实验报告：参数传递和系统调用

课程名称：操作系统	**年级：**2023 级	**上机实践成绩：**
指导教师：张民	**姓名：**张建夫	
上机实践名称：参数传递和系统调用	**学号：** 10235101477	**上机实践日期：** 2024/12/16
上机实践编号：	**组号：**	**上机实践时间：** 14:50~16:30 点

一、目的

了解参数传递并将其实现，实现用户程序和 OS 之间的系统调用，搞明白参数传递和系统调用在 pintos 里具体是如何实现的。（本次实验使用 cs162，已和张民老师和周孜为助教沟通过）

二、内容与设计思想

1.参数传递

要实现参数传递，先要搞明白 pintos 中进程是如何创建的，命令行参数肯定是在进程创建的同时传入的，对于 cs162，所有测试都是通过 run_task 函数创建一个用户进程进行测试：

```
/* Runs the task specified in ARGV[1]. */
static void run_task(char** argv) {
  const char* task = argv[1];

  printf("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait(process_execute(task));
#endif
  printf("Execution of '%s' complete.\n", task);
}
```

可以看到 pintos 的主程序调用了 process_wait 等待该进程完成，process_execute 创建该进程，而 task，就是整个用户传入的字符串（包括参数），因此要实现参数传递，肯定是在 process_execute 这个函数及其子函数里做手脚，现在再看 process_execute 这个函数（截图的是 cs162 github 上的原始代码，这样我就不用将改动代码删了再截图了）：

```
49      /* Starts a new thread running a user program loaded from
50         FILENAME.  The new thread may be scheduled (and may even exit)
51         before process_execute() returns.  Returns the new process's
52         process id, or TID_ERROR if the thread cannot be created. */
53      pid_t process_execute(const char* file_name) {
54        char* fn_copy;
55        tid_t tid;
56
57        sema_init(&temporary, 0);
58        /* Make a copy of FILE_NAME.
59           Otherwise there's a race between the caller and load(). */
60        fn_copy = palloc_get_page(0);
61        if (fn_copy == NULL)
62          return TID_ERROR;
63        strlcpy(fn_copy, file_name, PGSIZE);
64
65        /* Create a new thread to execute FILE_NAME. */
66        tid = thread_create(file_name, PRI_DEFAULT, start_process, fn_copy);
67        if (tid == TID_ERROR)
68          palloc_free_page(fn_copy);
69        return tid;
70      }
71
```

可以看到该函数简单的拷贝了一条命令行就创建了新线程（此处创建线程是因为 pintos 中所有进程都是由线程模拟的），没有任何的措施来实现父子进程的通信（此处要实现通信是为了给 wait 和 execute 两个系统调用做铺垫，父进程必须知道子进程的可执行表是否加载成功，没成功就要杀掉子进程），接着来看 thread_create 函数：

```
176   tid_t thread_create(const char* name, int priority, thread_func* function, void* aux) {
177     struct thread* t;
178     struct kernel_thread_frame* kf;
179     struct switch_entry_frame* ef;
180     struct switch_threads_frame* sf;
181     tid_t tid;
182
183     ASSERT(function != NULL);
184
185     /* Allocate thread. */
186     t = palloc_get_page(PAL_ZERO);
187     if (t == NULL)
188       return TID_ERROR;
189
190     /* Initialize thread. */
191     init_thread(t, name, priority);
192     tid = t->tid = allocate_tid();
193
194     /* Stack frame for kernel_thread(). */
195     kf = alloc_frame(t, sizeof *kf);
196     kf->eip = NULL;
197     kf->function = function;
198     kf->aux = aux;
199
200     /* Stack frame for switch_entry(). */
201     ef = alloc_frame(t, sizeof *ef);
202     ef->eip = (void (*)(void))kernel_thread;
203
204     /* Stack frame for switch_threads(). */
205     sf = alloc_frame(t, sizeof *sf);
206     sf->eip = switch_entry;
207     sf->ebp = 0;
208
209     /* Add to run queue. */
210     thread_unblock(t);
211
212     return tid;
213   }
```

该函数创建好测试主要运行的线程，并设置好该线程的运行函数，最后将该线程加入运行队列，回头看前一张图可以得知，运行函数为 start_process，因此我们看一下 start_process 函数：

```
72    /* A thread function that loads a user process and starts it
73       running. */
74    static void start_process(void* file_name_) {
75      char* file_name = (char*)file_name_;
76      struct thread* t = thread_current();
77      struct intr_frame if_;
78      bool success, pcb_success;
79
80      /* Allocate process control block */
81      struct process* new_pcb = malloc(sizeof(struct process));
82      success = pcb_success = new_pcb != NULL;
83
84      /* Initialize process control block */
85      if (success) {
86        // Ensure that timer_interrupt() -> schedule() -> process_activate()
87        // does not try to activate our uninitialized pagedir
88        new_pcb->pagedir = NULL;
89        t->pcb = new_pcb;
90
91        // Continue initializing the PCB as normal
92        t->pcb->main_thread = t;
93        strlcpy(t->pcb->process_name, t->name, sizeof t->name);
94      }
95
96      /* Initialize interrupt frame and load executable. */
97      if (success) {
98        memset(&if_, 0, sizeof if_);
99        if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
100       if_.cs = SEL_UCSEG;
101       if_.eflags = FLAG_IF | FLAG_MBS;
102       success = load(file_name, &if_.eip, &if_.esp);
103     }
```

```
104
105       /* Handle failure with succesful PCB malloc. Must free the PCB */
106       if (!success && pcb_success) {
107         // Avoid race where PCB is freed before t->pcb is set to NULL
108         // If this happens, then an unfortuantely timed timer interrupt
109         // can try to activate the pagedir, but it is now freed memory
110         struct process* pcb_to_free = t->pcb;
111         t->pcb = NULL;
112         free(pcb_to_free);
113       }
114
115       /* Clean up. Exit on failure or jump to userspace */
116       palloc_free_page(file_name);
117       if (!success) {
118         sema_up(&temporary);
119         thread_exit();
120       }
121
122       /* Start the user process by simulating a return from an
123          interrupt, implemented by intr_exit (in
124          threads/intr-stubs.S).  Because intr_exit takes all of its
125          arguments on the stack in the form of a `struct intr_frame',
126          we just point the stack pointer (%esp) to our stack frame
127          and jump to it. */
128       asm volatile("movl %0, %%esp; jmp intr_exit" : : "g"(&if_) : "memory");
129       NOT_REACHED();
130     }
131
```

这个函数才是初始化进程的部分，为进程分配 pcb 块（ucb 的 cs162 将 pcb 块单拎出来了（实际上就是自己写了个 struct），和学校使用的斯坦福的不同），加载可执行表（load 函数），通过模拟中断返回正式开启用户进程（最后的汇编部分 asm volatile），到这儿，我们就已经可以确定参数传递的代码修改范围（就是上面给出的几个函数）。

确定修改范围后，我们还要了解 pintos 是怎么将命令行参数传入的，对于 pintos，任何用户进程开始运行后（执行最后的汇编部分 asm volatile），都会运行如下函数：

```
void _start (int argc, char *argv[]) {
    exit (main (argc, argv));
}
```

（来自 cs162 pintos documentaton）

那么对于函数，传参的方式就是压栈，将所有的参数压栈即可，而对于初始的压栈，pintos 有一些额外的要求，以文档中的例子为例：

命令行参数为：/bin/ls -l foo bar

压栈的结果是：

The table below shows the state of the stack and the relevant registers right before the beginning of the user program, assuming `PHYS_BASE` is 0xc0000000:

Address	Name	Data	Type
0xbffffffc	argv[3][...]	bar\0	char[4]
0xbffffff8	argv[2][...]	foo\0	char[4]
0xbffffff5	argv[1][...]	-l\0	char[3]
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]
0xbfffffec	stack-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

可以看到命令行参数倒序推入栈，接着为了使接下来的指针部分保持 16 字节对齐（需要注意的是，这里的对齐指的是从 argv[argc]一直到假返回地址这个部分对齐，以上图为例，stack-align 的地址为 ec 结尾，那么假返回地址也必须是什么 c 结尾，这个部分最初我做的时候花了 3 天时间才搞明白这个对齐什么意思，之前一直理解为全部对齐，一直过不了测试）然后根据 C 的标准，需要推入 argv[argc]，再推入各个字符串的地址，接着推入 argc，参数的个数，最后推入假返回地址模拟函数传参。

2.系统调用

对于系统调用，我们也要首先知道 pintos 是怎么进行系统调用的，pintos 的系统调用是通过程序的‘陷阱’实现的，程序通过执行 int $0x30（一条汇编语句）进行系统调用（当然，再执行这一句之前程序已经将所有的参数推入栈上），至于为什么是这样可以观察下面这个函数：（位于 syscall.c 中）

```
void syscall_init(void) { intr_register_int(0x30, 3, INTR_ON, syscall_handler, "syscall"); }
```

该函数初始化了调用 int $0x30 时将会执行的函数（syscall_handler），保证在开中断的情况下才进行调用。

在了解了这些之后，观察我们要修改的主函数 syscall_handler：

```
static void syscall_handler(struct intr_frame* f UNUSED) {
  uint32_t* args = ((uint32_t*)f->esp);

  /*
   * The following print statement, if uncommented, will print out the syscall
   * number whenever a process enters a system call. You might find it useful
   * when debugging. It will cause tests to fail, however, so you should not
   * include it in your final submission.
   */

  /* printf("System call number: %d\n", args[0]); */

  if (args[0] == SYS_EXIT) {
    f->eax = args[1];
    printf("%s: exit(%d)\n", thread_current()->pcb->process_name, args[1]);
    process_exit();
  }
}
```

该函数只实现了一个 sys_exit，并且还实现不对（没有任何的参数地址检查，作为运行在内核层面的代码应该检测用户传来的任何东西，如果用户试图访问一个非法地址，例如没有页表映射的地址，内核应根据不同的系统调用进行相应的处理）。其他值得一提的就是 intr_frame 中的 esp（此处为何 esp 指向参数可以参阅 intr-stubs.S，pintos 所有中断都是从这进去从这出来）指向了用户传入的第一个参数（args[0]是系统调用号），由此实现用户参数的获取。

三、使用环境

1. 主机 os：Windows-11
2. 虚拟机 os:Linux
3. 虚拟环境：docker 的 linux 镜像
4. 代码编辑器:vscode

四、实验过程

1.参数传递：

首先修改 pcb 块和 tcb 块，并构建父子进程交流的结构体：

Tcb 块添加了：

```
int exit_status;                    /*退出状态*/
```

exit 系统调用需要用到

```
int cur_file_fd;                    /*下一个使用的文件描述符*/
struct list open_files;             /*所有打开的文件*/
```

用于处理线程打开的文件（文件相关的系统调用需要用到）

```
#ifdef USERPROG
  /* Owned by process.c. */
  struct list*child_process;        /*子进程的list*/
  struct process* pcb; /* Process control block if this thread is a userprog */
  struct list_elem elem_process;
  struct thread *father;            /*进程他爸*/
  struct semaphore wait_for_child;  /*调用wait时使用的信号量*/
  bool waited;                      /*该子进程是否已被等待过*/

  /*浮点数状态保存*/
  struct fpu_state fs;  /*当前进程的fpu状态*/
#endif
```

实现 wait 系统调用所需的各种参数（fpu_state 不用管，这个是 cs162 特有的测试）

Pcb 块：

此处所添加的字段注释已表达清楚

```
/* The process control block for a given process. Since
   there can be multiple threads per process, we need a separate
   PCB from the TCB. All TCBs in a process will have a pointer
   to the PCB, and the PCB will have a pointer to the main thread
   of the process, which is `special`. */
struct process {
  /* Owned by process.c. */
  uint32_t* pagedir;          /* Page directory. */
  char process_name[16];      /* Name of the main thread */
  struct thread* main_thread; /* Pointer to main thread */

  pid_t pid;
  bool is_child_loaded;             /*子进程是否加载可执行表成功*/
  struct semaphore from_child;      /*调用exec时使用的信号量*/
};
```

```
struct communcate{
    char*fn_copy;
    struct thread*father;
};
```

communcate 结构体用于将参数传给子进程，让子进程知道父进程是谁，同时传一份命令行字符串的拷贝）

接着正式修改函数：

先是 process_execute 函数（此处按照用户进程创建的逻辑讲述修改过程，实际上的修改过程是很复杂的，大部分情况是发现测试过不了就修改部分代码，后面又发现这个修改方式不能适配之后的要求，又不得不推翻重做，当时修改的时候蛮伤脑筋的）：

```c
/* Starts a new thread running a user program loaded from
   FILENAME.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   process id, or TID_ERROR if the thread cannot be created. */
pid_t process_execute(const char* file_name) {
  char* fn_copy;
  tid_t tid;
  thread_current()->pcb->is_child_loaded=false;

  sema_init(&thread_current()->pcb->from_child, 0);

  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page(0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy(fn_copy, file_name, PGSIZE);

  /*处理进程名字问题*/
  char*fn_copy_name = palloc_get_page(0);
  if (fn_copy_name == NULL)
    return TID_ERROR;

  /*防止进程名字后面的参数改变进程名字*/
  int i=0;
  while(file_name[i++]!=' ');
  strlcpy(fn_copy_name, file_name, i);

  /*分配变量以传进子进程*/
  struct communcate*commu=malloc(sizeof(struct communcate));
  commu->father=thread_current();
  commu->fn_copy=fn_copy;

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create(fn_copy_name, PRI_DEFAULT, start_process, commu);
  if (tid == TID_ERROR)
  {
    palloc_free_page(fn_copy);
    palloc_free_page(fn_copy_name);
    /*释放传递的变量*/
    free(commu);
    return tid;
  }

  /*释放得到名字的变量*/
  palloc_free_page(fn_copy_name);

  /*等待子进程加载完成*/
  sema_down(&thread_current()->pcb->from_child);

  /*释放传递的变量*/
  free(commu);

  /*执行表加载失败*/
  if(!thread_current()->pcb->is_child_loaded)
  {
    return TID_ERROR;
  }

  return tid;
}
```

用于判断子进程执行表是否加载成功，初始化为失败

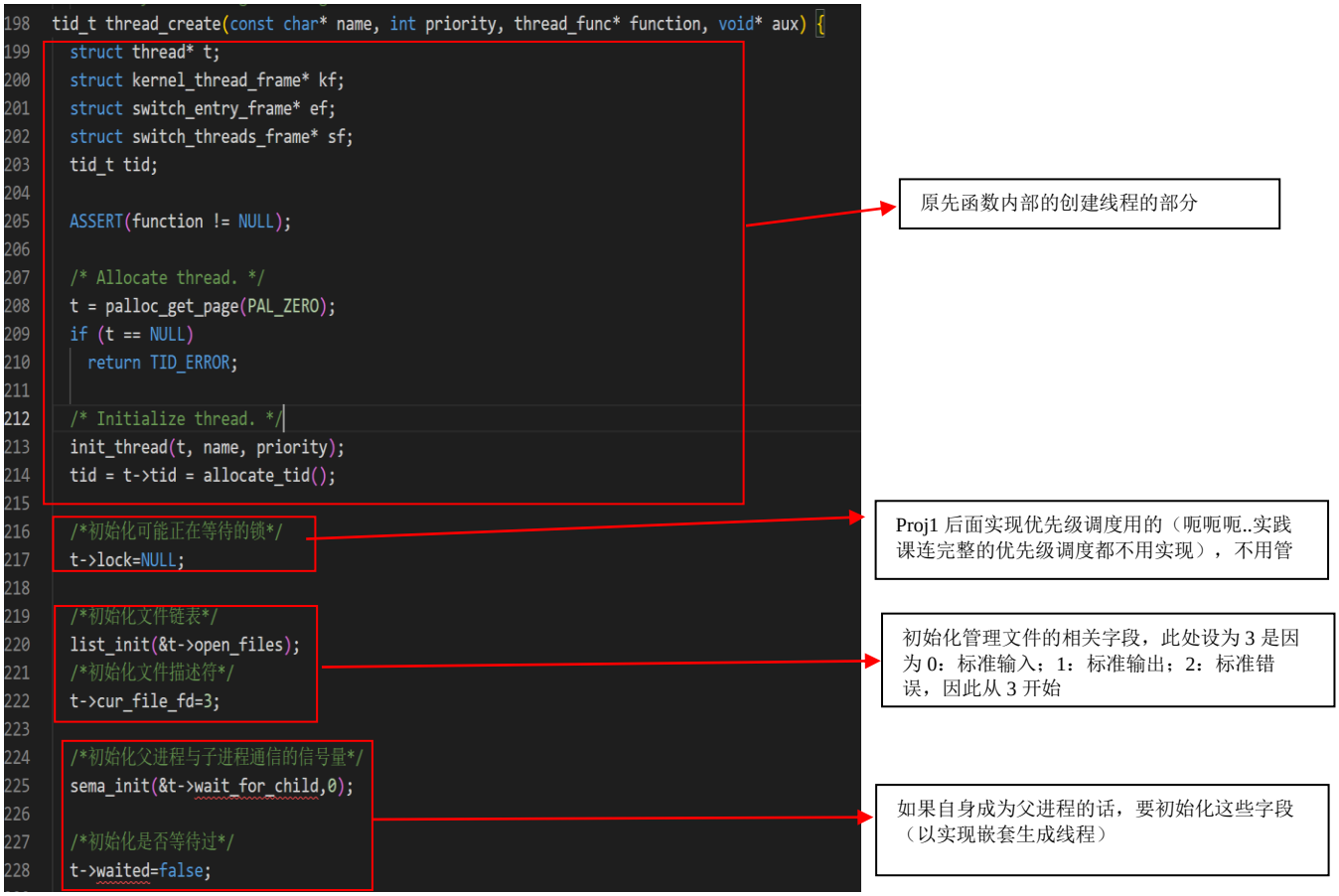
初始化父子进程的信号量，子进程将用这个通知父进程自己的初始化结束

此处是为了拿到该进程的名字，fn_copy_name 就是进程名，实际上就是测试的名字

通过 communcate 结构体将命令行参数传给子进程

资源释放和错误处理，同时父进程也要等待子进程可执行表加载完成（无论是否成功）

接着是 thread_create 函数：


tid_t thread_create(const char* name, int priority, thread_func* function, void* aux) {
struct thread* t;
struct kernel_thread_frame* kf;
struct switch_entry_frame* ef;
struct switch_threads_frame* sf;
tid_t tid;
ASSERT(function != NULL);
/* Allocate thread. */
t = palloc_get_page(PAL_ZERO);
if (t == NULL)
return TID_ERROR;
/* Initialize thread. */
init_thread(t, name, priority);
tid = t->tid = allocate_tid();
/*初始化可能正在等待的锁*/
t->lock=NULL;
/*初始化文件链表*/
list_init(&t->open_files);
/*初始化文件描述符*/
t->cur_file_fd=3;
/*初始化父进程与子进程通信的信号量*/
sema_init(&t->wait_for_child,0);
/*初始化是否等待过*/
t->waited=false;
原先函数内部的创建线程的部分
Proj1 后面实现优先级调度用的（呃呃呃..实践课连完整的优先级调度都不用实现），不用管
初始化管理文件的相关字段，此处设为 3 是因为 0：标准输入；1：标准输出；2：标准错误，因此从 3 开始
如果自身成为父进程的话，要初始化这些字段（以实现嵌套生成线程）

```
/* Stack frame for kernel_thread(). */
kf = alloc_frame(t, sizeof *kf);
kf->eip = NULL;
kf->function = function;
kf->aux = aux;

/* Stack frame for switch_entry(). */
ef = alloc_frame(t, sizeof *ef);
ef->eip = (void (*)(void))kernel_thread;

/* Stack frame for switch_threads(). */
sf = alloc_frame(t, sizeof *sf);
sf->eip = switch_entry;
sf->ebp = 0;
```

原先函数构建栈帧的部分

```
/*初始化fpu*/
fpu_init(t);

/* Add to run queue.*/
thread_unblock(t);

/* 若是优先级调度且优先级较高则释放cpu*/
if(active_sched_policy==SCHED_PRIO&&t->priority>thread_current()->priority)
thread_yield();

 return tid;
}
```

fpu 部分不用管，是为了过 cs162 特有的测试其他的更改为 proj1 的更改，也略过

接着看 start_process 函数（由于函数较长，与原函数相同的部分不再注解）：

```
124  static void start_process(void* argvs_) {
125    struct communcate*argss=(struct communcate*)argvs_;
126    char* argvs = argss->fn_copy;
127    struct thread* t = thread_current();
128    int total_len=strlen(argvs);
129    struct intr_frame if_;
130    bool success, pcb_success;
131
132    /* Allocate process control block */
133    struct process* new_pcb = malloc(sizeof(struct process));
134    success = pcb_success = new_pcb != NULL;
135
136    /*计算命令行参数个数并将命令行存入argv中*/
137    int argc=0;
138    char *argv[MAXARGV];
139    int args;
140    if(success){
141    char *saveptr;
142    char *token=strtok_r(argvs," ",&saveptr);
143    int index=0;
144    while(token!=NULL&&index<MAXARGV)
145    {
146
147      argv[index++]=token;
148      token=strtok_r(NULL," ",&saveptr);
149      argc++;
150    }
151    args=index;
152    }
153
```

从 argvs_得到 communcate，并通过 thread_current 函数得到当前线程，以及参数传递的重点 intr_frame（原本是线程切换，保存上下文用，这里用来传参和各种寄存器的初始化））

这里使用 c 库里的 strtok_r 函数做命令行参数分割，分割的结果保存在字符串数组 argv 中

```c
/*初始化子进程列表*/
struct list* cp=malloc(sizeof(struct list));
list_init(cp);
t->child_process=cp;

/*设置父子关系*/
t->father=argss->father;
list_push_back(argss->father->child_process,&t->elem_process);
```

初始化自身的子进程列表，方便 wait 系统调用实现，同时设置父子关系，并将自己加入到父进程的列表当中

```c
/* Initialize interrupt frame and load executable. */
if (success) {
  memset(&if_, 0, sizeof if_);
  if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
  if_.cs = SEL_UCSEG;
  if_.eflags = FLAG_IF | FLAG_MBS;
  success = load(argvs, &if_.eip, &if_.esp);

  /*加载成功*/
  if(success)
  {
    thread_current()->father->pcb->is_child_loaded=true;
  }
}
```

如果可执行表加载成功，将父进程内的判断子进程是否加载成功的 bool 变量设为 true，此处并没有将这个 bool 变量设为全局变量是因为使用全局变量会出现需要同步的情况，而且如果存在多个父子关系同时创建线程，全局变量要记住每一次的上下文，实现较为困难（实际上我第一次就是这样实现的，后面发现这样实现的话问题太多，就换成了目前这种，只是说 tcb 会大一些）

```c
uint32_t *addr_argv=malloc(sizeof(uint32_t)*MAXARGV);
if(!addr_argv)success=false;
if(success){

  /* 将命令行的所有参数推入栈*/
for(int i=0;i<args;i++)
{
  if_.esp=push_str_into_stack(argv[i],if_.esp);
  addr_argv[args-i-1]=if_.esp;
}
```

此处再分配一个 addr_argv 是为了能够记住推入栈的各个字符串的位置，这样在第二次入栈时（也就是要 16 字节对齐的那个部分）就可以使用这个 addr_argv 入栈

```c
  if(success){
  /*设置对齐*/
   uint8_t stack_align=16-(unsigned)(argc*4+12+total_len+1)%16;
   if_.esp-=stack_align;

  /*存入argv[argc]，保持栈结构*/
  if_.esp-=4;

  /*存入命令行参数的指针*/
  for(int i=0;i<args;i++)
  {
    if_.esp-=4;
    memcpy(if_.esp,&addr_argv[i],4);
  }

  /*存入命令行字符串数组的头位置*/
  uint32_t tmp=if_.esp;
  if_.esp-=4;
  memcpy(if_.esp,&tmp,4);

  /*存入argc*/
  if_.esp-=4;
  memcpy(if_.esp,&argc,4);

  /*存入fake_return_address*/
  if_.esp-=4;

  }

  /* Clean up. Exit on failure or jump to userspace */
  palloc_free_page(argvs);
```

这些部分都是根据第二部分的分析写成，注释已写得相当详细，值得一提的是 if_.esp 这个寄存器，他就是程序正式运行后得到命令行参数的寄存器

```
/* Handle failure with succesful PCB malloc. Must free the PCB */
if (!success && pcb_success) {
  // Avoid race where PCB is freed before t->pcb is set to NULL
  // If this happens, then an unfortuantely timed timer interrupt
  // can try to activate the pagedir, but it is now freed memory
  struct process* pcb_to_free = t->pcb;
  t->pcb = NULL;
  free(pcb_to_free);
}

/*释放存地址的空间和字符串*/
if(!addr_argv)
free(addr_argv);

if (!success) {
  sema_up(&t->father->pcb->from_child);
  thread_exit();
}

/*通知父进程*/
sema_up(&t->father->pcb->from_child);

  /* Start the user process byq simulating a return from an
   interrupt, implemented by intr_exit (in
   threads/intr-stubs.S).  Because intr_exit takes all of its
   arguments on the stack in the form of a `struct intr_frame',
   we just point the stack pointer (%esp) to our stack frame
   and jump to it. */
  asm volatile("movl %0,%%esp;jmp intr_exit" : : "g"(&if_) : "memory");

  NOT_REACHED();
}
```

释放函数内分配的资源

无论可执行表是否加载成功，告诉父进程自己搞完了，这样父进程就可以继续执行

2.系统调用

在实现系统调用之前，我们先要实现用户所传参数的合法性检查（合法性检查包括用户传递参数的位置（也就是 f->esp 的值的检查），以及用户传递参数自身的检查（如果用户传的是一个字符串，要逐字节检查字符串指针的合法性）），为了方便所有系统调用进行检查，我们将这部分功能封装成一个函数 check_ptr：

```
bool check_ptr(uint32_t* unchecked)
{
  for(int i=0;i<4;i++)
  {
    if(!is_user_vaddr(unchecked+i)||!pagedir_get_page(thread_current()->pcb->pagedir,unchecked+i))
    return false;
  }
  return true;
}
```

该函数检查传入的指针是否合法，指针本身是否有分配页面，这里 i<4 是因为 32 位机器内指针的长度等于 4 字节

而对于用户给的字符串指针，我们另外写一个函数查看字符串指针指向的位置是否合法：

```
bool check_string(const char*being_checked)
{
  if(being_checked==NULL||being_checked>PHYS_BASE)return false;
  int i=0;
  if(!is_user_vaddr(&being_checked[i])||!pagedir_get_page(thread_current()->pcb->pagedir,&being_checked[i]))
  return false;
  for(;being_checked[i]!='\0';i++)
  {
    if(!is_user_vaddr(&being_checked[i+1])||!pagedir_get_page(thread_current()->pcb->pagedir,&being_checked[i+1]))
    return false;
  }
  return true;
}
```

同样是逐字节检查其合法性，直到结束符'\0'

接着，由于各个系统调用要对不合法的指针进行处理，一般是终止进程，而终止进程需要提示（用例会对打印的提示信息进行检查），我们也将这部分封装成一个函数 sys_exit：

```
void sys_exit(int exit_status)
{
  printf("%s: exit(%d)\n", thread_current()->pcb->process_name, exit_status);
  thread_current()->exit_status=exit_status;
  process_exit();
}
```

需要注意的是，cs162 与学校的退出方式略有不同，cs162 为了更好地模拟进程，编写了 process_exit 函数，该函数内部调用了 thread_exit，本质上是一样的。

接着，我们在系统调用主函数的前面加上检查参数和系统调用号的部分：

```
if(!check_ptr((uint32_t*)f->esp))
{
  sys_exit(-1);
}

uint32_t*args = ((uint32_t*)f->esp);

if(args[0]<=0||args[0]>SYS_INUMBER)
{
  printf('不合法系统调用，硬件问题？');
  sys_exit(-1);
}
```

只要 f->esp 通过以上测试，就证明该系统调用目前是合法的。

接下来正式处理系统调用：

Exit 系统调用：

```
if (args[0] == SYS_EXIT) {
  if(!check_ptr(&args[1]))
  f->eax=-1;
  else
  f->eax = args[1];
  sys_exit(f->eax);
}
```

由于 exit_status 是一个 int 类型，只要检查存储位置的指针就行，检查不通过，退出代码变为 -1，检查通过，则为用户传入的退出代码

Halt 系统调用：

```
if(args[0]==SYS_HALT)
{
   shutdown_power_off();
}
```

最简单的系统调用，根据文档调用该函数即可

Exec 系统调用：

```
if(args[0]==SYS_EXEC)
{
  if(!check_ptr(&args[1])||!check_string(args[1]))
  {
    sys_exit(-1);
  }
  f->eax=process_execute(args[1]);
}
```

检查完用户的参数后调用 process_execute 函数即可（该函数的实现已在传参中说明）

Wait 系统调用：

```
if(args[0]==SYS_WAIT)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
  }
  int ch_pid=args[1];
  f->eax=process_wait(ch_pid);
}
```

主要是 process_wait 函数，该函数等待特定的子进程完成，并通过信号量告诉父进程自身结束

Create 系统调用：

```
if(args[0]==SYS_CREATE)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2])||!check_string(args[1]))
  {
    sys_exit(-1);
    return;
  }
  char*file=args[1];
  int initial_size=args[2];
  lock_acquire(&f_lock);
  f->eax=filesys_create(file,initial_size);
  lock_release(&f_lock);
}
```

得到用户参数后直接调用 filesys_create 函数（内置的线程不安全的文件创建函数，所以有一个锁用于同步，该锁不再这里讨论）即可，比较简单

Remove 系统调用：

```
if(args[0]==SYS_REMOVE)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  char*file=args[1];
  if(!check_string(file))
  {
    f->eax= false;
    return;
  }
  lock_acquire(&f_lock);
  bool success=filesys_remove(file);
  lock_release(&f_lock);
  f->eax=success;
}
```

该系统调用与 create 一样，参数合法并加锁后调用相应函数即可

Open 系统调用：

```
if(args[0]==SYS_OPEN)
{
  if(!check_ptr(&args[1])||!check_string(args[1]))
  {
    sys_exit(-1);
    return;
  }
  char*file=args[1];
  struct thread*cur=thread_current();

  struct thread_file* tmp=malloc(sizeof(struct thread_file));
  tmp->fd=cur->cur_file_fd++;
  strlcpy(tmp->name,file,sizeof(tmp->name));
  lock_acquire(&f_lock);
  tmp->f=filesys_open(file);
  lock_release(&f_lock);
  if(tmp->f==NULL)
  {
    f->eax=-1;
    free(tmp);//必须释放资源
    return;
  }
  list_push_back(&cur->open_files,&tmp->elem_tf);
  f->eax=tmp->fd;
}
```

这就与先前 tcb 块的修改有关系了，如果没有先前的修改，我们没有办法得知打开了哪些文件，而有了打开的文件列表，就可以将新打开的文件加入列表（list_push_back 语句）

需要注意的是文件结构体是位于堆上的，如果打开失败要释放内存。

Close 系统调用：

```
if(args[0]==SYS_CLOSE)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    return;
  }
  lock_acquire(&f_lock);
  file_close(tf->f);
  lock_release(&f_lock);
  list_remove(&tf->elem_tf);
  free(tf);
}
```

关闭文件，需要注意的点是要在关闭文件的同时将文件从列表中移除（list_remove 语句），同时释放文件结构体（free 语句）

Find_file 函数实现如下（是为找到句柄对应的文件结构体）：

```
struct thread_file*find_file(int fd)
{
  struct thread*cur=thread_current();
  struct list_elem*e;
  for(e=list_begin(&cur->open_files);e!=list_end(&cur->open_files);e=list_next(e))
  {
    struct thread_file*tmp=list_entry(e,struct thread_file,elem_tf);
    if(tmp->fd==fd)
    return tmp;
  }
  return NULL;
}
```

Filesize 系统调用：

```
if(args[0]==SYS_FILESIZE)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    f->eax=-1;
    return;
  }
  lock_acquire(&f_lock);
  f->eax=file_length(tf->f);
  lock_release(&f_lock);
}
```

返回文件长度，找到文件并调用 file_length 函数即可

Read 系统调用：

```
if(args[0]==SYS_READ)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2])||!check_ptr(&args[3]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  char*buffer=args[2];
  int size=args[3];

  if(!check_string(buffer))
  {
    sys_exit(-1);
    return;
  }

  if(fd==STDIN_FILENO)
  {
    int total=0;
    for(int i=0;i<size;i++)
    {
      buffer[i]=input_getc();
      total++;
    }
    f->eax=total;
    return;
  }
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    f->eax=-1;
    return;
  }
  lock_acquire(&f_lock);
  f->eax=file_read(tf->f,buffer,size);
  lock_release(&f_lock);
}
```

这里要处理输入是标准输入的情况（STDIN_FILENO 部分），对于标准输入，使用 input_getc 函数，对于文件输入，使用 file_read 函数。

Write 系统调用：

```
if(args[0]==SYS_WRITE)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2])||!check_ptr(&args[3]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  char*buffer=args[2];
  int size=args[3];

  if(!check_string(buffer))
  {
    sys_exit(-1);
    return;
  }
  if(fd==STDOUT_FILENO)
  {
    putbuf(buffer,size);
    f->eax=size;
    return;
  }
  struct thread_file*tf=find_file(fd);
  if(!tf)
  {
    f->eax=-1;
    return;
  }
  if(is_executing(tf->name))
  {
    f->eax=0;
    return;
  }
  lock_acquire(&f_lock);
  f->eax=file_write(tf->f,buffer,size);
  lock_release(&f_lock);
}
```

该系统调用要处理标准输出（STDOUT_FILENO 部分），同时由于如果文件正在被执行，是不能进行写操作的，因此在函数的后部有一个 is_executing 函数判断该文件是否正在被执行，该函数实现如下：

```
/*判断一个线程是否正在被执行*/
bool is_executing(const char*name)
{
  struct list_elem*e;
  for(e=list_begin(&all_list);e!=list_end(&all_list);e=list_next(e))
  {
    struct thread*tmp=list_entry(e,struct thread,allelem);
    if(strcmp(tmp->name,name)==0)return true;
  }
  return false;
}
```

（实际上就是看 all_list（该列表包含所有正在运行的线程，pintos 自带）里面有没有这个文件的名字）

Tell 系统调用：

```
if(args[0]==SYS_TELL)
{
  if(!check_ptr(&args[1]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  struct thread_file*tf=find_file(fd);
  if(tf!=NULL)
  {
    lock_acquire(&f_lock);
    f->eax=file_tell(tf->f);
    lock_release(&f_lock);
  }
}
```

返回文件指针的位置，检查参数无误后调用 file_tell 函数即可

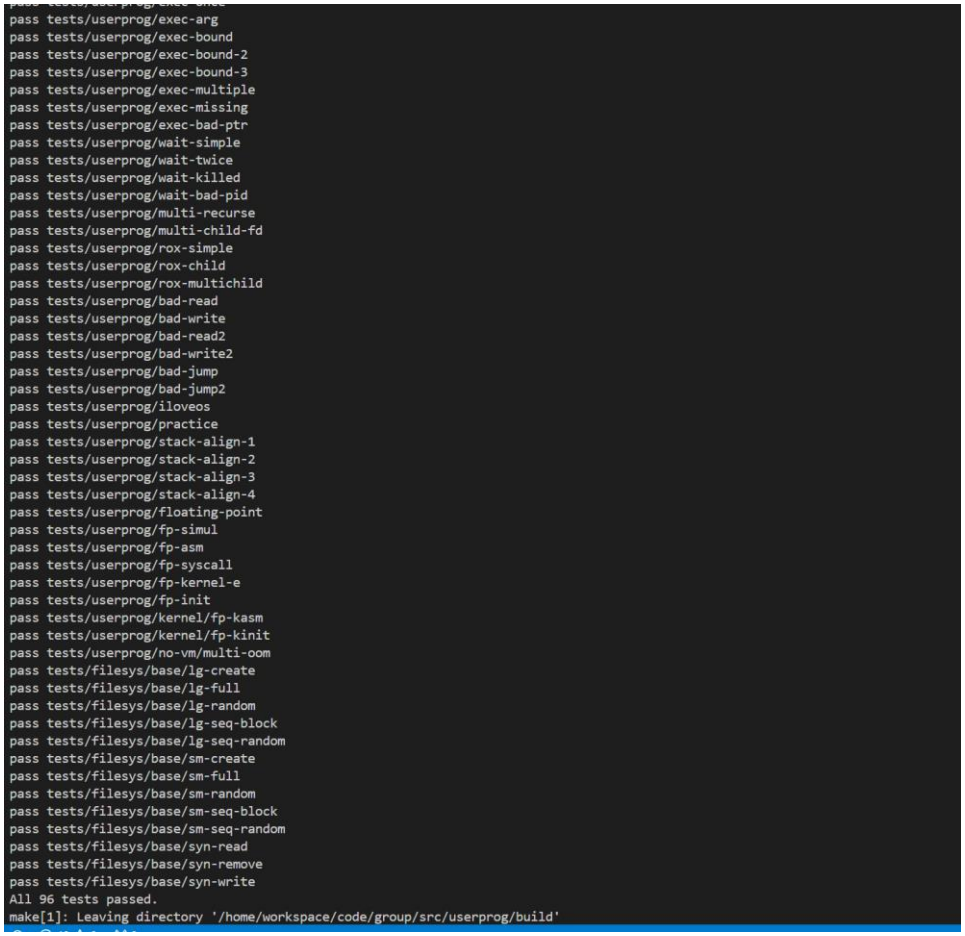
Seek 系统调用：

```
if(args[0]==SYS_SEEK)
{
  if(!check_ptr(&args[1])||!check_ptr(&args[2]))
  {
    sys_exit(-1);
    return;
  }
  int fd=args[1];
  unsigned pos=args[2];
  struct thread_file*tf=find_file(fd);
  if(tf!=NULL)
  {
    lock_acquire(&f_lock);
    file_seek(tf->f,pos);
    lock_release(&f_lock);
  }
}
```

改变文件指针的位置，同样检查参数无误后调用 file_seek 函数即可

3.通过截图：代码库：https://github.com/saydontgo/cs162-group

```
workspace@69d2794469c5 [07:02:40] userprog $ make check
cd build && make check
make[1]: Entering directory '/home/workspace/code/group/src/userprog/build'
pass tests/userprog/do-nothing
pass tests/userprog/stack-align-0
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
```



上面展示的修改并不能通过所有的测试，还有一些其他地方的修改（process_wait 函数以及 thread.c 里面的一些修改）才能通过所有 96 个测试，因为篇幅已经过长且关系不大（下次实验报告将会包含），此处不再展示。

五、总结

这个 lab 特别有含金量，做的过程是极其痛苦的（有时候一个测试做一天都过不了，但是觉得自己是对的），但是当测试都通过的那一瞬间又有发自内心的畅快，这么说呢，独立做完这个 lab 之后的那一周感觉自己可以手撕操作系统，很多具体的实现等到自己上手才知道理论很简单，实现很困难，要考虑的情况也相当多。总体来讲，做完这个收获还是很多的，最主要的就是加深了对操作系统的理解，而且是非常深刻的理解，能将每个调度细节和系统调用给你讲明白的那种理解。