华东师范大学软件学院实验报告

课程名称：操作系统	年级：2023 级本科	上机实践成绩：
指导教师：张民	姓名：张梓卫	
上机实践名称：Pintos Userprog Part 1	学号：10235101526	上机实践日期：2024/12/16
上机实践编号：（5）	组号：	上机实践时间：2 学时

目录

一 实验目的

本实验的目标是完成参数传递和部分系统调用（exit 和 write），使得 make check 通过 args 相关的 5 个测试。

本项目的主要任务：实现用户程序和 OS 之间的系统调用。本次实验作出修改的代码同时上传到了 Github 之上，仓库地址为：https://github.com/Shichien/ECNU-23-SEI-Homework

请在上传的 PDF 文件中直接点击粉色链接即可。

二 内容与设计思想

- 参数传递，系统调用

> **Notice**
>
> - 编译生成 build 文件夹
> - 创建用户磁盘
> - 修改 process_execute() 实现文件名与参数分离
> - 修改 load() 实现文件名与参数分离并将可执行文件存到 ESP 中
> - 修改 setup_stack() 实现参数堆栈
> - 注：上述方法均位于 userprog/process.c 中

该开始着手研究允许运行用户程序的系统部分了。基本代码已经支持加载和运行用户程序，但是无法进行 I / O 或交互。在此项目中，将使程序能够通过系统调用与 OS 进行交互，并为用户进程提供系统调用。在 Project 1 中，执行的操作都是在内核模式下运行的，而在这个 Project 中要求我们对非内核进行修改。

- src/userprog
- process.c/ process.h：进程代码（需修改）
- pagedir.c/pagedir.h：物理内存管理（不需修改，可以调用实现的）
- syscall.c/syscall.h：系统调用，目前只搭了骨架（需修改）
- exception.c/exception.h：异常处理情况（需修改）
- gdt.c/gdt.h：全局描述表（不需修改）
- tss.c/tss.h：任务状态段（不需修改）

三 使用环境

使用 Docker v27.1.1 进行 Pintos 的安装实验，基于 Windows 11 操作系统使用 WSL2。
实验报告使用 LaTeX 进行撰写，使用 VSCode + Vim 编辑器进行文本编辑。

四 实验过程与分析

1 进入实验背景

进入 `cd src/pintos/userprog` 中使用 `make` 命令编译
然后使用 `cd build` 进入到 build 文件夹中建立一个新的用户磁盘。

```
root@e60a96920998:~/pintos/src/userprog/build# pintos-mkdisk filesys.dsk --filesys-size=2
root@e60a96920998:~/pintos/src/userprog/build# pintos -f -q
Unknown option: f
Unknown option: q
root@e60a96920998:~/pintos/src/userprog/build# pintos -- -f -q
qemu-system-i386 -device isa-debug-exit -drive format=raw,media=disk,index=0,file=/tmp/YWZgP
media=disk,index=1,file=filesys.dsk -m 4 -net none -nographic -monitor null
Pintos hda1
Loading..........
Kernel command line: -f -q
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  157,081,600 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 189 sectors (94 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Formatting file system...done.
Boot complete.
Timer: 52 ticks
Thread: 0 idle ticks, 53 kernel ticks, 0 user ticks
hdb1 (filesys): 3 reads, 6 writes
Console: 581 characters output
```

图 1: 编译

注意到这里直接使用 `pintos -f -q` 会提示未知的参数，要修改为：`pintos -- -f -q`，才能编译成功。
接下来，执行下一条命令，将 echo.c 复制到磁盘中，且命名为 echo。

```
root@e60a96920998:~/pintos/src/userprog/build# pintos -p ../../examples/echo.c -a echo -- -q
Copying ../../examples/echo.c to scratch partition...
qemu-system-i386 -device isa-debug-exit -drive format=raw,media=disk,index=0,file=/tmp/8CQHhn1edQ.dsk -drive format=raw,
media=disk,index=1,file=filesys.dsk -m 4 -net none -nographic -monitor null
Pintos hda1
Loading..........
Kernel command line: -q extract
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  157,081,600 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 189 sectors (94 kB), Pintos OS kernel (20)
hda2: 4 sectors (2 kB), Pintos scratch (22)
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
scratch: using hda2
Boot complete.
```

图 2: 复制文件

在未实现参数传递时，我们直接执行 `pintos -- -q run 'echo x'` 这样的命令其实是很不合理的。这个输入的含义是：执行 echo 文件，传递的参数是 x。实际上，echo x 被当成了一个整体来处理。

```
root@e60a96920998:~/pintos/src/userprog/build# pintos -- -q run 'echo x'
qemu-system-i386 -device isa-debug-exit -drive format=raw,media=disk,index=0,
media=disk,index=1,file=filesys.dsk -m 4 -net none -nographic -monitor null
Pintos hda1
Loading..........
Kernel command line: -q run 'echo x'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  209,510,400 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 189 sectors (94 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo x':
Execution of 'echo x' complete.
Timer: 51 ticks
Thread: 0 idle ticks, 51 kernel ticks, 0 user ticks
hdb1 (filesys): 2 reads, 0 writes
Console: 612 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

图 3: Echo

2 开始分析代码

先查看官方文档中对目前清空的介绍：

3.3.3 Argument Passing

Currently, `process_execute()` does not support passing arguments to new processes. Implement this functionality, by extending `process_execute()` so that instead of simply taking a program file name as its argument, it divides it into words at spaces. The first word is the program name, the second word is the first argument, and so on. That is, `process_execute("grep foo bar")` should run `grep` passing two arguments `foo` and `bar`.

Within a command line, multiple spaces are equivalent to a single space, so that `process_execute("grep foo  bar")` is equivalent to our original example. You can

图 4: Pintos References

翻译如下：

目前，process_execute() 不支持向新进程传递参数。通过扩展 process_execute() 来实现此功能，而不是简单的 task。将程序 filename 作为其参数，它在空格处将其拆分为单词。第一个词是程序名，第二个单词是第一个参数，以此类推。也就是.process execute（“grep-foo-bar”）应该运行 grep，传递两个参数 foo 和 bar。在命令行中多个空格相当于一个空格，以便进程执行（“grep-foo-bar”）与我们最初的示例等效。

查看相关的代码处，我们可以看到以下内容：

```
tid_t process_execute (const char *file_name) {
  char *fn_copy;
  tid_t tid;

  /* 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);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

process

Notice

在传递参数时，传递 `process_execute` 函数的参数 `file_name` 既包括可执行文件的名称，也包含了可执行文件的参数。所以，要做的第一件事就是换把可执行文件名称和参数互相分开。我们可以使用 `string.h` 中的 `strtok_r()` 来分离参数。

```
tid_t process_execute (const char *file_name) {
  char *fn_copy;
  char *thread_name;
  char *save_ptr; // 用来保存strtok_r的状态

  thread_name = malloc(strlen(file_name) + 1);
  strlcpy(thread_name, file_name, strlen(file_name) + 1);
  // 拷贝 file_name 到动态分配的 thread_name，以便后续通过 strtok_r 截取线程名称
  thread_name = strtok_r(thread_name, " ", &save_ptr); // 创建一个线程用来运行"文件名"
  tid_t tid;

  /* 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);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  free(thread_name); // 释放由 malloc 创建的线程名
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

process

> **Notice**
>
> 这里我们首先在最开始添加两个 char * 类型变量，*save_ptr 指向 file_name 的一个拷贝，是为了实现保存从命令语句中分离出来的真正的可执行文件名，给 fn 分配与命令语句容量相等的内存空间。

2 .1 分析 start_process()

我们在函数中将 file_name 通过 load 存储到栈中。
下面这个是 Pintos 中的原始代码。

```
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;

  /* Initialize interrupt frame and load executable. */
  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 (file_name, &if_.eip, &if_.esp);

  /* If load failed, quit. */
  palloc_free_page (file_name);
  if (!success)
    thread_exit ();

  /* Start the user process by 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 ();
}
```

start_process

修改后的代码如下所示：

```
char *token, *save_ptr;
file_name = strtok_r(file_name, " ", &save_ptr); // 获取文件名
success = load (file_name, &if_.eip, &if_.esp);
```

process

2 .2 分析 load() 函数

主要是将从 FILE_NAME 获取的可执行的程序放入到当前线程中。
针对传参，我们需要第二次参数分离：传递过来的是file_name="args-single onearg"。

```
/* Separate file name and arguments. */
char *exec_name;
char *save_ptr;

/* Make a modifiable copy of file_name. */
char modifiable_file_name[PGSIZE];
strlcpy(modifiable_file_name, file_name, PGSIZE);

/* Extract the executable name (first token). */
```

```
  exec_name = strtok_r(modifiable_file_name, " ", &save_ptr);

  /* Open the executable file. */
  file = filesys_open(exec_name);
  if (file == NULL)
  {
    printf ("load: %s: open failed\n", exec_name);
    goto done;
  }
```

load

另外，在 load() 函数中，注释中写明了 `setup_stack()` 函数的作用是设置参数堆栈。

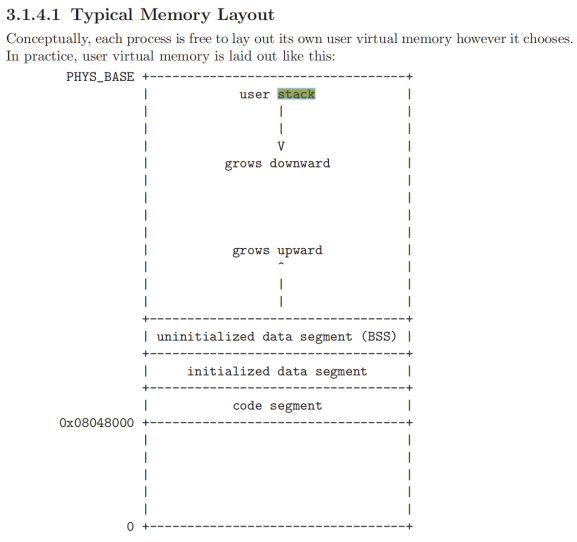
```
/* Set up stack. */
if (!setup_stack (esp))
  goto done;
```

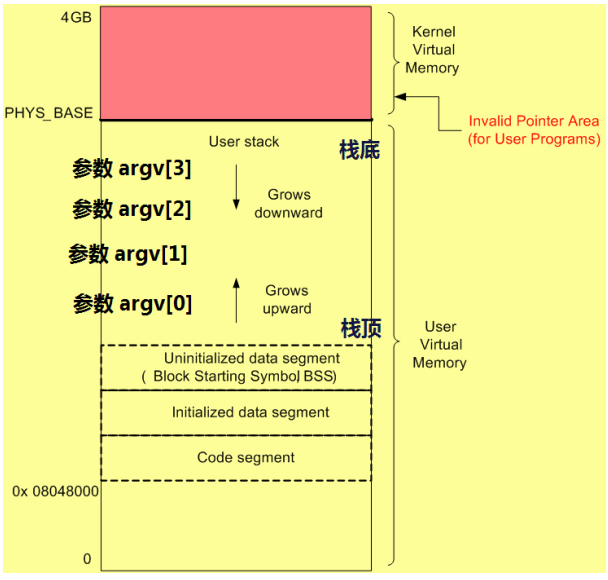
图 5: load()

而在 `setup_stack()` 函数中，它仅仅实现了页的初始化，并没有参数传递实现的代码。

因此我们需要将用 strok 分离出的 filename 以及 argv 传递进去。

一个实现思路是，在 `setup_stack()` 函数中添加参数 `file_name` 和 `save_ptr`





Notice

if_.esp 原来指向 esp 的内核和用户区边界为避免内存访问越界（访问到内核区）
所以需将`setup_stack()`函数中的`esp = PHYS_BASE` 修改成 `*esp = PHYS_BASE - 12;`
后来，文档中还附加了一条对于 **bin/ls -l foo bar** 命令执行时栈的增长情况：

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	word-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 (*) ()

In this example, the stack pointer would be initialized to 0xbffffffcc.

As shown above, your code should start the stack at the very top of the user virtual address space, in the page just below virtual address PHYS_BASE (defined in 'threads/vaddr.h').

图 7: 示例栈增长

根据 PPT 上的提示，我们编写以下代码来实现初始化栈：
另外，官方文档中还提到了我们可以使用 **hex_dump()** 函数来打印此时栈的信息，不妨添加进去。

As shown above, your code should start the stack *at the very top of the user virtual address space*, in the page just below virtual address `PHYS_BASE` (defined in `threads/vaddr.h`).

如上所示，你的代码应该在虚拟地址 `PHYS_BASE` （定义在 `threads/vaddr.h` ）下面的页面启动堆栈。

You may find the non-standard `hex_dump()` function, declared in `<stdio.h>`, useful for **debugging your argument passing code**. Here's what it would show in the above example:

你可能会发现非标准的 `hex_dump()` 函数，声明在 `<stdio.h>` 中，它对调试传递参数的代码很有用。这是它在上面的例子中显示的内容：

```
bffffc0                                00 00 00 00 |            ....|
bffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |................|
bffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 2f 62 69 |............/bi|
bfffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|
```

图 8: hex_dump()

在函数中，写好了为什么要用这样的方式实现的具体注释：

```
static bool
setup_stack (void **esp, char* file_name)
{
  uint8_t *kpage;
  bool success = false;

  // 分配一页用户内存并将其安装到用户栈的顶部
  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
        *esp = PHYS_BASE - 12;  // 初始化栈指针到栈顶
      else
        palloc_free_page (kpage);
    }

  // 创建一个临时拷贝的文件名字符串
  char *token, *temp_ptr;
  char *filename_cp = malloc(strlen(file_name) + 1);
  strlcpy (filename_cp, file_name, strlen(file_name) + 1);

  // 计算参数数量 (argc)
  enum intr_level old_level = intr_disable();
  int argc = 1; // 初始值为 1，因为至少有一个参数
```

```
    bool is_lastone_space = false; // 记录最后一个字符是否为空格
    for (int j = 0; j != strlen(file_name); j++) {
      if (file_name[j] == ' ') {
        if (!is_lastone_space)
          argc++; // 如果遇到新的空格且不是连续空格，参数计数加一
        is_lastone_space = true;
      } else {
        is_lastone_space = false;
      }
    }
    intr_set_level (old_level);

    // 为参数指针 (argv) 分配内存
    int *argv = calloc(argc, sizeof(int));

    // 将文件名分割为各个参数，并依次压入栈中
    int i;
    token = strtok_r (file_name, " ", &temp_ptr);
    for (i = 0; ; i++) {
      if (token) {
        *esp -= strlen(token) + 1; // 为参数字符串分配空间
        memcpy(*esp, token, strlen(token) + 1); // 将参数字符串复制到栈
        argv[i] = *esp; // 保存参数地址
        token = strtok_r (NULL, " ", &temp_ptr); // 获取下一个参数
      } else {
        break; // 没有更多参数时退出循环
      }
    }

    // 对齐到 4 字节边界 (Word Align)
    *esp -= ((unsigned)*esp % 4);

    // 空指针标志 (null pointer sentinel)，表示 argv[argc] 为空
    *esp -= sizeof(int);

    // 将参数地址依次压入栈中
    for (i = argc - 1; i >= 0; i--)
    {
      *esp -= sizeof(int);
      memcpy(*esp, &argv[i], sizeof(int));
    }

    // 压入 argv 的地址
    int tmp = *esp;
    *esp -= sizeof(int);
    memcpy(*esp, &tmp, sizeof(int));

    // 压入 argc (参数数量)
    *esp -= sizeof(int);
    memcpy(*esp, &argc, sizeof(int));

    // 压入返回地址 (return address)
    *esp -= sizeof(int);
    memcpy(*esp, &argv[argc], sizeof(int));

    // 打印栈的状态，便于调试
    printf("STACK SET. ESP: %p\n", esp);
    hex_dump((uintptr_t)esp, esp, 100, true); // 打印栈中前 100 字节的数据，便于调试

    // 释放临时分配的内存
    free(filename_cp);
    free(argv);

    return success;
```

```
    }
```

setup_stack

> **Notice**
>
> 在官方文档中提到，栈的增长是反方向的：栈地址是向下增长的，而字符串的地址是向上增长的，所以在放置参数的时候需要把参数倒序放置。

直到这里，即使我们使用 **make check** 命令，仍然无法看到输出，只会出现：

```
Run didn’t produce any output”。
```

因为用户程序的打印（输出到 stdout）也是通过写的系统调用实现的。

- The `write` system call for writing to fd 1, the system console. All of our test programs write to the console (the user process version of `printf()` is implemented this way), so they will all malfunction until `write` is available.

  `write` 系统调用，用于写入fd1，即系统控制台。我们所有的测试程序都是写控制台的（用户进程版本的 `printf()` 是这样实现的），所以它们都会出现故障，直到 `write` 可用。

图 9: write()

3 实现系统调用

除了我们要实现的 userprog/syscall.c，在 lib/user 文件夹下，还有一对 syscall.h 和 syscall.c 文件

> **Notice**
>
> 这组文件，是提供给用户的系统调用接口，其通过 int $0x30 中断指令以唤起一个系统调用，最终会落入 userprog/syscall.c 的 syscall_handler() 函数中。

其 enum 中定义了不同的系统调用的序号：

```
enum {
    /* Projects 2 and later. */
    SYS_HALT,                   /**< Halt the operating system. */
    SYS_EXIT,                   /**< Terminate this process. */
    SYS_EXEC,                   /**< Start another process. */
    SYS_WAIT,                   /**< Wait for a child process to die. */
    SYS_CREATE,                 /**< Create a file. */
    SYS_REMOVE,                 /**< Delete a file. */
    SYS_OPEN,                   /**< Open a file. */
    SYS_FILESIZE,               /**< Obtain a file's size. */
    SYS_READ,                   /**< Read from a file. */
    SYS_WRITE,                  /**< Write to a file. */
    SYS_SEEK,                   /**< Change position in a file. */
    SYS_TELL,                   /**< Report current position in a file. */
    SYS_CLOSE,                  /**< Close a file. */

    /* Project 3 and optionally project 4. */
    SYS_MMAP,                   /**< Map a file into memory. */
    SYS_MUNMAP,                 /**< Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,                  /**< Change the current directory. */
    SYS_MKDIR,                  /**< Create a directory. */
    SYS_READDIR,                /**< Reads a directory entry. */
    SYS_ISDIR,                  /**< Tests if a fd represents a directory. */
    SYS_INUMBER                 /**< Returns the inode number for a fd. */
  };
```

syscall.h

故我们需要在 **syscall_handler()** 中实现分发：根据类型分发给各个具体处理函数，逐个实现。返回值要放在 EAX 中传回调用者。

> **Notice**
>
> 在 Pintos 中，struct intr_frame 是用来保存中断处理器状态的结构体。在系统调用返回时，通常会将结果值存储到 f->eax 中，因为这是 x86 架构的约定——返回值通过 eax 寄存器传递。
>
> 对于系统调用 SYS_EXIT，需要返回一个退出码到用户进程，返回值就需要存放到 f->eax 中。类似地，SYS_WRITE 也会将写入的字节数返回到 f->eax。

```c
static void syscall_handler (struct intr_frame *f UNUSED) {
    int *p = f->esp;
    is_valid_addr(p);

    int system_call = *p;
    switch (system_call) {
        case SYS_HALT: syscall_halt(); break;
        case SYS_EXIT: syscall_exit(f); break; // 直接关机，不需要返回值，故是 void function
        case SYS_EXEC: f->eax = syscall_exec(f); break;
        case SYS_WAIT: f->eax = syscall_wait(f); break;
        case SYS_CREATE: f->eax = syscall_creat(f); break;
        case SYS_REMOVE: f->eax = syscall_remove(f); break;
        case SYS_OPEN: f->eax = syscall_open(f); break;
        case SYS_FILESIZE: f->eax = syscall_filesize(f); break;
        case SYS_READ: f->eax = syscall_read(f); break;
        case SYS_WRITE: f->eax = syscall_write(f); break;
        case SYS_SEEK: syscall_seek(f); break; // 调整文件指针的位置，也不需要返回值
        case SYS_TELL: f->eax = syscall_tell(f); break;
        case SYS_CLOSE: syscall_close(f); break;

        default:
        printf("Default %d\n",*p);
    }
}
```

syscall.c

在上方的函数中，有一些函数是不需要返回值的，这些 `void function` 在定义的时候要注意一下。
其次，Pintos 中已经使用了 `syscall_init` 将中断处理函数注册到寄存器：

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

init.c

另外要注意的是，Pintos 中不允许在系统调用中使用 malloc() 函数：

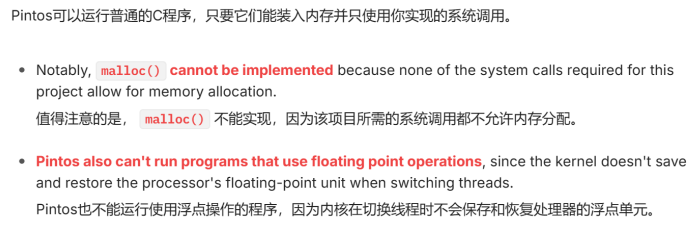


图 10: malloc()

4 处理内存非法访问

PPT 中的提示了使用 **syscall_create** 来实现：

图 11: PPT

为了保证内核不受到破坏，需要立即终止发出非法指针请求的进程。查看 userprog/pagedir.c 和 threads/vaddr.h，可以发现已经为我们定义了以下两个函数:

```
/* Looks up the physical address that corresponds to user virtual address UADDR in PD.
Returns the kernel virtual address
corresponding to that physical address, or a null pointer if UADDR is unmapped. */
void *pagedir_get_page (uint32_t *pd, const void *uaddr) {
uint32_t *pte;
ASSERT (is_user_vaddr (uaddr));
pte = lookup_page (pd, uaddr, false);
if (pte != NULL && (*pte & PTE_P) != 0)
    return pte_get_page (*pte) + pg_ofs (uaddr);
else
    return NULL;
}
```

pagedir_get_page()

```
/* Returns true if VADDR is a user virtual address. */
static inline bool is_user_vaddr (const void *vaddr) {
    return vaddr < PHYS_BASE;
}
```

is_user_vaddr()

除了空指针以外，这就是最常见的两个异常情况。当出现异常的时候都会返回异常信息。可以借助这两个函数来实现内存访问的安全控制。

我们可以跟着这个思路来编写代码：

```
// 检查传入的虚拟地址是否是用户空间的有效地址：
// 使用 is_user_vaddr 判断地址是否在用户地址范围内。如果是有效的用户地址，则使用 pagedir_get_page 从当前线程的页目录中获取该虚拟地址对应的物理页指针。
// 如果地址无效或无法找到对应的物理页，则调用 exit_process(-1) 退出当前进程。

void *is_valid_addr(const void *vaddr) {
    void *page_ptr = NULL;
    if (!is_user_vaddr(vaddr) || !(page_ptr = pagedir_get_page(thread_current()->pagedir, vaddr))) {
        // 地址无效，退出进程
        exit_process(-1);
```

```
        return 0;
    }
// 地址有效，返回物理页指针
return page_ptr;
}
```

syscall_create()

如果地址无效则以异常状态中止进程，有效则返回物理地址。
再编写 pop_stack() 函数，用于从栈中取得元素，方便取得参数：

```
void pop_stack(int *esp, int *a, int offset){
    int *tmp_esp = esp;
    *a = *((int *)is_valid_addr(tmp_esp + offset));
}
```

pop_stack()

这里是因为 Pintos 使用的是 4 字节对齐，所以 offset 参数可以方便地使用整数来取得参数。

Consider a function `f()` that takes three `int` arguments. This diagram shows a sample stack frame as seen by the callee at the beginning of step 3 above, supposing that `f()` is invoked as `f(1, 2, 3)`. The initial stack address is arbitrary:

考虑一个函数 `f()`，它接受三个 `int` 参数。这张图显示了上面步骤3开始时被调用者看到的一个样本栈帧，假设 `f()` 被调用为 `f(1, 2, 3)`。栈的初始地址是任意的：

```
                            +----------------+
                0xbffffe7c |        3       |
                0xbffffe78 |        2       |
                0xbffffe74 |        1       |
stack pointer --> 0xbffffe70 | return address |
                            +----------------+
```

图 12: 栈指针

5 完善系统调用

按照 PPT 要求，在 thread.h 中新增结构体变量：

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /**< Thread identifier. */
    int exit_code;                      /**< Exit code of the thread. */
    enum thread_status status;          /**< Thread state. */
    char name[16];                      /**< Name (for debugging purposes). */
    uint8_t *stack;                     /**< Saved stack pointer. */
    int priority;                       /**< Priority. */
    struct list_elem allelem;           /**< List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /**< List element. */

    bool load_success;                   /**< 标志线程加载是否成功（例如加载可执行文件时使用）。 */
    struct semaphore load_sema;          /**< 信号量，用于同步线程加载进程，例如父线程等待子线程加载完成。 */
    int exit_status;                     /**< 线程的退出状态码，表示线程终止时的状态。 */
    struct list children_list;           /**< 子线程列表，存储当前线程的所有子线程，用于管理父子线程关系。 */
    struct thread *parent;               /**< 指向当前线程的父线程，用于实现父子线程的交互（如退出时通知父线程）。 */
    struct file *self;                   /**< 当前线程正在执行的可执行文件，防止文件在执行期间被修改或删除。 */
    struct list opened_files;            /**< 打开文件的列表，存储当前线程打开的所有文件，用于管理文件资源的生命周期。 */

    int fd_count;                        /**< 文件描述符计数器，分配唯一的文件描述符给线程打开的文件。 */
    struct child_process *waiting_child; /**< 指向线程正在等待的子线程，用于实现父线程等待特定子线程结束的功能。 */
```

图 13: thread.h

同时定义全局锁：**struct lock filesys_lock**，用于文件系统操作的同步。该结构体 lock 位于 **synch.h** 中。

当然，不能忘记了子进程的结构体定义：

```
struct child_process {
    int tid;                        /**< 子进程的线程 ID。 */
    bool if_waited;                 /**< 子进程是否已被父进程等待。 */
    int exit_status;                /**< 子进程的退出状态。 */
    struct list_elem child_elem; /**< 子进程的元素。 */
    struct semaphore wait_sema;  /**< 等待子进程退出的信号量。 */
};
```

child_process

5 .1 halt

这里只需要简单地调用 **devices/shutdown.c** 下的函数接口即可：

```
    void syscall_halt(void){
        shutdown_power_off();
    }
```

6 syscall_exit

```
static void syscall_exit(struct intr_frame *f) {
    // exit_code在系统调用之后被解析为参数
    int exit_code = *(int *)check_read_user_ptr(f->esp + ptr_size, sizeof(int));
    thread_current()->exit_status = exit_code;
    thread_exit();
}
```

Notice

`syscall_exit` 函数实现了系统调用，用于终止当前进程。通过遍历查找对应的子进程描述结构（`child_process`），更新其退出状态码（`exit_status`）以及是否已被父线程等待的标志（`if_waited`）。同时，将当前线程的退出状态存储在其自身的 `exit_status` 字段中。为了避免多线程环境下的竞争问题，函数通过禁用中断（`intr_disable`）确保状态更新的原子性。

最后调用 `thread_exit` 终止当前线程，之所以选择 `thread_exit` 而非直接调用 `process_exit`，是因为前者还包含了对信号量的同步操作，并且可以用于终止用户进程和系统线程。

其中，子函数 `exit_process` 的实现如下：

```
    void exit_process(int status) {
        struct child_process *cp;
        struct thread *cur_thread = thread_current();
        enum intr_level old_level = intr_disable(); // 禁用中断以保证线程状态更新的安全性。

        // 遍历当前线程的父线程的子线程列表，更新当前线程在父线程记录中的状态。
        for (struct list_elem *e = list_begin(&cur_thread->parent->children_list);
             e != list_end(&cur_thread->parent->children_list);
             e = list_next(e)) {
            cp = list_entry(e, struct child_process, child_elem);
            if (cp->tid == cur_thread->tid) {
                cp->if_waited = true;        // 标记当前线程已被父线程等待过。
                cp->exit_status = status;  // 更新退出状态。
            }
        }
```

```
        cur_thread->exit_status = status;  // 设置当前线程的退出状态。
        intr_set_level(old_level); // 恢复中断状态
        thread_exit(); // 退出当前线程
    }
```

7 syscall_exec

```
    // 运行其名称在 cmd_line 中给出的可执行文件，并传递任何给定的参数，返回新进程的进程ID(pid)
    static void syscall_exec(struct intr_frame *f) {
        char *cmd_line = *(char **)check_read_user_ptr(f->esp + ptr_size, ptr_size);
        check_read_user_str(cmd_line);
        f->eax = process_execute(cmd_line);
    }
```

Notice

`syscall_exec` 函数实现了 exec 系统调用，用于运行用户程序指定的可执行文件并传递命令行参数。函数首先从用户栈中读取命令行字符串（`cmd_line`），这是用户程序传递的参数。

随后，通过 `check_read_user_ptr`验证命令行指针是否合法，确保指针指向的地址位于用户空间内，并进一步使用 `check_read_user_str` 验证字符串的内容是否安全可读，防止非法内存访问。验证通过后，函数调用 `process_execute(cmd_line)`，用于创建新进程并执行命令行中指定的程序。`process_execute` 返回新进程的进程 ID（pid），代表子进程的唯一标识符，函数将其存储在中断帧的 eax 寄存器中，供用户程序访问。

8 syscall_wait

```
    int syscall_wait(struct intr_frame *f) {
        tid_t child_tid;
        pop_stack(f->esp, &child_tid, 1);  // Extract the child process ID from the user stack.
        return process_wait(child_tid);    // Wait for the child process and return its exit status.
    }
```

syscall_wait()

Notice

`syscall_wait` 实现了 WAIT 系统调用，用于当前进程等待一个指定的子进程完成执行。它通过中断帧的栈指针（f->esp）从用户栈中获取传递的子进程线程 ID（tid）。

在未实现 `syscall_write` 之前，会出现这样的情况：

```
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/args-none
 2> tests/userprog/args-none.errors > tests/userprog/args-none.output
perl -I../.. ../../tests/userprog/args-none.ck tests/userprog/args-none
FAIL tests/userprog/args-none
Test output failed to match any acceptable form.

Acceptable output:
  (args) begin
  (args) argc = 1
  (args) argv[0] = 'args-none'
  (args) argv[1] = null
  (args) end
  args-none: exit(0)
```

```
Differences in `diff -u' format:
- (args) begin
- (args) argc = 1
- (args) argv[0] = 'args-none'
- (args) argv[1] = null
- (args) end
- args-none: exit(0)
+ Default 9
+ Default 9
+ Default 9
+ Default 9
+ Default 9
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/args-single
rg' < /dev/null 2> tests/userprog/args-single.errors > tests/userprog/args
perl -I../.. ../../tests/userprog/args-single.ck tests/userprog/args-sing
FAIL tests/userprog/args-single
Test output failed to match any acceptable form.
```

图 14: Combination of Failed Results

输出的结果显示为：**Default 9** 这在 `user/syscall.c` 中是 9 号，`SYS_WRITE`，用户程序的打印（输出到 stdout）也是通过写的系统调用实现的。

8 .1 syscall_write

```c
int syscall_write(struct intr_frame *f) {
    int ret;                    /**< 写入操作的返回值。 */
    int size;                   /**< 要写入的数据大小（字节）。 */
    void *buffer;               /**< 数据缓冲区的起始地址。 */
    int fd;                     /**< 文件描述符。 */

    // 从用户栈中读取系统调用参数，分别是 size、buffer 和 fd
    pop_stack(f->esp, &size, 7);
    pop_stack(f->esp, &buffer, 6);
    pop_stack(f->esp, &fd, 5);

    // 检查 buffer 是否为有效的用户地址
    if (!is_valid_addr(buffer))
        ret = -1;

    // 如果文件描述符是标准输出（fd == 1），直接将数据输出到控制台
    if (fd == 1) {
        putbuf(buffer, size); /**< 调用系统提供的 putbuf() 函数写入控制台。 */
        ret = size;           /**< 返回写入的字节数，即 size。 */
    } else {
        // 文件写入操作，需要查找文件描述符对应的文件指针
        enum intr_level old_level = intr_disable(); /**< 禁用中断，避免多线程竞争。 */
        struct process_file *pf = search_fd(&thread_current()->opened_files, fd);
        intr_set_level(old_level); /**< 恢复中断状态。 */
        // 如果文件描述符无效，返回错误
        if (pf == NULL) ret = -1;
        else {
            // 获取文件系统锁，确保文件写操作是线程安全的
            lock_acquire(&filesys_lock);
            ret = file_write(pf->ptr, buffer, size); /**< 写入数据到文件，返回写入字节数。 */
            lock_release(&filesys_lock);
        }
    }
    return ret;
}
```

syscall_write()

Notice

syscall_write 函数实现了系统调用，用于向文件或标准输出写入数据。函数首先通过从用户栈中读取参数（size、buffer、fd）完成参数传递，确保用户态与内核态的隔离。随后，检查缓冲区地址是否为有效的用户地址，以防止非法内存访问，保护操作系统的安全性。如果文件描述符为标准输出（fd == 1），直接调用 putbuf 将数据输出到控制台，绕过文件系统以提高效率。

对于其他文件描述符，函数通过禁用中断和文件系统锁保证线程安全，查询文件描述符对应的文件指针并执行文件写入操作，同时确保多线程环境下的资源管理。

五 实验结果

Question

使用 `make check` 命令时，但此时仍然显示：Run didn’t produce any output.
查询资料后显示，这是因为残留的 `process_wait` 未实现导致用户程序还没执行整个系统就关机了。
https://blog.csdn.net/Altair_alpha/article/details/126819252

根据博主写的彩蛋的内容作出相应的修改，重新 make，可以看到此时已经能够成功通过参数传递的五个测试点。

在 `thread.c` 中添加以下函数：

```
int thread_dead(tid_t tid) {
  struct list_elem *e;
  for (e = list_begin (&all_list); e != list_end (&all_list); e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allelem);
    if (t->tid == tid) return 0;
  }
  return 1;
}
```

将 `process_wait` 函数修改为如下所示：

Notice

之前的问题是系统在用户程序执行完成前就关闭了，导致没有机会输出预期结果。

通过在 `process_wait()` 中添加一个无限循环，主线程可以持续等待子进程完成。这样，用户程序在系统关闭前获得了足够的时间完成执行，并输出结果。

```
int process_wait (tid_t child_tid) {
  while (1) {
    thread_yield();
    if (thread_dead(child_tid)) break;
  }
  return -1;
}

int thread_dead(tid_t tid) {
struct list_elem *e;
for (e = list_begin (&all_list); e != list_end (&all_list); e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allelem);
    if (t->tid == tid) return 0;  // 如果找到目标线程，返回 0（未死亡）
}
return 1;                         // 如果目标线程不在 ‘all_list‘ 中，返回 1（已死亡）
}
```

- `all_list` 是所有线程的链表。一个线程在死亡后会从 `all_list` 中移除。
- 如果 `tid` 找不到匹配的线程，则认为线程已经死亡。

运行 `make clean 和 make check`

得到以下结果：

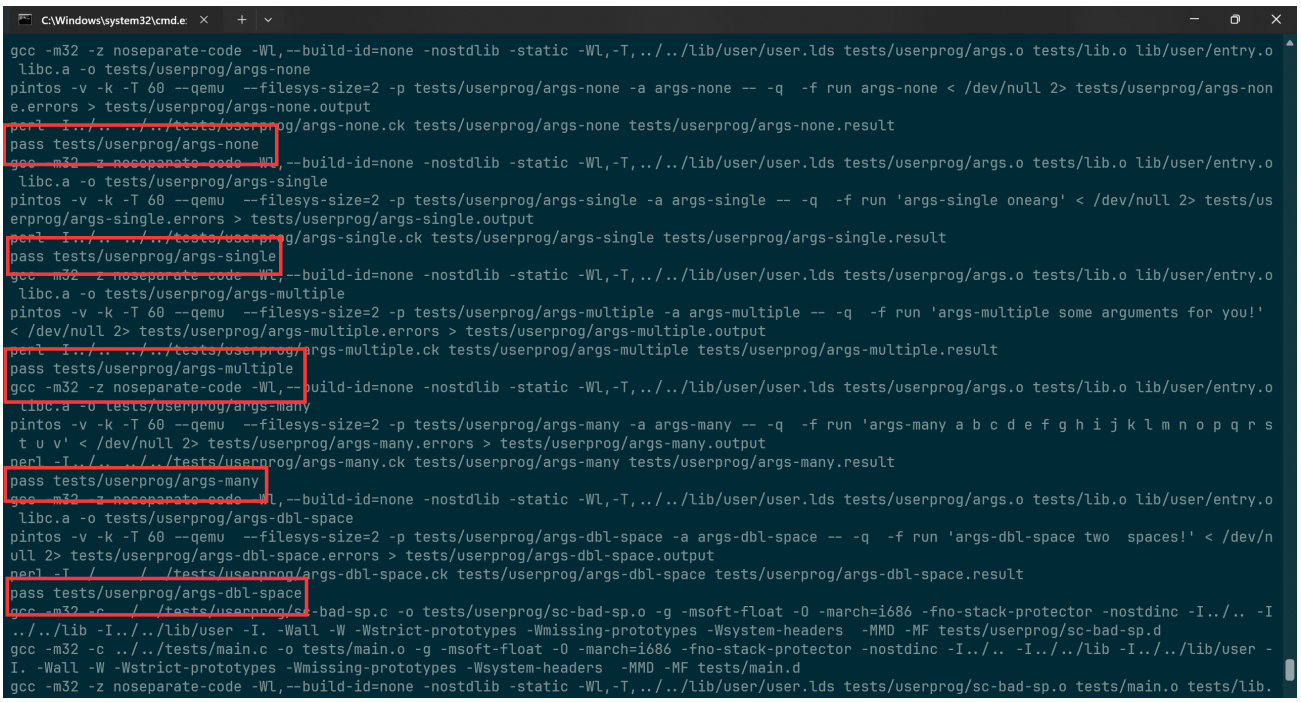


图 15: make check

等到程序最终运行完成时，可以看到 5 个与参数传递有关的测试点均已通过。

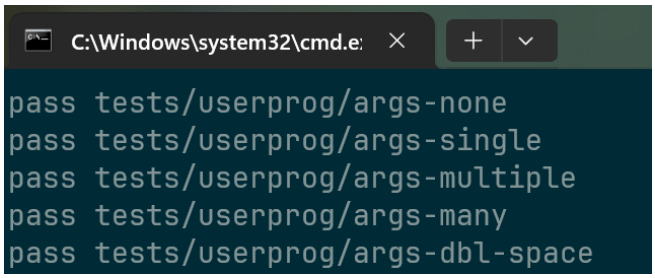


图 16: make check

六 附录

参考资料

- Pintos 单一测试点代码：https://pastebin.com/tFRfywvv
- Pintos Lab2：用户程序 User Programs（上）：https://blog.csdn.net/Altair_alpha/article/details/126819252
- Pintos 指导手册：https://pkuflyingpig.gitbook.io/pintos/project-description/lab2-user-programs/suggestions
- OS 实验：pintos project2：https://zhuanlan.zhihu.com/p/382326973
- 斯坦福大学 Pintos Project1、2 指南 + 总结：https://zhuanlan.zhihu.com/p/104497182