Project 2实验手册

>

现在是时候开始着手处理系统中允许**运行用户程序**的部分了。基本代码已经支持加载和运行用户程序，但不支持I/O或交互。在这个项目中，您将允许程序通过系统调用与操作系统交互。

您将在**userprog目录**下完成此任务，但您也将与Pintos的几乎所有其他部分进行交互。我们将在下面描述相关部分。

您可以在项目1提交的基础上构建项目2，也可以重新开始。**此任务不需要项目1中的代码**。“闹钟”功能在项目3和项目4中可能有用，但并非严格要求。

在上一个项目中，我们将测试代码直接编译到内核中，因此我们必须在内核中要求特定的函数接口。从现在起，我们将通过运行用户程序来测试您的操作系统。这给了你更大的自由。您必须确保用户程序界面符合此处描述的规范，但考虑到该约束，您可以随意重新构造或重写内核代码。

您将需要与此项目的文件系统代码进行接口，因为用户程序是从文件系统加载的，并且您必须实现的许多系统调用都与文件系统进行处理。但是，本项目的重点不是文件系统，因此我们在**filesys目录**中提供了一个简单但完整的文件系统。您需要查看 `filesys.h` **和** `file.h` **接口**，以了解如何使用文件系统，尤其是它的许多限制。
不需要修改此项目的文件系统代码，因此我们建议您不要修改。在文件系统上工作可能会分散您对本项目重点的注意力。

建议实现顺序

- 参数传递（参见第3.3.3节参数传递）
- 用户内存访问（参见第3.1.5节访问用户内存）。所有系统调用都需要读取用户内存。很少有系统调用需要写入用户内存。
- 系统调用基础设施（参见第3.3.4节系统调用）。实现足够的代码，以便从用户堆栈读取系统调用号，并基于它分派给处理程序。
- 退出系统调用。以正常方式完成的每个用户程序都调用exit。即使是从main（）返回的程序也会间接地调用exit（参见lib/user/entry.c中的_start（））。
- 写入系统调用，用于写入系统控制台fd 1。我们所有的测试程序都会写入控制台（printf（）的用户进程版本就是这样实现的），因此在写入可用之前，它们都会出现故障。
- 现在，将进程_wait（）更改为无限循环（一个永远等待的循环）。提供的实现会立即返回，因此Pintos将在任何进程实际运行之前关闭。您最终需要提供一个正确的实现。

任务介绍

- **3.3.2过程终止消息**
  每当用户进程因调用exit或任何其他原因而终止时，请打印该进程的名称和退出代码，其格式与打印F相同（“%s:exit（%d）\n”，...）；。打印的名称应该是传递给进程_execute（）的全名，忽略命令行参数。当不是用户进程的内核线程终止或调用halt系统调用时，不要打印这些消息。当进程加载失败时，该消息是可选的。
  除此之外，不要打印Pintos提供的尚未打印的任何其他邮件。在调试期间，您可能会发现额外的消息很有用，但它们会混淆分级脚本，从而降低您的分数。
- **3.3.3参数传递**

当前，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，这样它就不用简单地将程序文件名作为参数，而是在空格处将其划分为单词。第一个字是程序名，第二个字是第一个参数，依此类推。也就是说，process_execute（“grep foo bar”）应该通过两个参数foo和bar运行grep。

- **系统调用**
- 在 `userprog/syscall.c` 中实现系统调用处理程序。我们提供的框架实现通过终止进程来“处理”系统调用。它需要检索系统调用号，然后检索任何系统调用参数，并执行适当的操作。
    - **void halt（void）**
      通过调用shutdown_power_off（）（在 `devices/shutdown.h` 中声明）终止Pintos。这应该很少使用，因为您会丢失一些关于可能的死锁情况等的信息。
    - **void exit (int status)**
      终止当前用户程序，将状态返回内核。如果进程的父进程等待它（见下文），则将返回此状态。通常，状态为0表示成功，非零值表示错误。
    - **pid_t exec (const char *cmd_line)**
      运行名称在cmd_line中给定的可执行文件，传递任何给定参数，并返回新进程的程序id（pid）。如果程序由于任何原因无法加载或运行，则必须返回pid-1，否则该pid不应是有效的pid。因此，在知道子进程是否成功加载其可执行文件之前，父进程无法从exec返回。您必须使用适当的同步来确保这一点。
    - **int wait (pid_t pid)**

      等待子进程pid并检索子进程的退出状态。

      如果pid仍处于活动状态，则等待它终止。然后，返回pid传递给exit的状态。如果pid没有调用exit（），但被内核终止（例如，由于异常而终止），那么wait（pid）必须返回-1。父进程等待在父进程调用wait时已经终止的子进程是完全合法的，但是内核仍然必须允许父进程检索其子进程的退出状态，或者知道子进程已被内核终止。
    - **bool create (const char *file, unsigned initial_size)**

      创建一个名为file initially_size字节的新文件。如果成功，则返回true，否则返回false。创建新文件不会打开它：打开新文件是一个单独的操作，需要打开系统调用。
    - **bool remove (const char *file)**

      删除名为file的文件。如果成功，则返回true，否则返回false。无论文件是打开的还是关闭的，都可以将其删除，删除打开的文件不会将其关闭。有关详细信息，请参见删除打开的文件。
    - **int open (const char *file)**

      打开名为file的文件。返回称为“文件描述符”（fd）的非负整数句柄，如果无法打开文件，则返回-1。

      编号为0和1的文件描述符为控制台保留：fd 0（标准输入文件号）为标准输入，fd 1（标准输出文件号）为标准输出。开放系统调用永远不会返回这些文件描述符中的任何一个，它们仅作为系统调用参数有效，如下所述。

      每个进程都有一组独立的文件描述符。子进程不会继承文件描述符。

      当单个文件被多次打开时，无论是由单个进程还是不同的进程打开，每次打开都会返回一个新的文件描述符。
    - **int filesize (int fd)**

      返回作为fd打开的文件的大小（以字节为单位）。
    - **int read (int fd, void *buffer, unsigned size)**

      从作为fd打开的文件中读取大小字节到缓冲区中。返回实际读取的字节数（文件末尾为0），如果无法读取文件（由于文件结尾以外的条件），则返回-1。Fd 0使用input_getc（）从键盘读取数据。

- **int write (int fd, const void *buffer, unsigned size)**

  将大小字节从缓冲区写入打开的文件fd。返回实际写入的字节数，如果某些字节无法写入，则可能小于大小。

- **void seek (int fd, unsigned position)**

  将打开文件fd中要读取或写入的下一个字节更改为位置，以文件开头的字节表示。（因此，位置0是文件的起点。）
  超过文件当前结尾的搜索不是错误。稍后的读取获得0字节，表示文件结束。稍后的写入扩展文件，用零填充任何未写入的间隙。（然而，在Pintos中，在项目4完成之前，文件的长度是固定的，因此写入文件末尾将返回一个错误。）这些语义在文件系统中实现，在系统调用实现中不需要任何特殊的工作。

- **unsigned tell (int fd)**

  返回打开文件fd中要读取或写入的下一个字节的位置，以从文件开头开始的字节数表示。

- **void close (int fd)**

  关闭文件描述符fd。退出或终止进程会隐式关闭其所有打开的文件描述符，就像为每个描述符调用此函数一样。

该文件定义了其他系统调用。暂时忽略它们。您将在项目3中实现其中一些，在项目4中实现其余部分，因此请确保在设计系统时考虑可扩展性。

> 要实现系统调用，您需要提供在用户虚拟地址空间中读取和写入数据的方法。在获得系统调用号之前，您需要此功能，因为系统调用号位于用户虚拟地址空间中的用户堆栈上。这可能有点棘手：如果用户提供了无效的指针、指向内核内存的指针或其中一个区域的部分块，该怎么办？您应该通过终止用户进程来处理这些情况。我们建议在实现任何其他系统调用功能之前编写和测试此代码。有关更多信息，请参阅第3.1.5节访问用户内存。

您必须同步系统调用，以便任意数量的用户进程可以同时进行调用。特别是，一次从多个线程调用filesys目录中提供的文件系统代码是不安全的。系统调用实现必须将文件系统代码视为关键部分。不要忘记，process_execute（）也会访问文件。目前，我们建议不要修改filesys目录中的代码。

我们在`lib/user/syscall.c`中为每个系统调用提供了一个用户级函数。这些为用户进程提供了一种从C程序调用每个系统调用的方法。每种方法都使用少量内联汇编代码来调用系统调用，并（如果合适）返回系统调用的返回值。

- **3.3.5拒绝写入可执行文件**
  添加代码以拒绝写入用作可执行文件的文件。许多操作系统之所以这样做，是因为如果进程试图运行磁盘上正在更改的代码，会产生不可预测的结果。一旦在项目3中实现了虚拟内存，这一点就显得尤为重要，但即使现在也不会有什么坏处。
  您可以使用file_deny_write（）防止写入打开的文件。对该文件调用file_allow_write（）将重新启用它们（除非该文件被另一个打开程序拒绝写入）。关闭文件还将重新启用写入。因此，要拒绝对进程可执行文件的写入，必须在进程仍在运行时保持其打开状态。

目标开始

系统调用是什么?

在 Pintos 中，用户程序调用整数 $0x30进行系统调用，此时用户就会把没有权限干的活交给系统调用去干，系统调用的栈指针就是`esp`，返回值是`eax`。**我们需要干的事说白了就是根据`esp`指向栈的参数内容，完成系统调用对应的功能，最后把返回值放到`eax`里。**

`judge_pointer()`

根据Pintos文档描述

> 要实现系统调用，您需要提供在用户虚拟地址空间中读取和写入数据的方法。**在获得系统调用号之前，您需要此功能，因为系统调用号位于用户虚拟地址空间中的用户堆栈上。**这可能有点棘手：如果用户提供了无效的指针、指向内核内存的指针或其中一个区域的部分块，该怎么办？您应该通过终止用户进程来处理这些情况。**我们建议在实现** 任何其他系统调用功能 **之前编写和测试此代码。**有关更多信息，请参阅第3.1.5节访问用户内存。

因此，我们需要**自定义**一个判断函数，**来完成判断用户提供的指针指向的地址是否合法。**

支持读取和写入用户内存以进行系统调用。

至少有两种合理的方法可以正确地做到这一点。

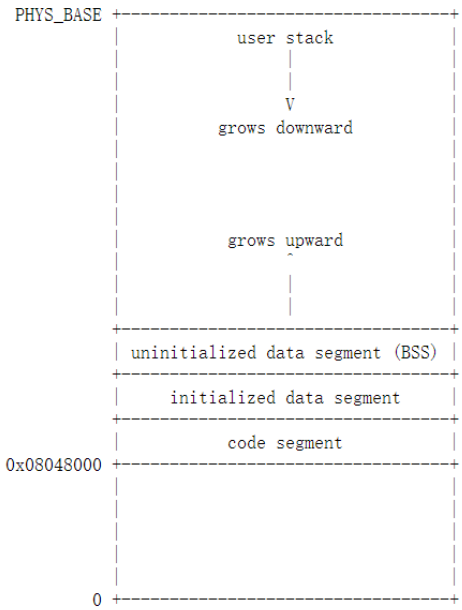
第一种方法是验证用户提供的指针的有效性，然后取消引用它。如果您选择这条路线，您将需要查看 `userprog/pagedir.c` 和 `threads/vaddr.h` 中的函数。这是处理用户内存访问的最简单方法。

第二种方法是只检查用户指针是否指向下方 `PHYS_BASE`，然后取消引用它。无效的用户指针将导致“页面错误”，你可以通过修改代码的处理 `page_fault()` 在 `userprog / exception.c` 。这种技术通常更快，因为它利用了处理器的 `MMU`，所以它往往用于实际内核（包括 `Linux`）。

如何判断此时指针指向的是属于用户的内存还是内核的内存呢？实验手册上给出了我们关于内存的介绍

3.1.4.1 Typical Memory Layout

Conceptually, each process is free to lay out its own user virtual memory however it chooses. In practice, user virtual memory is laid out like this:



下表显示了在**用户程序开始之前**堆栈和相关寄存器的状态，假设 `PHYS_BASE` 为0xc000000：

在本例中，堆栈指针将初始化为0xbffffffcc。

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 (*) ()`

如上所示，**您的代码应该从用户虚拟地址空间的最顶端开始堆栈**，在虚拟地址`PHYS_BASE`（在`threads/vaddr.h`中定义）下方的页面中。

因此，我们不仅需要检测指针指向的是用户存储空间还是内核存储空间，我们还需要判断当前指针是否有可能指向的是内核空间，将指针加4，如果加4后指向的地址属于内核存储空间（加4的原因是因为Type char[4]），则应该报错。

在`userprog/syscall.c`中定义函数`judge_ptr()`

`judge_pointer()`

```
void *
judge_pointer(const void *vaddr)
{
  void *ptr = pagedir_get_page (thread_current()->pagedir, vaddr);
  /* 判断是否属于用户地址空间，空间是否已经被映射（有效性） */
  if (!is_user_vaddr(vaddr) || ptr == NULL)
  {
    // 设置线程状态值为-1，表示线程错误
    thread_current()->st_exit = -1;
    // 线程退出
    thread_exit ();
  }
  /* 广度范围上的判断 */
  uint8_t *check_byteptr = (uint8_t *) vaddr;
  // 注意，最多加到3，因为4的时候已经跳出了一个内存单位
  for (uint8_t i = 0; i < 4; i++)
  {
    // 判断辅助函数
    if (get_user(check_byteptr + i) == -1)
    {
```

```
    thread_current()->st_exit = -1;
    thread_exit ();
    }
  }
  return ptr;
}
```

> 调用的相关函数

`is_user_vaddr()`

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

返回传入的地址指针是否指向的是用户虚拟空间

判断的方法也很简单，只要指针值小于定义的物理地址范围，就属于用户地址空间，返回`false`，否则返回`true`

`pagedir_get_page()`

```
/* 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;
}
```

查找与PD中的用户虚拟地址UADDR相对应的物理地址。返回与该物理地址对应的内核虚拟地址，如果UADDR未映射，则返回空指针

`get_user()`

实验手册中给了我们辅助函数的定义

causes a page fault, because there's no way to return an error code from a memory access. Therefore, for those who want to try the latter technique, we'll provide a little bit of helpful code:

```
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
  int error_code;
  asm ("movl $1f, %0; movb %b2, %1; 1:"
       : "=&a" (error_code), "=m" (*udst) : "q" (byte));
  return error_code != -1;
}
```

```c
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}
```

读取用户虚拟地址UADDR处的字节。UADDR必须低于PHYS_BASE。如果成功，则返回字节值；如果发生segfault，则返回-1。

这个函数我们也需要定义在`syscall.c`文件中，因为我们需要+4来判断用户调用的指针指向的虚拟地址是否指向了内核存储空间。

`page_fault()`

我们不仅需要主动判断指针指向内存错误，也需要在内存错误处理函数中处理这种错误

> 这些函数中的每一个都假设用户地址已被验证为低于PHYS_BASE。他们还假设您已经修改了page_fault（），因此内核中的页面错误只会将`eax`设置为0xffffffff，并将其以前的值复制到`eip`中。

在`execption.c`文件中

```c
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  asm ("movl %%cr2, %0" : "=r" (fault_addr));
```

```
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  // 新增的代码
  // 发生错误，cpu返回-1，退出程序
  if (!user)
  {
     f->eip = f->eax;//eip:寄存器存放下一个CPU指令存放的内存地址 EAX:返回值。bshd
     f->eax = -1;
     return;
  }

  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

syscall_init()

准备第二步，实现系统调用的第一步，也就是获取到系统调用

系统调用属于中断，因此初始程序中的第一行其实就是利用中断机制

在文件`src/lib/syscall-nr.h`中，我们可以找到所有系统调用的标识号

```
/* System call numbers. */
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. */
}
```

> C语言`enum`
>
> 枚举类，每一项的索引值默认从0开始依次递增

怎么获取到系统调用的栈指针指向的当前值呢?

我们可以发现源程序中的`syscall_call()`的参数出现了一个结构体：`intr_frame`，我们在`user/interrupt.h`文件中找出其具体定义

```
/* Interrupt stack frame. */
struct intr_frame
  {
    /* Pushed by intr_entry in intr-stubs.S. */
    uint16_t ss, :16;           /* Data segment for esp. */
  };
```

因此，这个结构体存储的就是当前系统调用的栈指针指向的值，所以，我们需要定义一个以此结构为类型的数组`syscallArray`，从中取出当前的指针值，根据系统调用类型的不同，执行不同的系统调用

在 Pintos 中，用户程序调用整数 $0x30进行系统调用，此时用户就会把没有权限干的活交给系统调用去干，系统调用的栈指针就是`esp`，返回值是`eax`

所以，第一行语句的函数`intr_register_int()`其实就是调用了Pintos的系统中断功能，将按照传入的值进行中断类型的分类与存储

```
static void
register_handler (uint8_t vec_no, int dpl, enum intr_level level,
                  intr_handler_func *handler, const char *name)
{
  ASSERT (intr_handlers[vec_no] == NULL);
  if (level == INTR_ON)
    idt[vec_no] = make_trap_gate (intr_stubs[vec_no], dpl);
  else
    idt[vec_no] = make_intr_gate (intr_stubs[vec_no], dpl);
  // 存储本次系统中断需要执行的操作，我们传入的是系统中断调用的函数名
  intr_handlers[vec_no] = handler;
  // 存储本次系统中断的名称，我们传入的是syscall，表示系统调用
  intr_names[vec_no] = name;
}

void
intr_register_int (uint8_t vec_no, int dpl, enum intr_level level,
                   intr_handler_func *handler, const char *name)
{
  ASSERT (vec_no < 0x20 || vec_no > 0x2f);
  register_handler (vec_no, dpl, level, handler, name);
}
```

第一步系统调用中断存储完成后，我们就可以在数组中存储系统调用对应的操作，使得之后在执行系统调用时执行数组中对应的操作即可

```
// src/userprog/syscall.c

// 存储系统调用类型的数组syscallArray
static void (*syscallArray[max_syscall])(struct intr_frame *);
// 系统调用
void sys_halt(struct intr_frame* f); /* syscall halt. */
void sys_exit(struct intr_frame* f); /* syscall exit. */
void sys_exec(struct intr_frame* f); /* syscall exec. */
void sys_create(struct intr_frame* f); /* syscall create */
void sys_remove(struct intr_frame* f); /* syscall remove */
void sys_open(struct intr_frame* f);/* syscall open */
```

```
void sys_wait(struct intr_frame* f); /*syscall wait */
void sys_filesize(struct intr_frame* f);/* syscall filesize */
void sys_read(struct intr_frame* f);  /* syscall read */
void sys_write(struct intr_frame* f); /* syscall write */
void sys_seek(struct intr_frame* f); /* syscall seek */
void sys_tell(struct intr_frame* f); /* syscall tell */
void sys_close(struct intr_frame* f); /* syscall close */

// 系统调用初始化
void
syscall_init(void) {
    // 存储中断类型为系统调用syscall，存储中断对应的操作syscall_handler函数，通过0x30识别为系统调用
    intr_register_int(0x30, 3, INTR_ON, syscall_handler, "syscall");
    // 存储系统调用对应的操作，SYS_EXEC为Pintos定义的系统调用标识号
    // 使用的是枚举类中定义的系统调用标识号，因此实际存储的数组为下标从0开始递增的数组
    syscallArray[SYS_EXEC] = &sys_exec;
    syscallArray[SYS_HALT] = &sys_halt;
    syscallArray[SYS_EXIT] = &sys_exit;
    syscallArray[SYS_WAIT] = &sys_wait;
    syscallArray[SYS_CREATE] = &sys_create;
    syscallArray[SYS_REMOVE] = &sys_remove;
    syscallArray[SYS_OPEN] = &sys_open;
    syscallArray[SYS_WRITE] = &sys_write;
    syscallArray[SYS_SEEK] = &sys_seek;
    syscallArray[SYS_TELL] = &sys_tell;
    syscallArray[SYS_CLOSE] = &sys_close;
    syscallArray[SYS_READ] = &sys_read;
    syscallArray[SYS_FILESIZE] = &sys_filesize;
}
```

syscall_handler()

`syscall_handler()` 函数执行系统调用函数

```
/* 检测指针是否正确，检测系统调用号是否正确，执行系统调用 */
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  int * ptr = f->esp;
  // 当前如果要执行系统调用，则需要判断指针是否指向正确，这一部分功能我们已经在上面的judge_pointer()函数中实现了，直接调用即可
  judge_pointer (ptr + 1);
  // 光指针正确了还不行，因为这样也可以调用不是系统调用的函数，因此，我们需要检测寄存器的值，在src/lib/syscall-nr.h文件中，系统调用总共有20个，因此，只要寄存器的值大于20或小于0，就说明当前调用了系统调用处理函数但是没有调用系统调用，此时程序应该报错并退出
  int type = * (int *)f->esp;
  if(type <= 0 || type >= 20){
    // 设置线程状态值为-1，表示线程错误
    thread_current()->st_exit = -1;
    // 线程退出
    thread_exit ();
  }
  // 上述判断都通过后，可以执行系统调用函数，f为系统调用函数相关参数
  syscallArray[type](f);
  //printf ("system call!\n");
  //thread_exit ();
}
```

这些工作完成后，我们就可以正式开始完成系统调用函数功能代码实现了。

halt()

> `void halt (void)` 通过调用 `shutdown_power_off()` （在 `devices/shutdown.h` 中声明）终止Pintos。这应该很少使用，因为您会丢失一些关于可能的死锁情况等的信息
>
> > `halt`：停止

我们首先来看一下 `shutdown_power_off()` 函数的功能

```
// devices/shutdown.h

/* Powers down the machine we're running on,
   as long as we're running on Bochs or QEMU. */
void
shutdown_power_off (void)
{
  const char s[] = "Shutdown";
  const char *p;
  // ...
}
```

关闭我们正在运行的机器的电源，只要我们使用 `Bochs` 或 `QEMU`

因此，我们按照实验文档，直接调用 `shutdown_power_off()` 函数即可

```
#include <devices/shutdown.h>

// 系统调用：halt()
// 关闭系统
void
sys_halt (struct intr_frame* f)
{
    shutdown_power_off();
}
```

exit()

> `void exit (int status)`
>
> 终止当前用户程序，将状态返回内核。如果进程的父进程等待它（见下文），则将返回此状态。通常，状态为0表示成功，非零值表示错误。

```
void
sys_exit(struct intr_frame *f) {
    uint32_t *ptr = f->esp;
    // ptr里指向系统调用函数名，ptr + 1里指向系统调用第一个参数
    // 也可以用f->esp + 4来表示ptr指向下一位
    judge_pointer(ptr + 1);
    // 指针移动，指向第一个参数
    *ptr++;
    // 第一个参数保存了int status，将其保存在进程状态中
    thread_current()->st_exit = *ptr;
    // 进程退出
    thread_exit();
}
```

不要忘记移动指针。

exec()

> `pid_t exec (const char *cmd_line)`
>
> 运行名称在 `cmd_line` 中给定的可执行文件，传递任何给定参数，**并返回新进程的程序id（pid）**。如果程序由于任何原因无法加载或运行，**则必须返回pid-1**，否则该pid不应是有效的pid。因此，在知道子进程是否成功加载其可执行文件之前，**父进程无法从exec返回**。您必须使用适当的同步来确保这一点。

是不是有点熟悉？没错，这个函数模拟的功能实际上就是 `Linux` 中的 `exec` 函数族的功能，只不过传递的参数类型不一样。

调用其他函数替代子进程，这个函数功能我们在Project 1中也用到过

```
// process.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
   thread id, or TID_ERROR if the thread cannot be created. */
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;
}
```

实际上就是复制了一份必要的参数，创建一个新的进程来执行指定的功能

因此，我们只需要调用这个函数就可以了，但是也要检测必要参数的正确性

```
void
sys_exec (struct intr_frame* f)
{
  uint32_t *ptr = f->esp;
  // ptr中存放的是系统调用函数名称，ptr+1中存放的是系统调用函数参数*cmd_line
  judge_pointer (ptr + 1);
  // 重点：参数需要检查，但是参数本身又是一个指针，指向了存放调用函数需要的参数地址，我们也需要检查这个指针指向的地址是否正确
  judge_pointer (*(ptr++));
  // 调用函数process_execute()，函数返回值是进程pid，函数返回值存放在寄存器eax中
  f->eax = process_execute((char*)* ptr);
}
```

wait()

> `int wait (pid_t pid)`
>
> 等待子进程pid并检索子进程的退出状态。
>
> 如果pid仍处于活动状态，则等待它终止。然后，返回pid传递给exit的状态。如果pid没有调用exit（），但被内核终止（例如，由于异常而终止），那么wait（pid）必须返回-1。父进程等待在父进程调用wait时已经终止的子进程是完全合法的，但是内核仍然必须允许父进程检索其子进程的退出状态，或者知道子进程已被内核终止。
>
> **如果以下任一条件为真，wait必须失败并立即返回-1：**
>
> - pid不引用调用进程的直接子进程。当且仅当调用进程从对exec的成功调用中接收到pid作为返回值时，pid才是调用进程的直接子进程。
>   请注意，子进程不是继承的：如果A生成子进程B，B生成子进程C，则A不能等待C，即使B已死亡。进程A对 `wait(C)` 的调用必须失败。类似地，如果孤立进程的父进程提前退出，则不会将其分配给新的父进程。
> - 调用wait的进程已经调用了pid上的wait。也就是说，一个进程最多只能等待一次给定的子进程。
>
> 进程可能产生任意数量的子进程，以任意顺序等待它们，甚至可能在没有等待部分或全部子进程的情况下退出。你的设计应该考虑等待发生的所有方式。必须释放进程的所有资源，包括其结构线程，无论其父进程是否等待它，也不管子进程是在其父进程之前还是之后退出。
>
> 您必须确保Pintos在初始进程退出之前不会终止。提供的Pintos代码试图通过从 `main()`（在 `threads/init.c` 中）调用 `process_wait()`（在 `userprog/process.c` 中）来实现这一点。**我们建议您根据函数顶部的注释实现 `process_wait()`，然后根据 `process_wait()` 实现wait系统调用。**
>
> **实现这个系统调用需要比其他任何调用都多得多的工作。**

根据提示，我们找到 `process_wait()` 函数

```
/* Waits for thread TID to die and returns its exit status.  If
   it was terminated by the kernel (i.e. killed due to an
   exception), returns -1.  If TID is invalid or if it was not a
   child of the calling process, or if process_wait() has already
   been successfully called for the given TID, returns -1
   immediately, without waiting.
   This function will be implemented in problem 2-2.  For now, it
   does nothing. */
int
process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

等待线程TID结束并返回其退出状态。如果它被**内核终止**（即由于异常而终止），则返回-1。如果**TID无效**，或者它不是调用进程的子进程，或者如果**已经为给定TID成功调用了** `process_wait()`，则立即返回-1，而不等待。

此功能将在问题2-2中实现。现在，它什么也没做。

在线程结构体中再加入下面的代码

```
struct list all_child_threads;      /* 存储所有子进程的结构体，为什么设置为list结构，后面会有解释 */
struct child_process * thread_child;/* 存储线程的子进程 */
int exit_status;                    /* 退出状态 */
```

创建子进程结构体 `child_process`

```
struct child_process
  {
    int tid;
    struct list_elem child_elem;          // 上面设置为了list，所以这里设置为list_elem类型
    struct semaphore sema;                // 控制等待的信号量
    bool iswait;           /* 子进程运行状态 */
    int exit_status_child;                // 子进程退出状态
  };
```

修改 `thread_create()` 函数

```
// 初始化子进程，分配存储空间
  t->thread_child = malloc(sizeof(struct child));
// tid = t->tid = allocate_tid ()，子进程的tid初始化为自己的tid，参考Linux
  t->thread_child->tid = tid;
// 初始化子进程的信号量
  sema_init (&t->thread_child->sema, 0);
// 将子进程放入到所有进程列表中，注意，我们放入的是list_elem类型的子进程指针，至于为什么这么做，后面有详细的解释
  list_push_back (&thread_current()->all_child_threads, &t->thread_child->child_elem);
// 子进程的退出状态设置为最大值
  t->thread_child->exit_status_child = UINT32_MAX;
// 子进程没有在运行
  t->thread_child->iswait = false;
```

思路：我们既然需要实现父进程对子进程的`wait()`函数，那么就需要对子进程进行相应的处理，首先，父进程需要知道，自己要`wait()`的子进程在哪，所以需要给进程结构体中增加`thread_child`，表示父进程拥有的子进程，也要增添退出状态记录，同理，也要单独增加子进程结构体，包含了子进程的进本信息。之后，我们就需要在创建进程的时候把子进程也增添到父进程的结构体中，同时将子进程tid赋值为当前tid，子进程信号量需要初始化，，将子进程的状态设置为false，表示不运行，子进程退出状态设置为最大值，并把初始化好的子进程放入父进程的所有子进程列表中，至此，一个带子进程创建过程的进程创建函数就完成了。

创建完成了，那么我们下一步的工作就是实现wait()功能，首先据需要按照给定的子进程tid，找出来你想要暂停哪个子进程，找出来后，判断该进程是否已经被wait，如果已经被wait，则返回-1表示错误，否则将iswait改为true，表示该进程已经被wait，然后阻塞该进程。如果没有找到该进程，则说明需要wait的进程不存在，返回-1表示错误，最后，将该子进程从总子进程列表中删除，并将子进程的退出状态作为该函数的返回值返回，按照题目要求。

> 正式开始之前，我们需要先了解一下Pintos定义的`list.c`文件

这种双链表的实现不需要使用动态分配的内存。相反，每个结构这是一个潜在的列表元素，必须嵌入一个结构列表元素成员所有列表函数都在这些结构上运行列出所有的元素。 **`list_entry`宏允许从结构列表元素返回到`包含`它的结构对象。**

因此，我们找到了**通过`list_elem`返回包含这一项结构体**的方法：`list_entry()`

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

取出列表中第一个元素的方法：`list_begin()`

```
/* Returns the beginning of LIST.  */
struct list_elem *
list_begin (struct list *list)
{
  ASSERT (list != NULL);
  return list->head.next;
}
```

取出当前元素的下一个元素的方法：`list_next()`

注意，该函数不需要传入原list表，因为`list`和`list_elem`的实现方式是链表

```
/* Returns the element after ELEM in its list.  If ELEM is the
   last element in its list, returns the list tail.  Results are
   undefined if ELEM is itself a list tail. */
struct list_elem *
list_next (struct list_elem *elem)
{
  ASSERT (is_head (elem) || is_interior (elem));
  return elem->next;
}
```

取出列表中最后一个元素的方法：`list_end()`

```
/* Returns LIST's tail.
   list_end() is often used in iterating through a list from
   front to back.  See the big comment at the top of list.h for
   an example. */
struct list_elem *
list_end (struct list *list)
{
  ASSERT (list != NULL);
  return &list->tail;
}
```

删除列表中指定元素的方法：`list_remove()`

```
/* Removes ELEM from its list and returns the element that
   followed it.  Undefined behavior if ELEM is not in a list.
   A list element must be treated very carefully after removing
   it from its list.  Calling list_next() or list_prev() on ELEM
   will return the item that was previously before or after ELEM,
   but, e.g., list_prev(list_next(ELEM)) is no longer ELEM!
   The list_remove() return value provides a convenient way to
   iterate and remove elements from a list:
   for (e = list_begin (&list); e != list_end (&list); e = list_remove (e))
     {
       ...do something with e...
     }
   If you need to free() elements of the list then you need to be
   more conservative.  Here's an alternate strategy that works
   even in that case:
   while (!list_empty (&list))
     {
       struct list_elem *e = list_pop_front (&list);
       ...do something with e...
     }
*/
struct list_elem *
list_remove (struct list_elem *elem)
{
  ASSERT (is_interior (elem));
  elem->prev->next = elem->next;
  elem->next->prev = elem->prev;
  return elem->next;
}
```

这些方法正好满足了我们遍历整个子进程数组寻找对应的子进程的需求，这也就是为什么前面结构体在定义时选择 `list` 和 `list_elem` 作为结构的原因。

开始吧！

```
int
process_wait (tid_t child_tid UNUSED)
{
  // 第一步，找出指定child_tid的子进程
  // 所有子进程的列表，注意，类型为list
  struct list *allchilds = &thread_current()->childs;
  // 定义一个子进程指针，注意，类型为list_elem
  struct list_elem *child_ptr;
  // 取出第一个子进程
  child_ptr = list_begin (l);
```

```
  // 定义一个子进程指针，注意，类型为child，我们之所以要使用list和list_elem类型，就是因为Pintos中对这种数据类型提供了方便的遍历方法，因此，我们需要用真正的子进程数据格式child来接收遍历出来的list_elem子进程类型
  struct child *child_ptr2 = NULL;
  // 开始遍历
  while (child_ptr != list_end (l))
  {
    // 根据list_elem返回包含list_elem的结构体child，我们通过这种巧妙地转变实现对子进程地遍历查找
    child_ptr2 = list_entry (child_elem_ptr, struct child, child_elem);
    // 判断是不是我们要找地子进程，通过pid来判断
    if (child_ptr2->tid == child_tid)
    {
      // 判断当前进程是否已经被wait
      if (child_ptr2->iswait == false)
      {
        // 如果没有被wait，则将其iswait属性状态改为true，表示该进程已经被wait了
        child_ptr2->iswait = true;
        // 调用sema_down()函数阻塞子进程
        sema_down (&child_ptr2->sema);
        // 找到目标函数后，直接退出while循环即可
        break;
      }
      // 当前进程已经被wait，根据实验要求，返回-1
      else
      {
        return -1;
      }
    }
    // 没有找到子进程，子进程指针指向所有子进程列表中地下一个子进程
    child_ptr = list_next (child_ptr);
  }
  // 如果直到找完整个所有子进程列表都还没有找到目标tid地子进程，判断条件是当前子进程指针是否等于所有子进程列表中地最后一个子进程
  if (child_ptr == list_end (l)) {
    // 找不到目标tid子进程，函数返回-1
    return -1;
  }
  // 在所有子进程列表中删除目标tid子进程
  list_remove (child_ptr);
  // 返回子进程地退出状态值
  return child_ptr2->exit_status_child;
}
```

相关函数调用说明

`sema_down()`

```
/* Down or "P" operation on a semaphore.  Waits for SEMA's value
   to become positive and then atomically decrements it.
   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but if it sleeps then the next scheduled
   thread will probably turn interrupts back on. */
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;
```

```
  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_insert_ordered (&sema->waiters, &thread_current ()->elem,
thread_cmp_priority, NULL);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

信号量上的向下或“P”操作。等待SEMA的值变为正值，然后按原子顺序递减。此函数可能处于休眠状态，因此不能在中断处理程序中调用。此函数可以在中断被禁用的情况下调用，但如果它处于休眠状态，则下一个调度线程可能会重新打开中断。

因此，该函数的功能是通过信号量的操作阻塞指定进程，该进程被阻塞并占有资源。

那么，怎么将已经退出的子进程的资源释放呢?

```
/* Up or "V" operation on a semaphore.  Increments SEMA's value
   and wakes up one thread of those waiting for SEMA, if any.
   This function may be called from an interrupt handler. */
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  sema->value++;
  intr_set_level (old_level);
}
```

信号量上的Up或“V”操作。增加SEMA的值并唤醒等待SEMA的线程（如果有）。此函数可以从中断处理程序调用。

因此，在进程退出函数中，调用`sema_up()`函数释放进程占有的还没有释放的资源即可。

```
// thread.c

struct thread *cur = thread_current();
printf ("%s: exit(%d)\n",cur->name, cur->ret); /* 输出进程name以及进程return值 */
// 记录子进程退出状态
cur->thread_child->exit_status_child = cur->exit_status;
// 子进程退出，释放资源
sema_up (&cur->thread_child->sema);
```

最后我们在函数`sys_wait()`里调用`process_wait()`即可

```
#include "process.h"

void
sys_wait (struct intr_frame* f)
{
  uint32_t *ptr = f->esp;
  // ptr存储系统调用函数名称，ptr+1存储系统调用参数指针
  judge_pointer (ptr + 1);
  // 指针移动一位
  *ptr++;
  // 子进程的退出状态值赋值给eax寄存器
  f->eax = process_wait(*ptr);
}
```

文件系统调用

接下来就要进入到文件调用系统了，我们需要做一些准备工作

> You must synchronize system calls so that any number of user processes can make them at once. In particular, it is not safe to call into the file system code provided in the `filesys` directory from multiple threads at once. Your system call implementation must treat the file system code as a critical section. Don't forget that `process_execute()` also accesses files. For now, we recommend against modifying code in the `filesys` directory.
>
> 您必须同步系统调用，以便任意数量的用户进程可以同时进行调用。特别是，一次从多个线程调用 `filesys` 目录中提供的文件系统代码是不安全的。系统调用实现必须将文件系统代码视为关键部分。不要忘记， `process_execute()` 也会访问文件。目前，我们建议不要修改 `filesys` 目录中的代码。

因此，根据Pintos官方实验文档的提示，我们需要提供一个文件锁，来防止在操作指定文件时其他进程来修改当前文件，同时，我们也需要提供文件资源释放机制，防止文件资源迟迟被一个资源占有而不被释放

因此，为了实现这两个功能，我们需要修改 `thread.h` 文件中关于进程结构体的定义，增添一项 `files` ，表示当前进程所拥有的全部文件资源，类型为 `list` ，因为可以方便调用Pintos为我们准备好的 `list` 遍历方法

```
// 当前进程所拥有的全部文件资源
struct list files;
```

同时，定义一个文件结构体

```
struct thread_file
{
    int fd; // 文件描述符
    struct file* file;  // 文件
    struct list_elem file_elem; // 用于找到整体的list_elem，详细作用在wait()函数中已经说明过了
};
```

`thread_file` 结构体就是进程结构体中 `files` 存储的文件资源结构

同时完成对文件申请锁与释放锁函数的实现

```
// 定义文件锁
static struct lock lock_f;

// 请求文件锁函数
```

```
void
acquire_lock_f()
{
    lock_acquire(&lock_f);
}

// 释放文件锁函数
void
release_lock_f() {
    lock_release(&lock_f);
}
```

由于定义了两个新的结构成员，因此我们需要分别初始化 `files` 与 `lock_f`

由于文件锁结构体只用初始化一次，因此我们在 `thread_init()` 函数中初始化 `lock_f`

```
// thread_init()

// 文件锁初始化
lock_init(&lock_f);
```

由于每个进程在初始化时都需要初始化自己进程结构体的 `files` 成员，因此我们在 `init_thread()` 函数中初始化 `files`

```
// init_thread()

// 初始化进程拥有的文件资源列表
list_init (&t->files);
```

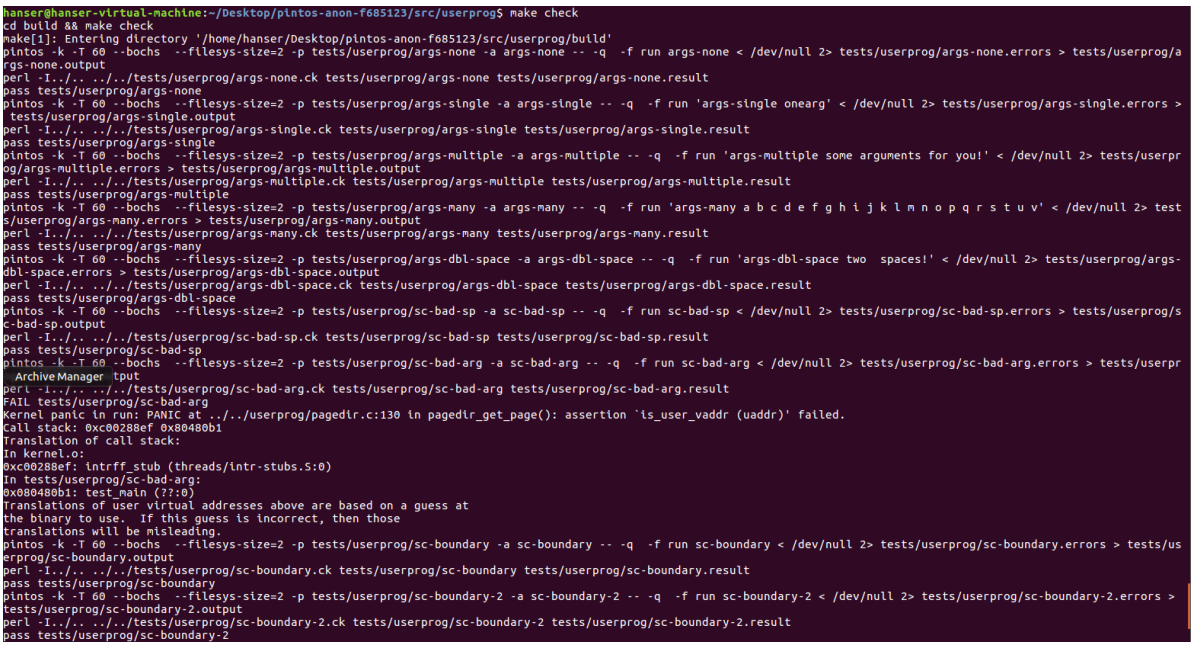
最后，我们在来实现当进程退出时，释放其所拥有的所有文件资源

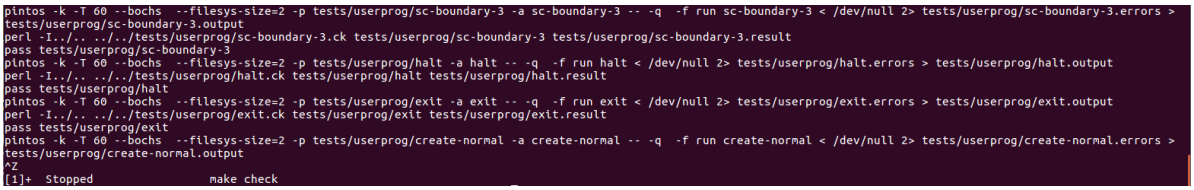
```
// thread_exit()

// 释放进程所拥有的所有文件资源
    struct list_elem *e;
    struct list *files = &thread_current()->files;
    while(!list_empty (files))
    {
        e = list_pop_front (files);
        struct thread_file *f = list_entry (e, struct thread_file, file_elem);
        // 申请锁
        acquire_lock_f ();
        // 关闭文件资源
        file_close (f->file);
        // 释放文件锁
        release_lock_f ();
        // 从文件列表中移除该文件资源
        list_remove (e);
        // 释放文件资源
        free (f);
    }
```

实现思路都写在注释里，使用的主要方法依然时上文提到的Pintos为我们提供的 `list` 遍历方法

write()

write()函数的具体实现方式还没有整理成笔记，但是写完write()测试点后，关于系统调用进程方面的测试点已经可以测试了，下面是相关部分测试点的过点情况





后面直接测试了`create()`函数，还没有写，所以测试点过不去，ok，下一个系统调用。

create()

> bool create (const char *file, unsigned initial_size)
>
> 创建一个名为`file`，`initially_size`字节的新文件。如果成功，则返回`true`，否则返回`false`。创建新文件不会打开它：**打开新文件是一个单独的操作，需要打开系统调用。**

创建一个文件，这么重要的函数Pintos有可能会帮我们写过，在文件`src/filesys/filesys.c`中，Pintos为我们写好了一个创建文件的函数`filesys_create()`

```
/* Creates a file named NAME with the given INITIAL_SIZE.
   Returns true if successful, false otherwise.
   Fails if a file named NAME already exists,
   or if internal memory allocation fails. */
bool
filesys_create (const char *name, off_t initial_size)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = dir_open_root ();
  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
```

```
                  && inode_create (inode_sector, initial_size)
                  && dir_add (dir, name, inode_sector));
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);

  return success;
}
```

我们要做的工作就是检验指针是否正确，然后调用该函数即可

注意，在调用`filesys_create()`函数时，参数指针ptr已经加过一了，因此，我们此时参数指针ptr指向的值就是参数的第二个参数，也就是要创建文件的文件名，使用`(cinst char *)*ptr`即可取地要创建文件的文件名，第三个参数是要创建文件的大小

```
void
sys_create(struct intr_frame* f)
{
    uint32_t *ptr = f->esp;
    // 检验指针是否正确
    judge_pointer(ptr + 5);
    judge_pointer(*(ptr + 4));
    *ptr++;
    // 申请锁
    acquire_lock_f ();
    // 注意*ptr此时已经指向了参数中的第二个参数
    f->eax = filesys_create ((const char *)*ptr, *(ptr+1));
    // 释放锁
    release_lock_f ();
}
```

`remove()`

> bool remove (const char *file)
>
> 删除名为`file`的文件。如果成功，则返回`true`，否则返回`false`。无论文件是打开的还是关闭的，都可以将其删除，删除打开的文件不会将其关闭。有关详细信息，请参见删除打开的文件。

删除文件这么重要的函数Pintos也很有可能帮我们写好了，在`src/filesys/filesys.c`文件中，Pintos帮我们定义了删除文件函数

```
/* Deletes the file named NAME.
   Returns true if successful, false on failure.
   Fails if no file named NAME exists,
   or if an internal memory allocation fails. */
bool
filesys_remove (const char *name)
{
  struct dir *dir = dir_open_root ();
  bool success = dir != NULL && dir_remove (dir, name);
  dir_close (dir);

  return success;
}
```

因此，类似于`create()`函数，我们检验指针正确性，并调用`filesys_remove()`函数即可

```
void
sys_remove(struct intr_frame* f)
{
  uint32_t *ptr = f->esp;
  // ptr：系统调用函数名
  judge_pointer (ptr);
  // ptr + 1：指针，系统调用第二个参数，指向系统调用第二个参数存储的地址
  judge_pointer (ptr + 1);
  // *(ptr + 1)：系统调用第二个参数，要删除的文件名
  judge_pointer (*(ptr + 1));
  // 不要忘记指针向后移一位
  *ptr++;
  // 申请锁
  acquire_lock_f ();
  // 调用filesys_remove()函数，注意将函数返回值放入寄存器eax中
  f->eax = filesys_remove ((const char *)*ptr);
  // 释放锁
  release_lock_f ();
}
```

open()

> int open (const char *file)
>
> 打开名为 `file` 的文件。返回称为“文件描述符”（fd）的非负整数句柄，**如果无法打开文件，则返回-1**。
> 编号为0和1的文件描述符为控制台保留：**fd 0（标准输入文件号）为标准输入，fd 1（标准输出文件号）为标准输出**。开放系统调用永远不会返回这些文件描述符中的任何一个，它们仅作为系统调用参数有效，如下所述。
>
> **每个进程都有一组独立的文件描述符。子进程不会继承文件描述符。**
>
> **当单个文件被多次打开时，无论是由单个进程还是不同的进程打开，每次打开都会返回一个新的文件描述符。**单个文件的不同文件描述符在单独的关闭调用中独立关闭，并且它们不共享文件位置。

由于每个进程的文件描述符都是相互独立的，因此，我们需要在进程结构体中加入一项进程当前所拥有的全部文件描述符，又因为文件描述符都是从小到大排列，因此，我们只需要记录文件描述符中的 最大项即可

```
// thread.h

int all_fd;       // 最大文件描述符
```

有了这个结构体成员，我们同样也需要在 `init_thread()` 函数中初始化，只用初始化一次，因为出去标准输入0，标准输出1，标准错误-1，所有进程的文件描述都从2开始，因此，我们只需要在创建进程的时候将 `all_fd` 初始化为2即可

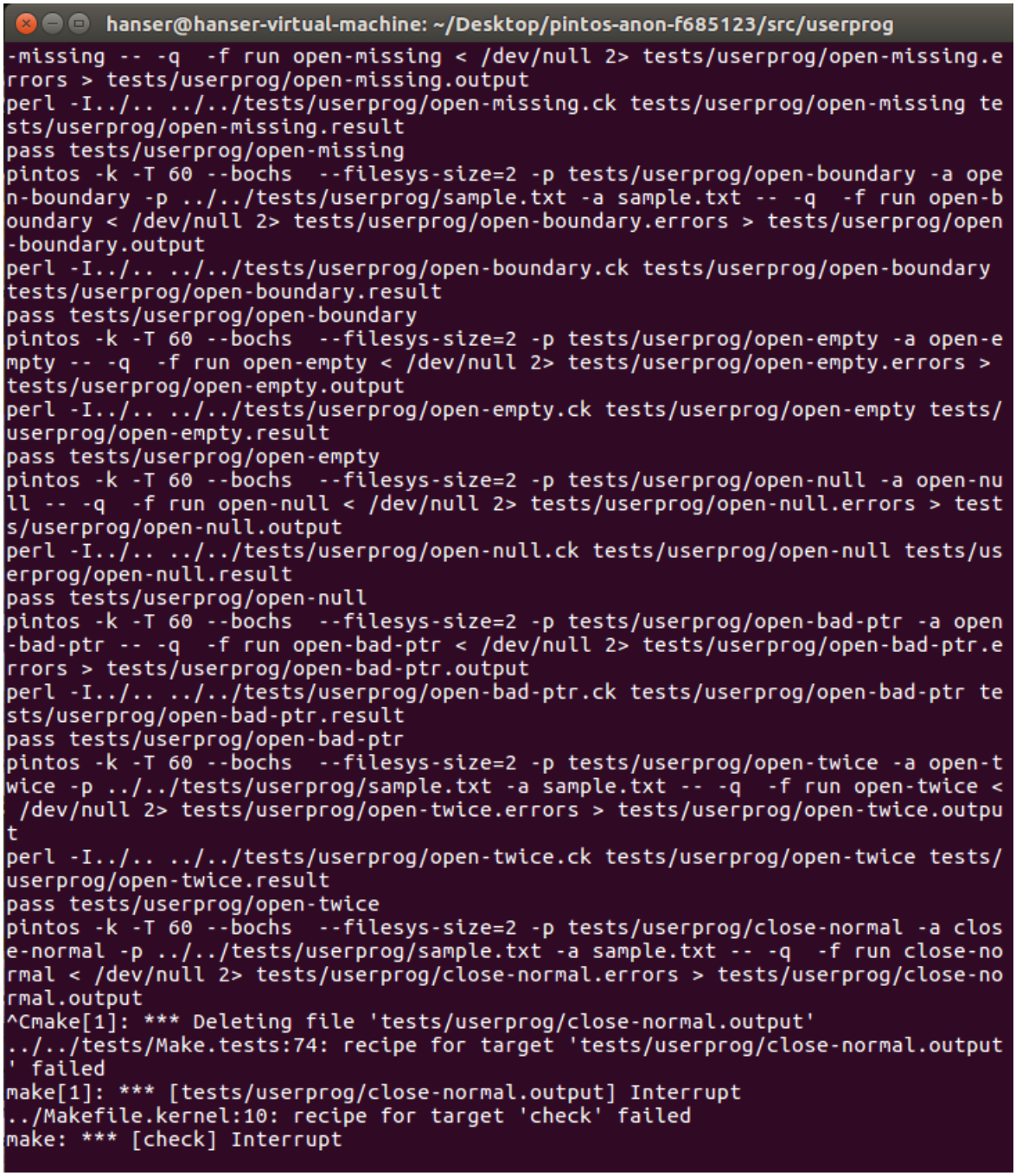
```
// init_thread()

// 初始化所有进程最大文件描述符all_fd = 2
t -> all_fd = 2;
```

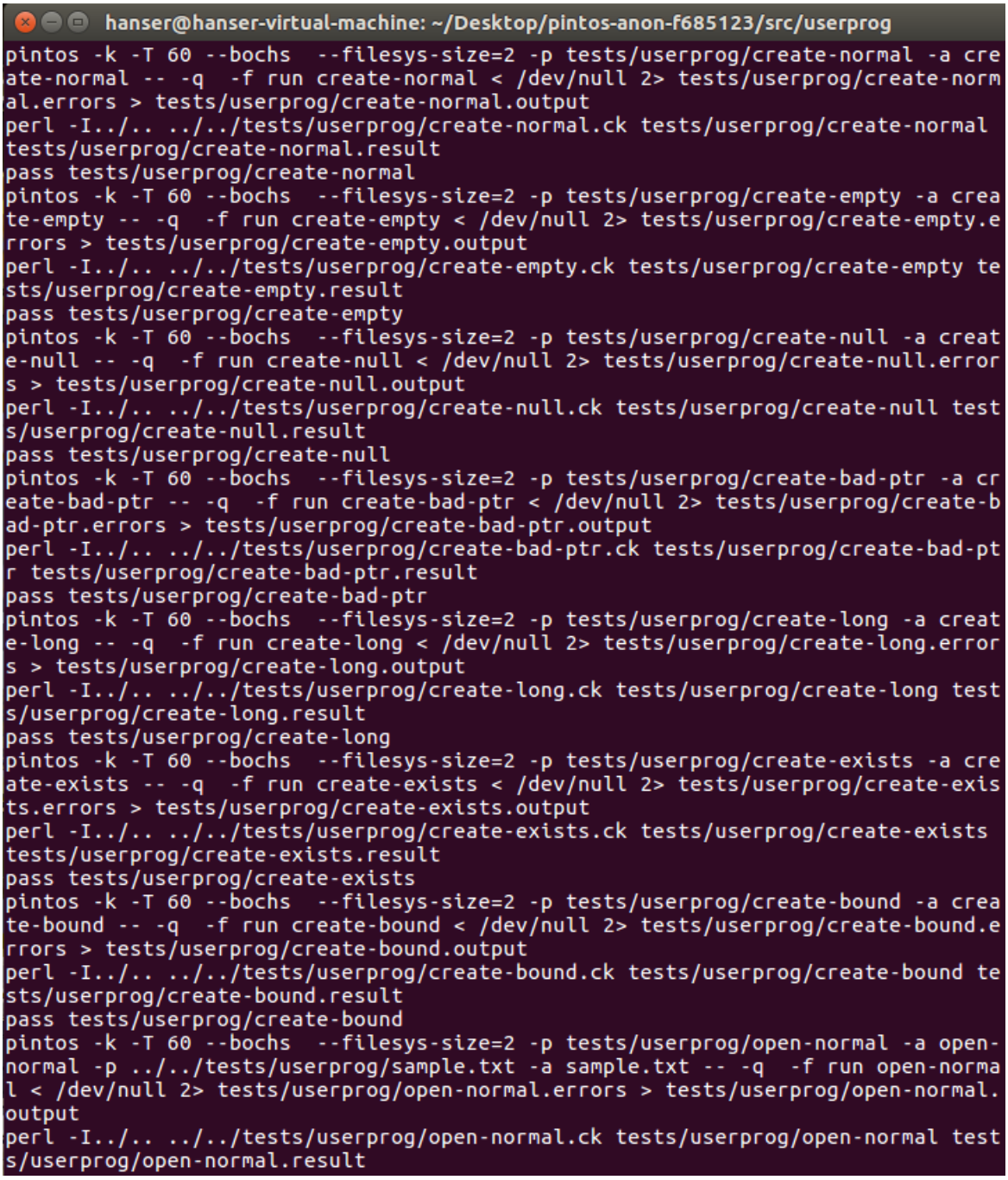
好了，准备工作完成了，接下来我们完成系统调用 `sys_open()` 函数

同样，Pintos为我们准备了文件打开函数 `filesys_open()` ，我们直接调用该函数即可，这里就不再详细说明该函数的实现过程

```
void
sys_open (struct intr_frame* f)
{
  uint32_t *ptr = f->esp;
  // 检查指针正确性
  // ptr + 1指针指向系统调用第二个参数：要打开文件的文件名存放地址
  judge_pointer (ptr + 1);
  // *(ptr + 1)为要打开文件的文件名
  judge_pointer (*(ptr + 1));
  // 不要忘记指针向下移动一位
  *ptr++;
  // 申请锁
  acquire_lock_f ();
  // 调用filesys_open()函数
  struct file * file_a = filesys_open((const char *)*ptr);
  // 释放锁
  release_lock_f ();
  // 准备为当前进程添加刚刚打开的文件资源，先定义当前进程结构体
  struct thread * t = thread_current();
  if (file_a)
  {
    // 先定义进程文件结构体，并为其申请存储空间
    struct thread_file *file_b = malloc(sizeof(struct thread_file));
    // 赋值文件fd，利用了fd从小到大的特点
    file_b->fd = t->all_fd++;
    // 添加文件资源
    file_b->file = file_a;
    // 将复制好的进程文件结构体放入当前进程的files列表中
    list_push_back (&t->files, &file_b->file_elem);//维护files列表
    // 根据题意，函数返回打开文件的文件描述符，因此将打开文件的文件描述符fd放到寄存器eax中
    f->eax = file_b->fd;
  }
  else
  {
    // 文件无法打开，返回-1
    f->eax = -1;
  }
}
```

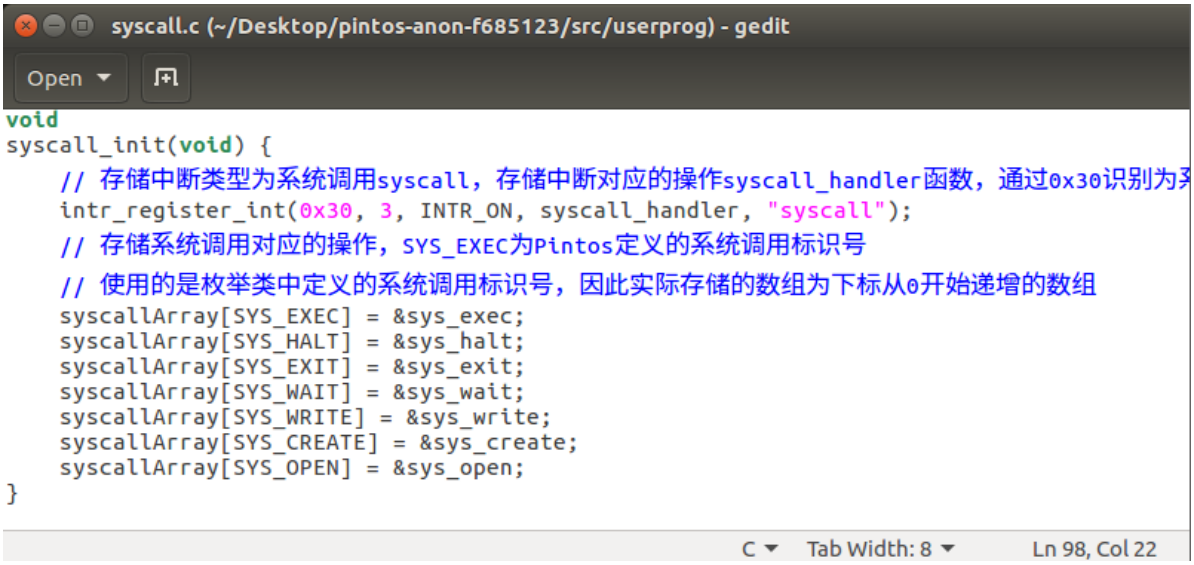
又写了三个有关于文件的系统调用函数，我们来测试一下相关测试点


hanser@hanser-virtual-machine: ~/Desktop/pintos-anon-f685123/src/userprog
-missing -- -q -f run open-missing < /dev/null 2> tests/userprog/open-missing.errors > tests/userprog/open-missing.output
perl -I../.. ../../tests/userprog/open-missing.ck tests/userprog/open-missing tests/userprog/open-missing.result
pass tests/userprog/open-missing
pintos -k -T 60 --bochs --filesys-size=2 -p tests/userprog/open-boundary -a open-boundary -p ../../tests/userprog/sample.txt -a sample.txt -- -q -f run open-boundary < /dev/null 2> tests/userprog/open-boundary.errors > tests/userprog/open-boundary.output
perl -I../.. ../../tests/userprog/open-boundary.ck tests/userprog/open-boundary tests/userprog/open-boundary.result
pass tests/userprog/open-boundary
pintos -k -T 60 --bochs --filesys-size=2 -p tests/userprog/open-empty -a open-empty -- -q -f run open-empty < /dev/null 2> tests/userprog/open-empty.errors > tests/userprog/open-empty.output
perl -I../.. ../../tests/userprog/open-empty.ck tests/userprog/open-empty tests/userprog/open-empty.result
pass tests/userprog/open-empty
pintos -k -T 60 --bochs --filesys-size=2 -p tests/userprog/open-null -a open-null -- -q -f run open-null < /dev/null 2> tests/userprog/open-null.errors > tests/userprog/open-null.output
perl -I../.. ../../tests/userprog/open-null.ck tests/userprog/open-null tests/userprog/open-null.result
pass tests/userprog/open-null
pintos -k -T 60 --bochs --filesys-size=2 -p tests/userprog/open-bad-ptr -a open-bad-ptr -- -q -f run open-bad-ptr < /dev/null 2> tests/userprog/open-bad-ptr.errors > tests/userprog/open-bad-ptr.output
perl -I../.. ../../tests/userprog/open-bad-ptr.ck tests/userprog/open-bad-ptr tests/userprog/open-bad-ptr.result
pass tests/userprog/open-bad-ptr
pintos -k -T 60 --bochs --filesys-size=2 -p tests/userprog/open-twice -a open-twice -p ../../tests/userprog/sample.txt -a sample.txt -- -q -f run open-twice < /dev/null 2> tests/userprog/open-twice.errors > tests/userprog/open-twice.output
perl -I../.. ../../tests/userprog/open-twice.ck tests/userprog/open-twice tests/userprog/open-twice.result
pass tests/userprog/open-twice
pintos -k -T 60 --bochs --filesys-size=2 -p tests/userprog/close-normal -a close-normal -p ../../tests/userprog/sample.txt -a sample.txt -- -q -f run close-normal < /dev/null 2> tests/userprog/close-normal.errors > tests/userprog/close-normal.output
^Cmake[1]: *** Deleting file 'tests/userprog/close-normal.output'
../../tests/Make.tests:74: recipe for target 'tests/userprog/close-normal.output' failed
make[1]: *** [tests/userprog/close-normal.output] Interrupt
../Makefile.kernel:10: recipe for target 'check' failed
make: *** [check] Interrupt



关于`create()`函数和`open()`函数通过了对应测试点，`remove()`函数由于还没有运行到对应的测试点Pintos就因为缺失对应的系统调用而超时了，所以现在无法测试

注意，先将系统调用初始化函数添加对应的系统调用函数



filesize()

> int filesize (int fd)
>
> 返回作为fd打开的文件的大小（以字节为单位）。

我们再次查看相关函数，发现Pintos为我们定义了返回文件字节大小的函数 `file_length()`

```
// src/filesys/file.c

/* Returns the size of FILE in bytes. */
off_t
file_length (struct file *file)
{
  ASSERT (file != NULL);
  return inode_length (file->inode);
}
```

我们调用在 `write()` 系统调用时定义的通过文件fd找文件的函数和 `file_length()` 函数即可

```
void
sys_filesize (struct intr_frame* f){
  uint32_t *ptr = f->esp;
  // ptr + 1:fd
  judge_pointer (ptr + 1);
  *ptr++;
  // 文件fd对应的文件结构体
  struct thread_file * file_a = find_file_id (*ptr);
  if (file_a)
  {
    // 申请锁
    acquire_lock_f ();
    // 返回对应的文件长度
    f->eax = file_length (file_a->file);
    // 释放锁
    release_lock_f ();
  }
  else
  {
    // 无法打开对应文件，返回-1
    f->eax = -1;
```

```
  }
}
```

有没有发现我们写的这几个关于文件的系统调用函数都长的有点像呢?

`read()`

> int read (int fd, void *buffer, unsigned size)
>
> 从作为fd打开的文件中读取大小字节到缓冲区中。**返回实际读取的字节数（文件末尾为0）**，如果无法读取文件（由于文件结尾以外的条件），则返回-1。 `Fd` 0使用 `input_getc()` 从键盘读取数据。

Pintos提示我们当fd = 0时使用 `input_getc()` 函数从键盘读取数据，如果fd != 0呢?

我们可以使用Pintos在 `src/filesys/file.c` 文件中为我们定义的函数 `file_read()`

```
/* Reads SIZE bytes from FILE into BUFFER,
   starting at the file's current position.
   Returns the number of bytes actually read,
   which may be less than SIZE if end of file is reached.
   Advances FILE's position by the number of bytes read. */
off_t
file_read (struct file *file, void *buffer, off_t size)
{
  off_t bytes_read = inode_read_at (file->inode, buffer, size, file->pos);
  file->pos += bytes_read;
  return bytes_read;
}
```

检查指针正确性，当fd = 0时调用 `input_getc()` 函数，当fd != 0时调用 `file_read()` 函数即可，注意函数返回值是实际读到的字节数

```
void
sys_read (struct intr_frame* f)
{
  /*
   * ptr：系统调用函数名，
   * ptr + 1：想要读取的文件fd，
   * ptr + 2：想要读入的数组地址，
   * ptr + 3：想要读入的文件字节数。
   */
  uint32_t *ptr = f->esp;
  judge_pointer (ptr);
  judge_pointer (ptr + 1);
  judge_pointer (ptr + 2);
  *ptr++;
  int fd = *ptr;
  uint8_t * buffer = (uint8_t*)*(ptr+1);
  off_t len = *(ptr+2);
  // fd = 0，标准输入，调用input_getc()函数
  if (fd == 0)
  {
    for (int i = 0; i < len; i++)
      buffer[i] = input_getc();
    // 返回实际读到的字节数
    f->eax = len;
  }
```

```
  // fd != 0，从文件读入
  else
  {
    struct thread_file * file_a = find_file_id (*ptr);
    if (file_a)
    {
      // 申请锁
      acquire_lock_f ();
      // 调用filesys_read()函数，函数返回值作为系统调用返回值放到寄存器eax中
      f->eax = file_read (file_a->file, buffer, len);
      // 释放锁
      release_lock_f ();
    }
    else
    {
      // 无法打开文件，返回-1
      f->eax = -1;
    }
  }
}
```

seek()

> void seek (int fd, unsigned position)
>
> **将打开文件fd中要读取或写入的下一个字节更改为position**，以文件开头的字节表示。（因此，位置0是文件的起点）
>
> **超过文件当前结尾的搜索不是错误**。稍后的读取获得0字节，表示文件结束。稍后的写入扩展文件，用零填充任何未写入的间隙。（然而，在Pintos中，在项目4完成之前，文件的长度是固定的，因此写入文件末尾将返回一个错误）**这些语义在文件系统中实现，在系统调用实现中不需要任何特殊的工作**。

Pintos实验文档写了那么多，其实总结起来就是：将打开文件fd中要读取或写入的下一个字节更改为`position`

但是，更改文件读取或写入字节位置怎么实现呢？别忘了`src/filesys/file.c`文件，里面正好有我们需要的更改文件读取或写入的下一个字节位置函数`file_seek()`

```
/* Sets the current position in FILE to NEW_POS bytes from the
   start of the file. */
void
file_seek (struct file *file, off_t new_pos)
{
  ASSERT (file != NULL);
  ASSERT (new_pos >= 0);
  file->pos = new_pos;
}
```

好像也就是将文件结构体的`pos`改为`new_pos`

好了，准备工作完成了，按照上面的常规步骤，实现`sys_seek()`函数

```
void
sys_seek(struct intr_frame* f)
{
  uint32_t *ptr = f->esp;
```

```
  judge_pointer (ptr + 5);
  // *(ptr + 1): fd
  *ptr++;
  // 按照fd找到文件结构体，并将文件结构体存储到文件结构体file_a中
  struct thread_file *file_a = find_file_id (*ptr);
  if (file_a)
  {
    acquire_lock_f ();
    file_seek (file_a->file, *(ptr+1));
    release_lock_f ();
  } else {
      // 文件打开错误
      f_eax = -1;
  }
}
```

Pintos实验文档中没有说 `sys_seek()` 函数如果无法打开对应文件应该如何处理，可能没有对应测试点，为了严谨，我们将无法打开文件对应返回值设置为和其他系统调用相同返回值-1

`tell()`

> unsigned tell (int fd)
>
> 返回打开文件 `fd` 中要读取或写入的下一个字节的位置，**以从文件开头开始的字节数表示。**

Pintos为我们提供了返回文件 `fd` 中要读取或写入的下一个字节的位置函数 `file_tell()`

```
// src/filesys/file,c

/* Returns the current position in FILE as a byte offset from the
   start of the file. */
off_t
file_tell (struct file *file)
{
  ASSERT (file != NULL);
  return file->pos;
}
```

按照前面系统调用函数的步骤，检查指针正确性，找出文件 `fd` 对应的文件结构体，调用 `file_tell()` 函数

```
void
sys_tell (struct intr_frame* f)
{
  uint32_t *ptr = f->esp;
  judge_pointer (ptr + 1);
  *ptr++;
  struct thread_file *thread_a = find_file_id (*ptr);
  if (thread_a != null)
  {
    acquire_lock_f ();
    f->eax = file_tell (thread_a->file);
    release_lock_f ();
  }else{
    f->eax = -1;
  }
}
```

close()

> void close (int fd)
>
> 关闭文件描述符 `fd` 。**退出或终止进程会隐式关闭其所有打开的文件描述符，就像为每个描述符调用此函数一样。**

Pintos为我们定义了关闭文件函数 `file_close()`

```
/* Closes FILE. */
void
file_close (struct file *file)
{
  if (file != NULL)
    {
      file_allow_write (file);
      inode_close (file->inode);
      free (file);
    }
}
```

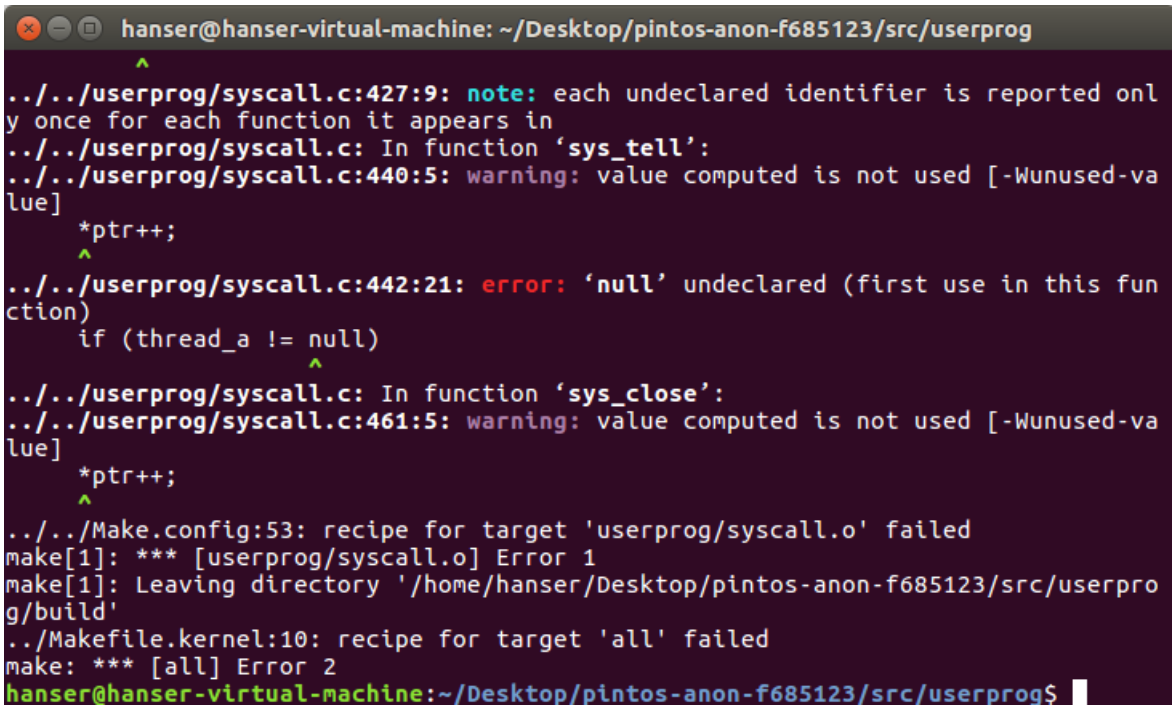
按照常规文件系统调用步骤来，检查指针正确性，按照文件 `fd` 找到对应文件结构体，调用 `file_close()` 函数

```
void
sys_close (struct intr_frame* f)
{
  uint32_t *ptr = f->esp;
  judge_pointer (ptr + 1);
  *ptr++;
  // 按照fd找到文件结构体，并将文件结构体存储到文件结构体file_a中
  struct thread_file * file_a = find_file_id (*ptr);
  if (file_a)
  {
    acquire_lock_f ();
    file_close (file_a->file);
    release_lock_f ();
    // Pintos提供的list方法，从主结构体中移除对应列表中元素
    list_remove (&file_a->file_elem);
    // 释放文件资源
    free (file_a);
  }
}
```

到现在为止，所有的系统调用已经写完了，将系统调用初始化函数补全，检查测试点

```
// syscall_init()

syscallArray[SYS_FILESIZE] = &sys_filesize;
syscallArray[SYS_READ] = &sys_read;
syscallArray[SYS_TELL] = &sys_tell;
syscallArray[SYS_SEEK] = &sys_seek;
syscallArray[SYS_REMOVE] = &sys_remove;
syscallArray[SYS_CLOSE] = &sys_close;
```

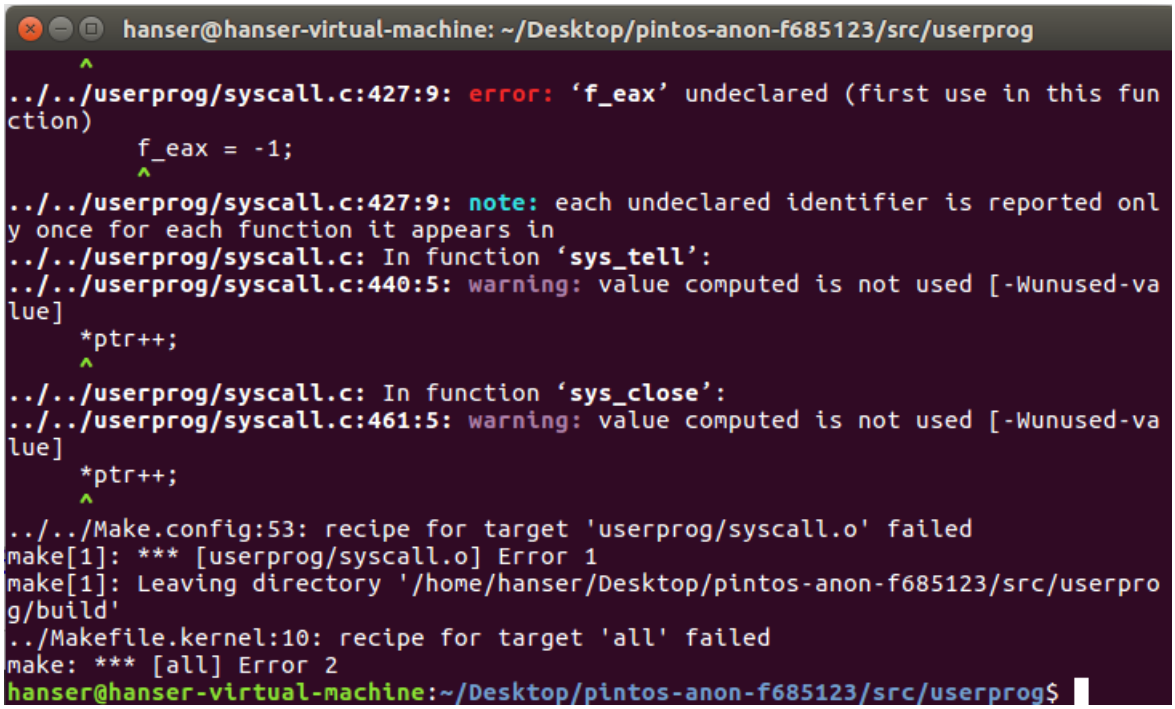
啊哦，出错了



原来Pintos中没有定义`null`，我们去掉`!= null`直接判断

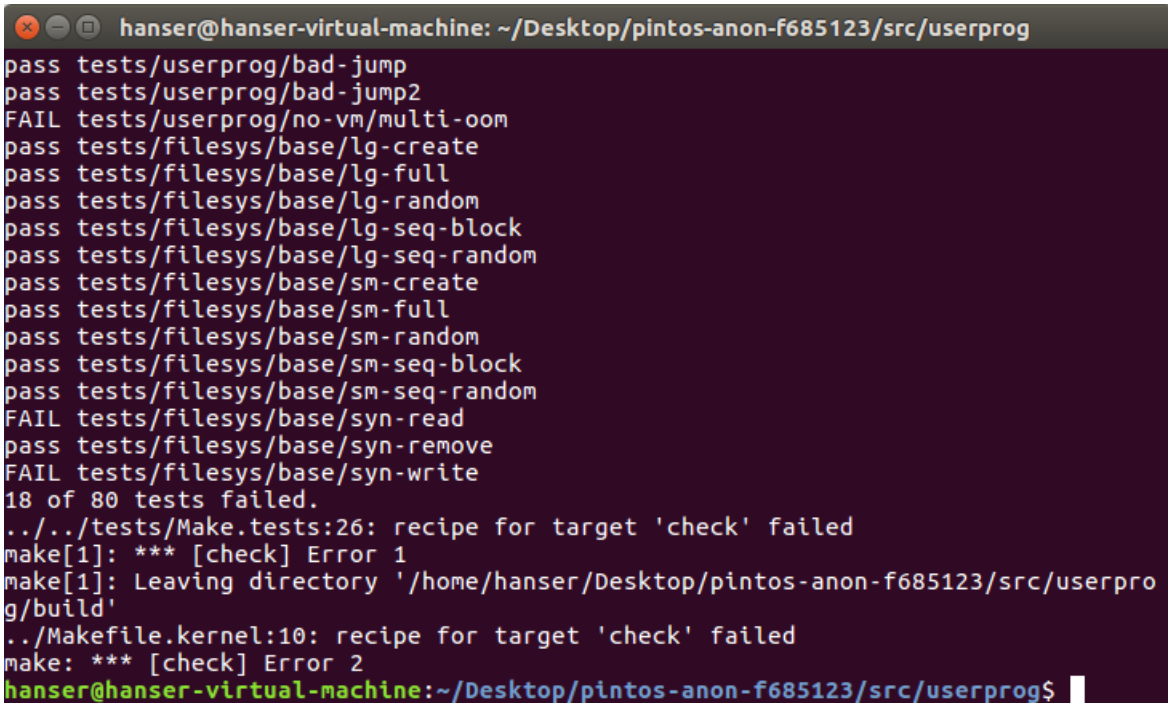
```
/*
 * 系统调用：tell()
 * 返回打开文件fd中要读取或写入的下一个字节的位置，以从文件开头开始的字节数表示。
 */
void
sys_tell (struct intr_frame* f)
{
    uint32_t *ptr = f->esp;
    judge_pointer (ptr + 1);
    *ptr++;
    struct thread_file *file_a = find_file_id (*ptr);
    if (file_a)
    {
        acquire_lock_f ();
        f->eax = file_tell (file_a->file);
        release_lock_f ();
    }else{
        f->eax = -1;
    }
}
```

又出错了



`->` 符号写成 `_` 符号，修改后再次运行Pintos

坏了



好吧，下面开始debug

debug

参数传递

7	sc-bad-arg	**将 `esp` 指向了栈顶下 4 字节（刚好放进了 `exit` 的系统调用号），试图在获取系统调用的参数时访问非法的内存区域，正常情况下应该以 exit(-1) 退出。**

```
pintos -k -T 60 --bochs  --filesys-size=2 -p tests/userprog/sc-bad-arg -a sc-bad
-arg -- -q  -f run sc-bad-arg < /dev/null 2> tests/userprog/sc-bad-arg.errors >
tests/userprog/sc-bad-arg.output
perl -I../.. ../../tests/userprog/sc-bad-arg.ck tests/userprog/sc-bad-arg tests/
userprog/sc-bad-arg.result
FAIL tests/userprog/sc-bad-arg
Kernel panic in run: PANIC at ../../userprog/pagedir.c:130 in pagedir_get_page()
: assertion `is_user_vaddr (uaddr)' failed.
Call stack: 0xc00288ff 0x80480b1
Translation of call stack:
In kernel.o:
0xc00288ff: intrff_stub (threads/intr-stubs.S:0)
In tests/userprog/sc-bad-arg:
0x080480b1: test_main (??:0)
Translations of user virtual addresses above are based on a guess at
the binary to use.  If this guess is incorrect, then those
translations will be misleading.
```

见`multi`测试点

`sys_exec()`

```
FAIL tests/userprog/exec-once
Test output failed to match any acceptable form.

Acceptable output:
  (exec-once) begin
  (child-simple) run
  child-simple: exit(81)
  (exec-once) end
  exec-once: exit(0)
Differences in `diff -u' format:
  (exec-once) begin
- (child-simple) run
- child-simple: exit(81)
- (exec-once) end
- exec-once: exit(0)
+ exec-once: exit(-1)
```

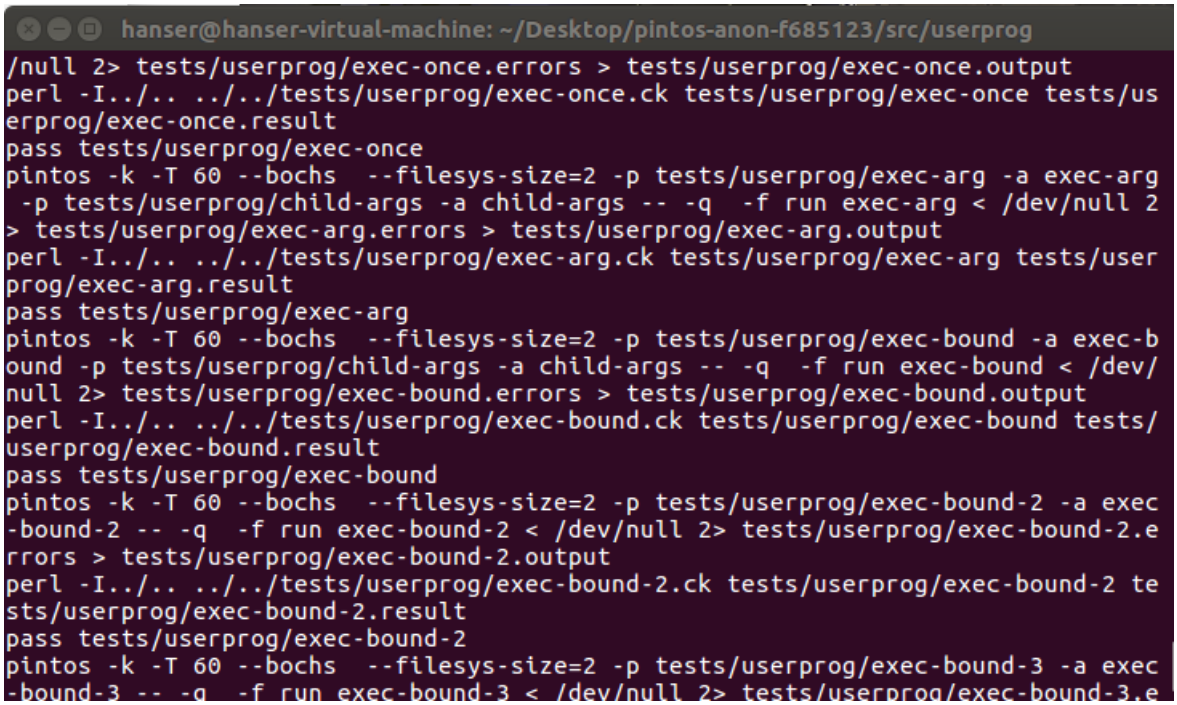
看来是没有输出直接判断指针错误退出了

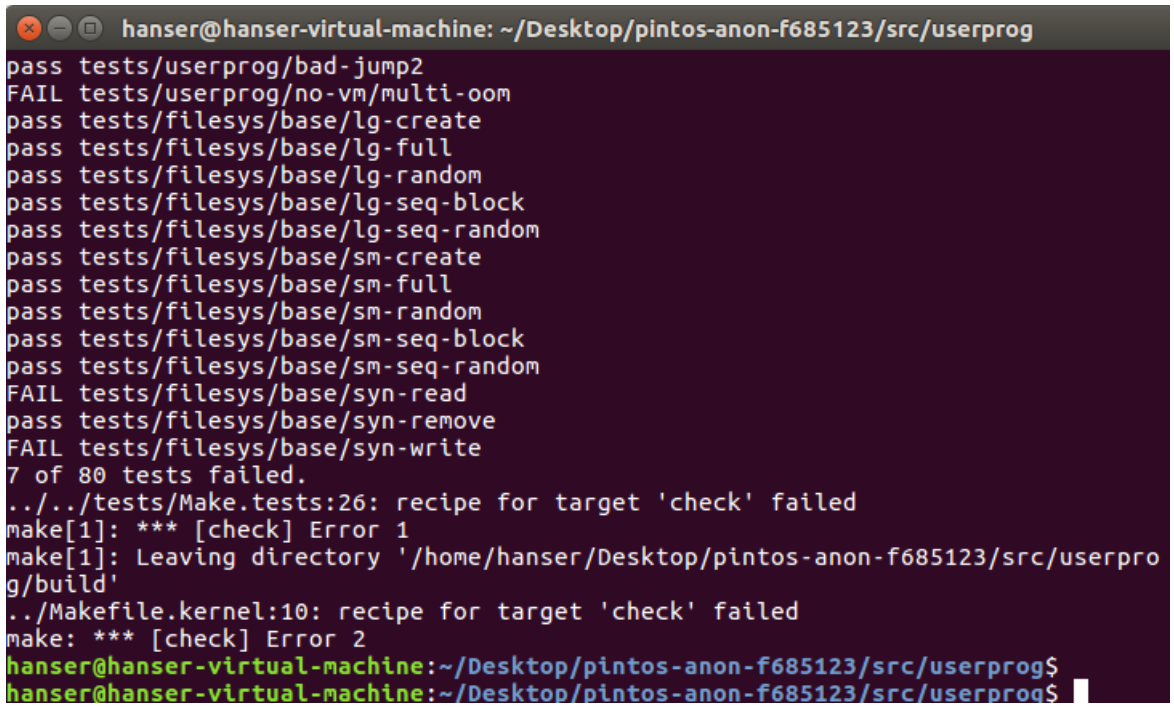
检查源代码

```
void
sys_exec (struct intr_frame* f)
{
    uint32_t *ptr = f->esp;
    // ptr中存放的是系统调用函数名称，ptr+1中存放的是系统调用函数参数*cmd_line
    // 也可以用f->esp + 4来表示ptr指向下一位
    judge_pointer ( vaddr: ptr + 1);
    // 重点：参数需要检查，但是参数本身又是一个指针，指向了存放调用函数需要的参数地址，我们也需要检查这个指针指向的地址是否正确
    judge_pointer (*(ptr++));
    // 调用函数process_execute()，函数返回值是进程pid，函数返回值存放在寄存器eax中
    f->eax = process_execute((char*)* ptr);
}
```

原来是写成了`*(ptr++)`，导致传入的参数还是`ptr`，但是`ptr`指向的并不是指针，所以自然就报错了。将`*(ptr)`修改为`*(++ptr)`

再次运行Pintos，关于`sys_exec()`测试点通过





`rox`

序号	名称	测试点
54	rox-simple	尝试改写自己的可执行文件（`write` 应该要返回 0）。
55	rox-child	父进程先写好子进程的可执行文件，然后执行子进程。接下来子进程尝试改写自己的可执行文件（`write` 应该要返回 0），之后会退出。然后父进程再次改写子进程的可执行文件（应该要成功）。
56	rox-multichild	父进程先写好子进程的可执行文件，然后递归地创建 5 个子进程且他们都试图改写自己的可执行文件；然后递归地退出，退出前也试图改写自己的可执行文件；最后一次父进程会再次改写子进程的可执行文件（这次应该要成功）。

这三个测试点没有通过

可执行文件在 Pintos 中的定义为用来创建进程的文件，即创建进程时打开的那一个文件。`filesys/file.c` 文件中定义了函数 `file_deny_write()`，该函数可以禁止对文件的写操作。我们需要在 `load` 时，禁止对该文件的写操作，在退出时回复。

同时这一块需要实现 `seek` 调用。需要用到 `file_seek()` 函数。

这一块很难debug，根据助教老师的实验说明文档和网上的资料，我慢慢发现这一部分出bug的原因是因为在加载文件的过程中也有可能将文件修改，因此，我们需要在加载文件函数 `load()` 加入文件锁操作函数，并且在加载函数中将对应的文件加入到文件队列中，这样就可以防止在文件加载过程中出现文件被修改的情况

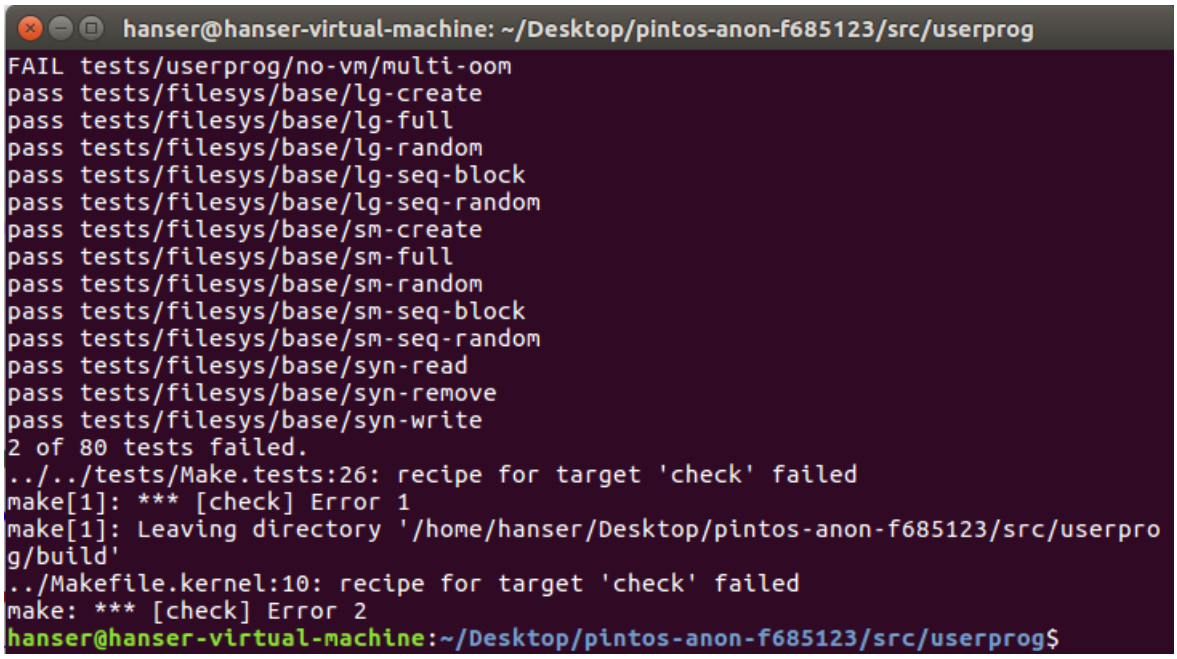
```
// load.c

bool
load (const char *file_name, void (**eip) (void), void **esp)
{
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();
  // 申请锁
  acquire_lock_f ();
  /* Open executable file. */
  file = filesys_open (file_name);

  if (file == NULL)
    {
      printf ("load: %s: open failed\n", file_name);
      goto done;
  }

  // 修改部分
  struct thread_file *thread_file_temp = malloc(sizeof(struct thread_file));
  thread_file_temp->file = file;
  // 加入队列
  list_push_back (&thread_current()->files, &thread_file_temp->file_elem);
  // 加载过程禁止写入
  file_deny_write(file);
  /* Read and verify executable header. */
  if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
      || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
      || ehdr.e_type != 2
      || ehdr.e_machine != 3
      || ehdr.e_version != 1
      || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
      || ehdr.e_phnum > 1024)
    {
      printf ("load: %s: error loading executable\n", file_name);
      goto done;
```

```
    }

  /* Read program headers. */
  file_ofs = ehdr.e_phoff;
  for (i = 0; i < ehdr.e_phnum; i++)
    {
      struct Elf32_Phdr phdr;

      if (file_ofs < 0 || file_ofs > file_length (file))
        goto done;
      file_seek (file, file_ofs);

      if (file_read (file, &phdr, sizeof phdr) != sizeof phdr)
        goto done;
      file_ofs += sizeof phdr;
      switch (phdr.p_type)
        {
        case PT_NULL:
        case PT_NOTE:
        case PT_PHDR:
        case PT_STACK:
        default:
          /* Ignore this segment. */
          break;
        case PT_DYNAMIC:
        case PT_INTERP:
        case PT_SHLIB:
          goto done;
        case PT_LOAD:
          if (validate_segment (&phdr, file))
            {
              bool writable = (phdr.p_flags & PF_W) != 0;
              uint32_t file_page = phdr.p_offset & ~PGMASK;
              uint32_t mem_page = phdr.p_vaddr & ~PGMASK;
              uint32_t page_offset = phdr.p_vaddr & PGMASK;
              uint32_t read_bytes, zero_bytes;
              if (phdr.p_filesz > 0)
                {
                  /* Normal segment.
                     Read initial part from disk and zero the rest. */
                  read_bytes = page_offset + phdr.p_filesz;
                  zero_bytes = (ROUND_UP (page_offset + phdr.p_memsz, PGSIZE)
                                - read_bytes);
                }
              else
                {
                  /* Entirely zero.
                     Don't read anything from disk. */
                  read_bytes = 0;
                  zero_bytes = ROUND_UP (page_offset + phdr.p_memsz, PGSIZE);
                }
              if (!load_segment (file, file_page, (void *) mem_page,
                                 read_bytes, zero_bytes, writable))
                goto done;
            }
          else
            goto done;
          break;
```

```
        }
    }

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

  /* Start address. */
  *eip = (void (*) (void)) ehdr.e_entry;

  success = true;

 done:
  /* We arrive here whether the load is successful or not. */
  // 释放锁
  release_lock_f();
  return success;
}
```

这一部分真的找了好久资料

好了，修改完后我们再次运行Pintos



我们发现，错误点从7个变成了两个，最后一个测试点和倒数第三个测试点也通过了，可能这两个测试点与`load()`函数修改有关

multi

```
pintos -k -T 360 --bochs  --filesys-size=2 -p tests/userprog/no-vm/multi-oom -a
multi-oom -- -q  -f run multi-oom < /dev/null 2> tests/userprog/no-vm/multi-oom.
errors > tests/userprog/no-vm/multi-oom.output
perl -I../.. ../../tests/userprog/no-vm/multi-oom.ck tests/userprog/no-vm/multi-
oom tests/userprog/no-vm/multi-oom.result
FAIL tests/userprog/no-vm/multi-oom
Kernel panic in run: PANIC at ../../userprog/pagedir.c:130 in pagedir_get_page()
: assertion `is_user_vaddr (uaddr)' failed.
Call stack: 0xc00288ff 0x804aaf4
Translation of call stack:
In kernel.o:
0xc00288ff: intrff_stub (threads/intr-stubs.S:0)
In tests/userprog/no-vm/multi-oom:
0x0804aaf4: open (??:0)
Translations of user virtual addresses above are based on a guess at
the binary to use.  If this guess is incorrect, then those
translations will be misleading.
```

资源释放

序号	名称	测试点
63	multi-oom	全实验最难的一个点，但也有可能是最简单的一个点。

该部分旨在探究同学们编码过程中，对于资源的合理理由，务必要做到每一个资源申请后，都会有释放。包括前面提到的文件的退出，消除开辟的内存空间，同时注意到进程退出时，关闭所有打开的文件，以及 `open`、`close`、`create`、`remove`、`execute`、`wait` 等所有过程中开辟的空间都需要关闭。

看起来他的报错信息是因为

```
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;
}
```

断言判断错误，而 `is_use_vddr(uaddr)` 函数其实很简单

```
is_user_vaddr (const void *vaddr)
{
  return vaddr < PHYS_BASE;
}
```

而结合第一个报错点 `sc-bad-args`

```
pintos -k -T 60 --bochs  --filesys-size=2 -p tests/userprog/sc-bad-arg -a sc-bad
-arg -- -q  -f run sc-bad-arg < /dev/null 2> tests/userprog/sc-bad-arg.errors >
tests/userprog/sc-bad-arg.output
perl -I../.. ../../tests/userprog/sc-bad-arg.ck tests/userprog/sc-bad-arg tests/
userprog/sc-bad-arg.result
FAIL tests/userprog/sc-bad-arg
Kernel panic in run: PANIC at ../../userprog/pagedir.c:130 in pagedir_get_page()
: assertion `is_user_vaddr (uaddr)' failed.
Call stack: 0xc00288ff 0x80480b1
Translation of call stack:
In kernel.o:
0xc00288ff: intrff_stub (threads/intr-stubs.S:0)
In tests/userprog/sc-bad-arg:
0x080480b1: test_main (??:0)
Translations of user virtual addresses above are based on a guess at
the binary to use.  If this guess is incorrect, then those
translations will be misleading.
```

两者的报错信息几乎一样，因此我们合理推测 `multi-oom` 报错点是因为 `sc-bad-args` 报错点导致的，见参数传递

既然报错信息是在函数 `pagedir_get_page()` 中的断言中，调用 `is_user_vaddr(uaddr)` 函数报错，那么我们就在调用 `pagedir_get_page()` 函数前就使用 `is_user_vaddr(uaddr)` 函数判断，如果判断结果为 `false`，则直接以 `-1` 退出即可。

修改 `judge_pointer()` 函数

```c
// 判断指针指向内存的合理性
void *
judge_pointer(const void *vaddr) {
    if(!is_user_vaddr(vaddr)) {
        // 设置线程状态值为-1，表示线程错误
        thread_current()->exit_status = -1;
        // 线程退出
        thread_exit();
    }
    void *ptr = pagedir_get_page(thread_current()->pagedir, vaddr);
    /* 判断是否属于用户地址空间，空间是否已经被映射（有效性） */
    if (!is_user_vaddr(vaddr) || ptr == NULL) {
        // 设置线程状态值为-1，表示线程错误
        thread_current()->exit_status = -1;
        // 线程退出
        thread_exit();
    }
    /* 广度范围上的判断 */
    uint8_t *check_byteptr = (uint8_t *) vaddr;
    // 注意，最多加到3，因为4的时候已经跳出了一个内存单位
    for (uint8_t i = 0; i < 4; i++) {
        // 判断辅助函数
        if (get_user(check_byteptr + i) == -1) {
            thread_current()->exit_status = -1;
            thread_exit();
        }
    }
    return ptr;
}
```

再次测试

```
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
```

测试点全部通过，bingo!