Lab8&Lab6实验报告

1. Lab8

1.1 练习1:完成读文件操作的实现

1.1.1 打开文件的处理流程

首先会在用户态先通过 `open()` 到系统调用 `SYS_open` 进行处理，入口在 `syscall.c` 中，转到 `sysfile_open()` 。

代码块

```
//syscall
static int
sys_open(uint64_t arg[])
{
    const char *path = (const char *)arg[0];
    uint32_t open_flags = (uint32_t)arg[1];
    return sysfile_open(path, open_flags);
}
```

接着 `sysfile_open()` 会拷贝用户路径字符串，在 `sysfile.c` 中通过 `copy_path()` 把用户路径拷到内核缓冲区，然后调用 `file_open(path,flags)` 。首先看到这里的 `sysfile_open()` 函数，这是系统调用 `open()` 的内核入口，用户态路径在用户地址空间里；在 `copy_path()` 成功后， `path` 是内核堆上的一份路径字符串，再调用 `file_open` 真正走VFS打开流程，得到fd或者错误码。接着看 `copy_path()` 里面具体做了什么，首先会取当前进程的用户地址空间，接着分配一个内核缓冲区 `buffer` ，给这个空间加锁，确保在拷贝过程中mm不被并发修改；接着会调 `copy_string` ，在这个里面检查用户指针 `from` 是否在用户地址空间且可读，逐页检查知道找到 `\0` 或超过最大长度，成功则把用户路径复制到内核 `buffer` 中去。

代码块

```
//sysfile.c
int
sysfile_open(const char *__path, uint32_t open_flags) {
    int ret;
    char *path;
    if ((ret = copy_path(&path, __path)) != 0) {
        return ret;
    }
```

```
    ret = file_open(path, open_flags);
    kfree(path);
    return ret;
}

static int
copy_path(char **to, const char *from) {
    struct mm_struct *mm = current->mm;
    char *buffer;
    if ((buffer = kmalloc(FS_MAX_FPATH_LEN + 1)) == NULL) {
        return -E_NO_MEM;
    }
    lock_mm(mm);
    if (!copy_string(mm, buffer, from, FS_MAX_FPATH_LEN + 1)) {
        unlock_mm(mm);
        goto failed_cleanup;
    }
    unlock_mm(mm);
    *to = buffer;
    return 0;

failed_cleanup:
    kfree(buffer);
    return -E_INVAL;
}
```

在 `file.c` 中会有 `file_open()` 的逻辑，在这个里面会分配fd并走VFS。在这里，首先通过 `open_flags & O_ACCMODE` 判断只读或者只写或者读写，接着设置 `readable`或者 `writable` 。设置好模式后，在当前进程的fd表里找空位， `fd_array_alloc(NO_FD, &file)` 分配一个 `struct file` 槽位（状态从 `FD_NONE` 变为 `FD_INIT` ），如果fd表满了或者参数非法，会直接返回错误码。接着会通过VFS打开路径， `vfs_open(path, open_flags, &node)` 会做路径解析，创建并返回 `inode` ；若失败，必须释放刚分配的槽位。接着会初始化文件位置，默认 `file->pos = 0` ，如果 `O_APPEND` ，用 `vop_fstat(node, stat)` 拿到文件大小，把 `file->pos` 设置为 `st_size` ，确保从尾部写入。接着绑定需要 `node` 和读写权限，最后返回 `fd` 给上层，这也是用户看到的文件描述符。

代码块

```
//file.c
int
file_open(char *path, uint32_t open_flags) {
    bool readable = 0, writable = 0;
    switch (open_flags & O_ACCMODE) {
    case O_RDONLY: readable = 1; break;
    case O_WRONLY: writable = 1; break;
```

```
8       case O_RDWR:
9           readable = writable = 1;
10          break;
11      default:
12          return -E_INVAL;
13      }
14      int ret;
15      struct file *file;
16      if ((ret = fd_array_alloc(NO_FD, &file)) != 0) {
17          return ret;
18      }
19      struct inode *node;
20      if ((ret = vfs_open(path, open_flags, &node)) != 0) {
21          fd_array_free(file);
22          return ret;
23      }
24      file->pos = 0;
25      if (open_flags & O_APPEND) {
26          struct stat __stat, *stat = &__stat;
27          if ((ret = vop_fstat(node, stat)) != 0) {
28              vfs_close(node);
29              fd_array_free(file);
30              return ret;
31          }
32          file->pos = stat->st_size;
33      }
34      file->node = node;
35      file->readable = readable;
36      file->writable = writable;
37      fd_array_open(file);
38      return file->fd;
39  }
```

在 `vfsfile.c` 中，会有上面 `vfs_open()` 的实现过程，在这里会进行路径的解析和创建，以及 `inode` 打开位置。这里 `open_flags & O_ACCMODE` 只允许 `O_RDONLY` 、 `O_WRONLY` 和 `O_RDWR` 。如果是只写或者读写模式， `can_write = 1` ；如果设置了 `O_TRUNC` 但没写权限，直接返回 `-E_INVAL` 。接着会解析 `O_EXCL` 和 `O_CREAT` 的语义， `excl = O_EXCL` ， `create = O_CREAT` 。接着会使用 `vfs_lookup` 找已有 `inode` ，如果找不到，就找父目录，然后使用 `vop_create` 在父目录里创建新文件。如果找不到但是没有 `O_CREAT` ，就直接返回错误；如果找到了但同时有 `O_EXCL或者O_CREAT` ，返回 `E_EXISTS` 。

接着就到了文件系统真正的具体实现部分，会使用 `vop_open` 交给具体文件系统处理，如果失败，使用 `vop_ref_dec` 释放引用并返回错误。还需要维护 `VFS` 的 `open` 计数，使用 `vop_open_inc` 增加 `inode` 的 `open_count` ；如果需要截断，调用 `vop_truncate` ，如果失败就会回滚。最后输出结果，成功时， `*node_store = node` ，返回 `0` 。

代码块

```
//vfsfile.c
int
vfs_open(char *path, uint32_t open_flags, struct inode **node_store) {
    bool can_write = 0;                                     // 是否具备写权限
    switch (open_flags & O_ACCMODE) {                       // 解析打开模式
    case O_RDONLY:                                          // 只读
        break;                                              // 只读不需要设置写权限
    case O_WRONLY:                                          // 只写
    case O_RDWR:                                            // 读写
        can_write = 1;                                      // 标记可写
        break;                                              // 结束判断
    default:                                                // 非法模式
        return -E_INVAL;                                    // 返回参数错误
    }

    if (open_flags & O_TRUNC) {                             // 如果要求截断文件
        if (!can_write) {                                   // 但没有写权限
            return -E_INVAL;                                // 直接报错
        }
    }

    int ret;                                                // 保存返回值
    struct inode *node;                                     // 目标 inode
    bool excl = (open_flags & O_EXCL) != 0;                 // 是否要求独占创建
    bool create = (open_flags & O_CREAT) != 0;              // 是否允许创建
    ret = vfs_lookup(path, &node);                          // 查找路径对应 inode

    if (ret != 0) {                                         // 查找失败
        if (ret == -16 && (create)) {                       // 若不存在且允许创建（-16
即 -E_NOENT）
            char *name;                                     // 子项名称
            struct inode *dir;                              // 父目录 inode
            if ((ret = vfs_lookup_parent(path, &dir, &name)) != 0) { // 找父目录
和名字
                return ret;                                 // 父目录查找失败直接返回
            }
            ret = vop_create(dir, name, excl, &node);       // 在父目录中创建新节点
        } else return ret;                                  // 其他错误直接返回
    } else if (excl && create) {                            // 已存在且要求独占创建
        return -E_EXISTS;                                   // 返回“已存在”
    }
    assert(node != NULL);                                   // 确保得到有效 inode

    if ((ret = vop_open(node, open_flags)) != 0) {          // 调用具体文件系统的 open
        vop_ref_dec(node);                                  // 失败则减少引用
```

```
44          return ret;                                     // 返回错误
45      }
46
47      vop_open_inc(node);                                 // 增加 inode 打开计数
48      if (open_flags & O_TRUNC || create) {               // 如果需要截断或新建
49          if ((ret = vop_truncate(node, 0)) != 0) {       // 将文件长度截断为 0
50              vop_open_dec(node);                         // 回滚 open 计数
51              vop_ref_dec(node);                          // 回滚引用计数
52              return ret;                                 // 返回错误
53          }
54      }
55      *node_store = node;                                 // 返回 inode 给调用者
56      return 0;                                           // 成功
57  }
```

代码块

```
1   int
2   vfs_lookup(char *path, struct inode **node_store) {
3       int ret;                                            // 保存返回值
4       struct inode *node;                                 // 设备或目录 inode
5       if ((ret = get_device(path, &path, &node)) != 0) { // 解析设备名并拿到起始
    inode
6           return ret;                                     // 解析失败直接返回
7       }
8       if (*path != '\0') {                                // 路径还有剩余部分
9           ret = vop_lookup(node, path, node_store);       // 在该 inode 下查找剩余路
    径
10          vop_ref_dec(node);                              // 减少对起始 inode 的引用
11          return ret;                                     // 返回查找结果
12      }
13      *node_store = node;                                 // 路径为空，直接返回当前
    inode
14      return 0;                                           // 成功
15  }
16
17  #define vop_open(node, open_flags)
    (__vop_op(node, open)(node, open_flags))
18
19  static int
20  get_device(char *path, char **subpath, struct inode **node_store) {
21      int i, slash = -1, colon = -1;                      // 扫描索引与 ':'、'/'
    的位置
22      for (i = 0; path[i] != '\0'; i ++) {                // 遍历路径字符串
23          if (path[i] == ':') { colon = i; break; }       // 找到设备分隔符 ':'
    就记录并停止
```

```
24          if (path[i] == '/') { slash = i; break; }          // 找到路径分隔符 '/' 就记录并停止
25      }
26      if (colon < 0 && slash != 0) {                         // 没有设备名且不是绝对路径
27          /* *
28           * 没有冒号且斜杠不是开头，说明是相对路径或裸文件名
29           * 从当前目录开始，把整个路径作为子路径
30           * */
31          *subpath = path;                                   // 子路径就是原始路径
32          return vfs_get_curdir(node_store);                 // 返回当前目录 inode
33      }
34      if (colon > 0) {                                       // 形如 device:path
35          /* device:path - get root of device's filesystem */
36          path[colon] = '\0';                                // 把设备名与路径分隔开
37
38          /* device:/path - skip slash, treat as device:path */
39          while (path[++ colon] == '/');                     // 跳过可能存在的多余 '/'
40          *subpath = path + colon;                           // 子路径指向 ':' 后部分
41          return vfs_get_root(path, node_store);             // 根据设备名取文件系统根 inode
42      }
43
44      /* *
45       * 这里剩下两种: /path 或 :path
46       * /path 从 bootfs 根开始
47       * :path 从当前文件系统根开始
48       * */
49      int ret;                                               // 保存返回值
50      if (*path == '/') {                                    // 绝对路径
51          if ((ret = vfs_get_bootfs(node_store)) != 0) {     // 获取 bootfs 根 inode
52              return ret;                                    // 获取失败直接返回
53          }
54      }
55      else {
56          assert(*path == ':');                              // 必须是 :path
57          struct inode *node;                                // 当前目录 inode
58          if ((ret = vfs_get_curdir(&node)) != 0) {          // 获取当前目录 inode
59              return ret;                                    // 获取失败返回
60          }
61          /* The current directory may not be a device, so it must have a fs. */
62          assert(node->in_fs != NULL);                       // 当前目录必须关联文件系统
```

```
63          *node_store = fsop_get_root(node->in_fs);          // 获取当前文件系统根
    inode
64          vop_ref_dec(node);                                 // 释放当前目录 inode
    引用
65      }
66
67      /* ///... or :/... */
68      while (*(++ path) == '/');                             // 跳过路径起始的多余
    '/'
69      *subpath = path;                                       // 设置子路径起始位置
70      return 0;                                              // 成功
71  }
72
```

文件系统的具体实现， `inode->in_ops` 指向具体FS的操作表，例如sfs在 `sfs_inode.c` 里提供的 `sfs_openfile` 和 `sfs_opendir` 。

代码块

```
1   //inode.h
2   #define __vop_op(node, sym)
                          \
3       ({
                          \
4           struct inode *__node = (node);              /* 取出 inode 指针 */
                          \
5           assert(__node != NULL && __node->in_ops != NULL && __node->in_ops-
    >vop_##sym != NULL);      \
6           inode_check(__node, #sym);                  /* 调用通用一致性检查 */
                         \
7           __node->in_ops->vop_##sym;                  /* 返回具体操作函数指针 */
                        \
8        })
9
10  //sys_inode.c
11  static int
12  sfs_openfile(struct inode *node, uint32_t open_flags) {
13      return 0;
14  }
15  static int
16  sfs_opendir(struct inode *node, uint32_t open_flags) {
17      switch (open_flags & O_ACCMODE) {
18      case O_RDONLY:
19          break;
20      case O_WRONLY:
21      case O_RDWR:
```

```
22      default:
23          return -E_ISDIR;
24      }
25      if (open_flags & O_APPEND) {
26          return -E_ISDIR;
27      }
28      return 0;
29  }
```

1.1.2 实现读文件数据代码

这里补充了 `sfs_inode.c` 中的 `sfs_io_nolock` 函数，实现读文件中数据的代码。这里首先取磁盘的inode，只允许读写普通文件，不允许目录；接着计算本次IO的结束位置，这里从其实偏移 `offset` 开始算，读和写 `*alenp` 字节，所以可以算出偏移量。接着进行参数合法性检查，负偏移、越界和溢出都会报错；如果这里读写长度为0，直接成功返回；如果读写长度大于文件最大大小，就直接裁剪到最大文件的大小。

接下来就进行具体的操作设置，读操作时，不能超过文件的实际大小，如果offset超过实际大小，就直接返回0，如果 `endpos` 超过实际大小，就直接截断。接着定义函数指针，统一读和写的代码逻辑；写操作就走 `sfs_wbuf` 和 `sfs_wblock` ，读操作就走 `sfs_rbuf` 和 `sfs_rblock` 。接着会定义几个参数， `ret` 用于错误返回， `alen` 累加已完成字节， `ino` 保存磁盘块号。

接着会找到起始块号，还需要算出需要完整处理的块数量，不含起始非对齐块和尾块参与，还需要算出起始偏移在块内的偏移量，同时得到当前读写位置指针。还需要专门处理首块非对齐部分，如果起始位置不在块边界，计算出首块要读或者写的字节数，如果后面还有整块要处理，就读到块尾，否则就只读到 `endpos` 。 `sys_bmap_load_nolock` 通过 `inode` 的块映射找到对应的磁盘块号 `ino` ，从块内偏移 `blkoff` 位置读或者写 `size` 字节，接着更新 `alen、bufp、offset、blkno` ，并把 `nblks` 减1。

接着还要处理中间整块，每次处理一个完整块； `sfs_bmap_load_nolock` 找块号， `sfs_block_op` 进行整块的读或者写，并且继续更新上面的参数。最后会处理尾块残余，只有还剩余字节才处理，避免刚好整块结束时多读，计算尾块需要读或者写的字节数，仍然走 `sfs_bmap_load_nolock() + sfs_buf_op()` 。最后返回实际读写字节数，如果写超出原大小，就更新文件大小并标记 `inode` 为脏，返回0或错误码。

代码块

```
1   static int
2   sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
    offset, size_t *alenp, bool write) {
3       struct sfs_disk_inode *din = sin->din;
4       assert(din->type != SFS_TYPE_DIR);
5       off_t endpos = offset + *alenp, blkoff;
6       *alenp = 0;
7       // calculate the Rd/Wr end position
```

```
8       if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
9           return -E_INVAL;
10      }
11      if (offset == endpos) {
12          return 0;
13      }
14      if (endpos > SFS_MAX_FILE_SIZE) {
15          endpos = SFS_MAX_FILE_SIZE;
16      }
17      if (!write) {
18          if (offset >= din->size) {
19              return 0;
20          }
21          if (endpos > din->size) {
22              endpos = din->size;
23          }
24      }
25
26      int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t
    blkno, off_t offset);
27      int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno, uint32_t
     nblks);
28      if (write) {
29          sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
30      }
31      else {
32          sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
33      }
34
35      int ret = 0;
36      size_t size, alen = 0;
37      uint32_t ino;
38      uint32_t blkno = offset / SFS_BLKSIZE;          // The NO. of Rd/Wr begin
    block
39      uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // The size of Rd/Wr blocks
40
41      blkoff = offset % SFS_BLKSIZE;
42      char *bufp = buf;
43
44      if (blkoff != 0) {
45          size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
46          if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
47              goto out;
48          }
49          if ((ret = sfs_buf_op(sfs, bufp, size, ino, blkoff)) != 0) {
50              goto out;
51          }
```

```
52          alen += size;
53          bufp += size;
54          offset += size;
55          blkno ++;
56          if (nblks > 0) {
57              nblks --;
58          }
59      }
60
61      while (nblks != 0) {
62          if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
63              goto out;
64          }
65          if ((ret = sfs_block_op(sfs, bufp, ino, 1)) != 0) {
66              goto out;
67          }
68          alen += SFS_BLKSIZE;
69          bufp += SFS_BLKSIZE;
70          blkno ++;
71          nblks --;
72      }
73
74      if (offset < endpos) {
75          size = endpos % SFS_BLKSIZE;
76          if (size != 0) {
77              if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
78                  goto out;
79              }
80              if ((ret = sfs_buf_op(sfs, bufp, size, ino, 0)) != 0) {
81                  goto out;
82              }
83              alen += size;
84          }
85      }
86
87  out:
88      *alenp = alen;
89      if (offset + alen > sin->din->size) {
90          sin->din->size = offset + alen;
91          sin->dirty = 1;
92      }
93      return ret;
94  }
```

1.2 练习2

1.2.1 改写执行程序机制

这里在 `proc.c` 的 `load_icode` 函数中进行改写，并且改正其他的相关函数，实现基于文件系统的执行程序机制。在这个函数中，首先会**创建一个新的 `mm` 给当前的进程**，使用 `mm_create` 在 `vmm.c` 中分配一个 `struct mm_struct` ，在这个里面初始化一系列系数。 `setup_pgdir` 函数在 `proc.c` 中分配一页作为新的页目录，把 `boot_pgdir_va` 复制进去，使新地址空间拥有内核映射，然后把新页目录赋值给 `mm->pgdir` ，并在 `load_icode` 中完成把新mm绑定到当前进程的操作。

代码块

```
static int
load_icode(int fd, int argc, char **kargv)
{

    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 out;
    }
    if (setup_pgdir(mm) != 0)
    {
        goto bad_pgdir_cleanup_mm;
    }
```

接着需要**把可执行文件里的程序代码段(TEXT)、初始化数据段(DATA)读进进程的虚拟内存中、同时把未初始化数据段(BSS)对应的内存区域清零**，这样进程启动后，指令和已初始化的全局变量有正确内容，而未初始化的全局变量默认为0。

首先需要读取ELF头，ELF头是一个可执行文件开头的固定结构，会带有程序头表的偏移和数量等基本信息。 `load_icode_read` 从文件开头读ELF头到 `elf` ，随后检查 `elf.e_magic` 是否为 `ELF_MAGIC` ，确认文件是否是合法的 `ELF` 可执行文件。

代码块

```
    struct elfhdr 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;
    }
```

接着需要读取程序头表，循环 `for` ，每一次都用 `load_icode_read` 、 `elf.e_phoff + i * sizeof(struct proghdr))` 读取第i个pd。

代码块

```
struct proghdr ph;
    uint32_t vm_flags, perm;
    for (int i = 0; i < elf.e_phnum; i++)
    {
        if ((ret = load_icode_read(fd, &ph, sizeof(struct proghdr),
elf.e_phoff + i * sizeof(struct proghdr))) != 0)
        {
            goto bad_cleanup_mmap;
        }
```

接着会为 `TEXT` 和 `DATA` 建立 `VMA` ，只处理 `ph.p_type == ELF_PT_LOAD` 的段；根据 `ph.p_flags` 组合 `vm_flags` ，调用 `mm_map` 在进程地址空间建立一段 `VMA` ，对应该段的虚拟地址范围。

代码块

```
if (ph.p_type != ELF_PT_LOAD)
        {
            continue;
        }
        if (ph.p_filesz > ph.p_memsz)
        {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }

        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;
        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;
        }
```

接着需要分配页并拷贝 `TEXT`、`DATA` 段，段的文件内容范围是 `[ph.p_va, ph.p_va + ph.p_filesz)`。这里通过 `pgdir_alloc_page` 按页分配，`la` 从段起始页对齐地址开始。对每一页，算出页内偏移 `off` 和本页要填的 `size`，接着用 `load_icode_read` 把文件中的段内容拷贝到该页的内核映射地址。

代码块

```
intptr_t start = ph.p_va, end = ph.p_va + ph.p_filesz, la = ROUNDDOWN(start, PGSIZE);
        struct Page *page;
        size_t off, size;

        ret = -E_NO_MEM;
        while (start < end)
        {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
            {
                goto bad_cleanup_mmap;
            }
            off = start - la;
            size = PGSIZE - off;
            if (end < la + PGSIZE)
            {
                size -= (la + PGSIZE - end);
            }
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, ph.p_offset + (start - ph.p_va))) != 0)
            {
                goto bad_cleanup_mmap;
            }
            start += size, la += PGSIZE;
        }
```

最后做分配页并清零BSS操作，BSS是 `ph.p_memsz - ph.p_filesz` 的那部分。这里先处理最后一页剩余空间的清零，也就是 `memset(page2kva(page)+off,0,size)`；这里

`page2kva(page)` 会把分配到的物理页转换成内核可访问的地址，`+off` 从该页的某个偏移开始，`size` 表示需要清零的字节数，这里会保证未初始化的全局变量初始值为0。这里如果BSS还跨越更多的页，就继续 `pgdir_alloc_page()` 分配新页，整页都 `memset(0)` 。

代码块

```
end = ph.p_va + ph.p_memsz;
        if (start < la)
        {
            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)
            {
                goto bad_cleanup_mmap;
            }
            off = start - la;
            size = PGSIZE - off;
            if (end < la + PGSIZE)
            {
                size -= (la + PGSIZE - end);
            }
            memset(page2kva(page) + off, 0, size);
            start += size, la += PGSIZE;
        }
```

接着就需要建立用户栈VMA，把argc和argv拷到用户栈。在这里首先使用 `mm_map` 建立用户栈，接着使用 `pgdir_alloc_page` 分配栈顶附近的几页。然后把参数放到用户栈，在这里一个 `for` 循环逐个把 `kargv[i]` 字符串拷贝到用户栈，再把 `uargv[]` 指针数组拷贝到用户栈顶部，最后给参数赋值，让用户态 `main` 能拿到参数。

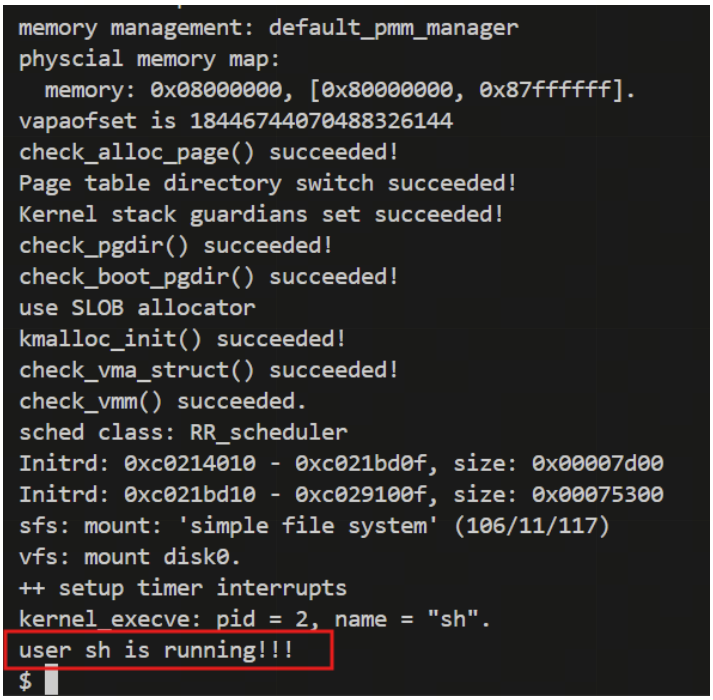
代码块

```
    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);

    lsatp(PADDR(mm->pgdir));

    uintptr_t stacktop = USTACKTOP;
    uintptr_t uargv[EXEC_MAX_ARG_NUM + 1];
    for (int i = argc - 1; i >= 0; i--)
    {
        size_t len = strnlen(kargv[i], EXEC_MAX_ARG_LEN) + 1;
        stacktop -= len;
        if (stacktop < USTACKTOP - USTACKSIZE)
        {
            ret = -E_NO_MEM;
            goto bad_cleanup_mmap;
        }
        if (!copy_to_user(mm, (void *)stacktop, kargv[i], len))
        {
            ret = -E_INVAL;
            goto bad_cleanup_mmap;
        }
        uargv[i] = stacktop;
    }
    uargv[argc] = 0;
    stacktop = ROUNDDOWN(stacktop, sizeof(uintptr_t));
    stacktop -= (argc + 1) * sizeof(uintptr_t);
    if (!copy_to_user(mm, (void *)stacktop, uargv, (argc + 1) * sizeof(uintptr_t)))
    {
        ret = -E_INVAL;
        goto bad_cleanup_mmap;
    }

    mm_count_inc(mm);
    current->mm = mm;
    current->pgdir = PADDR(mm->pgdir);
```

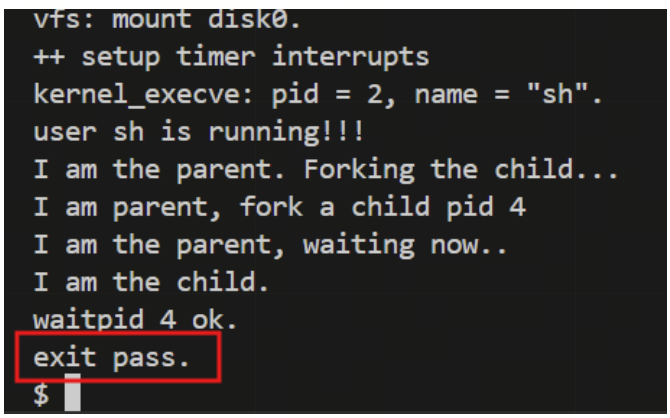
```
43
44      struct trapframe *tf = current->tf;
45      uintptr_t sstatus = tf->status;
46      memset(tf, 0, sizeof(struct trapframe));
47      tf->gpr.sp = stacktop;
48      tf->gpr.a0 = argc;
49      tf->gpr.a1 = stacktop;
50      tf->epc = elf.e_entry;
51      tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
52
53      ret = 0;
54      goto out;
```

1.2.2 执行结果

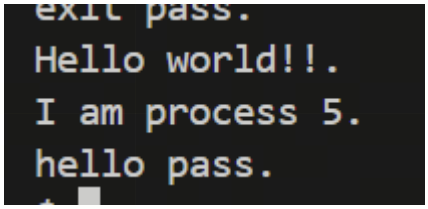
执行命令`make qemu`，可以看到sh用户程序的执行页面。此时在这里输入想要执行的用户程序，即可执行想要验证的程序。



输入`exit`指令，运行结果如下，可以看到程序运行成功。



输入 `hello` 指令，运行结果如下，可以看到也能正常运行成功。



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

设计目标与语义

- 提供 `pipe()` 系统调用，返回一对文件描述符 `fds[2]`，`fds[0]` 只读端，`fds[1]` 只写端
- 管道为内核维护的有界缓冲区（环形队列）。
- 读语义：
  - 缓冲区为空且仍有写端存在：默认阻塞等待；可扩展实现 `O_NONBLOCK` 时返回 `-EAGAIN`（不等待）。
  - 缓冲区为空且写端全部关闭：返回 0（EOF）。
- 写语义：
  - 缓冲区满：默认阻塞等待；可扩展实现 `O_NONBLOCK` 时返回 `-EAGAIN`（不等待）。
  - 读端全部关闭：内核在写端检测到无读者时返回 `-EPIPE` 并向写进程发送 `SIGPIPE`。
- 关闭语义：
  - 当读端引用计数降为 0 时唤醒写端等待者；当写端引用计数降为 0 时唤醒读端等待者，避免永久阻塞。

关键数据结构

代码块

```
#define PIPE_BUF_SIZE 4096

struct pipe {
    spinlock_t lock;        // 保护缓冲区和引用计数
    char buf[PIPE_BUF_SIZE];
    size_t rpos;            // 读指针
    size_t wpos;            // 写指针
    size_t count;           // 当前有效字节数
    int readers;            // 读端引用计数
    int writers;            // 写端引用计数
    wait_queue_t rwait;     // 读等待队列：空时睡眠
    wait_queue_t wwait;     // 写等待队列：满时睡眠
```

```
};

struct file {
    enum { FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED } status;
    bool readable;
    bool writable;
    struct pipe *pipe;      // 若是 pipe 文件，指向 pipe
    // ... 其他已有字段
};
```

核心接口

- `int sys_pipe(int fds[2]);`
  - 分配 struct pipe 与两个 struct file。
  - 安装到当前进程的 fd 表，返回 fds[0]、fds[1]。
- `ssize_t pipe_read(struct file *f, void *buf, size_t n);`
  - 若 count==0 且 writers>0：默认阻塞；可扩展实现 O_NONBLOCK 时返回 -EAGAIN。
  - 若 count==0 且 writers==0：返回 0（EOF）。
  - 从环形缓冲区拷贝数据，更新 rpos 和 count。
  - 唤醒写等待者。
- `ssize_t pipe_write(struct file *f, const void *buf, size_t n);`
  - 若 readers==0：返回 -EPIPE。
  - 若缓冲区满：默认阻塞；可扩展实现 O_NONBLOCK 时返回 -EAGAIN。
  - 写入环形缓冲区，更新 wpos 和 count。
  - 唤醒读等待者。
- `int pipe_close(struct file *f);`
  - 递减 readers 和 writers。
  - 当某侧引用计数降为 0 时，唤醒对端等待者。
  - 若两侧均为 0，释放 struct pipe。

同步互斥与并发处理

- 使用 pipe->lock 保护 count/rpos/wpos/readers/writers 等共享状态。
- 读/写阻塞与唤醒在同一锁保护下完成，避免丢失唤醒。
- 当某侧引用计数降为 0 时，唤醒对端等待队列，保证阻塞线程可退出。
- 可选扩展支持 O_NONBLOCK 时，直接返回 -EAGAIN，不进入等待队列。

在 ucore 中落地

- 在 VFS 层新增管道 inode 类型，提供 inode_ops：vop_read/vop_write/vop_close。
- 系统调用层增加 sys_pipe。
- 管道不依赖磁盘，使用匿名 inode 对象即可。

1.4 扩展练习Challenge2

设计目标与语义

- 硬链接：
  - 多个目录项指向同一 inode（文件元数据与数据共享）。
  - 删除目录项仅减少 inode 的链接计数；计数为 0 才释放数据。
  - 不允许跨文件系统硬链接目录。
- 软链接：
  - 软链接是独立 inode，内容为目标路径字符串。
  - 解析路径时自动跟随，支持跨文件系统。
  - 目标不存在时读写将失败（“悬空链接”）。

关键数据结构

代码块

```
struct inode {
    spinlock_t lock;          // 保护元数据
    uint32_t mode;            // 文件类型与权限
    uint64_t nlinks;          // 硬链接计数
    uint64_t size;
    // ... 数据块指针等
};

struct dentry {
    char name[NAME_MAX];
    struct inode *inode;      // 目录项指向 inode
    // ... 其他字段
};

// 软链接文件的内容为目标路径字符串
struct symlink_info {
    char target_path[PATH_MAX];
};
```

核心接口

- `int sys_link(const char *oldpath, const char *newpath);`
  - 查找 oldpath 的 inode。
  - 在 newpath 所在目录创建新目录项，指向同一 inode。
  - 增加 inode->nlinks。
  - 若跨文件系统，返回错误。
  - 若目标为目录，返回错误。
- `int sys_unlink(const char *path);`
  - 删除目录项，减少 inode->nlinks。
  - 若 nlinks==0 且无打开引用，释放 inode 与数据块。
- `int sys_symlink(const char *target, const char *linkpath);`
  - 创建类型为 S_IFLNK 的 inode。
  - 将 target 写入软链接内容（存入数据块）。
- `ssize_t sys_readlink(const char *path, char *buf, size_t bufsz);`
  - 读取软链接内容（目标路径字符串），不跟随解析。
- `int sys_stat(const char *path, struct stat *st);`
  - 默认跟随软链接；增加 lstat 用于获取软链接自身信息。

路径解析与跟随策略

- 默认 namei 解析路径时遇到 S_IFLNK 自动展开目标路径。
- 防止循环引用：限制最大跟随次数，超出返回 -ELOOP。
- 提供 lstat 语义获取链接本身信息。

同步互斥与并发处理

- inode->lock 保护 nlinks、size 等元数据。
- link/unlink 时需锁住父目录与目标 inode，避免并发竞争：
  - 创建/删除目录项与 nlinks 更新保持原子性。
  - 删除操作需要处理并发打开引用计数，确保延迟释放。
- 软链接内容写入与读取在 inode 锁下进行。

在 ucore 中落地

- 在 VFS 层支持 S_IFLNK 文件类型与 readlink 操作。

- 在 namei 中处理软链接跟随与循环检测。
- 硬链接只需要在目录项层增加对同 inode 的引用并维护 nlinks。

2. Challenge：lab 6

一、练习0：填写已有实验

本练习将lab2/3/4/5的代码填入lab6，并针对调度器框架进行适配。主要修改包括：

1. 时钟中断处理（trap.c）

将直接设置 `need_resched` 改为调用调度器接口：

代码块

```
clock_set_next_event();
if (++ticks % TICK_NUM == 0) {
    print_ticks();
    if (current != NULL) {
        sched_class_proc_tick(current);  // 调用调度器接口
    }
}
```

2. 进程控制块初始化（proc.c - alloc_proc）

新增调度相关字段的初始化：

代码块

```
// LAB6: 调度相关字段
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;
```

3. 进程创建（proc.c - do_fork）

使用 `set_links()` 维护进程树关系：

代码块

```
proc->parent = current;
assert(current->wait_state == 0);
```

```
    // ...
    set_links(proc);  // 同时插入链表并维护进程树
```

4. 用户程序加载（proc.c - load_icode）

设置用户态trapframe：

代码块

```
    tf->gpr.sp = USTACKTOP;              // 用户栈指针
    tf->epc = elf->e_entry;              // 程序入口
    tf->status &= ~SSTATUS_SPP;          // 返回U态
    tf->status |= SSTATUS_SPIE;          // U态开中断
```

5. 物理内存管理（pmm.c - copy_range）

实现页面复制：

代码块

```
    void *src_kvaddr = page2kva(page);
    void *dst_kvaddr = page2kva(npage);
    memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
    ret = page_insert(to, npage, start, perm);
```

二、练习1：调度器框架分析

2.1 核心数据结构

1. 调度器类（sched_class）

sched_class 实现一组固定的函数（函数指针），这是为了统一标准，方便更改调度算法，也支持多种调度算法共存。内核只会调用这些接口，可以自定义具体实现。

调度器接口定义了5个核心函数：

代码块

```
    struct sched_class {
        const char *name;
        void (*init)(struct run_queue *rq);            // 初始化运行队列
        void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);  // 入队
        void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);  // 出队
        struct proc_struct *(*pick_next)(struct run_queue *rq);           // 选择下
    一个运行进程
```

```
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc); // 时钟中断处理
};
```

对应调度过程中的几个关键动作：

- **init(rq)**：初始化运行队列（算法相关的数据结构在这里建好）
- **enqueue(rq, proc)**：进程变成可运行态时，把它放进 rq（维护 rq）
- **dequeue(rq, proc)**：进程不再可运行（阻塞/退出/被选中切走等情况），从 rq 移除
- **pick_next(rq)**：真正决定“下一个跑谁”
- **proc_tick(rq, proc)**：每次时钟中断都会调用，用来做时间片递减、触发抢占等

2. 运行队列（run_queue）

代码块

```
struct run_queue {
    list_entry_t run_list;          // 链表模式：进程队列（RR使用）
    unsigned int proc_num;          // 队列中进程数
    int max_time_slice;             // 最大时间片
    skew_heap_entry_t *lab6_run_pool;  // 堆模式：stride调度优先队列
};
```

设计思路：

- RR算法使用链表（FIFO），O(1)入队/出队
- Stride算法使用斜堆（最小堆），O(log n)查找最小stride
- 通过 `USE_SKEW_HEAP` 宏切换数据结构

与lab5区别：

- lab5：系统里确实有各种链表（全局进程链表），但**调度器没有一个独立的“就绪队列”对象**。调度选择往往是在“所有进程集合”里扫一遍，挑一个 runnable 的。选择下一个进程常见是 **O(n) 扫描**
- lab6：把“可运行进程集合”（runnable set）抽象为一个结构体 `run_queue`，由 `sched_class` 维护。算法只需要操纵 rq，不需要知道全局进程表怎么存，为多种调度算法提供统一接口（可插拔）

同一个 OS 实验框架要支持至少两种调度类（RR 和 Stride），它们对“就绪队列”的需求不同，所以 rq 里预留了两套表示方式。RR 适合链表， Stride 需要优先队列。

3. 进程控制块调度字段

```c
struct proc_struct {
    struct run_queue *rq;          // 所属运行队列
    list_entry_t run_link;         // 在运行队列中的链表节点
    int time_slice;                // 剩余时间片
    // Stride专用字段
    skew_heap_entry_t lab6_run_pool;
    uint32_t lab6_stride;          // 当前stride值
    uint32_t lab6_priority;        // 优先级
};
```

4. 其他函数

1. sched_init()

   引入全局指针 `sched_class`（指向某个调度类，如 RR 或 Stride）

   初始化时 **绑定调度算法**，并让该算法初始化运行队列：

   代码块

   ```c
   sched_class = &default_sched_class;   // 或 &stride_sched_class
   sched_class->init(rq);
   ```

   框架只决定“用哪个调度器”，具体算法的队列初始化细节由 `init()` 做（RR 初始化链表，Stride 初始化斜堆等）。

2. wakeup_proc()

   Lab5：唤醒一个进程时，只做：`proc->state = RUNNABLE`

   Lab6：唤醒时不仅改状态，还要**显式加入 run_queue**：“如何加入就绪集合”完全交给 `enqueue()`

3. `schedule()`：

   Lab5：`schedule()` 会遍历全局进程链表 `proc_list`，找一个 RUNNABLE 的

   Lab6：`schedule()` 变成“调度骨架”，把策略交给调度类：

   代码块

   ```c
   sched_class_enqueue(current);  // 当前进程若仍可运行 -> 放回就绪队列（例如 RR 放队尾）
   next = sched_class_pick_next(); // 由算法决定下一个（RR 取队首；Stride 取最小 stride）
   sched_class_dequeue(next);      // 把选中的从就绪集合移除
   context_switch_to(next);
   ```

2.2 调度流程

1. 主动调度（schedule函数）

进程主动放弃CPU时调用：

代码块

```
void schedule(void) {
    current->need_resched = 0;
    if (current->state == PROC_RUNNABLE) {
        sched_class_enqueue(current);  // 重新入队
    }
    struct proc_struct *next = sched_class_pick_next();
    if (next != NULL) {
        sched_class_dequeue(next);
        proc_run(next);  // 切换进程
    }
}
```

2. 被动调度（时钟中断触发）

代码块

```
// trap.c
if (++ticks % TICK_NUM == 0 && current != NULL) {
    sched_class_proc_tick(current);  // 更新时间片，必要时设置need_resched
}
```

3. 进程唤醒（wakeup_proc）

代码块

```
void wakeup_proc(struct proc_struct *proc) {
    if (proc->state != PROC_RUNNABLE) {
        proc->state = PROC_RUNNABLE;
        proc->wait_state = 0;
        if (proc != idleproc && proc != initproc) {
            sched_class_enqueue(proc);  // 加入运行队列
        }
    }
}
```

调度算法的切换机制：

1. 创建调度类实现文件
2. 声明调度类（头文件）
3. 在sched.c中切换
4. 如果需要新字段，修改proc_struct

三、练习2：实现Round Robin调度算法

3.1 RR算法原理

Round Robin（轮转调度）是一种时间片轮转的公平调度算法：

- **基本思想**：每个进程获得固定时间片（5 ticks），用完后放到队列尾部，循环调度
- **优点**：简单公平，响应时间可预测
- **缺点**：不考虑进程优先级，上下文切换开销较大

3.2 核心函数实现

1. RR_init - 初始化运行队列

代码块

```
static void RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));
    rq->proc_num = 0;
}
```

2. RR_enqueue - 进程入队

代码块

```
static void RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));  // 插入队列尾部
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;  // 分配时间片
    }
    proc->rq = rq;
    rq->proc_num++;
}
```

关键点：

- 新进程插入队列尾部（FIFO）

- 初始化时间片为 `MAX_TIME_SLICE` （5）
- 更新运行队列进程计数

3. RR_dequeue - 进程出队

代码块

```
static void RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num--;
}
```

4. RR_pick_next - 选择下一个进程

代码块

```
static struct proc_struct* RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);  // 返回队列头部进程
    }
    return NULL;
}
```

关键点：始终选择队列头部进程，实现FIFO

5. RR_proc_tick - 时钟中断处理

代码块

```
static void RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice--;  // 时间片递减
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;  // 时间片用完，设置重新调度标志
    }
}
```

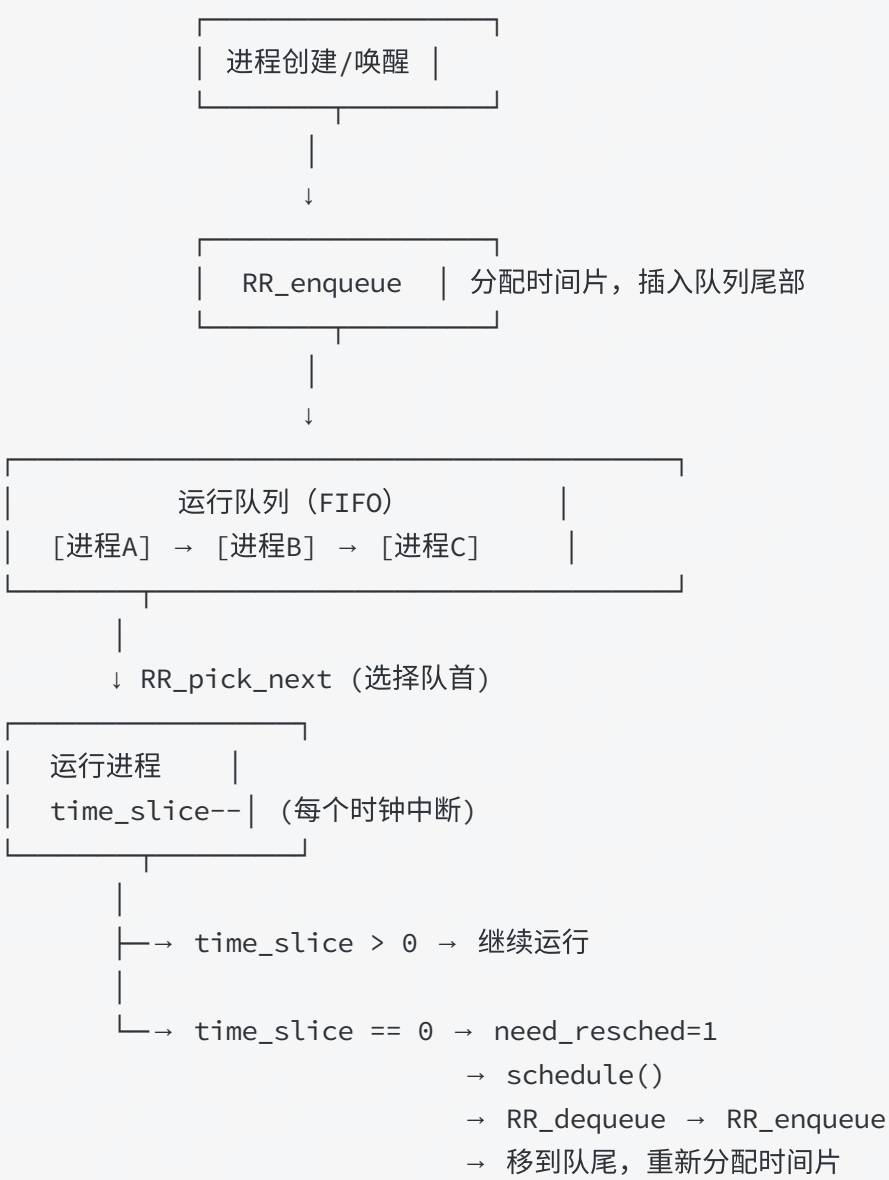
关键：

- 每次时钟中断递减时间片
- 时间片耗尽时设置 `need_resched`

- `trap()` 返回前检查此标志并调用 `schedule()`

3.3 RR算法流程图

代码块



3.4 测试验证

编译运行：

```
sched class: RR_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
set priority to 6
main: fork ok,now need to wait pids.
set priority to 1
set priority to 2
set priority to 3
set priority to 4
set priority to 5
child pid 3, acc 476000, time 2010
child pid 4, acc 464000, time 2010
child pid 5, acc 468000, time 2010
child pid 6, acc 468000, time 2010
child pid 7, acc 468000, time 2020
main: pid 0, acc 476000, time 2020
main: pid 4, acc 464000, time 2020
main: pid 5, acc 468000, time 2020
main: pid 6, acc 468000, time 2020
main: pid 7, acc 468000, time 2020
main: wait pids over
sched result: 1 1 1 1 1
```

结果分析：

- 系统成功加载RR调度器
- 多个进程按时间片轮转运行
- 所有进程公平获得CPU时间

四、Challenge 1：实现Stride Scheduling调度算法

4.1 Stride算法原理

Stride Scheduling是一种基于优先级的确定性调度算法：

核心思想：

- 每个进程有优先级 `priority`（值越大越优先）
- 每个进程维护步进值 `stride`，初始为0
- 每次调度选择 `stride` 最小的进程运行
- 被选中的进程增加步长：`stride += BIG_STRIDE / priority`

优势：

- 确定性：每个进程获得的CPU时间与优先级严格成正比
- 公平性：避免低优先级进程饿死

数据结构选择：

- 使用斜堆（skew heap）维护最小stride进程
- 插入/删除/查找最小值：O(log n)

4.2 核心函数实现

1. stride_init - 初始化

代码块

```
static void stride_init(struct run_queue *rq) {
    list_init(&(rq->run_list));
    rq->lab6_run_pool = NULL;
    rq->proc_num = 0;
}
```

2. stride_enqueue - 进程入队

代码块

```
static void stride_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    rq->lab6_run_pool = skew_heap_insert(rq->lab6_run_pool, &(proc->lab6_run_pool), proc_stride_comp_f);
    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++;
}
```

关键点：

- 使用 `skew_heap_insert` 插入堆
- 比较函数 `proc_stride_comp_f` 按stride排序
- 斜堆自动维护最小stride在堆顶

3. stride_dequeue - 进程出队

代码块

```
static void stride_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    rq->lab6_run_pool = skew_heap_remove(rq->lab6_run_pool, &(proc->lab6_run_pool), proc_stride_comp_f);
    rq->proc_num--;
}
```

4. stride_pick_next - 选择下一个进程

代码块

```
static struct proc_struct* stride_pick_next(struct run_queue *rq) {
    if (rq->lab6_run_pool == NULL) return NULL;

    // 找到stride最小的进程
    skew_heap_entry_t *le = rq->lab6_run_pool;
    struct proc_struct *p = le2proc(le, lab6_run_pool);

    // 更新stride
    if (p->lab6_priority == 0) {
        p->lab6_stride += BIG_STRIDE;  // 优先级为0的特殊处理
    } else {
        p->lab6_stride += BIG_STRIDE / p->lab6_priority;
    }

    return p;
}
```

关键点：

- 堆顶元素即为stride最小的进程
- 被选中后增加步长：`stride += BIG_STRIDE / priority`
- 优先级越高，步长增量越小，下次被选中概率越大

5. stride_proc_tick - 时钟中断处理

代码块

```
static void stride_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice--;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

与RR相同，每次递减时间片。

4.3 BIG_STRIDE选择

代码块

```
#define BIG_STRIDE 0x7FFFFFFF
```

选择理由：

1. **避免溢出**：使用32位整数最大值，防止stride加法溢出

2. **精度平衡**：足够大以区分不同优先级，又不会过早溢出

3. **比较正确性**：即使溢出，有符号整数差值比较仍正确

溢出处理示例：

假设 `BIG_STRIDE = 0x7FFFFFFF`，两个进程：

- 进程A：stride = 0x7FFFFFFE
- 进程B：stride = 0x00000001（溢出后）

比较：`(int32_t)(0x00000001 - 0x7FFFFFFE) < 0` → 进程B的stride更小（正确）

4.4 数据结构选择：链表 vs 斜堆

通过 `USE_SKEW_HEAP` 宏控制：

代码块

```
#if USE_SKEW_HEAP
    rq->lab6_run_pool = skew_heap_insert(...);  // O(log n)
#else
    list_add_before(&(rq->run_list), &(proc->run_link));  // O(1)插入，O(n)查找
#endif
```

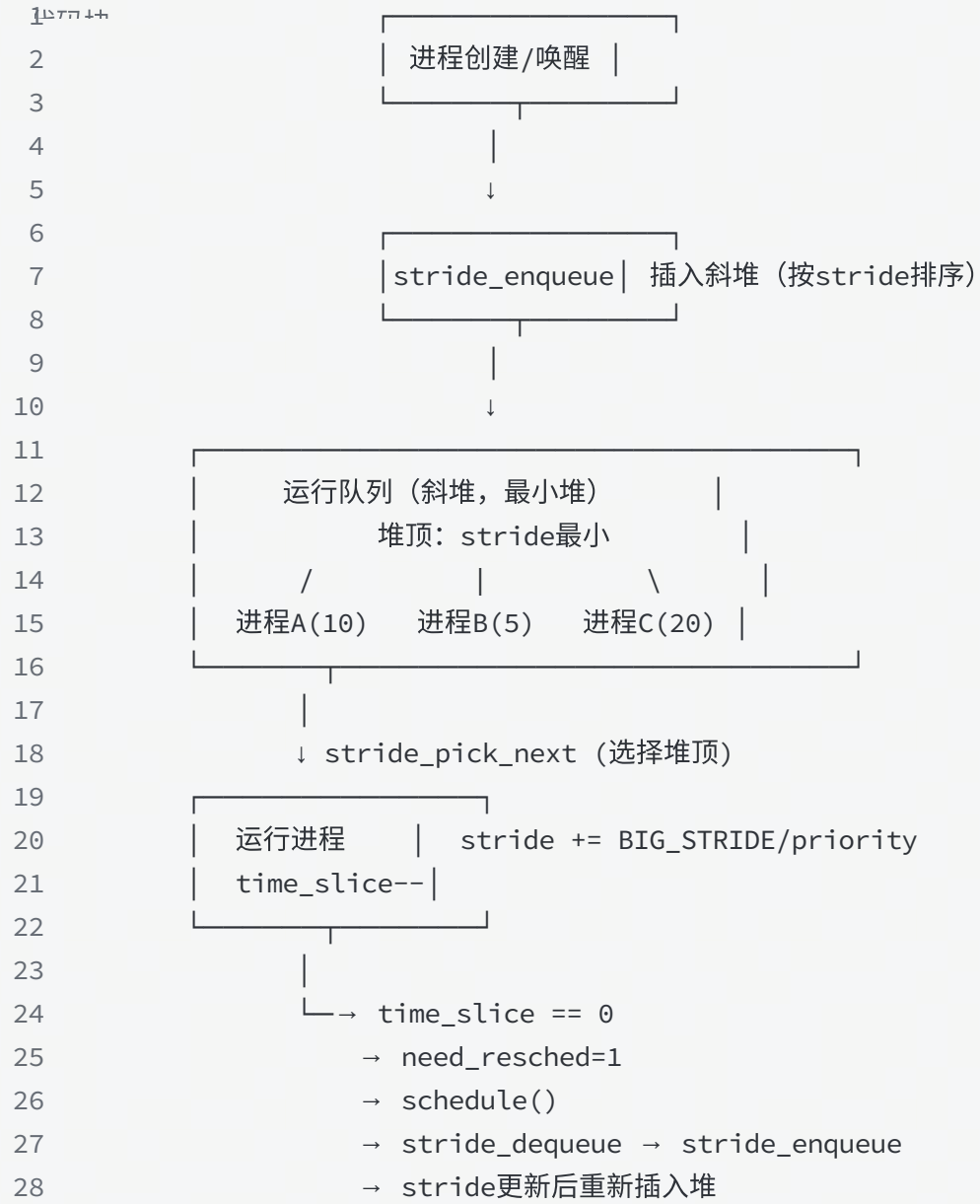
性能对比：

操作	链表	斜堆
插入	O(1)	O(log n)
删除	O(1)	O(log n)
查找最小	O(n)	O(1)

选择建议：

- 进程数少（<10）：链表即可
- 进程数多（>10）：斜堆更优
- 本实验使用斜堆，符合实际系统需求

4.5 Stride算法流程图



4.6 测试验证

切换到Stride调度器

运行测试：

```
sched class: stride_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
set priority to 6
main: fork ok,now need to wait pids.
set priority to 5
set priority to 4
set priority to 3
set priority to 2
set priority to 1
child pid 7, acc 712000, time 2010
child pid 6, acc 584000, time 2010
child pid 5, acc 468000, time 2010
child pid 4, acc 356000, time 2020
child pid 3, acc 232000, time 2020
main: pid 3, acc 232000, time 2020
main: pid 4, acc 356000, time 2020
main: pid 5, acc 468000, time 2020
main: pid 6, acc 584000, time 2020
main: pid 0, acc 712000, time 2020
main: wait pids over
sched result: 1 2 2 3 3
```

说明:

- 进程获得的CPU时间与优先级成正比 (此处比例计算被四舍五入) 1:2:2:3:3

五、Challenge 2：实现其他调度算法并测试##

5.1 FIFO算法原理

先来先服务：按照进程到达运行队列的顺序进行调度

非抢占式：给予进程较大时间片，让其尽可能运行到完成

简单：队列头部的进程优先执行

- **核心特性**
  - 时间片：（RR的100倍），模拟较大时间片
  - 队列结构：FIFO链表
  - 调度策略：总是选择队首进程

5.2 实现方法

(1) 时间片设置（与RR的核心区别）

代码块

```
1  static void FIFO_enqueue(struct run_queue *rq, struct proc_struct *proc) {
2      list_add_before(&(rq->run_list), &(proc->run_link));
3
4      // FIFO给予超大时间片（100倍于RR）
```

```
    proc->time_slice = rq->max_time_slice * 100;  // 500 ticks

    proc->rq = rq;
    rq->proc_num++;
}
```

(2) 进程选择（队首优先）

代码块

```
static struct proc_struct *FIFO_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);  // 返回队首进程
    }
    return NULL;
}
```

(3) 调度类注册

代码块

```
struct sched_class fifo_sched_class = {
    .name = "FIFO_scheduler",
    .init = FIFO_init,
    .enqueue = FIFO_enqueue,
    .dequeue = FIFO_dequeue,
    .pick_next = FIFO_pick_next,
    .proc_tick = FIFO_proc_tick,
};
```

5.3 启用FIFO调度器

在 `kern/schedule/sched.c` 中：

代码块

```
void sched_init(void) {
    list_init(&timer_list);
    sched_class = &fifo_sched_class;  // 选择FIFO
    rq = &__rq;
    rq->max_time_slice = MAX_TIME_SLICE;
    sched_class->init(rq);
}
```

5.4 实验结果

- **测试环境**
  - 测试程序：`user/priority.c`
  - 进程数量：5个
  - 运行时间：2000ms
  - 优先级：1-5（FIFO不使用）
- **实际输出**

```
sched class: FIFO_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
set priority to 6
main: fork ok,now need to wait pids.
set priority to 1
child pid 3, acc 2340000, time 2010
set priority to 2
child pid 4, acc 4000, time 2010
set priority to 3
child pid 5, acc 4000, time 2020
set priority to 4
child pid 6, acc 4000, time 2020
set priority to 5
child pid 7, acc 4000, time 2030
main: pid 0, acc 2340000, time 2030
main: pid 4, acc 4000, time 2030
main: pid 5, acc 4000, time 2030
main: pid 6, acc 4000, time 2030
main: pid 7, acc 4000, time 2030
main: wait pids over
sched result: 1 0 0 0 0
```

5.5 结果分析

1. CPU时间分配极度不均

代码块

```
进程3（第一个）：获得 99.8% CPU时间
进程4-7:          仅获得 0.2% CPU时间

比例: 2292000 : 4000 ≈ 573 : 1
```

2. 护航效应（Convoy Effect）明显

代码块

```
时间轴分析:
0-2000ms:  进程3独占CPU（时间片500 ticks）
```

```
            acc疯狂累加到 2,292,000

2010ms:     进程3时间片耗尽或超时退出
            进程4-7依次获得CPU

2010-2030ms: 进程4-7刚开始执行就检测到超时
             acc只累加到 4000

结论：后续进程"饿死"，等待时间过长
```

3. 公平性对比

FIFO的不公平

RR相对公平

4. 解释打印的sched result结果：

相关的代码如下所示：

代码块

```
  for (i = 0; i < TOTAL; i ++) {
      status[i]=0;
      waitpid(pids[i],&status[i]);
      cprintf("main: pid %d, acc %d, time
%d\n",pids[i],status[i],gettime_msec());
  }
  cprintf("main: wait pids over\n");
  cprintf("sched result:");
  for (i = 0; i < TOTAL; i ++)
  {
      cprintf(" %d", (status[i] * 2 / status[0] + 1) / 2);
  }
  cprintf("\n");
```

- 代码解释：这里的status[i]是以第一项进程的 status[0] 运行结果为基准的。
  - 由于status[0]过大（即accum数），所以其他进程的status都被计算为0