华中科技大学

操作系统原理课程实验报告

姓　　名：　　王　国　豪

学　　院：　　计算机科学与技术

专　　业：　　计算机科学与技术

班　　级：　　CS2203

学　　号：　　U202215643

指导教师：　　张　杰

分数	
教师签名	

2024 年 12 月 25 日

目　录

实验一 打印用户程序调用栈

1.1 实验目的

- 理解完整的系统调用过程。
- 通过分析反汇编代码理解用户栈结构，懂得如何通过 fp 寄存器寻找各个栈帧和函数的返回地址。
- 了解 elf 文件中的内容信息，掌握如何通过函数的返回地址，将虚拟地址转换为源代码中的符号。

1.2 实验内容

1.2.1 完整系统调用过程

我们需要回顾一下 lab1_1 学到的系统调用的知识点，来了解一下系统调用完整的流程。我们先从我们的 user/app_print_backtrace.c 出发，通过一步步的函数调用最终会从用户态陷入 S 态，并处理系统调用，见图 1- 1。

```
user/app_print_backtrace.c --> user/user_lib.c --> kernel/strap_vector.S -->
kernel/strap.c --> kernel/syscall.c
```

图 1- 1 基本的系统调用流程

我们发现 print_backtrace 函数没有声明和定义，我们在 user_lib.c 中进行声明定义，并在 syscall.h 中定义一个宏来增加一个新的系统调用号，见表 1- 1。

表 1- 1 print_backtrace 声明与定义

```c
// syscall.h
#define SYS_user_printbacktrace (SYS_user_base +2)

// user_lib.h
int print_backtrace(int a);

// user_lib.c
int print_backtrace(int depth) {
    return do_user_call(SYS_user_printbacktrace, depth, 0, 0, 0, 0, 0, 0);
}
```

在 print_backtrace 函数中我们需要调用 do_user_call 函数来实现系统调用，第一个参数就是我们刚刚定义的系统调用号，第二个参数就一次放置我们需要传入

的参数，我们这里需要传入的就是我们想要递归的深度。

经过一系列的转换 handle_syscall 将 a0~a7 寄存器的值交给函数 do_syscall，我们接着会进入 strap.c 的 do_syscall 函数，这个函数通过 switch case, 来对应处理我们的系统调用请求，此处我们应该增加一个 case 来处理我们的 SYS_user_printbacktrace，见表 1- 2，接着我们需要再次定义一个处理的函数来解决我们的栈回溯和函数名打印任务。

表 1- 2 S 态下系统调用处理函数

```
// strap.c
// do_syscall
    case SYS_user_printbacktrace:
        return sys_user_printbacktrace(a1);

// sys_user_printbacktrace
ssize_t sys_user_printbacktrace(uint64 depth) {
    // todo: .....
    return 0;
}
```

1.2.2 用户栈结构理解和查找函数返回地址

首先我们需要了解 riscv 中的栈结构，通过查看指导手册，函数调用的栈帧结构见图 1- 2。

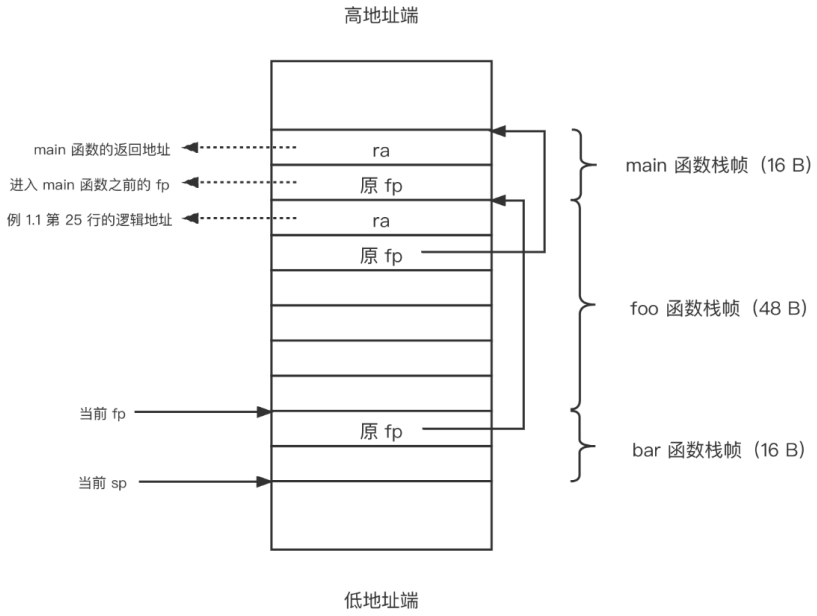

图 1- 2 riscv 的栈结构示意图

值得注意的是，我们当前用户栈是处于 do_user_call 函数的栈帧中，并且它的栈帧结构和 print_backtrace 以及 f1~f8 的栈帧结构不太相同，这个需要我们去

对文件进行反汇编，查看汇编代码以及尝试打印 do_user_call 的栈结构来具体分析。

```
0000000081000000 <f8>:
    81000000:   1141                    addi    sp,sp,-16      # 分配空间
    81000002:   e406                    sd  ra,8(sp)           # 保存ra(即返回地址)
    81000004:   e022                    sd  s0,0(sp)           # 保存 s0(即fp)
    81000006:   0800                    addi    s0,sp,16
    81000008:   451d                    li  a0,7
    8100000a:   160000ef            jal ra,8100016a <print_backtrace>   # 此时会把epc+4的地
址放入 ra 中
    8100000e:   60a2                    ld  ra,8(sp)
    ......
```

```
00000000810000ca <do_user_call>:
    810000ca:   1101                    addi    sp,sp,-32      # 分配空间
    810000cc:   ec22                    sd  s0,24(sp)          # 保存s0
    810000ce:   1000                    addi    s0,sp,32
    810000d0:   00000073            ecall
    ......
```

图 1- 3 部分代码的反汇编代码

我们首先需要了解基本的 jal 指令的运作流程，jal（Jump and link）会进行两个操作，跳转并链接，会将 epc+4 的地址放到 ra 中，也就是说返回地址放入 ra 中，之后再跳转到指定的目的地址。

通过分析图 1- 3，我们可以看到在 do_user_call 函数中，旧 s0（也就是 print_traceback 函数的 fp）保存在了 do_user_call 的 fp-8 的位置，而且在 f8 的代码中，旧 s0（也就是 f7 的 fp）保存在 f8 的 fp-16 的位置，并且 f8 中还保存了返回地址 ra 到 fp-8 的位置，这个信息也印证了图 1- 2 的内容。

通过上面的分析以及对堆栈的打印，我们就可以通过循环的方式得到 f1~f8 的返回地址，要注意 do_user_call 的堆栈不太一样，我们要特别处理，具体的流程见表 1- 3，我们先由当前的 fp（do_user_call 的 fp）来获得 print_backtrace 的 fp，然后通过循环依次获得返回地址。

表 1- 3 递归查找返回地址代码

```
// sys_user_printbacktrace
ssize_t sys_user_printbacktrace(uint64 depth) {
    uint64 fp_user_call = current->trapframe->regs.s0;  // 获得当前的fp
    uint64 fp_traceback = *((uint64*)(fp_user_call - 8));  // 获得
print_backtrace的fp
    uint64 beforefp = fp_traceback;
    for (int i = 0; i < depth; i++) {
        uint64 nowfp = *((uint64*)(beforefp - 16));
        uint64 nowra = *((uint64*)(beforefp - 8));  // 获得返回地址
        if (find_closest_function(nowra) == 0)  return 0;  // 通过返回地址
得到函数名
```

```
        beforefp = nowfp;
    }
    return 0;
}
```

1.2.3 通过函数返回地址得到函数名

通过上面的分析，我们可以得到 f1~f8 的返回地址，那么如何通过这些返回地址来得到我们想要的函数名呢？我们需要了解一下 elf 文件的结构，见图 1- 4。

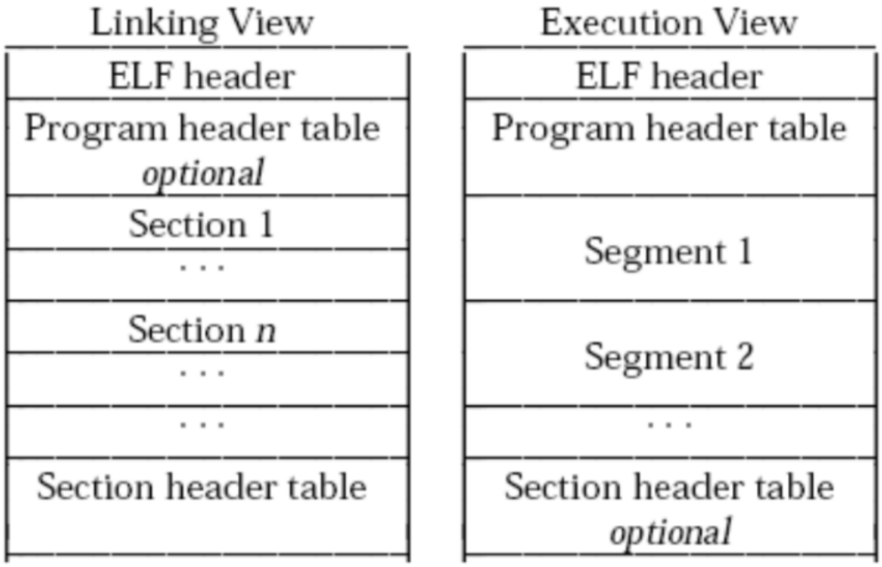

图 1- 4 elf 文件结构

我们先分析 elf 头存储的内容，见 shoff 记录了第一个 Section Header 的位置，shentsize 代表了一个 Section Header 的大小，shnum 代表一共有多少个 Section Header，shstrndx 代表了 String Table 的索引值，而且我们需要注意.shstr 节，这个节包含了所有节的字符串，但是函数名不在这里。

表 1- 4 elf 头结构体

```
typedef struct elf_header_t {
uint32 magic;
uint8 elf[12];
uint16 type;      /* Object file type */
uint16 machine;   /* Architecture */
uint32 version;   /* Object file version */
uint64 entry;     /* Entry point virtual address */
uint64 phoff;     /* Program header table file offset */
uint64 shoff;     /* Section header table file offset */
uint32 flags;     /* Processor-specific flags */
uint16 ehsize;    /* ELF header size in bytes */
```

```
uint16 phentsize; /* Program header table entry size */
uint16 phnum;     /* Program header table entry count */
uint16 shentsize; /* Section header table entry size */
uint16 shnum;     /* Section header table entry count */
uint16 shstrndx;  /* Section header string table index */
} elf_header;

typedef struct elf_shdr_t
{
  uint32 sh_name;
  uint32 sh_type;
  uint64 sh_flags;
  uint64 sh_addr;
  uint64 sh_offset;
  uint64 sh_size;
  uint32 sh_link;
  uint32 sh_info;
  uint64 sh_addralign;
  uint64 sh_entsize;
} elf_shdr;
```

我们首先需要获取这个节头，读取后放入表 1- 4 中的 elf_shdr_t 结构体中，其中有三个特别重要的字段，分别是 name，offset 和 size，name 是节名字符串在.shstr 的字节偏移，offset 是该节相对于 elf 文件头的字节偏移量，size 就是这节的大小，到这里我们就可以将.shstr 节的内容到 shstr 变量中，然后通过节名获取任意节在 elf 文件中的位置。

接下来我们就可以读取节的内容，通过 elf_fpread 还能进一步将.strtab 节和.symtab 加载到内存，接着我们就可以通过我们的函数返回地址来获取我们的函数名，具体实现如下：遍历.symtab 中的 elf_sym 结构，看看哪个结构的[st_value, st_value + st_size)包含该指令的地址，根据刚刚找到的 elf_sym 的 st_name 字段，在.strtab 中找到相应的字符串，即为当前指令所处的函数的名称。

1.3 实验调试及心得

1.3.1 实验调试与过程分析

- 问题 1：无法确定栈帧结构，在查看用户栈的时候由于对 riscv 的栈结构没有了解，不知道如何找到返回地址

- 解决：对代码进行反汇编，分析汇编代码，结合任务书进行分析，并且因为 do_user_call 的栈和其他函数的栈还有一些区别，故手动调试，将 do_user_call 的 fp 指针上下的地址都打印出来，分析 print_backtrace 的 fp 在什么位置，具体的调试代码见表 1- 5。
- 结果：通过深入了解堆栈情况后，就可以清楚的通过循环来找到每一个函数的返回地址，成功解决。

表 1- 5 手动调试堆栈的代码

```
// 调试代码
for (int i = -3; i <= 3; i++) {
    uint64 fpp = current->trapframe->regs.s0 + i * 8;
    uint64 values0 = (*((uint64*)fpp));
    sprint("now's address %p   value %p\n", fpp, values0);
}
```

1.3.2 实验心得

第一次做挑战实验，耗时很长，也十分考察综合能力，通过一次次查找资料，也渐渐对 riscv 的堆栈，elf 文件，以及系统调用的具体流程有了清楚的认识，总而言之，收获颇丰。

头哥上所有测试集已通过。

1.4 实验建议

在这个实验中，在 elf 文件中查找函数名难度过大，耗时很久，希望能够削减实验难度，增加一些接口函数，不用自己手搓读取 elf 文件的工具函数。

实验二 复杂缺页异常

1.1 实验目的

- 了解缺页异常发生的原因，区分不同原因的缺页处理方式
- 实现缺页处理的程序

1.2 实验内容

1.2.1 缺页异常的原因

通过查阅资料以及结合任务书 lab2_3 的基础实验，我们可以分析得出在我们实验中缺页有两种情况，分别是在栈和堆中，本质情况上都是访问的虚拟地址和物理地址之间没有建立起映射关系，这样不一定是因为我们访问的虚拟地址是非法的，要具体情况具体考虑。

（1）堆中的缺页：

动态内存分配：程序使用如 malloc（C 语言）或 new（C++）等函数在堆上分配内存。当分配的内存区域对应的虚拟地址尚未映射到物理内存时，首次访问该内存区域就会触发缺页。例如，程序通过 malloc 申请了一大块内存，但操作系统可能并未立即将所有虚拟地址映射到物理页，当程序试图读写这块新分配内存中尚未映射的部分时，缺页就会发生。另外一种就是如果访问到了分配空间之外的地址，就是我们常说的数组下标访问越界，也会触发缺页异常，但这种就是非法的行为，这也是这个挑战实验要完成的判断内容。

（2）栈中的缺页：

递归调用：递归函数调用时，每次调用的函数上下文（包括参数、局部变量和返回地址等）都要压入栈中。当递归深度过大，栈空间不断扩展，新扩展的栈空间对应的虚拟地址若没有映射到物理页，就会引发缺页。例如，一个没有正确终止条件的递归函数，会不断消耗栈空间，最终可能触发栈缺页。

大局部变量：在函数内部声明大型数组或结构体等局部变量时，这些变量占用的栈空间可能较大。如果栈空间不足，且对应的虚拟地址没有映射到物理页，访问这些局部变量时也可能导致缺页。这个也是 lab2_3 中所要实现的，每当出现缺页的时候，就将未建立地址映射关系的虚拟地址映射到空闲的物理地址上。

区分好是堆还是栈上的缺页很重要，这是区分 lab2_3 和挑战实验的关键。

1.2.2 如何实现缺页异常处理程序

有了 lab2_3 中的处理经验，其实挑战实验的本质就是区分是堆上的缺页还是栈上的缺页，我们通过查看 menlayout.h 文件，发现已经设置了栈顶的地址，我们就再设置一个栈底的地址，见表 2- 1。当然，这个范围只是我们假定的，我这里假定的栈的大小有 1024 个 Pagesize，值得注意的是，栈是从高地址向低地址生长的。我们假定我们设置的栈范围以外的空间就是堆的空间。

表 2- 1 栈范围设置

```
// default stack size
#define STACK_SIZE 4096

// virtual address of stack top of user process
#define USER_STACK_TOP 0x7ffff000

#define USER_STACK_BOTTOM (0x7ffff000-PGSIZE * 1024)
```

我们只用在 lab2_3 的基础上，对缺页中断进行一个额外的判断，如果当前的虚拟地址是处于堆的位置，那么我们就调用 panic 函数，进行报错提醒，到这里这一关就可以通过了。

```
// void handle_user_page_fault
case CAUSE_STORE_PAGE_FAULT:
    if (stval >= USER_STACK_BOTTOM && stval <= USER_STACK_TOP)
        map_pages(current->pagetable, ROUNDDOWN(stval, PGSIZE), PGSIZE,
(uint64)alloc_page(), prot_to_type(PROT_READ | PROT_WRITE, 1));
    else {
        panic("this address is not available!");
    }
```

1.3 实验调试与心得

这一个挑战实验的本质就是区分好堆和栈上的缺页问题，没有遇到啥问题，但是也是对我理解缺页问题有很大的帮助。

头哥上所有测试集已通过。

1.4 实验建议

相比与其他的实验的挑战实验，这个挑战实验过于开放，且难度较低，希望能够在文档中增加关于堆栈范围的内容，这样更加具有引导性。

实验三 进程等待和数据段复制

1.1 实验目的

- 理解进程管理和进程切换
- 清楚子进程的创建过程，明白如何为子进程分配资源，如何复制父进程的代码段，数据段，堆栈等信息。
- 实现进程等待，以及父进程的返回值设置。

1.2 实验内容

需要有文字解释设计思路，只能贴关键代码或数据结构！

1.2.1 数据段复制

Lab3 的基础实验中已经实现了堆栈以及代码段的复制工作，通过查看代码，我们也得知，这些内容在初始化的时候都是分配了一个页面大小，后续我们处理数据段的时候也只用拷贝一个页面就行，值得注意的是，和代码段不同的是代码段由于不需要修改，可以和父进程共享一块物理内存，而我们的数据段就不同了，父进程和子进程的数据应该存储在不同的位置，见表 3-1。

表 3-1 数据段拷贝代码

```
// process.c  do_fork函数
case DATA_SEGMENT:
    do {
        child->mapped_info[child->total_mapped_region].va =
parent->mapped_info[i].va;   // 同一虚拟地址
        uint64 pa = (uint64)alloc_page();   // 分配物理空间（一个页）
        user_vm_map((pagetable_t)child->pagetable,
parent->mapped_info[i].va, PGSIZE, pa, prot_to_type(PROT_WRITE |
PROT_READ, 1));   // 建立虚拟地址和物理地址映射
        memcpy((void*)pa, (void*)lookup_pa(parent->pagetable,
parent->mapped_info[i].va), PGSIZE);  // 拷贝内容
        child->mapped_info[child->total_mapped_region].npages =
parent->mapped_info[i].npages;
        child->mapped_info[child->total_mapped_region].seg_type =
DATA_SEGMENT;
        child->total_mapped_region++;
    } while (0);
```

注意，实验中定义的 mapped_info 数组会记录当前存在有多少个段的信息，我们新增一个就要让 total_mapped_region 加一，而且我们的数据段的权限应该设置为可读和可写。

1.2.2 实现进程等待

我们还需要实现一个 wait 函数来实现进程等待，这个过程也是需要实现一个系统调用，和我们之前实现的系统调用的流程一样，我们需要增加一个系统调用号 SYS_user_wait，后续在调用 do_user_call 之后我们还会进入 s 态处理系统调用，我们在 do_syscall 函数中增加一个 case 来选择处理 wait 系统调用。

首先，我们先定义几个数据结构，见表 3-2，来方便处理父进程等待子进程所需要的信息，具体的功能我都写在了代码的注释中。

表 3-2 数据结构的定义

```
process* bolck_queue_list = NULL;         // 阻塞队列
process* father_queue_list[NPROC] = {};   // 记录父进程
int except_pid[NPROC] = {};    // 父进程等待的具体的子进程的 pid
```

如何初始化我们的数据结构呢，在父进程调用 fork 函数的时候我们就需要记录每一个子进程的父进程，见表 3-3。并且我们仿照了插入就绪队列的代码仿写了插入阻塞队列的代码。

表 3-3 记录父进程代码和插入阻塞队列代码

```
// do_fork
sprint("will fork a child from parent %d.\n", parent->pid);
process* child = alloc_process();

father_queue_list[child->pid] = parent;  // 记录父进程


void insert_to_block() {  // 插入阻塞队列
    if (bolck_queue_list == NULL) {
        current->status = BLOCKED;
        current->queue_next = NULL;
        bolck_queue_list = current;
    }
    else {
        current->status = BLOCKED;
        current->queue_next = bolck_queue_list;
        bolck_queue_list = current;
    }
}
```

有了上面的前置准备，我们就可以开始根据实验的具体要求，来编写我们的处理函数，根据任务书的要求，当父进程调用 wait 函数时候如果传入的参数 pid

不同，要进行不同的处理：

（1） 当 pid 为-1 时，父进程等待任意一个子进程退出即返回子进程的 pid，这种情况下，我们就直接让父进程变为阻塞状态，然后进行进程调度函数就行。

（2） 当 pid 大于 0 时，父进程等待进程号为 pid 的子进程退出即返回子进程的 pid，这种情况也是类似，不过需要在我们定义的 except_pid 中记录，方便后续我们处理。

（3） 如果 pid 不合法或 pid 大于 0 且 pid 对应的进程不是当前进程的子进程，返回-1，这种情况需要我们判断 pid 是否合法，可以简单判断 pid 是否小于-1 或者在我们的 father_queue_list 结构中是否保存了两个进程的父子关系。

具体的实现代码见表 3- 4。

表 3- 4 根据具体 pid 处理的代码

```
ssize_t sys_user_wait(int pid) {
    if (pid == -1) {
        insert_to_block();  // 插入阻塞队列
        schedule();
    }
    else if (pid > 0) {
        int flag = 1;
        if (father_queue_list[pid] == NULL ||
father_queue_list[pid]->pid != current->pid) {   // 判断是否存在父子进程的关系
            return -1;
        }
        else {
            except_pid[current->pid] = pid;   // 记录当前父进程想要等待的子进程的pid
            insert_to_block();
            schedule();
        }
    }
    // 处理pid小于0等的情况
    return -1;
}
```

到这里还有一个关键的知识点没有考虑，如何在子进程退出的时候，把自己的 pid 传给父进程呢？这个知识点十分关键，fa->trapframe->regs.a0 = proc->pid，十分巧妙的就是，我们可以直接获取父进程，然后设置父进程保存返回值的寄存器 a0 就可以间接设置父进程的返回值。

每一个进程在退出的时候都会调用 free_process 函数，我们可以在这个函数里面增加代码，在子进程结束的时候唤醒父进程并设置父进程的返回值。具体的代码实现见表 3- 5。

表 3- 5 子进程退出的设置代码

```
int free_process(process* proc) {
    proc->status = ZOMBIE;
    if (father_queue_list[proc->pid]) {  //如果这个进程有父进程才会处理
        process* fa = father_queue_list[proc->pid]; // 取出父进程
        int fa_pid = (int)fa->pid;
        if (except_pid[fa_pid]) { //如果这个父进程规定了等待的子进程的id
            if (except_pid[fa_pid] == proc->pid) {
                fa->trapframe->regs.a0 = proc->pid; // 设置返回值
                insert_to_ready_queue(fa);
                except_pid[fa_pid] = 0; // 清空
            }
        }
        else {
            insert_to_ready_queue(fa);
            fa->trapframe->regs.a0 = proc->pid;
        }
        father_queue_list[proc->pid] = NULL;
    }
    return 0;
}
```

通过实现上面的代码，就可以实现任务啦。

1.3 实验调试及心得

1.3.1 实验调试以及结果分析

- 在本地进程调试的时候出现了下面的报错，显示就绪队列为空，但是还有未处理的进程。

ready queue empty, but process 0 is not in free/zombie state:3

ready queue empty, but process 1 is not in free/zombie state:1

Not handled: we should let system wait for unfinished processes.

解决措施：通过打印中间过程，发现是在唤醒父进程的时候，只是单纯的将父进程的 status 设置为 READY，没有将其放入就绪队列中，正确的做法就是调用 insert_to_ready_queue 中就可以解决问题。

1.3.2 心得体会

在这个实验中我感觉是最有趣的，把进程调度的各个流程都理解的十分清晰，我觉得设计新的数据结构来完成挑战实验是一件有趣的事情，合理的数据结构可以让我们写代码更加方便，完成代码后不断的 debug 也是对自己调试能力的考验，总而言之，乐在其中。

实验四 相对路径

1.1 实验目的

- 理解 riscv 中文件结构，了解系统调用中如何进行文件操作
- 通过修改 PKE 文件系统代码，提供解析相对路径的支持
- 完成用户层 pwd 函数（显示进程当前工作目录）、cd 函数（切换进程当前工作目录）

1.2 实验内容

1.2.1 实现 pwd 显示当前的工作目录

和之前的实验一样，我们这里也是需要通过调用系统函数来实现我们的目标，我们增加新的系统调用号，然后在我们仿照其他的调用函数编写两个系统调用函数，见表 4-1，表中还包含了 cd 切换工作目录的系统函数，一并写在一起。

表 4-1 系统调用函数

```
ssize_t sys_read_cwd(char* path) {
    char* pfn = (char*)user_va_to_pa((pagetable_t)(current->pagetable),
(void*)path);
    return do_read_cwd(pfn);
}

ssize_t sys_change_cwd(char* path) {
    char* pfn = (char*)user_va_to_pa((pagetable_t)(current->pagetable),
(void*)path);
    return do_change_cwd(pfn);
}
```

有一个细节需要注意，我们需要将我们的虚拟地址先转换成物理地址再进行处理，然后我们可以通过 current->pfiles->cwd->name 来获得当前的文件名，然后通过 pre = pre->parent 来获取上一级的 dentry，然后循环操作直到根目录，我们最后还需要加上根目录的‘/’。

表 4-2 循环查找目录代码

```
int do_read_cwd(char* path) {
    char answer[80] = {};
    struct dentry* pre = current->pfiles->cwd;
```

```
    // 循环查找上一级目录
    while (pre->parent) { // 只有不是根目录才处理
        char temp[32] = {};
        strcpy(temp, pre->name);
        char b[34] = { '/','\0' };
        strcat(b, temp);
        strcat(b, answer);
        strcpy(answer, b);
        }
        pre = pre->parent;
    }
    if (strlen(answer) == 0) {
        answer[0] = '/';   // 单独处理根目录
    }
    strcpy(path, answer);
    //sprint(name);
    return 0;
}
```

1.2.2 实现 cd 切换工作目录

我们这个切换工作目录可以换个角度思考而不用去修改底层的文件系统，我们可以直接将相对路径转换成绝对路径，这样我们进行切换的时候就不会出现问题，从而巧妙解决我们的问题。

首先我们先写一个负责将相对路径转换成绝对路径的函数，它接收当前绝对路径前缀 prefix 和相对路径 path，先以'/'分割相对路径，对前缀末尾斜杠特殊处理，接着遍历相对路径各部分，遇到'.'跳过，遇到'..'则回退上级目录（通过调整前缀指针位置），其他情况将其拼接到前缀后，最后在合适位置添加字符串结束符'\0'，并返回转换后的绝对路径（即修改后的 prefix）。具体的代码实现见表 4-3。

表 4-3 实现工作目录切换的代码

```
// 将相对路径转换成绝对路径
char* relative_to_absolute(char* prefix, char* path) {
    char* dir = strtok(path, "/");
    size_t sz = strlen(prefix);
    if (prefix[sz - 1] == '/') {
        sz -= 1;
    }
    for (; dir; dir = strtok(NULL, "")) {
        if (strcmp(dir, ".") == 0) {
            continue;
```

```
        }
        else if (strcmp(dir, "..") == 0) {
            while (sz > 0 && prefix[--sz] != '/');
        }
        else {
            prefix[sz] = '/';
            while ((prefix[++sz] = *dir++));
        }
    }
    prefix[MAX(1, sz)] = '\0';
    return prefix;
}


int do_change_cwd(char* path) {

    char abs_path[MAX_PATH_LEN], path_cpy[MAX_PATH_LEN];
    strcpy(path_cpy, path);
    if (path[0] == '/')  // 如果已经是绝对路径
        strcpy(abs_path, path);
    else {  // 如果是相对路径就要转换成绝对路径
        do_read_cwd(abs_path);
        relative_to_absolute(abs_path, path_cpy);
    }
    char buf[32];
    current->pfiles->cwd = lookup_final_dentry(abs_path,
&vfs_root_dentry, buf);
    return 0;
}
```

有了将绝对路径转换成相对路径的函数，我们就可以开始处理代码的逻辑部分，如果传进来的地址以‘/’开头，说明已经是绝对地址了，就不需要转换，否则我们将调用我们写好的函数进行处理，最后，我们将 current->pfiles->cwd 赋值为我们新的目录的 dentry 就可以完成任务了。

1.3 实验调试及心得

1.3.1 实验调试与过程分析

- 一开始写好代码运行的时候，运行时报错，查找后发现是没有将虚拟地址转

换为物理地址，在内核态处理时用到的应该是物理地址，利用 User_va_to_pa 后可以通过

测试完 bug 后提交到头哥，圆满通关。

1.3.2 心得与体会

这个挑战实验花费时间最长，主要精力都花在阅读文档上了，各种函数的增加极大考验了我的能力，通过查阅资料和寻求他人的帮助，我对 linux 的文件系统有了更深的印象和理解，对平常经常使用的 pwd 和 cd 指令的工作原理有了大概的了解，受益匪浅。