

Lab8

练习0：填写已有实验

如下是初始化进程控制块的代码，在这里面新增了文件结构指针的初始化，下面将使用注释的形式解释。

```
1 // alloc_proc - alloc a proc_struct and init all fields of proc_struct
2 static struct proc_struct *
3 alloc_proc(void)
4 {
5     struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
6     if (proc != NULL)
7     {
8         // LAB4:填写你在lab4中实现的代码 已填写
9         /*
10          * below fields in proc_struct need to be initialized
11          *      enum proc_state state;                // Process
12          state
13          *      int pid;                                // Process ID
14          *      int runs;                                // the running
15          times of Proces
16          *      uintptr_t kstack;                        // Process
17          kernel stack
18          *      volatile bool need_resched;            // bool value:
19          need to be rescheduled to release CPU?
20          *      struct proc_struct *parent;            // the parent
21          process
22          *      struct mm_struct *mm;                  // Process's
23          memory management field
24          *      struct context context;                // switch here
25          to run process
26          *      struct trapframe *tf;                  // Trap frame
27          for current interrupt
28          *      uintptr_t pgdir;                        // the base
29          addr of Page Directroy Table(PDT)
30          *      uint32_t flags;                          // Process flag
31          *      char name[PROC_NAME_LEN + 1];          // Process name
32          */
33
34          // LAB5:填写你在lab5中实现的代码 (update LAB4 steps)已填写
35          /*
36          * below fields(add in LAB5) in proc_struct need to be initialized
37          *      uint32_t wait_state;                    // waiting
38          state
39          *      struct proc_struct *cptr, *yptr, *optr; // relations
40          between processes
41          */
42
43          // LAB6:填写你在lab6中实现的代码 (update LAB5 steps)已填写
44          /*
45          * below fields(add in LAB6) in proc_struct need to be initialized
```

```

35     *      struct run_queue *rq;                                // run queue
contains Process
36     *      list_entry_t run_link;                                // the entry
linked in run queue
37     *      int time_slice;                                       // time slice
for occupying the CPU
38     *      skew_heap_entry_t lab6_run_pool;                       // entry in the
run pool (lab6 stride)
39     *      uint32_t lab6_stride;                                   // stride value
(lab6 stride)
40     *      uint32_t lab6_priority;                                 // priority
value (lab6 stride)
41     */
42
43     //LAB8 YOUR CODE : (update LAB6 steps)
44     /*
45     * below fields(add in LAB6) in proc_struct need to be initialized
46     *      struct files_struct * filesp;                          file struct
point
47     */
48     proc->state = PROC_UNINIT;
49     proc->pid = -1;
50     proc->runs = 0;
51     proc->kstack = 0;
52     proc->need_resched = 0;
53     proc->parent = NULL;
54     proc->mm = NULL;
55     memset(&(proc->context), 0, sizeof(struct context));
56     proc->tf = NULL;
57     proc->pgdir = boot_pgdir_pa;
58     proc->flags = 0;
59     memset(proc->name, 0, PROC_NAME_LEN);
60     // LAB5新增
61     proc->wait_state = 0;
62     proc->cptr = NULL;
63     proc->optr = NULL;
64     proc->yptr = NULL;
65
66     // Lab6新增
67     proc->rq = NULL;
68     list_init(&(proc->run_link));
69     proc->time_slice = 0;
70     skew_heap_init(&(proc->lab6_run_pool));
71     proc->lab6_stride = 0;
72     proc->lab6_priority = 0;
73     // lab8 add
74     proc->filesp = NULL;
75
76
77 }
78 return proc;
79 }

```

如下是进程切换函数的新增内容，下面将使用注释的形式解释

```

1 // proc_run - make process "proc" running on cpu
2 // NOTE: before call switch_to, should load base addr of "proc"'s new PDT
3 void proc_run(struct proc_struct *proc)
4 {
5     // LAB4:填写你在lab4中实现的代码
6     /*
7      * Some Useful MACROS, Functions and DEFINES, you can use them in
8      below implementation.
9      * MACROS or Functions:
10     * local_intr_save():      Disable interrupts
11     * local_intr_restore():   Enable Interrupts
12     * lcr3():                 Modify the value of CR3 register
13     * switch_to():            Context switching between two
14     processes
15     */
16     //LAB8 YOUR CODE : (update LAB4 steps)
17     /*
18     * below fields(add in LAB6) in proc_struct need to be initialized
19     * before switch_to();you should flush the tlb
20     * MACROS or Functions:
21     * flush_tlb():             flush the tlb
22     */
23     unsigned long intrflag;
24     struct proc_struct *prev = current;
25
26     local_intr_save(intrflag);
27     current = proc;
28
29     lsatp(proc->pgdir);
30     proc->need_resched = 0;
31     proc->runs++;
32     // 刷新TLB (快表)
33     flush_tlb();
34
35     switch_to(&(prev->context), &(proc->context));
36     local_intr_restore(intrflag);
37 }

```

练习1 完成读文件操作的实现（需要编码）

如下是 `sfs_io_nolock()` 函数，下面将使用注释解释

```

1 /*
2  * sfs_io_nolock - Rd/Wr a file content from offset position to offset+
3  length disk blocks<-->buffer (in memroy)
4  * @sfs:      sfs file system
5  * @sin:      sfs inode in memory
6  * @buf:      the buffer Rd/Wr
7  * @offset:   the offset of file
8  * @alenp:    the length need to read (is a pointer). and will RETURN the
9  really Rd/Wr lenght
10 * @write:    BOOL, 0 read, 1 write
11 */
12 static int

```

```

11 sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
offset, size_t *alenp, bool write) {
12     struct sfs_disk_inode *din = sin->din;
13     assert(din->type != SFS_TYPE_DIR);
14     off_t endpos = offset + *alenp, blkoff;
15     *alenp = 0;
16     // calculate the Rd/wr end position
17     if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
18         return -E_INVAL;
19     }
20     if (offset == endpos) {
21         return 0;
22     }
23     if (endpos > SFS_MAX_FILE_SIZE) {
24         endpos = SFS_MAX_FILE_SIZE;
25     }
26     if (!write) {
27         if (offset >= din->size) {
28             return 0;
29         }
30         if (endpos > din->size) {
31             endpos = din->size;
32         }
33     }
34
35     int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t
blkno, off_t offset);
36     int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno,
uint32_t nblks);
37     if (write) {
38         sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
39     }
40     else {
41         sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
42     }
43
44     int ret = 0;
45     size_t size, alen = 0;
46     uint32_t ino;
47     uint32_t blkno = offset / SFS_BLKSIZE;          // The NO. of Rd/wr
begin block
48     uint32_t nblks = endpos / SFS_BLKSIZE - blkno; // The size of Rd/wr
blocks
49
50     //LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf,
sfs_rblock,etc. read different kind of blocks in file
51     /*
52     * (1) If offset isn't aligned with the first block, Rd/wr some content
from offset to the end of the first block
53     *     NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
54     *     Rd/wr size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) :
(endpos - offset)
55     * (2) Rd/wr aligned blocks
56     *     NOTICE: useful function: sfs_bmap_load_nolock, sfs_block_op

```

```

57      * (3) If end position isn't aligned with the last block, Rd/wr some
content from begin to the (endpos % SFS_BLKSIZE) of the last block
58      *      NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
59      */
60
61      // (1) 处理第一个块（可能未对齐）
62      blkoff = offset % SFS_BLKSIZE;
63      if (blkoff != 0) {
64          // (1) 计算第一部分的大小
65          size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
66          // 获取逻辑块号对应的物理块号
67          if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
68              goto out;
69          }
70          // 执行部分块读写
71          if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
72              goto out;
73          }
74          // 更新已处理的长度和缓冲区指针
75          alen += size;
76          buf += size;
77          // 如果还有更多块需要处理
78          if (nblks == 0) {
79              goto out;
80          }
81          blkno++;
82          nblks--;
83      }
84
85      // (2) 处理中间的对齐块
86      while (nblks > 0) {
87          // 获取当前块的物理块号
88          if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
89              goto out;
90          }
91          // 读写多个完整块
92          if ((ret = sfs_block_op(sfs, buf, ino, nblks)) != 0) {
93              goto out;
94          }
95          // 更新已处理的长度和缓冲区指针
96          alen += nblks * SFS_BLKSIZE;
97          buf += nblks * SFS_BLKSIZE;
98          blkno += nblks;
99          nblks -= nblks;
100      }
101
102      // (3) 处理最后一个块
103      size = endpos % SFS_BLKSIZE;
104      // 如果结束位置不在块边界上
105      if (size != 0) {
106          if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
107              goto out;
108          }
109          // 从块的开始位置读写部分数据
110          if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {

```

```

111         goto out;
112     }
113     alen += size;
114 }
115
116 out:
117     *alenp = alen;
118     if (offset + alen > sin->din->size) {
119         sin->din->size = offset + alen;
120         sin->dirty = 1;
121     }
122     return ret;
123 }

```

在这里出现了一些问题，后来排查发现在处理中间对齐块的时候需要调用 *sfs_block_op()* 函数而不是 *sfs_buf_op()* 函数，这样才能一次性读取所有的块，否则读取部分块可能导致错误。

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）

如下是我们编写的 *load_icode()* 函数，下面将使用注释的形式解释。

```

1  // load_icode - called by sys_exec-->do_execve
2
3  static int
4  load_icode(int fd, int argc, char **kargv)
5  {
6      /* LAB8:EXERCISE2 YOUR CODE  HINT:how to load the file with handler fd
7      in to process's memory? how to setup argc/argv?
8          * MACROS or Functions:
9          * mm_create      - create a mm
10         * setup_pgdir     - setup pgdir in mm
11         * load_icode_read - read raw data content of program file
12         * mm_map          - build new vma
13         * pgdir_alloc_page - allocate new memory for TEXT/DATA/BSS/stack
14         parts
15         * lsatp           - update Page Directory Addr Register -- CR3
16         */
17         //You can Follow the code form LAB5 which you have completed to
18         complete
19         /* (1) create a new mm for current process
20         * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
21         * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
22         * (3.1) read raw data content in file and resolve elfhdr
23         * (3.2) read raw data content in file and resolve proghdr based on
24         info in elfhdr
25         * (3.3) call mm_map to build vma related to TEXT/DATA
26         * (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read
27         contents in file
28         * and copy them into the new allocated pages
29         * (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero
30         in these pages
31         * (4) call mm_map to setup user stack, and put parameters into user
32         stack

```

```

26     * (5) setup current process's mm, cr3, reset pgidr (using lsatp MARCO)
27     * (6) setup uargc and uargv in user stacks
28     * (7) setup trapframe for user environment
29     * (8) if up steps failed, you should cleanup the env.
30     */
31     int ret = -E_NO_MEM;
32     struct mm_struct *mm;
33
34     // (1) 创建新内存管理结构
35     if ((mm = mm_create()) == NULL) {
36         goto bad_mm;
37     }
38
39     // (2) 建立页目录表
40     if (setup_pgdir(mm) != 0) {
41         goto bad_pgdir_cleanup_mm;
42     }
43
44     // (3) 复制TEXT/DATA/BSS段内容到进程的内存空间
45     struct Page *page;
46     struct elfhdr elf;
47     struct proghdr ph;
48
49     // 解析ELF文件头
50     if ((ret = load_icode_read(fd, &elf, sizeof(struct elfhdr), 0)) != 0) {
51         goto bad_elf_cleanup_pgdir;
52     }
53
54     // ELF程序是否有效
55     if (elf.e_magic != ELF_MAGIC) {
56         ret = -E_INVALID_ELF;
57         goto bad_elf_cleanup_pgdir;
58     }
59
60     uint32_t vm_flags, perm;
61     uintptr_t elf_entry = elf.e_entry;
62
63     // 加载程序段
64     for (uint32_t i = 0; i < elf.e_phnum; i++) {
65         off_t phoff = elf.e_phoff + sizeof(struct proghdr) * i;
66         if ((ret = load_icode_read(fd, &ph, sizeof(struct proghdr), phoff))
67         != 0) {
68             goto bad_cleanup_mmap;
69         }
70
71         // 读取每个程序段头
72         if (ph.p_type != ELF_PT_LOAD) {
73             continue;
74         }
75         if (ph.p_filesz > ph.p_memsz) {
76             ret = -E_INVALID_ELF;
77             goto bad_cleanup_mmap;
78         }
79
80         // 设置段权限

```

```

80     vm_flags = 0, perm = PTE_U | PTE_V;
81     if (ph.p_flags & ELF_PF_X)
82         vm_flags |= VM_EXEC;
83     if (ph.p_flags & ELF_PF_W)
84         vm_flags |= VM_WRITE;
85     if (ph.p_flags & ELF_PF_R)
86         vm_flags |= VM_READ;
87
88     // 将ELF权限转换为页表权限
89     if (vm_flags & VM_READ)
90         perm |= PTE_R;
91     if (vm_flags & VM_WRITE)
92         perm |= (PTE_W | PTE_R);
93     if (vm_flags & VM_EXEC)
94         perm |= PTE_X;
95     // 建立虚拟内存区域
96     if ((ret = mm_map(mm, ph.p_va, ph.p_memsz, vm_flags, NULL)) != 0) {
97         goto bad_cleanup_mmap;
98     }
99
100     size_t off, size;
101     uintptr_t start = ph.p_va, end, la = ROUNDDOWN(start, PGSIZE);
102     ret = -E_NO_MEM;
103
104     // 加载文件内容到内存
105     end = ph.p_va + ph.p_filesz;
106
107     // 复制文本和数据段
108     while (start < end) {
109         if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
110             goto bad_cleanup_mmap;
111         }
112         off = start - la, size = PGSIZE - off, la += PGSIZE;
113         if (end < la) {
114             size -= la - end;
115         }
116
117         // 从文件中读取
118         if ((ret = load_icode_read(fd, page2kva(page) + off, size,
ph.p_offset + (start - ph.p_va))) != 0) {
119             goto bad_cleanup_mmap;
120         }
121         start += size;
122     }
123
124     // 建立BSS段
125     end = ph.p_va + ph.p_memsz;
126     if (start < la) {
127         if (start == end) {
128             continue;
129         }
130         off = start + PGSIZE - la, size = PGSIZE - off;
131         if (end < la) {
132             size -= la - end;
133         }

```



```

134         memset(page2kva(page) + off, 0, size);
135         start += size;
136         assert((end < 1a && start == end) || (end >= 1a && start ==
1a));
137     }
138     while (start < end) {
139         if ((page = pgdir_alloc_page(mm->pgdir, 1a, perm)) == NULL) {
140             goto bad_cleanup_mmap;
141         }
142         off = start - 1a, size = PGSIZE - off, 1a += PGSIZE;
143         if (end < 1a) {
144             size -= 1a - end;
145         }
146         memset(page2kva(page) + off, 0, size);
147         start += size;
148     }
149 }
150
151 sysfile_close(fd);
152
153 // (4) 设置用户栈
154 vm_flags = VM_READ | VM_WRITE | VM_STACK;
155 if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags,
NULL)) != 0) {
156     goto bad_cleanup_mmap;
157 }
158 if (pgdir_alloc_page(mm->pgdir, USTACKTOP - PGSIZE, PTE_USER) == NULL)
{
159     ret = -E_NO_MEM;
160     goto bad_cleanup_mmap;
161 }
162 if (pgdir_alloc_page(mm->pgdir, USTACKTOP - 2 * PGSIZE, PTE_USER) ==
NULL) {
163     ret = -E_NO_MEM;
164     goto bad_cleanup_mmap;
165 }
166 if (pgdir_alloc_page(mm->pgdir, USTACKTOP - 3 * PGSIZE, PTE_USER) ==
NULL) {
167     ret = -E_NO_MEM;
168     goto bad_cleanup_mmap;
169 }
170 if (pgdir_alloc_page(mm->pgdir, USTACKTOP - 4 * PGSIZE, PTE_USER) ==
NULL) {
171     ret = -E_NO_MEM;
172     goto bad_cleanup_mmap;
173 }
174
175 // (5) 设置内存管理结构
176 mm_count_inc(mm);
177 current->mm = mm;
178 current->pgdir = PADDR(mm->pgdir);
179 lsatp(PADDR(mm->pgdir));
180
181 // (6) 设置命令行参数
182 uint32_t stacktop = USTACKTOP;

```

```

183     uint32_t argv_size = 0;
184     for (int i = 0; i < argc; i++) {
185         argv_size += strlen(kargv[i]) + 1;
186     }
187
188     // 计算参数所需空间
189     uint32_t argv_strs_size = argv_size;
190     uint32_t argv_array_size = (argc + 1) * sizeof(uintptr_t);
191
192     // 放置字符串
193     uint32_t argv_strs_start = stacktop - argv_strs_size;
194     argv_strs_start = ROUNDDOWN(argv_strs_start, 4); // 对齐4字节
195
196     uint32_t argv_array = argv_strs_start - argv_array_size;
197     argv_array = ROUNDDOWN(argv_array, 4); // 对齐4字节
198
199     // 复制参数字符串到用户栈
200     uintptr_t argv_strs[EXEC_MAX_ARG_NUM];
201     uint32_t current_pos = argv_strs_start;
202     for (int i = 0; i < argc; i++) {
203         uint32_t len = strlen(kargv[i]) + 1;
204         argv_strs[i] = current_pos;
205
206         // 确保对应的页面存在
207         uintptr_t la = ROUNDDOWN(current_pos, PGSIZE);
208         pte_t *ptep;
209         struct Page *p = get_page(mm->pgdir, la, &ptep);
210         if (p == NULL) {
211             p = pgdir_alloc_page(mm->pgdir, la, PTE_USER);
212             if (p == NULL) {
213                 ret = -E_NO_MEM;
214                 goto bad_cleanup_current_mm;
215             }
216         }
217         memcpy(page2kva(p) + (current_pos - la), kargv[i], len);
218         current_pos += len;
219     }
220
221     // 复制参数指针数组
222     uintptr_t la = ROUNDDOWN(argv_array, PGSIZE);
223     pte_t *ptep;
224     struct Page *p = get_page(mm->pgdir, la, &ptep);
225     if (p == NULL) {
226         p = pgdir_alloc_page(mm->pgdir, la, PTE_USER);
227         if (p == NULL) {
228             ret = -E_NO_MEM;
229             goto bad_cleanup_current_mm;
230         }
231     }
232     uintptr_t *argv_ptr = (uintptr_t *) (page2kva(p) + (argv_array - la));
233     for (int i = 0; i < argc; i++) {
234         argv_ptr[i] = argv_strs[i];
235     }
236     argv_ptr[argc] = 0; // 空终止符
237

```

```

238 // (7) 初始化陷阱帧
239 struct trapframe *tf = current->tf;
240 uintptr_t sstatus = tf->sstatus;
241 memset(tf, 0, sizeof(struct trapframe));
242
243 /* set user stack pointer */
244 tf->gpr.sp = (uintptr_t)argv_array;
245
246 /* set program entry point (sepc) */
247 tf->epc = (uintptr_t)elf_entry;
248
249 /* set return value for exec in user mode (argc) */
250 tf->gpr.a0 = (uintptr_t)argc;
251
252 /* set argv pointer */
253 tf->gpr.a1 = (uintptr_t)argv_array;
254
255 /* Adjust sstatus: clear SPP (so sret goes to user-mode), set SPIE
256 (enable interrupts after sret) */
257 tf->sstatus = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
258
259 ret = 0;
260 out:
261 return ret;
262 bad_cleanup_current_mm:
263 // (8) 清理当前mm
264 lsatp(boot_pgdir_pa);
265 if (mm_count_dec(mm) == 0) {
266     exit_mmap(mm);
267     put_pgdir(mm);
268     mm_destroy(mm);
269 }
270 current->mm = NULL;
271 current->pgdir = boot_pgdir_pa;
272 goto out;
273 bad_cleanup_mmap:
274     exit_mmap(mm);
275 bad_elf_cleanup_pgdir:
276     put_pgdir(mm);
277 bad_pgdir_cleanup_mm:
278     mm_destroy(mm);
279 bad_mm:
280     goto out;

```

经过上面的修改后，又对`trap.c`中的错误处理部分进行简化处理，去除了`pgfault_handler()`函数，下面将使用注释的形式解释。

```

1 // 具体错误处理
2 void exception_handler(struct trapframe *tf)
3 {
4     int ret;
5     switch (tf->cause)
6     {
7         case CAUSE_MISALIGNED_FETCH:

```

```

8         cprintf("Instruction address misaligned\n");
9         break;
10    case CAUSE_FETCH_ACCESS:
11        cprintf("Instruction access fault\n");
12        break;
13    case CAUSE_ILLEGAL_INSTRUCTION:
14        cprintf("Illegal instruction\n");
15        break;
16    case CAUSE_BREAKPOINT:
17        cprintf("Breakpoint\n");
18        break;
19    case CAUSE_MISALIGNED_LOAD:
20        cprintf("Load address misaligned\n");
21        break;
22    case CAUSE_LOAD_ACCESS:
23        cprintf("Load access fault\n");
24        break;
25    case CAUSE_MISALIGNED_STORE:
26        panic("AMO address misaligned\n");
27        break;
28    case CAUSE_STORE_ACCESS:
29        cprintf("Store/AMO access fault\n");
30        break;
31    case CAUSE_USER_ECALL:
32        // cprintf("Environment call from U-mode\n");
33        tf->epc += 4;
34        syscall();
35        break;
36    case CAUSE_SUPERVISOR_ECALL:
37        cprintf("Environment call from S-mode\n");
38        tf->epc += 4;
39        syscall();
40        break;
41    case CAUSE_HYPERVISOR_ECALL:
42        cprintf("Environment call from H-mode\n");
43        break;
44    case CAUSE_MACHINE_ECALL:
45        cprintf("Environment call from M-mode\n");
46        break;
47    // 指令取值异常
48    case CAUSE_FETCH_PAGE_FAULT:
49        cprintf("Instruction page fault at 0x%08x\n", tf->tval);
50        print_trapframe(tf);
51        if (current != NULL) {
52            do_exit(-E_KILLED);
53        } else {
54            panic("kernel page fault");
55        }
56        break;
57    // 数据加载异常
58    case CAUSE_LOAD_PAGE_FAULT:
59        cprintf("Load page fault at 0x%08x\n", tf->tval);
60        print_trapframe(tf);
61        if (current != NULL) {
62            do_exit(-E_KILLED);

```

```

63     } else {
64         panic("kernel page fault");
65     }
66     break;
67     // 数据存储异常
68     case CAUSE_STORE_PAGE_FAULT:
69         cprintf("Store/AMO page fault at 0x%08x\n", tf->tval);
70         print_trapframe(tf);
71         if (current != NULL) {
72             do_exit(-E_KILLED);
73         } else {
74             panic("kernel page fault");
75         }
76         break;
77     default:
78         print_trapframe(tf);
79         break;
80 }
81 }

```

经过上面的改动，在使用`make qemu`命令时终于可以看到`sh`用户程序的执行界面，输入`exit`、`hello`命令均能够执行，说明实验基本成功！结果如下所示：

```

tom@ubuntu: ~/Downloads/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-ubuntu14/labcode/labcode/lab8
create bin/sfs.Img (disk) successfully.
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.Img
OpenSBI v0.4 (Jul  2 2019 11:53:53)

OpenSBI
i

Platform Name       : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs   : 8
Current Hart        : 0
Firmware Base       : 0x80000000
Firmware Size       : 112 KB
Runtime SBI Version  : 0.1

MPM0: 0x0000000000000000-0x0000000000000000 (A)
MPM1: 0x0000000000000000-0xffffffffffffffff (A,R,M,X)
(THU.CST) os is loading ...

Special kernel symbols:
entry 0xc020004a (virtual)
etext 0xc020072a (virtual)
edata 0xc0201060 (virtual)
end   0xc0206910 (virtual)
kernel executable memory footprint: 603KB
DTB init
HartID: 0
DTB Address: 0xc2200000
Physical Memory from DTB:
Base: 0x0000000000000000
Size: 0x0000000000000000 (128 MB)
End: 0x0000000007fffffff
DTB init completed
Memory management: default_pmm_manager
Physical memory map:
memory: 0x00000000, (0x00000000, 0x07fffffff).
vpaoffset is 184674407048326144
check_alloc_page() succeeded!
Page table directory switch succeeded!
kernel stack guardians set succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloct_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
sched class: RR scheduler
Initrd: 0xc0214010 - 0xc021bd0f, size: 0x00007d00
Initrd: 0xc021bd10 - 0xc021e00f, size: 0x00075900
sfs: mount: 'simple file system' (106/11/117)
vfs: mount disk0.
+ setup timer interrupts
kernel execve: pid = 2, name = "sh".
user sh is running!!!
I am the parent. Forking the child...
I am parent, fork a child pid 4
I am the parent, waiting now..
I am the child.
waitpid 4 ok.
exit pass.
hello world!
I am process 5.
hello pass.

```