创建的 `mm` 结构关联到当前进程，并设置当前进程的页目录地址。
9. **设置用户程序参数**：
   a. 计算参数在用户栈的位置。
   b. 将参数字符串拷贝到用户栈中。
   c. 设置用户栈指针（`sp`）和用户程序入口地址（`epc`）。
   d. 设置状态寄存器 `sstatus`。

10. **清理**：如果发生错误，执行相应的清理工作，包括释放内存、关闭文件等。

效果演示

至此，已基本完成了本实验所有的代码补充与修改，`make qemu` 后可以观察到 sh 的用户界面，接着键入 `hello\yield\spin\sleep` 等指令也可以成功执行（`user`目录下支持的指令较多，此处就不一一展示了）

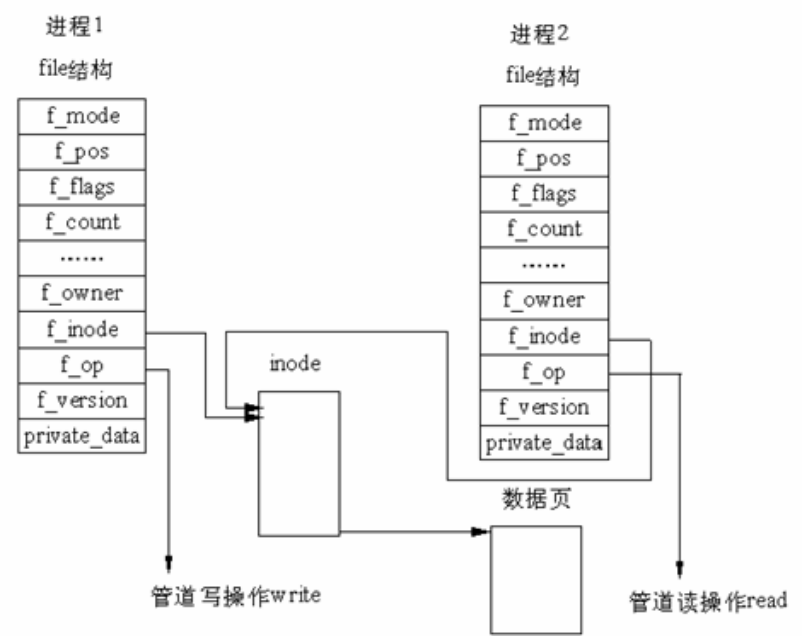


`make grade` 结果如下：

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。

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从而实现的.



管道可以看作是由内核管理的一个缓冲区，一端连接进程 A 的输出，另一端连接进程 B 的输入。进程 A 会向管道中放入信息，而进程 B 会取出被放入管道的信息。当管道中没有信息，进程 B 会等待，直到进程 A 放入信息。当管道被放满信息的时候，进程A会等待，直到进程 B 取出信息。当两个进程都结束的时候，管道也自动消失。管道基于fork 机制建立，从而让两个进程可以连接到同一个 PIPE 上。

基于上述理论基础，我们可以模仿 UNIX 来对 PIPE 机制进行简单设计：

- 首先我们需要在磁盘上保留一定的区域用来作为 PIPE 机制的缓冲区，或者创建一个文件为 PIPE 机制服务
- 对系统文件初始化时将 PIPE 也初始化并创建相应的 inode
- 在内存中为 PIPE 留一块区域，以便高效完成缓存
- 当两个进程要建立管道时，那么可以在这两个进程的进程控制块上新增变量来记录进程的这种属性
- 当其中一个进程要对数据进行写操作时，通过进程控制块的信息，可以将其先对临时文件 PIPE 进行修改
- 当一个进行需要对数据进行读操作时，可以通过进程控制块的信息完成对临时文件 PIPE 的读取
- 增添一些相关的系统调用支持上述操作

在设计方案中，我们将使用一个简单的管道结构 `pipe` 来记录管道的状态，同时扩展进程控制块（PCB）以记录管道信息。此外，需要设计系统调用来支持管道的创建、写入和读取等操作。

数据结构

```
// 管道结构
struct pipe {
    char buffer[PIPE_SIZE];  // 管道缓冲区
    int write_pos;           // 写入位置
    int read_pos;            // 读取位置
    int count;               // 缓冲区中的数据数量
    struct semaphore mutex;  // 用于同步的互斥锁
    struct semaphore empty;  // 缓冲区为空的信号量
    struct semaphore full;   // 缓冲区已满的信号量
};

// 扩展后的进程控制块
struct process_control_block {
    // 其他进程控制块信息...
    struct pipe *pipe;  // 指向管道结构体的指针
```

```
    int pipe_write_fd;  // 记录写端文件描述符
    int pipe_read_fd;   // 记录读端文件描述符
};

// 管道相关的系统调用参数结构
struct pipe_syscall_args {
    int fd;     // 文件描述符
    void *buf; // 数据缓冲区指针
    size_t count; // 读取或写入的字节数
};
```

函数接口

```
// 管道相关系统调用
int pipe(struct process_control_block *pcb, int *fd);
ssize_t read_pipe(struct process_control_block *pcb, int fd, void *buf, size_t
count);
ssize_t write_pipe(struct process_control_block *pcb, int fd, const void *buf,
size_t count);
int close_pipe(struct process_control_block *pcb, int fd);
```

同步互斥处理

在管道结构中，使用信号量 `mutex` 来实现对管道的互斥访问。`empty` 和 `full` 信号量用于在缓冲区空和满时进行等待和唤醒操作。

在写入和读取管道时，使用信号量来进行同步控制，防止多个进程同时操作管道造成数据混乱。当缓冲区为空时，读取进程会等待 `empty` 信号量，而写入进程会在写入后发出 `full` 信号量。当缓冲区已满时，写入进程会等待 `full` 信号量，而读取进程会在读取后发出 `empty` 信号量。

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。）

由于硬链接是有着相同 inode 号仅文件名不同的文件，因此硬链接存在以下几点特性：

- 文件有相同的 inode 及 data block；
- 只能对已存在的文件进行创建；
- 不能交叉文件系统进行硬链接的创建；
- 不能对目录进行创建，只可对文件创建；
- 删除一个硬链接文件并不影响其他有相同 inode 号的文件

而软链接的创建与使用没有类似硬链接的诸多限制：

- 软链接有自己的文件属性及权限等；
- 可对不存在的文件或目录创建软链接；
- 软链接可交叉文件系统；

- 软链接可对文件或目录创建；
- 创建软链接时，链接计数 i_nlink 不会增加；
- 删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软连接被称为死链接（即 dangling link，若被指向路径文件被重新创建，死链接可恢复为正常的软链接）。

且由于存在磁盘上的 inode 信息均存在一个 `nlinks` 变量用于表示当前文件的被链接的计数，因而支持实现硬链接和软链接机制；

- ***创建硬链接 link 时***，为 new_path 创建对应的 file，并把其 inode 指向 old_path 所对应的 inode，inode 的引用计数加1。
- ***创建软链接 link 时***，创建一个新的文件（inode 不同），并把 old_path的内容存放到- 文件的内容中去，将该文件保存在磁盘上时 disk_inode 类型为 `SFS_TYPE_LINK`，再完善对于该类型 inode 的操作即可。
- ***删除一个软链接 B 时***，直接将其在磁盘上的 inode 删掉即可
- ***删除一个硬链接 B 时***，除了需要删除掉 B 的 inode 之外，还需要将 B 指向的文件 A 的被链接计数减1，如果减到了0，则需要将A删除掉；
- 访问软/硬链接的方式是一致的；

具体设计方案如下：

数据结构

在设计硬链接和软链接机制时，需要对 inode 进行扩展，并引入新的数据结构表示链接。

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

// 文件信息结构（inode）扩展
struct inode_extension {
    int nlinks;  // 被链接计数
    struct symlink *symlink;  // 指向软链接结构的指针
};
```

函数接口

```
// 创建硬链接
int create_hardlink(const char *old_path, const char *new_path);
// 创建软链接
int create_symlink(const char *target_path, const char *link_path);
// 删除链接
int unlink(const char *path);
// 读取链接目标
int read_link(const char *path, char *buf, size_t size);
```

- **创建硬链接：**
  - 在创建硬链接时，为新的链接路径 `new_path` 创建一个新的文件（`inode`）。
  - 将新文件的 `inode` 指向被链接路径 `old_path` 所对应的 `inode`。
  - 增加 `old_path` 对应 `inode` 的被链接计数 `nlinks`。
- **创建软链接：**

    - 在创建软链接时，创建一个新的文件（`inode` 不同），并将 `target_path` 的内容存放到文件的内容中。
    - 在磁盘上保存时，文件的 `disk_inode` 类型标记为 `SFS_TYPE_LINK`，需要完善对该类型 `inode` 的操作。
- **删除链接：**
    - 在删除软链接时，直接删除其在磁盘上的 `inode`。
    - 在删除硬链接时，除了删除 `new_path` 的 `inode` 外，还需减少被链接文件 `old_path` 的 `inode` 的被链接计数 `nlinks`。
        - 如果 `nlinks` 减到 0，需要将 `old_path` 的 `inode` 删除。
- **访问链接：**
    - 访问软链接和硬链接的方式是一致的，通过链接路径即可访问到被链接的文件。
- **同步互斥问题处理：**
    - 在修改 `nlinks` 和删除链接等操作上使用锁机制，确保对链接结构的访问是原子的。
    - 对硬链接计数进行适当的同步，防止多个进程同时修改硬链接计数导致数据不一致。