

# PINTOS Project Report

10185101210 陈俊潼, East China Normal University 2019.12

## 目录 / Table of Content

---

### PINTOS Project Report

#### 目录 / Table of Content

##### 准备工作

安装与调试

代码规范

版本管理

##### Project 1: Threads

Overview

###### Mission 1: Alarm Clock (忙等待问题)

Requirement

Analysis

Solution

Result

###### Misson 2: Priority Scheduling (优先级调度问题)

Requirements

Analysis

Solution

Result

###### Mission 3: Advanced Scheduler (高级调度问题)

Requirements

Analysis

Solution

Result

Remark

##### Project 2: User Programs

Overview

Interlude: 针对 macOS 的额外修改

###### Mission 1: Argument Passing (参数传递)

Requirements

Analysis

Solution

Result

### Mission 2: Process Termination Messages (进程终止消息)

Requirements

Analysis

Solution

### Mission 3 System Calls (系统调用)

Requirements

Analysis

Solution

```
void halt (void)
pid_t exec (const char *cmd_line)
int wait (pid t pid)
void exit (int status)
bool create (const char *file, unsigned initial_size)
bool remove (const char *file)
int open (const char *file)
int filesize (int fd)
int read (int fd, void *buffer, unsigned size)
int write (int fd, const void *buffer, unsigned size)
void seek (int fd, unsigned position)
unsigned tell (int fd)
void close (int fd)
```

Structure

### Mission 4 Denying Writes to Executables (拒接写可执行文件)

Requirements

Analysis

Solution

Result

Remark

# 准备工作

---

## 安装与调试

为了方便使用 VSCode 做实验，避免安装一个繁重 Ubuntu 虚拟机，便尝试直接在 macOS 上安装 pintos。使用的 pintos 来自：

[https://github.com/maojie/pintos\\_mac](https://github.com/maojie/pintos_mac)

下载后使用以下 port 命令安装依赖库、gdb 和 bochs：

```
1 sudo port install i386-elf-binutils
2 sudo port install i386-elf-gcc
3 sudo port install sdl
4 sudo port install gdb # 用于调试，安装后需要使用命令 ggdb 调试
5 sudo port install bochs -smp +gdbstub
```

其中 `sudo port install bochs -smp +gdbstub` 后面的两个参数是为了开启 gdb 调试。因为 port 默认安装的 pintos 没有 `--enable-gdb-stub` 参数（可以通过查阅 port 的 variant 得到，如下图）。

```
* billchen@Bills-MacBook-Pro-2018 ~ ➔ port variant bochs
bochs has the variants:
[+]avx: Enable AVX support
    debugger: Enable bochs internal debugger
        * conflicts with gdbstub
    gdbstub: Enable GDB stub debugging
        * conflicts with debugger smp
    sdl: Enable SDL GUI
[+]sdl2: Enable SDL2 GUI
[+]smp: Enable symmetric multi-processor support
    * conflicts with gdbstub
[+]term: Enable text-mode GUI
    universal: Build for multiple architectures
    x11: Enable X11 support
```

为了能够直接输入 gdb 运行 ggdb，可以 `vim ~/.bash_profile`，加入一行 `alias gdb='ggdb' ;`。

接着讲将 pintos 放入任意目录，在终端中将 utils 目录 export PATH：

```
1 # 后面的目录是 utils 所在的目录
2 export
    PATH=$PATH:~/OneDrive/Workspace/LearningRepo/Course/OSConcepts/pintos
        /utils
```

可以尝试运行：

```
billchen@Bills-MacBook-Pro-2018 ~ ➜ pintos --run
Prototype mismatch: sub main::SIGVTALRM () vs none at /Users/billchen/OneDrive/Workspace/LearningRepo/Course/OSConcepts/pintos/utils/pintos line 934.
Constant subroutine SIGVTALRM redefined at /Users/billchen/OneDrive/Workspace/LearningRepo/Course/OSConcepts/pintos/utils/pintos line 926.
Cannot find kernel
```

发现无法找到内核。进入 threads 目录运行 make：

```
y-donate-chain.o tests/threads/mlfqqs-load-1.o tests/threads/mlfqqs-load-60.o tests/threads/mlfqqs-load-avg.o tests/threads/mlfqqs-load-random.o  
i386-elf-objcopy -R .note -R .comment -S kernel.o kernel.bin  
i386-elf-gcc -c ../../threads/loader.S -o threads/loader.o -Wa,--gstabs -nostdinc -I../../include -I../../lib  
i386-elf-ld -N -e 0 -Ttext 0x7c00 --oformat binary -o loader.bin threads/loader.o  
billchen@Bills-MacBook-Pro-2018 ~/OneDrive/Workspace/Pintos/src/threads master
```

一般就可以执行了。但经过多次尝试你发现偶尔还会出现找不到内核的情况。如果还是无法运行，可以尝试修改 `kernal.o` 和 `loader.o` 的位置。在 `/utils/pintos` 的第 256 行：

```
254 sub find_disks {
255     # Find kernel, if we don't already have one.
256     if (!exists $parts{KERNEL}) {
257         my $name = find_file ('/Users/billchen/OneDrive/Workspace/LearningRepo/Course/OSConcepts/pintos/thread/build/kernel');
258         die "Cannot find kernel\n" if !defined $name;
259         do_set_part ('KERNEL', 'file', $name);
260     }
261 }
```

/utils/pintos.pm 第 362 行:

```
360 ✓ sub read_loader {
361     my ($name) = @_;
362     $name = find_file ("loader.bin") if !defined $name;
363     die "Cannot find loader\n" if !defined $name;
364
365     my ($handle);
366     open ($handle, <$name) or die "Cannot open $name\n";
367     my $content = do { local $/; <$handle> };
368     close ($handle);
```

/utils/pintos-gdb 第 4 行，调整 GDBMACROS 的目录：

```
-  
3 # Path to GDB macros file. Customize for your site.  
4 GDBMACROS=/usr/class/cs140/pintos/pintos/src/misc/gdb-macros  
5
```

经过后面的实验，发现如果手动修改了目录，需要在进行 Project 2 的时候把 `loader` 和 `kernel` 的位置指向 `userprog` 目录下的 `build` 内的 `kernel` 和 `loader`。

### 接着测试调试

## 输入命令

```
1 sudo pintos --adb -s -- run alarm-multiple
```

新建终端 进入 threads/build 目录下输入命令:

```
1 ggdb kernel.o # 使用 port 安装的 mac 下的 gdb 应输入 ggdb  
2 target remote localhost:1234
```

回到 Bochs 运行界面，可以发现已经连接成功：

```
0000000000001[SIM] quit_sim called with exit code 1
* billchen@Bills-MacBook-Pro-2018 ~ sudo pintos --gdb -s -- run alarm-multiple
Password:
bochs -q
=====
Bochs x86 Emulator 2.6.8
Built from SVN snapshot on May 3, 2015
Compiled on Dec 5 2019 at 11:56:27
=====
00000000000i[      ] LTDL_LIBRARY_PATH not set. using compile time default '/opt/local/lib/bochs/plugins'
00000000000i[      ] BXSHARE not set. using compile time default '/opt/local/share/bochs'
00000000000i[      ] lt_dlhandle is 0x7fc527e287b0
00000000000i[PLUGIN] loaded plugin libbx_unmapped.so
00000000000i[      ] lt_dlhandle is 0x7fc527e28c00
00000000000i[PLUGIN] loaded plugin libbx_biosdev.so
00000000000i[      ] lt_dlhandle is 0x7fc527d00780
00000000000i[PLUGIN] loaded plugin libbx_speaker.so
00000000000i[      ] lt_dlhandle is 0x7fc527f00720
00000000000i[PLUGIN] loaded plugin libbx_extfpuirq.so
00000000000i[      ] lt_dlhandle is 0x7fc529800230
00000000000i[PLUGIN] loaded plugin libbx_parallel.so
00000000000i[      ] lt_dlhandle is 0x7fc5299015b0
00000000000i[PLUGIN] loaded plugin libbx_serial.so
00000000000i[      ] reading configuration from bochssrc.txt
00000000000e[      ] bochssrc.txt:8: 'user_shortcut' will be replaced by new 'keyboard' option.
00000000000i[      ] Enabled gdbstub
00000000000i[      ] lt_dlhandle is 0x7fc527f03940
00000000000i[PLUGIN] loaded plugin libbx_nogui.so
00000000000i[      ] installing nogui module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
Waiting for gdb connection on port 1234
Connected to 127.0.0.1
```

这时已经可以在 gdb 中输入命令 `make check` 查看检查信息，得到以下反馈：

```
1 (gdb) target remote localhost:1234
2 Remote debugging using localhost:1234
3 warning: No executable has been specified and target does not support
4 determining executable automatically. Try using the "file" command.
5 0x00000000 in ?? ()
6 (gdb) make check
7 pass tests/threads/alarm-single
8 pass tests/threads/alarm-multiple
9 pass tests/threads/alarm-simultaneous
10 FAIL tests/threads/alarm-priority
11 pass tests/threads/alarm-zero
12 pass tests/threads/alarm-negative
13 FAIL tests/threads/priority-change
14 FAIL tests/threads/priority-donate-one
15 FAIL tests/threads/priority-donate-multiple
16 FAIL tests/threads/priority-donate-multiple2
17 FAIL tests/threads/priority-donate-nest
18 FAIL tests/threads/priority-donate-sema
19 FAIL tests/threads/priority-donate-lower
20 FAIL tests/threads/priority-fifo
21 FAIL tests/threads/priority-preempt
```

```
22 FAIL tests/threads/priority-sema
23 FAIL tests/threads/priority-condvar
24 FAIL tests/threads/priority-donate-chain
25 FAIL tests/threads/mlfqs-load-1
26 FAIL tests/threads/mlfqs-load-60
27 FAIL tests/threads/mlfqs-load-avg
28 FAIL tests/threads/mlfqs-recent-1
29 pass tests/threads/mlfqs-fair-2
30 pass tests/threads/mlfqs-fair-20
31 FAIL tests/threads/mlfqs-nice-2
32 FAIL tests/threads/mlfqs-nice-10
33 FAIL tests/threads/mlfqs-block
34 20 of 27 tests failed.
35 make: *** [check] Error 1
```

确认环境配置完成，实验正式开始。

在 pintos 运行时，会在 build 目录下创建 bochssrc.txt 文件，用于给 bochs 虚拟机提供配置文件。运行时的输出都会输出在终端窗口中。可以使用 intos run alar0m-multiple > log 将输出重定向到文本文件保存。

## 代码规范

官方文档推荐在开始实现之前，阅读附录中的代码规范。指出项目应当遵循 [GNU Coding Standards](#)：

### C.1 Style

Style, for the purposes of our grading, refers to how readable your code is. At minimum, this means that your code is well formatted, your variable names are descriptive and your functions are decomposed and well commented. Any other factors which make it hard (or easy) for us to read or use your code will be reflected in your style grade.

The existing Pintos code is written in the GNU style and largely follows the [GNU Coding Standards](#). We encourage you to follow the applicable parts of them too, especially chapter 5, “Making the Best Use of C.” Using a different style won’t cause actual problems, but it’s ugly to see gratuitous differences in style from one function to another. If your code is too ugly, it will cost you points.

规范指出：

- 不应当使任意一行代码超过 79 个字符
- 支持 C99 标准库中的新特性
- 应当为每一个函数写明注释
- 不要使用 `strcpy()`、`strcat()`、`sprintf()` 等不安全的函数

## 版本管理

为了方便维护项目完成进度，或者在出现无法解决的问题是回滚至旧版代码，将整个项目文件夹托管到 GitHub 上。

首先在 GitHub 网站上新建一个 Repository，在 Pintos 目录下使用命令 `git init` 新建空仓库，然后将所有文件添加到项目管理 `git add -A`。使用 `git commit -m "init Project"` 进行一次提交，最后使用 `remote add pintos https://github.com/BillChen2000/Pintos` 和 `git push origin master` 将初始项目上传至 GitHub。

项目 GitHub 地址：<https://github.com/BillChen2000/Pintos>。

## Project 1: Threads

---

### Overview

在开始之前，初步了解 PintOS 目录下的几个文件夹的内容：

```
1 threads/: 内核的源代码  
2 userprog/: 用户程序加载代码  
3 vm/: 虚拟内存目录  
4 filesystem/: 文件系统目录  
5 devics/: I/O 设备驱动目录  
6 lib/: 包含部分标准 C 语言的函数  
7 lib/kernel: 部分只在 Pintos 中有的 C 语言函数  
8 lib/user: 包含一些头文件，只在 Pintos 中有的一些 C 语言函数  
9 tests/: 每个 Project 的测试案例  
10 examples/: 在 Projcet 2 的一些案例  
11 misc/ & utils/: 官方不推荐修改的两个文件夹
```

除此之外，浏览官方文档的附录 Debugging Tools，了解到两个常用的调试工具的用法。第一个是 `ASSERT`，官方的描述如下：

#### E.2 ASSERT

Assertions are useful because they can catch problems early, before they'd otherwise be noticed. Ideally, each function should begin with a set of assertions that check its arguments for validity. (Initializers for functions' local variables are evaluated before assertions are checked, so be careful not to assume that an argument is valid in an initializer.) You can also sprinkle assertions throughout the body of functions in places where you suspect things are likely to go wrong. They are especially useful for checking loop invariants.

`ASSERT` 的作用是测试括号内的表达式，如果表达式不为真，这会出现 kernel panic，将出现错误的详细信息打印在屏幕上。

第二个调试工具是 `printf()`，使用方法同 C 标准库函数。

# Mission 1: Alarm Clock (忙等待问题)

## Requirement

在这个任务中需要重新实现 `devices/timer/c` 目录下的 `timer_sleep()`。虽然当前代码提供了一个实现方式，但它的实现方为忙等待，即它在循环中检查当前时间是否已经过去 `ticks` 个时钟，并循环调用 `thread_yield()` 直到循环结束。需要重新实现这个函数来避免忙等待。

对于 `timer_msleep()`、`timer_usleep()`、`timer_nsleep()` 等函数，将会自动定期调用 `timer_sleep()`，所有不需要修改。

## Analysis

这个函数将会在 pintos 原先的实现中，`timer_sleep()` 的代码如下：

```
1  /* Sleeps for approximately TICKS timer ticks.  Interrupts must
2   * be turned on. */
3  void
4  timer_sleep (int64_t ticks)
5  {
6      int64_t start = timer_ticks ();
7
8      ASSERT (intr_get_level () == INTR_ON);
9      while (timer_elapsed (start) < ticks)
10         thread_yield ();
11 }
```

- 首先，`start` 记录了进入这个函数的当前时间。
- 然后判断 `intr_get_level()` 的返回值是否为真，即是否启用了中断，如果没有则 kernel panic。
- 利用 `timer_elapsed ()` 判断当前经过的时间和 `start` 之间的差值。
- 判断当前流逝的时间是否超过给定的 `ticks`，如果没有，执行 `thread_yield ()`，让线程休眠。

接下来查找 `thread_yield()` 函数。这个函数位于 `threads/thread.c`：

```
1  /* Yields the CPU.  The current thread is not put to sleep and
2   * may be scheduled again immediately at the scheduler's whim. */
3  void
4  thread_yield (void)
5  {
6      struct thread *cur = thread_current ();
7      enum intr_level old_level;
```

```

8
9     ASSERT (!intr_context ());
10
11    old_level = intr_disable ();
12    if (cur != idle_thread)
13        list_push_back (&ready_list, &cur->elem);
14    cur->status = THREAD_READY;
15    schedule ();
16    intr_set_level (old_level);
17 }

```

- 首先通过 `thread_current()` 获得当前正在运行的线程。这个结构体指针中的线程结构体包含以下字段：

```

1   struct thread
2   {
3       /* Owned by thread.c. */
4       tid_t tid;                                /* Thread identifier.
5   */
5       enum thread_status status;                /* Thread state. */
6       char name[16];                           /* Name (for debugging
purposes). */
7       uint8_t *stack;                          /* Saved stack
pointer. */
8       int priority;                           /* Priority. */
9       struct list_elem allelem;               /* List element for
all threads list. */
10
11      /* Shared between thread.c and synch.c. */
12      struct list_elem elem;                  /* List element. */
13
14 #ifdef USERPROG
15     /* Owned by userprog/process.c. */
16     uint32_t *pagedir;                     /* Page directory. */
17 #endif
18
19     /* Owned by thread.c. */
20     unsigned magic;                        /* Detects stack
overflow. */
21 };

```

包含有线程 ID，线程状态，线程名，栈指针，优先级，线程链表，线程项，项目目录，和一个用于检查栈溢出的量。而其中的 `thread_status` 又包含有以下几种线程状态：

```

1  /* States in a thread's life cycle. */
2  enum thread_status
3  {
4      THREAD_RUNNING,      /* Running thread. */
5      THREAD_READY,        /* Not running but ready to run. */
6      THREAD_BLOCKED,      /* Waiting for an event to trigger. */
7      THREAD_DYING         /* About to be destroyed. */
8  };

```

这些文件都通过 `thread.c` 实现。

2. 获取 `old_level`，即调用函数的时候的中断状态。同时发现在 `intr_disable()` 函数中会暂时禁用中断。
3. 判断当前的线程是否为 `idle_thread`，如果不是，则将现在这个线程 push 到 `ready_list` 的尾部。
4. 调度当前线程，同时将中断状态设置回刚调用 `thread_yield()` 时的状态。

回顾 `timer_sleep()` 函数，可以发现这个函数是一个自旋锁，将会一直循环检查当前经过的时间并且调用 `thread_yield()` 来让线程休眠，会出现忙等待的现象。也就是为了让线程休眠特定时间，这个函数就在这一段时间里反复把线程从运行状态丢到就绪列表的最后。当调度到来时，如果时间没到，又一次把线程放在最后，这样做效率低下。

观察到 `thread_block()` 函数和 `thread_unblock` 函数，观察其注释：

```

1  /* Puts the current thread to sleep. It will not be scheduled
2   again until awoken by thread_unblock().
3
4   This function must be called with interrupts turned off. It
5   is usually a better idea to use one of the synchronization
6   primitives in synch.h. */
7
8  void
9  thread_block (void)
10 {
11     ASSERT (!intr_context ());
12     ASSERT (intr_get_level () == INTR_OFF);
13
14     thread_current ()->status = THREAD_BLOCKED;
15     schedule ();
16 }

```

可知如果一个线程被设置为阻塞状态后，在调用 `thread_unblock()` 之前将不再会被 `schedule()` 函数调度，而这正是我们需要的。

## Solution

为了解决这个问题，可以在线程第一次调用 `timer_sleep()` 的时候，通过调用 `thread_block()` 函数来讲线程的状态设置为阻塞，同时记录下需要阻塞的时间，至此，`timer_sleep()` 就完成了自己的工作，把唤醒的工作留给其他函数。

这样的话，当在每一次中断，即调用 `timer_interrupt()` 的时候，都需要把所有被阻塞的线程内记录的阻塞时间信息 -1，如果减到了 0，则 `unblock` 线程，供后续调度。

所以需要对代码进行以下修改：

1. 在 `thread` 的结构体，也就是 PCB 中加入一项 `blocked_ticks`：

```
83 ~ struct thread
84 ~ {
85     /* Owned by thread.c. */
86     tid_t tid;                      /* Thread identifier. */
87     enum thread_status status;      /* Thread state. */
88     char name[16];                 /* Name (for debugging purposes) */
89     uint8_t *stack;                /* Saved stack pointer. */
90     int priority;                 /* Priority. */
91     struct list_elem allelem;      /* List element for all threads */
92
93     /* Shared between thread.c and synch.c. */
94     struct list_elem elem;          /* List element. */
95     uint16_t blocked_ticks;        /* Blocked ticks. */
```

2. 在初始化线程调用 `thread_create()` 的时候，将 `blocked_ticks` 设置为 0：

```
● 183  /* Initialize thread. */
184  init_thread (t, name, priority);
185  tid = t->tid = allocate_tid ();
186
187  /* ++ Set blocked ticks to 0 */
188  t->blocked_ticks = 0;
```

3. 新建一个 `thread_check_blocked(struct thread *t)` 函数检查线程的阻塞时间记录情况。之所以需要增加第二个 `void *aux` 指针的原因是这个函数将会被 `thread_foreach` 调用，而这个函数将会给 `thread_chcheck_blocked` 传递一个 `aux` 指针。

```
593  /* ++ Check if the thread should be unblocked as it's
594  |   blocked ticks reached to 0. */
595  void thread_check_blocked (struct thread *t, void *aux UNUSED){
596      if (t->status == THREAD_BLOCKED && t->blocked_ticks > 0){
597          t->blocked_ticks--;
598          if (t->blocked_ticks == 0) {
599              thread_unblock(t);
600          }
601      }
602  }
```

同时在 `thread.h` 头文件中添加：

```
1 void thread_check_blocked(struct thread *, void * aux UNUSED);
```

4. 在 `timer_interrupt()` 调用的时候，对每一个线程都使用 `thread_for_each()` 函数调用 `thread_check_blocked()` 来处理阻塞状态：

```
178 /* Timer interrupt handler. */
179 static void
180 timer_interrupt (struct intr_frame *args UNUSED)
181 {
182     ticks++;
183     thread_tick ();
184     /* ++ Check block status for each thread */
185     thread_foreach(thread_check_blocked, NULL);      You, a few seconds ago • Uncomm
186 }
187
```

5. 最后修改改 `timer_sleep()` 函数，使线程通过阻塞休眠而不是忙等待：

```
● 90  timer_sleep (int64_t ticks)
91  {
92      int64_t start = timer_ticks ();
93
94      ASSERT (intr_get_level () == INTR_ON);
95
96      /* ++ Use block to handle sleep */      You, a few seconds ago • Uncommitte
97  if(ticks > 0){
98      enum intr_level old_intr_level = intr_disable();
99      struct thread *t = thread_current();
100     t->blocked_ticks = ticks;
101     thread_block();      You, a few seconds ago • Uncommitte
102     intr_set_level(old_intr_level);
103 }
104 /* -- Old Implementations */
105 // while (timer_elapsed (start) < ticks)
106 //   thread_yield ();
107 }
108 }
```

## Result

使用命令 `pintos -- run alarm-multiple` 检查运行结果，可以得到：

```
1 (alarm-multiple) begin
2 (alarm-multiple) Creating 5 threads to sleep 7 times each.
3 (alarm-multiple) Thread 0 sleeps 10 ticks each time,
4 (alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
5 (alarm-multiple) If successful, product of iteration count and
6 (alarm-multiple) sleep duration will appear in nondescending order.
7 (alarm-multiple) thread 0: duration=10, iteration=1, product=10
8 (alarm-multiple) thread 0: duration=10, iteration=2, product=20
9 (alarm-multiple) thread 1: duration=20, iteration=1, product=20
```

```
10 (alarm-multiple) thread 0: duration=10, iteration=3, product=30
11 (alarm-multiple) thread 2: duration=30, iteration=1, product=30
12 (alarm-multiple) thread 0: duration=10, iteration=4, product=40
13 (alarm-multiple) thread 1: duration=20, iteration=2, product=40
14 (alarm-multiple) thread 3: duration=40, iteration=1, product=40
15 (alarm-multiple) thread 0: duration=10, iteration=5, product=50
16 (alarm-multiple) thread 4: duration=50, iteration=1, product=50
17 (alarm-multiple) thread 0: duration=10, iteration=6, product=60
18 (alarm-multiple) thread 1: duration=20, iteration=3, product=60
19 (alarm-multiple) thread 2: duration=30, iteration=2, product=60
20 (alarm-multiple) thread 0: duration=10, iteration=7, product=70
21 (alarm-multiple) thread 1: duration=20, iteration=4, product=80
22 (alarm-multiple) thread 3: duration=40, iteration=2, product=80
23 (alarm-multiple) thread 2: duration=30, iteration=3, product=90
24 (alarm-multiple) thread 1: duration=20, iteration=5, product=100
25 (alarm-multiple) thread 4: duration=50, iteration=2, product=100
26 (alarm-multiple) thread 1: duration=20, iteration=6, product=120
27 (alarm-multiple) thread 2: duration=30, iteration=4, product=120
28 (alarm-multiple) thread 3: duration=40, iteration=3, product=120
29 (alarm-multiple) thread 1: duration=20, iteration=7, product=140
30 (alarm-multiple) thread 2: duration=30, iteration=5, product=150
31 (alarm-multiple) thread 4: duration=50, iteration=3, product=150
32 (alarm-multiple) thread 3: duration=40, iteration=4, product=160
33 (alarm-multiple) thread 2: duration=30, iteration=6, product=180
34 (alarm-multiple) thread 3: duration=40, iteration=5, product=200
35 (alarm-multiple) thread 4: duration=50, iteration=4, product=200
36 (alarm-multiple) thread 2: duration=30, iteration=7, product=210
37 (alarm-multiple) thread 3: duration=40, iteration=6, product=240
38 (alarm-multiple) thread 4: duration=50, iteration=5, product=250
39 (alarm-multiple) thread 3: duration=40, iteration=7, product=280
40 (alarm-multiple) thread 4: duration=50, iteration=6, product=300
41 (alarm-multiple) thread 4: duration=50, iteration=7, product=350
42 (alarm-multiple) end
```

duration 和 iteration 的乘积已经是不减排序了，修改完成。运行 `make check` 查看结果：

```
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
FAIL tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
```

除了在 Mission 2 中要解决的 `alarm-priority` 以外都显示通过测试。

## Misson 2: Priority Scheduling (优先级调度问题)

### Requirements

这一部分要求在 Pintos 中实现优先级调度。

在 thread 的 PCB 中已经具有了 `priority` 项，优先级最低从 `PRI_MIN` 到最高 `PRI_MAX`。当 ready list 中出现了一个比当前正在运行的线程优先级更高的线程的时候，当前的线程将会立即让出 CPU。同时，当线程在等待一个信号量的时候，最高优先级的线程将会被第一个唤醒。

除此之外，实验还要求每一个线程可以在任意时候提高或降优先级，并且在降低优先级后如果不是当前系统中优先级最高的线程，立即让出 CPU。

除此之外，需要实现的问题还有优先级倒置、优先级捐赠。需要完成 `thread.c` 中的 `void thread_set_priority (int new_priority)` 函数和 `int thread_get_priority (void)` 函数。

### Analysis

操作系统目前的调度函数为 `schedule()`，代码如下：

```
1  /* Schedules a new process. At entry, interrupts must be off and
2   the running process's state must have been changed from
3   running to some other state. This function finds another
4   thread to run and switches to it.
5
6   It's not safe to call printf() until thread_schedule_tail()
7   has completed. */
8 static void
9 schedule (void)
10 {
11     struct thread *cur = running_thread ();
12     struct thread *next = next_thread_to_run ();
13     struct thread *prev = NULL;
14
15     ASSERT (intr_get_level () == INTR_OFF);
16     ASSERT (cur->status != THREAD_RUNNING);
17     ASSERT (is_thread (next));
18
19     if (cur != next)
20         prev = switch_threads (cur, next);
21     thread_schedule_tail (prev);
22 }
```

可以看到在第 12 行，函数通过调用 `next_thread_to_run()` 获取要调度的下一个线程，然后在第 20 行调用 `switch_threads(cur, next)` 把当前线程和要调度的下一个线程进行交换。

观察目前 `next_thread_to_run()` 的代码：

```
1  /* Chooses and returns the next thread to be scheduled. Should
2   return a thread from the run queue, unless the run queue is
3   empty. (If the running thread can continue running, then it
4   will be in the run queue.) If the run queue is empty, return
5   idle_thread. */
6  static struct thread *
7  next_thread_to_run (void)
8  {
9    if (list_empty (&ready_list))
10      return idle_thread;
11    else
12      return list_entry (list_pop_front (&ready_list), struct thread,
13      elem);
14 }
```

可以看到这个函数只是简单的返回 `ready_list` 中最前面的线程。如果队列为空，则返回一 `idle_thread` 空线程。

而观察 `thread.c` 中添加线程方式，发现有三个函数在向 `ready_list` 中添加线程，分别是 `thread_list()`、`init_thread()`、和 `thread_yield()`。而这三个函数都使用了 `list_push_back (&ready_list, &cur->elem);` 来添加。因此，pintos 目前使用的算法是 FIFO 调度。

为了实现优先级调度，首先应当使线程进入队列的时候按照优先级的大小插入，而不是简单的放在后面。

发现 `list.c` 内已经具有了方法 `list_insert_ordered()`，可以将项目按照顺序插入：

```
1  /* Inserts ELEM in the proper position in LIST, which must be
2   sorted according to LESS given auxiliary data AUX.
3   Runs in O(n) average case in the number of elements in LIST. */
4  void
5  list_insert_ordered (struct list *list, struct list_elem *elem,
6                      list_less_func *less, void *aux){...}
```

因此，对于该过程，需要：

- 自行实现 `list_less_func` 用于比较优先级

- 然后把这三个具有安排线程的函数中的调度函数从 `list_push_back()`；更改成 `list_insert_ordered()`。

查看官方文档：

Implement priority scheduling in Pintos. When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread. Similarly, when threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first. A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU.

除了当新的线程出现的时候需要根据优先级进行调度和安排，当某一个线程的优先级改变并且比当前线程的优先级高的时候，也需要内核立即让出程序，即调用 `thread_yield`。所以，当有新的线程出现（在 `init_thread` 中）或者当某一线程优先级改变的时候（在 `thread_set_priority()` 中），也需要抢占式地改变当前正在运行的程序。

在实验手册的 F&Q 部分也有进一步说明：

#### **Can a thread added to the ready list preempt the processor?**

Yes. If a thread added to the ready list has higher priority than the running thread, the correct behavior is to immediately yield the processor. It is not acceptable to wait for the next timer interrupt. The highest priority thread should run as soon as it is runnable, preempting whatever thread is currently running.

高优先级的线程在具备运行的机会时候，应当以及运行。因此，在这里需要做两个更改：

- 当新的线程建立时，判断新线程的优先级，如果高则抢占
- 更新一个线程的优先级时，判断更新后的优先级，如果高则抢占。

另外，对于锁的唤醒问题，当有一系列程序等待锁释放的时候，需要最先唤醒优先级最高的程序。对于这一个要求，观察现有的锁函数（位于 `synch.c`）：

```

1  /* Acquires LOCK, sleeping until it becomes available if
2   necessary.  The lock must not already be held by the current
3   thread.
4
5   This function may sleep, so it must not be called within an
6   interrupt handler.  This function may be called with
7   interrupts disabled, but interrupts will be turned back on if
8   we need to sleep. */
9 void
10 lock_acquire (struct lock *lock)
11 {
12     ASSERT (lock != NULL);
13     ASSERT (!intr_context ());

```

```

14     ASSERT (!lock_held_by_current_thread (lock));
15
16     sema_down (&lock->semaphore);
17     lock->holder = thread_current ();
18 }
19

```

观察 `sema_down()` 函数的注释：

```

1  /* Down or "P" operation on a semaphore.  Waits for SEMA's value
2   to become positive and then atomically decrements it.
3
4   This function may sleep, so it must not be called within an
5   interrupt handler.  This function may be called with
6   interrupts disabled, but if it sleeps then the next scheduled
7   thread will probably turn interrupts back on. */
8 void
9 sema_down (struct semaphore *sema)
10 {...}

```

可以发现 `sema_down()` 是信号量的 P 操作。而对于这个信号量，原代码中的定义如下：

```

1  /* A counting semaphore. */
2  struct semaphore
3  {
4      unsigned value;           /* Current value. */
5      struct list waiters;    /* List of waiting threads. */
6  };

```

可以看到维持了一个等待线程的队列。但进一步观察发现所有对 `waiters` 队列的操作都是同样采用的 FIFO 算法，所以这里需要：

- 修改所有对 `waiters` 队列的，使其变成优先级队列。

对于 **优先级倒转**，官方的文档中有如下叙述：

One issue with priority scheduling is “priority inversion”. Consider high, medium, and low priority threads  $H$ ,  $M$ , and  $L$ , respectively. If  $H$  needs to wait for  $L$  (for instance, for a lock held by  $L$ ), and  $M$  is on the ready list, then  $H$  will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is for  $H$  to “donate” its priority to  $L$  while  $L$  is holding the lock, then recall the donation once  $L$  releases (and thus  $H$  acquires) the lock.

也就是说，当一个低优先级线程占有了一个资源锁并已经在等待队列中，一个中优先级线程也在等待队列中在低优先级线程之前。如果这个时候一个高优先级的线程，也就是正在运行的线程申请使用这个资源，就会陷入死锁的状态。因为高优先级线程为了使用资源必须要让低优先级线程释放资源锁，但低优先级线程却已经被放在了队列的后方，永远无法被调度出来。

所以这里需要有一个优先级翻转的过程，即先让这个低优先级的线程暂时获得比较高的优先级并处理，让它把资源锁释放；释放了之后再将优先级恢复成之前的状态。恢复后再放回队列中的位置，不会影响之前已经实现了的调度策略。

通过阅读官方文档，可以使用优先级捐赠算法解决这一问题。关于优先级捐赠，官方的实验手册还有如下说明：

#### What happens to the priority of a donating thread?

Priority donation only changes the priority of the donee thread. The donor thread's priority is unchanged. Priority donation is not additive: if thread *A* (with priority 5) donates to thread *B* (with priority 3), then *B*'s new priority is 5, not 8.

捐赠不会导致子线程的优先级超过捐赠者，也不会改变捐赠者自身的优先级。此外，除了单个线程对单个线程的优先级捐赠以外，还需要考虑多个线程之间的捐赠情况：

Implement priority donation. You will need to account for all different situations in which priority donation is required. Be sure to handle multiple donations, in which multiple priorities are donated to a single thread. You must also handle nested donation: if *H* is waiting on a lock that *M* holds and *M* is waiting on a lock that *L* holds, then both *M* and *L* should be boosted to *H*'s priority. If necessary, you may impose a reasonable limit on depth of nested priority donation, such as 8 levels.

当出现了高优先级线程访问中优先级线程的锁，同时中优先级线程访问低优先级线程的锁的情况的时候，需要发生递归捐赠，低优先级的线程最终会被提升到高优先级线程的优先级，然后在锁释放后恢复原先的优先级。

如果没有递归捐赠的步骤，那么高优先级的线程会把优先级捐赠给中优先级线程，但此时之前中优先级线程为了不让低优先级线程死锁而捐赠的优先级就会没有用了。所以这个时候需要把低优先级线程的优先级也提升上来。

而现有的锁的定义为：

```
1  /* Lock. */
2  struct lock
3  {
4      struct thread *holder;      /* Thread holding lock (for
debugging). */
5      struct semaphore semaphore; /* Binary semaphore controlling
access. */
6  };
```

不支持多个正在访问这个锁的线程的记录，需要稍后修改。

总结起来为了解决优先级捐赠问题需要有以下修改步骤：

- 需要修改线程的结构体定义来存储线程是否被捐赠了优先级，之前的优先级和新的优先级。
- 在一个线程获取一个锁的时候，如果拥有这个锁的线程优先级比自己低就提高它的优先级；如果还有其他线程在使用这个锁，也会相应地提升这个线程的优先级
- 如果线程同时被多个线程捐赠优先级，线程的优先级将会变成这些捐赠线程的优先级中的最高值

## Solution

首先为了实现 `list_insert_ordered()` 函数，首先需要自行编写一个比较优先级的函数。在 `thread.c` 加入以下比较函数：

```
1  /* ++1.2 Compare priority */
2  bool compare_priority(const struct list_elem *a, const struct
list_elem *b, void *aux UNUSED) {
3      int pa = list_entry(a, struct thread, elem)->priority;
4      int pb = list_entry(b, struct thread, elem)->priority;
5      return pa > pb;
6  }
```

同时在 `thread.h` 中声明：

```
1  /* +++1.2 */
2  bool compare_priority(const struct list_elem *, const struct
list_elem *, void *);
```

修改 `yield()` 中的调度语句：

```

● 317 void
318   thread_yield (void)
319 {
320     struct thread *cur = thread_current ();
321     enum intr_level old_level;
322
323     ASSERT(!intr_context ());
324
325     old_level = intr_disable ();
326     if (cur != idle_thread)
327       //list_push_back (&ready_list, &cur->elem);
328     /* +++1.2 Put elements ordered */
329     list_insert_ordered(&ready_list, &cur->elem, (list_less_func *) &compare_priority, NULL);
330
331     cur->status = THREAD_READY;
332     schedule ();
333     intr_set_level (old_level);
334 }

```

You, 5 days ago • Init project.

另两个函数 `thread_unblock()` 同样做相同更改：

```

1  //list_push_back (&ready_list, &t->lelem);
2  /* +++1.2 Priority */
3  list_insert_ordered(&ready_list, &t->elem, (list_less_func
*)&compare_priority, NULL);

```

对 `init_thread()`：传递的参数为 `all_list` 和 `allelem`。

```

1  //list_push_back (&all_list, &t->allelem);
2  /* ++1.2 Priority */
3  list_insert_ordered(&all_list, &t->allelem, (list_less_func
*)&compare_priority, NULL);

```

接下来修改设置优先级函数中的语句：

```

357 void
358   thread_set_priority (int new_priority)
359 {
360   /* ++1.2 Handle priority */
361   int old_priority = thread_current()->priority;
362
363   You, a minute ago • Uncommitted changes
364   thread_current()->priority = new_priority;
365   /* +++ 1.2 Handle priority */
366   if (new_priority < old_priority) {
367     thread_yield();
368   }

```

至此，修改了 `thread.c` 中的优先级调度程序，已经可以通过部分和优先级有关的测试了：

```
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
```

接下来对于优先级捐赠问题，首先需要 `thread` 定义中加入以下数据结构：

```
97  /* ++1.2 Data members to handle priority */ You, a few seconds ago • Uncommitted changes
98  int base_priority;          /* Base priority. */
99  struct list locks;         /* Locks that the thread is holding. */
100 struct lock *lock_waiting; /* The lock that the thread is waiting for. */
101
```

然后为 `lock` 的定义加入以下两个数据结构。前者是当前线程在信号量队列中位置（在原始的队列中，当前线程一定位于队列的首部），后者则是表示该锁的信号量队列中线程的最高优先级（用于优先级的捐赠）：

```
► 20  /* Lock. */
21  struct lock
22  {
23      struct thread *holder;      /* Thread holding lock (for debugging). */
24      struct semaphore semaphore; /* Binary semaphore controlling access. */
25
26      /* ++1.2 Priority Donation */
27      struct list_elem elem;     /* List element for priority donation. */
28      int max_priority;         /* Max priority among the threads acquiring the lock. */
29  };
30
```

为了让修改的数据结构能够得到初始化，修改线程的初始化函数 `init_thread` 和锁的初始化函数 `lock_init`：

```
495  /* ++1.2 Priority */
496  t->base_priority = priority; You, a few seconds ago • Uncor
497  list_init(&t->locks);
498  t->lock_waiting = NULL;
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
```

由于全程都在涉及关于锁的操作，所以先对获取锁的函数 `lock_acquire()` 进行修改。也就是要让每一个锁在获得的时候首先检测比较想要获得这个锁的线程的优先级是否比锁内存储的 `max_priority` 大，如果更大，则需要将这个锁内存储的 `max_priority` 设置为最大的优先级，并且对线程进行优先级捐赠操作：

```
1  /* Acquires LOCK, sleeping until it becomes available if
2   necessary. The lock must not already be held by the current
3   thread.
4
5   This function may sleep, so it must not be called within an
6   interrupt handler. This function may be called with
7   interrupts disabled, but interrupts will be turned back on if
8   we need to sleep. */
```

```

9 void
10 lock_acquire (struct lock *lock)
11 {
12     ASSERT (lock != NULL);
13     ASSERT (!intr_context ());
14     ASSERT (!lock_held_by_current_thread (lock));
15
16     /** ++1.2 Priority Donate */
17     struct thread *current_thread = thread_current ();
18     struct lock *l;
19     enum intr_level old_level;
20
21     if (lock->holder != NULL && !thread_mlfqs) {
22         current_thread->lock_waiting = lock;
23         l = lock;
24         while (l && current_thread->priority > l->max_priority) {
25             l->max_priority = current_thread->priority;
26             thread_donate_priority(l->holder);
27             l = l->holder->lock_waiting;
28         }
29     }
30
31     sema_down (&lock->semaphore);
32     old_level = intr_disable ();
33     current_thread = thread_current ();
34     if (!thread_mlfqs) {
35         current_thread->lock_waiting = NULL;
36         lock->max_priority = current_thread->priority;
37         thread_hold_the_lock (lock);
38     }
39     lock->holder = current_thread;
40     intr_set_level (old_level);
41     // sema_down (&lock->semaphore);
42     // lock->holder = thread_current ();
43 }

```

对应的在 `thread.c` 中实现 `thread_donate_priority` 和 `thread_hold_the_lock` 函数。

其中 `thread_donate_priority` 需要自行实现一个对 `t` 进行优先级更新的函数，因为被捐赠优先级的线程不一定是正在运行的线程，之前程序自带的 `thread_set_priority()` 只能满足更新当前线程的优先级，所以需要修改。

而 `thread_hold_the_lock` 则是让线程获得当前锁。由于如果线程拥有一个锁，那么线程的优先级一定要是拥有这个锁的队列中的最大值，所以如果锁的优先级大于线程的优先级，需要相应地更新线程的优先级，然后把这个锁加入到线程拥有的锁的队列中。

这两个函数的代码如下：

```
1  /* ++1.2 Let thread hold a lock */
2  void thread_hold_the_lock(struct lock *lock) {
3      enum intr_level old_level = intr_disable();
4      list_insert_ordered(&thread_current()>locks, &lock->elem,
5                          lock_cmp_priority, NULL);
6
7      if (lock->max_priority > thread_current()>priority) {
8          thread_current()>priority = lock->max_priority;
9          thread_yield();
10     }
11
12 }
13
14 /* ++1.2 Donate current priority to thread t. */
15 void thread_donate_priority(struct thread *t) {
16     enum intr_level old_level = intr_disable();
17     thread_update_priority(t);
18
19     if (t->status == THREAD_READY) {
20         list_remove(&t->elem);
21         list_insert_ordered(&ready_list, &t->elem, compare_priority,
22                             NULL);
23     }
24     intr_set_level(old_level);
25 }
```

接下来对于 `thread_update_priority()` 函数编写如下。当前已经有的修改函数 `thread_set_priority()` 只能用于修改当前正在运行的优先级。不过这个函数可以修改一下用于更新当前运行线程的 `base_priority` 并根据新的优先级来判断是否需要 `yield` 线程。

在这之前，先完善修改当前线程优先级的函数 `thread_set_priority()` 如下：

```
1  /* Sets the current thread's priority to NEW_PRIORITY. */
2  void
3  thread_set_priority (int new_priority)
4  {
5      /* ++1.2 Handle priority */
```

```

6   int old_priority = thread_current()->priority;
7   enum intr_level old_level = intr_disable();
8   struct thread *current_thread = thread_current();
9   current_thread->base_priority = new_priority;
10
11  if (list_empty(&current_thread->locks) || new_priority >
old_priority) {
12      current_thread->priority = new_priority;
13      thread_yield();
14  }
15  intr_set_level(old_level);
16  /* +++ 1.2 (OLD )Handle priority */
17  // if (new_priority < old_priority) {
18  //   thread_yield();
19  // }
20 }
```

此后再编写一个 `thread_update_priority()` 函数来更新当前正在运行的线程的优先级。使用这个函数来应对多个线程同时在对一个线程进行优先级捐赠的情况。为了使设置的优先级为锁的最大值，需要对某个线程所获得的锁进行排序，然后取出队列中最前的元素作为当前线程的新优先级。

```

1  /* ++1.2 Used to update priority. */
2  void thread_update_priority(struct thread *t) {
3      enum intr_level old_level = intr_disable();
4      int max_priority = t->base_priority;
5      int lock_priority;
6
7      if (!list_empty(&t->locks)) {
8          list_sort(&t->locks, lock_cmp_priority, NULL);
9          lock_priority = list_entry(list_front(&t->locks), struct lock,
elem)->max_priority;
10         if (lock_priority > max_priority)
11             max_priority = lock_priority;
12     }
13
14     t->priority = max_priority;
15     intr_set_level(old_level);
16 }
```

显然为了实现关于锁的排序函数，需要对应地实现 `lock_cmp_priority` 函数如下：

```

1  /* ++1.2 Compare priority in locks */
2  bool lock_cmp_priority(const struct list_elem *a, const struct
3      list_elem *b, void *aux UNUSED) {
4      return list_entry(a, struct lock, elem)->max_priority >
            list_entry(b, struct lock, elem)->max_priority;
5  }

```

同时在 `thread.h` 中加入刚刚编写的几个函数的声明:

```

150  /* +++1.2 */ You, an hour ago • Uncommitted changes
151  bool compare_priority(const struct list_elem *, const struct list_elem *, void *);
152  void thread_update_priority(struct thread *);
153  bool lock_cmp_priority(const struct list_elem *, const struct list_elem *, void *);
154  void thread_remove_lock(struct lock *);
155  void thread_donate_priority(struct thread *);
156  void thread_hold_the_lock(struct lock *);
157

```

以上实现了获取锁的逻辑。而对于锁的释放，需要先把锁从线程的 `lock` 队列中删除，然后将锁设置为不被任何线程占用，最后再进行信号量的 `V` 操作。这里编写一个 `thread_remove_lock` 函数来实现:

```

1  /* ++1.2 Remove a lock. */
2  void thread_remove_lock(struct lock *lock) {
3      enum intr_level old_level = intr_disable();
4      list_remove(&lock->elem);
5      thread_update_priority(thread_current());
6      intr_set_level(old_level);
7  }

```

在进行 `lock_release()` 的时候调用该函数:

```

1  /* ++1.2 */
2  if(!thread_mlfqs){
3      thread_remove_lock(lock);
4  }

```

最后将剩下的队列修改为优先级队列。首先修改 `synch.c` 中的 `cond_signal` 函数:

```

348  if (!list_empty (&cond->waiters))
349  /* ++1.2 Priority */
350      list_sort(&cond->waiters, cond_sema_cmp_priority, NULL); You, 5
351      sema_up (&list_entry (list_pop_front (&cond->waiters),
352          struct semaphore_elem, elem)->semaphore);
353  }

```

完善其排序函数:

```

1  /* ++1.2 cond sema comparation function */
2  bool cond_sema_cmp_priority(const struct list_elem *a, const struct
3      list_elem *b, void *aux UNUSED) {
4      struct semaphore_elem *sa = list_entry(a, struct semaphore_elem,
5          elem);
6      struct semaphore_elem *sb = list_entry(b, struct semaphore_elem,
7          elem);
8      return list_entry(list_front(&sa->semaphore.waiters), struct
9          thread, elem)->priority > list_entry(list_front(&sb-
10         >semaphore.waiters), struct thread, elem)->priority;
11  }

```

再修改 `waiters` 为优先级队列。对 `sema_down` 和 `sema_up` 修改如下：

```

61  sema_down (struct semaphore *sema)
62  {
63      enum intr_level old_level;
64
65      ASSERT (sema != NULL);
66      ASSERT (!intr_context ());
67
68      old_level = intr_disable ();
69      while (sema->value == 0) {
70
71          /* ++1.2 Priority */
72          list_insert_ordered(&sema->waiters, &thread_current()->elem, compare_priority, NULL);
73          //list_push_back(&sema->waiters, &thread_current()->elem);
74          You, 3 minutes ago • Uncommitted changes
75          thread_block();
76      }
77      sema->value--;
78      intr_set_level (old_level);
79  }
80
81
112  sema_up (struct semaphore *sema)
113  {
114      enum intr_level old_level;
115
116      ASSERT (sema != NULL);
117
118      old_level = intr_disable ();
119      if (!list_empty (&sema->waiters)) {
120
121          /* ++1.2 */
122          list_sort(&sema->waiters, compare_priority, NULL);      You, a few seconds ago •
123          thread_unblock(list_entry(list_pop_front(&sema->waiters),
124          ||| struct thread, elem));
125      }
126
127      sema->value++;
128      intr_set_level (old_level);
129  }
130

```

## Result

完成以上步骤以后，可以看到 `priority` 部分的测试已经全部通过，优先级调度部分的修改完成。

```
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
```

1 7 of 27 tests failed.

## Mission 3: Advanced Scheduler (高级调度问题)

### Requirements

在这个部分中，需要自行实现类似于BSD调度程序的多级反馈队列调度程序，以减少在系统上运行作业的平均响应时间。

与优先级调度程序一样，高级调度程序同样基于线程的优先级来调度线程。但是高级调度程序不会执行优先级捐赠。必须编写必要的代码，以允许在Pintos启动时选择调度算法策略。默认情况下，优先级调度程序必须处于活动状态，但必须能够使用 `-mlfq` 内核选项选择4.4BSD调度程序。在 `main()` 函数中 `parse_options()` 解析选项时，传递此选项会将 `threads / thread.h` 中声明的 `thread_mlfqs` 设置为 `true`。所以为了完成这个任务，在 Mission 2 中的关于优先级捐赠的代码需要加入 `if(!thread_mlpfs)` 进行判断，暂时关闭优先级调度的代码。

启用 BSD 调度程序后，线程不再直接控制自己的优先级。应忽略 `thread_create()` 的优先级参数，以及对 `thread_set_priority()` 的任何调用，并且 `thread_get_priority()` 应返回调度程序设置的线程的当前优先级。高级调度程序不会在以后的任何项目中使用。

而对于每个线程的优先级的更新，应当使用官方文档附录中的算法实现。这一部分官方文档给出了三个计算流程：

The following formulas summarize the calculations required to implement the scheduler. They are not a complete description of scheduler requirements.

Every thread has a *nice* value between -20 and 20 directly under its control. Each thread also has a priority, between 0 (PRI\_MIN) through 63 (PRI\_MAX), which is recalculated using the following formula every fourth tick:

$$\text{priority} = \text{PRI\_MAX} - (\text{recent\_cpu} / 4) - (\text{nice} * 2).$$

*recent\_cpu* measures the amount of CPU time a thread has received “recently.” On each timer tick, the running thread’s *recent\_cpu* is incremented by 1. Once per second, every thread’s *recent\_cpu* is updated this way:

$$\text{recent\_cpu} = (2*\text{load\_avg})/(2*\text{load\_avg} + 1) * \text{recent\_cpu} + \text{nice}.$$

*load\_avg* estimates the average number of threads ready to run over the past minute. It is initialized to 0 at boot and recalculated once per second as follows:

$$\text{load\_avg} = (59/60)*\text{load\_avg} + (1/60)*\text{ready\_threads}.$$

where *ready\_threads* is the number of threads that are either running or ready to run at time of update (not including the idle thread).

需要计算 `priority` , `recent_cpu` 和 `load_avg` 三个值。第一个参数为整数，但后面两个参数的计算结果为定点数。但 pintos 本身的内核不支持定点数的运算：

## B.6 Fixed-Point Real Arithmetic

In the formulas above, *priority*, *nice*, and *ready\_threads* are integers, but *recent\_cpu* and *load\_avg* are real numbers. Unfortunately, Pintos does not support floating-point arithmetic in the kernel, because it would complicate and slow the kernel. Real kernels often have the same limitation, for the same reason. This means that calculations on real quantities must be simulated using integers. This is not difficult, but many students do not know how to do it. This section explains the basics.

The fundamental idea is to treat the rightmost bits of an integer as representing a fraction. For example, we can designate the lowest 14 bits of a signed 32-bit integer as fractional bits, so that an integer  $x$  represents the real number  $x/2^{14}$ . This is called a 17.14 fixed-point number representation, because there are 17 bits before the decimal point, 14 bits after it, and one sign bit.<sup>1</sup> A number in 17.14 format represents, at maximum, a value of  $(2^{31} - 1)/2^{14} \approx 131,071.999$ .

所以在这个任务中，还需要我们自行实现定点数的计算。

## Analysis

通用调度程序的目标是平衡线程的不同调度需求。执行大量I/O的线程需要快速响应时间以保持输入和输出设备忙，但需要很少的CPU时间。另一方面，绑定计算的线程需要花费大量CPU时间来完成其工作，但不需要快速响应时间。其他线程介于两者之间，I/O周期被计算周期打断，因此需求随时间变化。精心设计的调度程序通常可以同时满足具有所有这些要求的线程。

我们必须实现附录B中描述的调度程序。调度程序类似于 [McKusick] 中描述的调度程序，是多级反馈队列调度程序的一个示例。这种类型的调度程序维护几个可立即运行的线程队列，其中每个队列包含具有不同优先级的线程。在任何给定时间，调度程序从最高优先级的非空队列中选择一个线程。如果最高优先级队列包含多个线程，则它们以“循环”顺序运行。

调度程序的多个方面需要在一定数量的计时器滴答之后更新数据。在每种情况下，这些更新应该在任何普通内核线程有机会运行之前发生，这样内核线程就不可能看到新增的 `timer_ticks()` 值而是旧的调度程序数据值。

首先需要关注的是对于每个线程的 `nice` 值，即处理器亲和度。`nice` 值为0不会影响线程优先级。`nice` 值从1至20，会降低线程的优先级，并导致它放弃一些原本会收到的CPU时间。另一种情况，`nice` 值从-20到-1，往往会从其他线程中抢占CPU时间。每4个时间周期进行  $priority = PRI\_MAX - (recent\_cpu/4) - (nice * 2)$  算出线程的新优先级。在 `thread.c` 中，已经预留了两个需要填补的函数：

```
1 int thread_get_nice (void);
2 void thread_set_nice (int new_nice);
```

对于公式中的 `recent_cpu`，需每个 `timer_tick` 都计算一次：

$$recent\_cpu = (2 * load\_avg) / (2 * load\_avg + 1) * recent\_cpu + nice.$$

我们希望 `recent_cpu` 可以表征每个进程“最近”收到多少CPU运行时间。此外，作为一种改进，最新收到的CPU时间应该比其之前的CPU时间权重更大。一种方法是使用  $n$  个元素的数组来跟踪在最后  $n$  秒的每一个中接收的CPU时间。然而，这种方法每线程需要  $O(n)$  空间，并且每次计算新加权平均值需要  $O(n)$  时间。

相反，实验手册建议使用指数加权平均数计算：

$$\begin{aligned} x(0) &= f(0), \\ x(t) &= ax(t-1) + (1-a)f(t), \\ a &= k/(k+1), \end{aligned}$$

其中  $x(t)$  是整数时间  $t \geq 0$  的移动平均值， $f(t)$  是被平均的函数， $k > 0$  控制衰减速率。通过以下几个步骤迭代公式：

$$\begin{aligned} x(1) &= f(1), \\ x(2) &= af(1) + f(2), \\ &\vdots \\ x(5) &= a^4f(1) + a^3f(2) + a^2f(3) + af(4) + f(5). \end{aligned}$$

其中  $f(t)$  的值在时间  $t$  的权重为1，在时间  $t+1$  的权重为  $a$ ，在时间  $t+2$  的权重为  $2$ ，等等。我们还可以将  $x(t)$  与  $k$  相关联： $f(t)$  在时间  $t+k$  具有大约  $1/e$  的权重，在时间  $t+2k$  具有大约  $1/e^2$  等等。从相反方向， $f(t)$  在时间  $t+\log_a w$  衰减到  $f(t)*w$ 。

在创建的第一个线程中，`recent_cpu` 的初始值为0，或者在其他新线程中为其父进程值。每次发生定时器中断时，除非空闲线程正在运行，否则 `recent_cpu` 仅对正在运行的线程递增1。此外，每秒一次，使用以下公式为每个线程（无论是运行，准备还是阻塞）重新计算 `recent_cpu` 的值：其中 `load_avg` 是准备运行的线程数的移动平均值（见下文）。如果 `load_avg` 为1，表示单个线程正在竞争CPU，那么 `recent_cpu` 的当前值在  $\log_2 / 3 0.1 \approx 6$  秒内衰减到原值的0.1。如果 `load_avg` 为2，则衰减到原值的0.1需要  $\log_3 / 4 0.1 \approx 8$  秒。结果是 `recent_cpu` 估计了线程“最近”收到的CPU时间量，衰减率与竞争CPU的线程数成反比。

一些测试所做的假设要求在系统计数器每达到一秒完全重新计算 `recent_cpu`，即条件 `timer_ticks() % TIMER_FREQ == 0` 成立。

对于具有负 `nice` 值的线程，`recent_cpu` 的值可能为负，所以不应当假设 `recent_cpu` 的值一直大于0。

此外，还需要考虑此公式中的计算顺序。先计算 `recent_cpu` 的系数，然后再做乘法，否则可能会产生溢出。

必须实现在 `threads/thread.c` 中的 `thread_get_recent_cpu()` 函数：

```
1 int thread_get_recent_cpu (void);
```

而为了计算 `recent_cpu`，又需要计算 `load_avg`，即系统的平均负载量。它估计在过去一分钟内，在准备队列中的平均线程数。像 `recent_cpu` 一样，它也是指数加权的平均值。与 `priority` 和 `recent_cpu` 不同，`load_avg` 是系统范围的，而不是特定于线程的。在系统启动时，它被初始化为0。此后每个 `timer_ticker` 更新一次，根据以下公式更新：

$$recent\_cpu = (2 * load\_avg) / (2 * load\_avg + 1) * recent\_cpu + nice,$$

`ready_thread` 是在更新时运行或准备运行的线程数（不包括空闲线程）。

有些测试假设当计时器达到1秒倍数时，即当 `timer_ticks()%TIMER_FREQ == 0` 时，必须要重新更新 `load_avg`，而不是在其它时间。必须实现位于 `threads/thread.c` 中的函数：

```
1 int thread_get_load_avg(void);
```

最后，为了计算出 `recent_cpu` 值和 `load_avg` 值，需要我们自行实现定点数的运算。

根据官方实验手册的知道，实现定点运算基本思想是将定点数运算转换成整数的运算。**将整数的最右边的几位视为表示分数**。例如，我们可以将带符号的32位整数的最低14位指定为小数位，这样整数x代表实数  $x/2^{14}$ 。叫做17.14定点数，最大能够表示  $(2^{31}-1)/2^{14}$ ，近似于131071.999。

假设我们使用 p.q 的定点数格式，并且设  $f = 2^q$ 。根据上面的定义，我们可以通过乘以  $f$  将整数或实数转换为 p.q 格式。例如，基于 17.14 格式的定点数转换  $59/60$ ， $(59/60)2^{14} = 16110$ 。将定点数转换为整数则除以  $f$ 。

C 中的 / 运算符向零舍入，也就是说，它将正数向下舍入，向负数向上舍入。要舍入到最近，将  $f/2$  先与正数相加再除，或者先在负数中减去  $f/2$  再除。

实验手册提供了总结了如何在 C 中实现定点的算术运算。在表中， $x$  和  $y$  是定点数， $n$  是整数，定点数是带符号的 p.q 格式，其中  $p + q = 31$ ， $f$  的值应当为  $1 \ll q$ ：

The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table,  $x$  and  $y$  are fixed-point numbers,  $n$  is an integer, fixed-point numbers are in signed p.q format where  $p + q = 31$ , and  $f$  is  $1 \ll q$ :

Convert $n$ to fixed point:	$n * f$
Convert $x$ to integer (rounding toward zero):	$x / f$
Convert $x$ to integer (rounding to nearest):	$(x + f / 2) / f$ if $x \geq 0$ , $(x - f / 2) / f$ if $x \leq 0$ .
Add $x$ and $y$ :	$x + y$
Subtract $y$ from $x$ :	$x - y$
Add $x$ and $n$ :	$x + n * f$
Subtract $n$ from $x$ :	$x - n * f$
Multiply $x$ by $y$ :	$((int64_t) x) * y / f$
Multiply $x$ by $n$ :	$x * n$
Divide $x$ by $y$ :	$((int64_t) x) * f / y$
Divide $x$ by $n$ :	$x / n$

所以我们要自行定义定点数的运算逻辑，并在后面进行实现。

## Solution

首先，新的算法需要用到之前的内核中不支持的定点数运算。为了实现定点数的运算，在 `thread` 目录下新建一个 `fixed_point.h` 并编写如下宏：

```
1  /* Basic definitions of fixed point. */
2  #define int fixed_t;
3  /* 16 LSB used for fractional part. */
4  #define FP_SHIFT_AMOUNT 16
5  /* Convert a value to fixed-point value. */
6  #define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
7  /* Add two fixed-point value. */
8  #define FP_ADD(A,B) (A + B)
```

```

9  /* Add a fixed-point value A and an int value B. */
10 #define FP_ADD_MIX(A,B) (A + (B << FP_SHIFT_AMOUNT))
11 /* Subtract two fixed-point value. */
12 #define FP_SUB(A,B) (A - B)
13 /* Subtract an int value B from a fixed-point value A */
14 #define FP_SUB_MIX(A,B) (A - (B << FP_SHIFT_AMOUNT))
15 /* Multiply a fixed-point value A by an int value B. */
16 #define FP_MULT_MIX(A,B) (A * B)
17 /* Divide a fixed-point value A by an int value B. */
18 #define FP_DIV_MIX(A,B) (A / B)
19 /* Multiply two fixed-point value. */
20 #define FP_MULT(A,B) (((fixed_t)((int64_t) A) * B) >>
21 FP_SHIFT_AMOUNT)
22 /* Divide two fixed-point value. */
23 #define FP_DIV(A,B) (((fixed_t)((((int64_t) A) << FP_SHIFT_AMOUNT) /
24 B))
25 /* Get integer part of a fixed-point value. */
26 #define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
27 /* Get rounded integer of a fixed-point value. */
28 #define FP_ROUND(A) (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >>
29 FP_SHIFT_AMOUNT) \
30 : ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT))

```

这里用 16 位数 (FP\_SHIFT\_AMOUNT) 作为定点数的小数部分，也就是所有的运算都需要维持整数部分从第17位开始。

有了定点数的运算，便可以修改原代码了。首先在线程的结构体定义中加入如下新定义：

```

1  /* ++1.3 Nice */
2  int nice; /* Niceness. */
3  fixed_t recent_cpu;

```

有了新的数据成员，也需要在 `init_thread()` 线程初始化时，将 `nice` 与 `recent_cpu` 置零。注意 `recent_cpu` 是定点数0。我们需要在 `thread.c` 中定义全局变量 `load_avg`，注意这里使用的是我们自行定义的定点数类型。在 `thread_start()` 函数中初始化为0。

```

1  /* ++1.3 mlfqs */
2  t->nice = 0;
3  t->recent_cpu = FP_CONST(0);

```

初次之外，还需要在 `thread.c` 中加入全局变量 `load_avg`：

```

1  /** 1.3 */
2  fixed_t load_avg;

```

接下来处理多级反馈调度的逻辑实现。

根据实验说明，我们可以知道 `bool` 变量 `thread_mlfqs` 指示是否启用高级调度程序，并且高级调度程序不应包含优先级捐赠的内容，所以，在 Mission 2 中实现的优先级捐赠代码，需要使用 `if` 判断以保证在使用高级调度程序时，不启用优先级捐赠。

然后修改 `timer.c` 中的 `timer_interrupt` 函数，在完成任务 1 修改的基础下加入如下代码：

```
1  /* ++1.3 mlfqs */
2  if (thread_mlfqs)
3  {
4      thread_mlfqs_increase_recent_cpu_by_one ();
5      if (ticks % TIMER_FREQ == 0)
6          thread_mlfqs_update_load_avg_and_recent_cpu ();
7      else if (ticks % 4 == 0)
8          thread_mlfqs_update_priority (thread_current ());
9  }
```

实现 `thread_mlfqs_increase_recent_cpu_by_one`

`()`、`thread_mlfqs_update_load_avg_and_recent_cpu`  
`()`、`thread_mlfqs_update_priority (thread_current ())` 如下。

首先是 `thread_mlfqs_increase_recent_cpu_by_one(void)`，若当前进程不是空闲进程则当前进程加1，在函数中的所有运算都采用定点数加法。

```
1  /* ++1.3 mlfqs */
2  /* Increase recent_cpu by 1. */
3  void thread_mlfqs_increase_recent_cpu_by_one(void) {
4      ASSERT(thread_mlfqs);
5      ASSERT(intr_context());
6
7      struct thread *current_thread = thread_current();
8      if (current_thread == idle_thread)
9          return;
10     current_thread->recent_cpu = FP_ADD_MIX(current_thread->recent_cpu,
11                                              1);
11 }
```

接下在 `thread_mlfqs_update_load_avg_and_recent_cpu(void)` 函数中首先根据就绪队列的大小计算 `load_avg` 的值，随后根据 `load_avg` 的值，更新所有进程的 `recent_cpu` 值及 `priority` 值。

```
1  /* ++1.3 Every per second to refresh load_avg and recent_cpu of all
   threads. */
```

```

2 void thread_mlfqs_update_load_avg_and_recent_cpu(void) {
3     ASSERT(thread_mlfqs);
4     ASSERT(intr_context());
5
6     size_t ready_threads = list_size(&ready_list);
7     if (thread_current() != idle_thread)
8         ready_threads++;
9     load_avg = FP_ADD(FP_DIV_MIX(FP_MULT_MIX(load_avg, 59), 60),
10                      FP_DIV_MIX(FP_CONST(ready_threads), 60));
11
12     struct thread *t;
13     struct list_elem *e = list_begin(&all_list);
14     for (; e != list_end(&all_list); e = list_next(e)) {
15         t = list_entry(e, struct thread, allelem);
16         if (t != idle_thread) {
17             t->recent_cpu = FP_ADD_MIX(FP_MULT(FP_DIV(FP_MULT_MIX(load_avg,
18                 FP_ADD_MIX(FP_MULT_MIX(load_avg, 2), 1)), t->recent_cpu), t-
19                 >nice));
20             thread_mlfqs_update_priority(t);
21         }
22     }
23 }
```

最后，通过 `thread_mlfqs_update_priority (struct thread *t)` 函数，更新当前进程的 `priority` 值。并且一定要保证每个线程的优先级介于0(`PRI_MIN`)到63(`PRI_MAX`)之间，所以在最后加入一个关于上下限的逻辑判断。

```

1 /* ++1.3 Update priority. */
2 void thread_mlfqs_update_priority(struct thread *t) {
3     if (t == idle_thread)
4         return;
5
6     ASSERT(thread_mlfqs);
7     ASSERT(t != idle_thread);
8
9     t->priority = FP_INT_PART(FP_SUB_MIX(FP_SUB(FP_CONST(PRI_MAX),
10                                     FP_DIV_MIX(t->recent_cpu, 4)), 2 * t->nice));
11    t->priority = t->priority < PRI_MIN ? PRI_MIN : t->priority;
12    t->priority = t->priority > PRI_MAX ? PRI_MAX : t->priority;
13 }
```

然后将新定义的若干个函数加入到 `thread.h` 中：

```
162 /* ++1.3 mlfqs */
163 void thread_mlfqs_update_priority(struct thread *); You, a few seconds a
164 void thread_mlfqs_update_load_avg_and_recent_cpu(void);
165 void thread_mlfqs_increase_recent_cpu_by_one(void);
166
```

最后，在 `thread.c` 中还有之前定义好的几个和 `nice`、`load_avg`、`recent_cpu` 有关和几个函数：

```
394 /* Sets the current thread's nice value to NICE. */
395 void
396 thread_set_nice (int nice UNUSED)
397 {
398     /* Not yet implemented. */
399 } You, 9 days ago • Init project.
400
401 /* Returns the current thread's nice value. */
402 int
403 thread_get_nice (void)
404 {
405     /* Not yet implemented. */
406     return 0;
407 }
408
409 /* Returns 100 times the system load average. */
410 int
411 thread_get_load_avg (void)
412 {
413     /* Not yet implemented. */
414     return 0;
415 }
416
417 /* Returns 100 times the current thread's recent_cpu value. */
418 int
419 thread_get_recent_cpu (void)
420 {
421     /* Not yet implemented. */
422     return 0;
423 }
```

简单完善这四个函数的代码后如下。在设置进程的处理器亲和度之后需要及时让出进程并处理新的进程优先级。

```
1  /* ++1.3 Sets the current thread's nice value to NICE. */
2  void thread_set_nice(int nice) {
3      thread_current()->nice = nice;
4      thread_mlfqs_update_priority(thread_current());
5      thread_yield();
6  }
7
```

```
8  /* ++1.3 Returns the current thread's nice value. */
9  int thread_get_nice(void) {
10    return thread_current()->nice;
11  }
12
13  /* ++1.3 Returns 100 times the system load average. */
14  int thread_get_load_avg(void) {
15    return FP_ROUND(FP_MULT_MIX(load_avg, 100));
16  }
17
18  /* ++1.3 Returns 100 times the current thread's recent_cpu value. */
19  int thread_get_recent_cpu(void) {
20    return FP_ROUND(FP_MULT_MIX(thread_current()->recent_cpu, 100));
21  }
```

## Result

编写完以上代码后，执行 `make check`，可以看到以下结果：

```
pass tests/threads/mlfqs-block
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
pass tests/threads/priority-donate-one
pass tests/threads/priority-donate-multiple
pass tests/threads/priority-donate-multiple2
pass tests/threads/priority-donate-nest
pass tests/threads/priority-donate-sema
pass tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
pass tests/threads/priority-sema
pass tests/threads/priority-condvar
pass tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
```

27个测试已经完全通过。

## Remark

至此，Project 1 的 27 个测试已经全部通过了。

通过本次实验，从 0 开始分析一个操作系统，对我而言也是一个前所未有的挑战。不过通过阅读官方文档的资料和来自互联网上的资源，我逐渐对 Pintos 中线程调度这一部分有了更深刻的理解。从修改一个简单的忙等待问题，到完善优先级调度和多级反馈的调度策略，我将在操作系统理论课程里学导的知识运用到了实践当中，体会到了这些调度算法真正的实用之处，获益匪浅。

算法是无止境的，总会有更优的算法来解决问题，这也是我在修改操作系统中的一大体会。为了满足更加复杂的需求，实现更加智能与合理的调度，需要不断完善操作系统的数据结构和运行原理才能做到。通过学习前人的思路与逻辑，“踩在巨人的肩膀上”，我也理解到现有的这些算法都是无数前人智慧的结晶，最终才能融合出一个成熟的操作系统。

其次，这也是我第一次接触到这么底层的代码。虽然 pintos 的结构相比成熟的操作系统而言还非常简单，但尝试阅读这个操作系统代码的过程还是让我对整个操作系统的架构都有了更清晰的认识。在这之前，操作系统对我来说还是一块完全陌生而不敢触及的领域。经历过无数次无法成功 make 的绝望之后最后改出结果的成就感也是前所未有的。

## Project 2: User Programs

### Overview

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，而在本 Project 中要求我们对非内核进行修改。

对于 Project 2 主要修改的 `userprog` 目录，官方文档对目录下的主要文件有以下解释：

File	Note
process.c	用于加载 ELF 二进制文件和初始化进程
pagedir.c	一个页表的管理文件
syscall.c	包含了系统调用函数的文件
exception.c	用于处理用户程序的非法操作的文件
gdt.c	Global Descriptor Table (GDT)
tss.c	Task-State Segement 任务状态块

这个部分不需要用到之前在 Project 1 中已经完成的线程部分。

由于用户程序需要使用文件系统来进行加载，所以也需要了解一下 Pintos 中文件系统的实现。对于 Pintos 中的文件系统，是在 `filesys` 目录下实现的，已经为我们提供了一个简单的实现方式。为了正确的使用文件系统内，需要注意的有以下几点：

- 没有内部同步机制。也就是两个同时出现的访问会出现问题；
- 文件大小在创建的时候已经固定；
- 文件数据是整块分配的；
- 没有子目录；
- 文件名最长为 14 个字符；
- 系统崩溃可能会导致文件无法自动修复；

例如，可以使用以下命令创建一个磁盘文件：

```

1 pintos-mkdisk fileys.dsk --fileys-size=2
2 pintos -f -q
3 pintos -p ../../examples/echo -a echo -- -q
4 pintos -q run 'echo x'
```

命令 `pintos -p` 和命令 `pintos -g` 提供了将文件添加进 pintos 和从 pintos 中获取的方法。

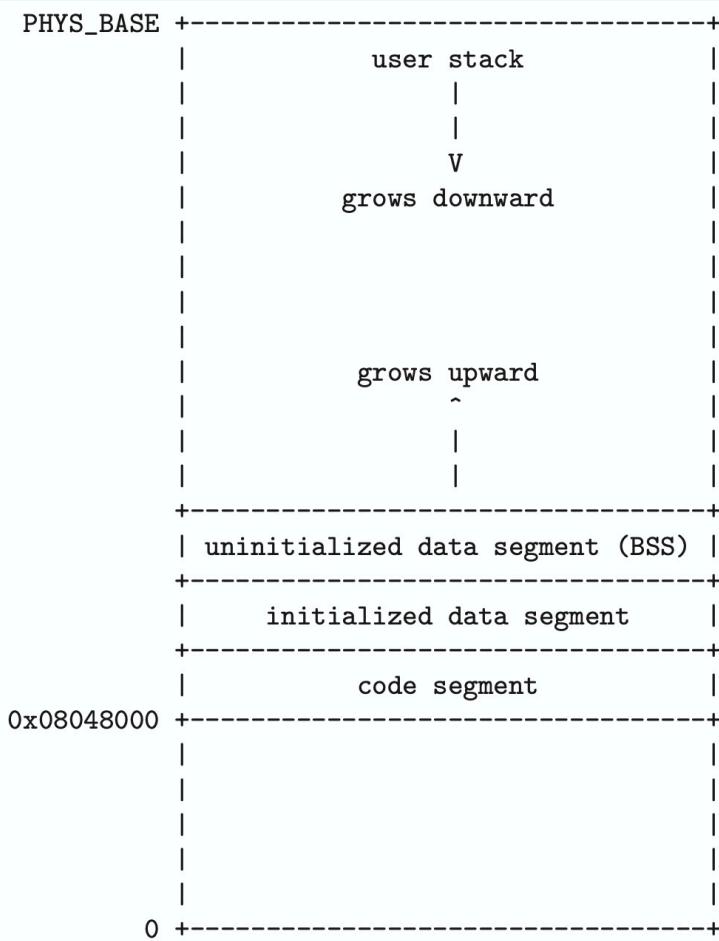
You'll need a way to copy files in and out of the simulated file system. The `pintos -p` ("put") and `-g` ("get") options do this. To copy '`file`' into the Pintos file system, use the command '`pintos -p file -- -q`'. (The '`--`' is needed because '`-p`' is for the `pintos` script, not for the simulated kernel.) To copy it to the Pintos file system under the name '`newname`', add '`-a newname`': '`pintos -p file -a newname -- -q`'. The commands for copying files out of a VM are similar, but substitute '`-g`' for '`-p`'.

继续了解用户程序在 Pintos 中的执行方式。在 Pintos 中可以执行正常的 C 语言程序，只要符合内存大小的要求并且没有调用没有实现的系统调用。注意 `malloc()` 函数是无法使用的。在 `src/example` 目录下具有一些简单的用户程序的实例，我们可以把这些程序编译进自己的程序。

此外，Pintos 还可以加载 `ELF` 可执行文件，使用在 `userprog/process.c` 中的加载器加载。

在 Pintos 中的每一个进程都具有如下的虚拟内存结构：

虚拟内存结构



在 Project 2 中，`user stack` 的大小是固定的。代码段开始的虚拟地址为 `0x08048000`，大概有 128 M 的大小。如果想要查看某一个特定可执行文件的结构，可以执行 `-p objdump` 命令查看。

对于系统调用中内存访问的部分，需要注意的是用农户可能传递过一个空指针，或者一个内核空间的指针，这些行为都是不被允许的。需要我们在以后的实现中注意。

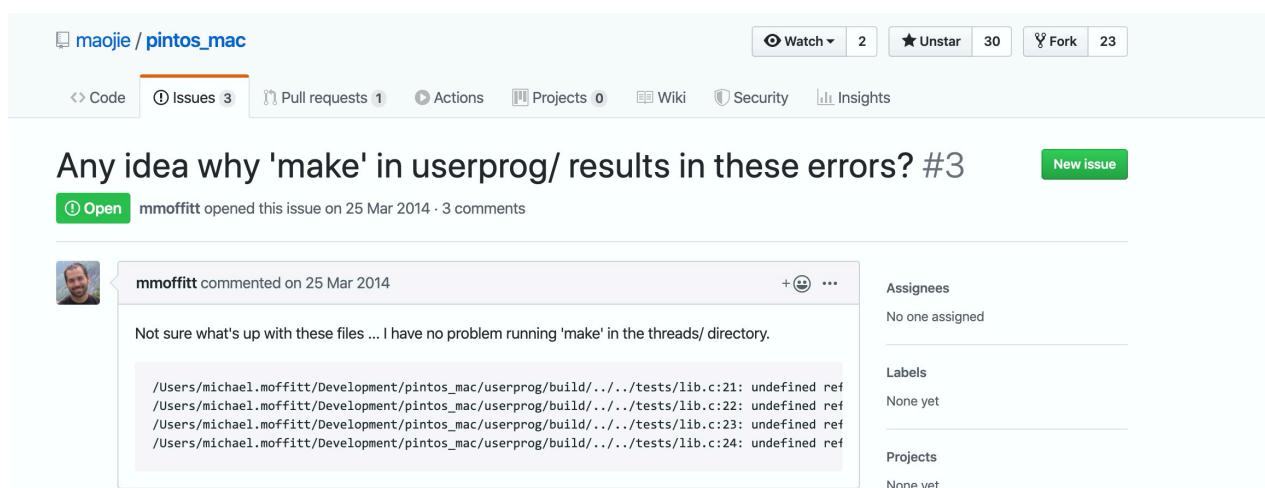
我将按照官方文档建议的执行顺序完成 Project，即首先完成参数传递的部分（这样才能避免每一个用户程序都出现页错误），然后完成用户的安全内存访问，然后完成系统调用。先完成基础函数和 `exit()` 系统调用，最后完成 `write()` 系统调用。然后完成进程终止消息，最后完成禁止写入可执行文件的部分。

## Interlude: 针对 macOS 的额外修改

在开始试验之前，在 `userprog` 下尝试 `make` 的时候，始终会提示无法成功：

```
warning: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/ranlib: warning for library: libc.a the table of contents is empty (no object file members in the library define global symbols)
i386-elf-gcc -nostdlib -static -Wl,-T,./lib/user/user.lds tests/userprog/args.o tests/lib.o lib/user/entry.o libc.a -o tests/userprog/args-none
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o in function 'vmsg':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:21: undefined reference to `snprintf'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:22: undefined reference to `vsnprintf'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:23: undefined reference to `strncpy'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:24: undefined reference to `write'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o in function 'fail':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:48: undefined reference to `exit'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o in function 'shuffle':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:74: undefined reference to `random_ulong'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o: in function `exec_children':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:87: undefined reference to `snprintf'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:88: undefined reference to `exec'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o in function 'wait_children':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:100: undefined reference to `wait'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o: in function `compare_bytes':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:171: undefined reference to `memcmp'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:191: undefined reference to `hex_dump'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:193: undefined reference to `hex_dump'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o in function 'check_file_handle':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:118: undefined reference to `filesize'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:133: undefined reference to `read'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: tests/lib.o: in function `check_file':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:155: undefined reference to `open'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: /Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../tests/lib.c:159: undefined reference to `close'
/opt/local/lib/gcc/i386-elf/9.2.0/../../../../i386-elf/bin/ld: lib/user/entry.o: in function `_start':
:/Users/billchen/OneDrive/Workspace/Pintos/src/userprog/build/../../lib/user/entry.c:9: undefined reference to `exit'
collect2: 错误: ld 返回 1
make[1]: *** [tests/userprog/args-none] Error 1
make: *** [all] Error 2
```

发现在 `pintos_mac` 的原作者 `maojie` 的 Issues 页面有人提出了同样的问题（[https://github.com/maojie/pintos\\_mac/issues/3](https://github.com/maojie/pintos_mac/issues/3)）：



Any idea why 'make' in userprog/ results in these errors? #3

mmoffitt opened this issue on 25 Mar 2014 · 3 comments

mmoffitt commented on 25 Mar 2014

Not sure what's up with these files ... I have no problem running 'make' in the threads/ directory.

/Users/michael.moffitt/Development/pintos\_mac/userprog/build/../../tests/lib.c:21: undefined ref  
/Users/michael.moffitt/Development/pintos\_mac/userprog/build/../../tests/lib.c:22: undefined ref  
/Users/michael.moffitt/Development/pintos\_mac/userprog/build/../../tests/lib.c:23: undefined ref  
/Users/michael.moffitt/Development/pintos\_mac/userprog/build/../../tests/lib.c:24: undefined ref

Assignees  
No one assigned

Labels  
None yet

Projects  
None yet

经过探索发现是由于 `gcc-elf-i386` 在 `Catalina` 系统下的链接出了问题。而另一位作者 `fork` 过去的库（[https://github.com/jinmel/pintos\\_mac](https://github.com/jinmel/pintos_mac)）对 `Makefile.userprog` 进行了进一步的完善，修改了对跨平台编译库中 `libc.a` 的引用，解决了该问题。

在 `GitHub Desktop` 中检视所做的修改：

```

src/Makefile.userprog

@@ -22,7 +22,7 @@ lib/user_SRC += lib/user/console.c # Console code.

22 22
23 23 LIB_OBJ = $(patsubst %.c,%o,$(patsubst %.S,%o,$(lib_SRC) $(lib/user_SRC)))
24 24 LIB_DEP = $(patsubst %.o,%d,$(LIB_OBJ))
25 25 -LIB = lib/user/entry.o libc.a
+LIB = lib/user/entry.o $(LIB_OBJ)

26 26
27 27 PROGS_SRC = $(foreach prog,$(PROGS),$( $(prog)_SRC))
28 28 PROGS_OBJ = $(patsubst %.c,%o,$(patsubst %.S,%o,$(PROGS_SRC)))
@@ -38,14 +38,9 @@ endef

38 38
39 39 $(foreach prog,$(PROGS),$(eval $(call TEMPLATE,$(prog))))
40 40

41 41 -libc.a: $(LIB_OBJ)
42 42 - rm -f $@
43 43 - ar r $@ $^
44 44 - ranlib $@
45 45 -
46 46 clean:::
47 47 rm -f $(PROGS) $(PROGS_OBJ) $(PROGS_DEP)
48 48 - rm -f $(LIB_DEP) $(LIB_OBJ) lib/user/entry.[do] libc.a
+ rm -f $(LIB_DEP) $(LIB_OBJ) lib/user/entry.[do]

49 44
50 45 .PHONY: all clean
51 46

```

写死的 `libc.a` 目录会导致错误。替换了新的 `Makefile.userprog` 后，在 `userprog` 目录下再次运行 `make` 即可成功：

```

billchen@bill-MacBook-Pro-2018 ~/OneDrive/Workspace/Pintos/src/userprog master • make
cd build && /Applications/Xcode.app/Contents/Developer/usr/bin/make all
make[1]: Nothing to be done for `all'.

```

进入 `userprog/build` 文件夹下运行 `make SIMULATOR==bochs check` 也可以检查成功了（虽然目前还是 76 个 test 全 fail 的状态）：

```

FAIL tests/filesys/base/sm-create
FAIL tests/filesys/base/sm-full
FAIL tests/filesys/base/sm-random
FAIL tests/filesys/base/sm-seq-block
FAIL tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
FAIL tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
76 of 76 tests failed.
make: *** [check] Error 1

```

接下来正式开始实验。

## Mission 1: Argument Passing (参数传递)

### Requirements

### 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 impose a reasonable limit on the length of the command line arguments. For example, you could limit the arguments to those that will fit in a single page (4 kB). (There is an unrelated limit of 128 bytes on command-line arguments that the `pintos` utility can pass to the kernel.)

目前的 `process_execute()` 函数并不支持将参数传递到新的进程。目前的执行方式为当 `main()` 函数初始完后，调用 `run_action()` 函数，如果检测到用户输入了 `run`，则调用 `run_task()` 运行程序，进一步调用 `process_wait()` 来等待 `process_execute()` 完成任务。

需要注意的是，Pintos 的命令行语句长度最大为 128 字节。

## Analysis

分析代码创建进程的函数 `process_execute`：

```
1  /* Starts a new thread running a user program loaded from
2   FILENAME. The new thread may be scheduled (and may even exit)
3   before process_execute() returns. Returns the new process's
4   thread id, or TID_ERROR if the thread cannot be created. */
5  tid_t
6  process_execute (const char *file_name)
7  {
8      char *fn_copy;
9      tid_t tid;
10
11     /* Make a copy of FILE_NAME.
12      Otherwise there's a race between the caller and load(). */
13     fn_copy = palloc_get_page (0);
14     if (fn_copy == NULL)
15         return TID_ERROR;
16     strlcpy (fn_copy, file_name, PGSIZE);
17
18     /* Create a new thread to execute FILE_NAME. */
19     tid = thread_create (file_name, PRI_DEFAULT, start_process,
20                         fn_copy);
21     if (tid == TID_ERROR)
22         palloc_free_page (fn_copy);
```

```
22     return tid;
23 }
```

可以看到，新建一个进程具有如下过程：

- 首先调用 `process.c` 中的 `process_execute` 函数；
- 操作系统申请一块内存
- 调用 `thread_create` 函数调用子进程并且传递一个函数指针 `load`
- 在函数 `load` 中通过加载 `file_name` 等参数，保存了可执行文件的参数指针等。

因此，我们需要做的主要做的任务便是要将命令行参数传递进 `load` 和 `setup_stack` 中，然后将参数按照正确的顺序压进栈指针 `*esp` 中。

官方文档的 F&Q 部分阐释了当输入 `bin/ls -l foo bar` 的时候，栈的结构示意图供我们参考：

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
0xbfffffc	argv[3] [...]	'bar\0'	char [4]
0xbfffff8	argv[2] [...]	'foo\0'	char [4]
0xbfffff5	argv[1] [...]	'-l\0'	char [3]
0xbfffffd	argv[0] [...]	'/bin/ls\0'	char [8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbfffffc	char *
0xbfffffe0	argv[2]	0xbfffff8	char *
0xbfffffdc	argv[1]	0xbfffff5	char *
0xbfffffd8	argv[0]	0xbfffffd	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

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

在该过程中，存在同步问题。`load` 函数会直接分配页目录，打开文件，然后把参数压进栈中。我们已经知道了 pintos 的文件系统不会在多个文件同时访问的时候正常工作，所以需要通过一个信号量来解决该问题。

## Solution

首先需要将参数分隔开来。传递个 `process_execute` 函数的参数 `file_name` 既包括了可执行文件的名称，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。这个函数的具体使用方法如下：

**DESCRIPTION**

This interface is obsoleted by `strsep(3)`.

The `strtok()` function is used to isolate sequential tokens in a null-terminated string, `str`. These tokens are separated in the string by at least one of the characters in `sep`. The first time that `strtok()` is called, `str` should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, `sep`, must be supplied each time, and may change between calls.

在 `process_execute` 加入如下语句：

```

1  char *thread_name;
2  struct thread * current_thread = thread_current();
3  char *save_ptr;
4
5  thread_name = malloc(strlen(file_name)+1);
6  strlcpy (thread_name, file_name, strlen(file_name)+1);
7  thread_name = strtok_r (thread_name, " ", &save_ptr);
8
9  tid = thread_create (thread_name, PRI_DEFAULT, start_process,
10   fn_copy);
11 free(thread_name);

```

这样可以将文件名传递到 `thread_name` 参数中，然后交给 `thread_create` 函数进行线程的创建。

然后在 `start_process` 中将 `file_name` 通过 `load` 存储到栈中：

```
1  success = load (file_name, &if_.eip, &if_.esp);
```

`load` 函数目前的实现是直接将整个 `file_name` 作为文件名打开，但实际情况可能还含有参数。所以需要处理一下 `load` 中的打开过程：

```

1  /* Open executable file. */
2
3  char *fn_cp = malloc(strlen(file_name) + 1);
4  strlcpy(fn_cp, file_name, strlen(file_name) + 1);
5  char *temp_ptr;
6  fn_cp = strtok_r(fn_cp, " ", &temp_ptr);
7  file = filesys_open(fn_cp);
8  free(fn_cp);

```

然后在 `setup_stack` 函数中处理关键的参数处理部分。编写参数处理的语句如下，一些关键的步骤已经打上注释：

```

1  /* ++2 Argument passing */
2  char *token, *temp_ptr;
3
4  char * filename_cp = malloc(strlen(file_name)+1);

```

```

5   strlcpy (filename_cp, file_name, strlen(file_name)+1);
6
7   // calculate argc
8   enum intr_level old_level = intr_disable();
9   int argc=1;
10  bool is_lastone_space=false; //keep a record that if the last char
11  is space. for the use of two-space situation
12
13  for(int j=0;j!=strlen(file_name); j++){
14      if(file_name[j] == ' '){
15          if(!is_lastone_space)
16              argc++;
17          is_lastone_space=true;
18      }
19      else
20          is_lastone_space=false;
21
22  intr_set_level (old_level);
23
24  int *argv = calloc(argc,sizeof(int));
25
26  int i;
27  token = strtok_r (file_name, " ", &temp_ptr);
28  for (i=0; ; i++){
29      if(token){
30          *esp -= strlen(token) + 1;
31          memcpy(*esp,token,strlen(token) + 1);
32          argv[i]=*esp;
33          token = strtok_r (NULL, " ", &temp_ptr);
34      }else{
35          break;
36      }
37
38      // word align
39      *esp -= ((unsigned)*esp % WORD_SIZE);
40
41      //null ptr sentinel: null at argv[argc]
42      *esp-=sizeof(int);
43
44      //push address
45      for(i=argc-1;i>=0;i--)
46      {
47          *esp-=sizeof(int);
48          memcpy(*esp,&argv[i],sizeof(int));

```

```

49     }
50
51     //push argv address
52     int tmp = *esp;
53     *esp-=sizeof(int);
54     memcpy(*esp,&tmp,sizeof(int));
55
56     //push argc
57     *esp-=sizeof(int);
58     memcpy(*esp,&argc,sizeof(int));
59
60     //return address
61     *esp-=sizeof(int);
62     memcpy(*esp,&argv[argc],sizeof(int));
63
64     free(filename_cp);
65     free(argv);
66

```

- 首先处理文件名，即文件名，作为第一个参数
- 然后禁用中断，一个字节一个字节的读取参数，每当读取到一段连续空格结束就给参数个数 + 1
- 接着为读取到的参数 `argv[]` 字符串数组申请相应的内存空间
- 然后按照 4 字节对齐的方式依次把参数压入栈中。这里使用的是 `sizeof(int)` 来处理字节对齐的问题。注意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。另外还需要让 `argv[argc]` 应当是一个空指针。
- 最后释放申请的临时指针。

到此，参数就可以正确地在程序初始化的时候压入栈中了。

不过在做完修改步骤之后，执行检查的时候会发现所有测试仍然会导致内核 `PANIC`。查阅文档得知，如果调度器被过早调用，就会导致运行的时候出现以下消息：

```

FAIL tests/userprog/open-normal
Kernel panic in run: PANIC at ../../threads/vaddr.h:84 in vtop(): assertion `is_kernel_vaddr (vaddr)' failed.
Call stack: 0xc0027c98

```

分析在 Project 1 中做过的修改可以得知，这是由于之前在实现优先级调度的时候，在每一个 `sema_up` 操作之后都进行了 `thread_yiled()` 导致的。所以这里可以考虑修改 `synch.c` 中的 `sema_up` 函数：

```
1  /** ++2 If it's user program, don't yield */
2  #ifdef USERPROG
3      // thread_yield ();
4  #else
5      thread_yield();
6  #endif
7
```

加入一个判断逻辑即可解决该问题。

最后，为了处理临界区问题，我们单独在在 `load` 的临界区进入前后加入一个对 `filesys_lock` 锁的申请。

```
1  lock_acquire(&filesys_lock);
2  //loading the file
3  lock_release(&filesys_lock);
```

同时在 `thread/thread.h` 中线程的结构体定义中也应当相应地加入：

```
1  //a global lock on filesystem operations, to ensure thread safety.
2  struct lock fileys_lock
```

这样既可避免冲突访问。这个信号量同时还用在了后续任务中关于文件的系统调用函数中，会在后面的部分详细说明。

至此，参数传递部分已经实现完成。

## Result

在 `userprog/build` 目录下执行命令

```
1  make SIMULATOR=--bochs check
```

可以发现参数传递部分的测试已经通过：

```
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
```

## Mission 2: Process Termination Messages (进程终止消息)

### Requirements

当一个用户进程终止的时候，会调用 `exit()` 函数，并且打印当前进程名和退出代号。其中，终止消息必须按照以下格式打印：

```
1 printf ("%s: exit(%d)\n")
```

其中，进程的名称必须和传递给 `process_execute()` 的参数一样，并忽略命令行参数。

### 3.3.2 Process Termination Messages

Whenever a user process terminates, because it called `exit` or for any other reason, print the process's name and exit code, formatted as if printed by `printf ("%s: exit(%d)\n", ...);`. The name printed should be the full name passed to `process_execute()`, omitting command-line arguments. Do not print these messages when a kernel thread that is not a user process terminates, or when the `halt` system call is invoked. The message is optional when a process fails to load.

需要注意的是，当内核线程终止或者 `halt` 调用的时候不应当打印这些信息，而当进程加载失败的时候，这些消息应当是一个可选项。除了这条消息以外不应当打印任何额外的信息，否则会降低评分脚本给出的评分。

## Analysis

既然要打印返回值，就得用一个变量保存返回值，于是在每个线程的结构体 `thread` 中加入一个数据结构 `ret` 用于存储返回值，并且在 `init_thread()` 的时候将初始化设置为 `INIT_EXIT_STAT`。

而当线程退出的时候，需要将返回值保存到 `exit\_status` 中，可以在系统调用 `exit()` 函数的时候保存。

考虑到每个现车航结束的时候，都一定会调用 `thread_exit()` 函数。观察这个函数的代码：

```
1 /* Deschedules the current thread and destroys it. Never
2      returns to the caller. */
3 void
4 thread_exit (void)
5 {
6     ASSERT (!intr_context ());
7
8 #ifdef USERPROG
9     process_exit ();
10 #endif
11
12     /* Remove thread from all threads list, set our status to dying,
13         and schedule another process. That process will destroy us
14         when it calls thread_schedule_tail(). */
15     intr_disable ();
```

```
16     list_remove (&thread_current()>allelem);
17     thread_current ()>status = THREAD_DYING;
18     schedule ();
19     NOT_REACHED ();
20 }
```

在前面的文档中已经了解到每个用户进程只有一个线程，而内核进程具有多个线程。所以如果是用户进程，这里会调用 `process_exit()` 函数直接终止掉整个进程。如果是用户进程，则页表一定不是空，也就是 `pd!=NULL` 成立。所以可以在 `process_exit()` 函数中加入打印语句即可。

## Solution

在 `thread.h` 中新建如下定义：

```
1  /* ++ 2 */
2  #define INIT_EXIT_STAT -2333
3  ...
4  int exit_status;
```

并在 `thread_init` 中加入如下初始化：

```
1  c->exit_status = t->exit_status;
```

最后，在 `process_exit()` 函数中加入终止消息的打印语句：

```
1  /* ++2 Terminate Message */
2  int exit_status = current_thread->exit_status;
3  if (exit_status == INIT_EXIT_STAT)
4      exit_process(-1);
5
6
7  printf("%s: exit(%d)\n", current_thread->name, exit_status);
```

对于没有加载成功的进程，这里也进行了信息的打印处理。如果进程的状态仍然为 `INIT_EXIT_STAT`，即表示进程创建失败，直接将 -1 作为终止码终止进程。

## Mission 3 System Calls (系统调用)

### Requirements

官方文档对这一部分的要求如下：

### 3.3.4 System Calls

Implement the system call handler in ‘userprog/syscall.c’. The skeleton implementation we provide “handles” system calls by terminating the process. It will need to retrieve the system call number, then any system call arguments, and carry out appropriate actions.

也就是要实现 `syscall.c` 下的系统调用函数，完善 `syscall_handler` 的架构。

可以看见目前 `syscall.c` 内的这一函数只有如下语句：

```
1 static void
2 syscall_handler (struct intr_frame *f UNUSED)
3 {
4     printf ("system call!\n");
5     thread_exit ();
6 }
```

接下来查看 `lib/user/` 下的 `syscall_nr.h` 中的中断代号：

```
4 /* System call numbers. */      You, 11 days ago • Init project.
5 enum
6 {
7     /* Projects 2 and later. */
8     SYS_HALT,                  /* Halt the operating system. */
9     SYS_EXIT,                  /* Terminate this process. */
10    SYS_EXEC,                  /* Start another process. */
11    SYS_WAIT,                  /* Wait for a child process to die. */
12    SYS_CREATE,                /* Create a file. */
13    SYS_REMOVE,                /* Delete a file. */
14    SYS_OPEN,                  /* Open a file. */
15    SYS_FILESIZE,              /* Obtain a file's size. */
16    SYS_READ,                  /* Read from a file. */
17    SYS_WRITE,                 /* Write to a file. */
18    SYS_SEEK,                  /* Change position in a file. */
19    SYS_TELL,                  /* Report current position in a file. */
20    SYS_CLOSE,                 /* Close a file. */

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

24    /* Project 4 only. */
25    SYS_CHDIR,                 /* Change the current directory. */
26    SYS_MKDIR,                 /* Create a directory. */
27    SYS_READDIR,               /* Reads a directory entry. */
28    SYS_ISDIR,                 /* Tests if a fd represents a directory. */
29    SYS_INUMBER                /* Returns the inode number for a fd. */

30 };
31 
```

可以发现一系列的中断号码。后面关于虚拟内存和目录的终端代码是 Project 3 和 Project 4 的内容，对于 Project 2，需要实现前 20 行中的系统终端处理。

## Analysis

先分析如何使用 `syscall_handler` 处理不同的中断信息。

Pintos 已经先使用 `syscall_init` 将中断处理函数注册到寄存器：

```
1 void
2 syscall_init (void)
3 {
4     intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
5 }
```

在 `syscall_handler` 内部，考虑使用一个 `switch` 语句来让系统处理对应的终端代码。而为了获取到当前的中断是什么，阅读官方文档的指引：

In the 80x86 architecture, the ‘int’ instruction is the most commonly used means for invoking system calls. This instruction is handled in the same way as other software exceptions. In Pintos, user programs invoke ‘int \$0x30’ to make a system call. The system call number and any additional arguments are expected to be pushed on the stack in the normal fashion before invoking the interrupt (see Section 3.5 [80x86 Calling Convention], page 35).

Thus, when the system call handler `syscall_handler()` gets control, the system call number is in the 32-bit word at the caller’s stack pointer, the first argument is in the 32-bit word at the next higher address, and so on. The caller’s stack pointer is accessible to

`syscall_handler()` as the ‘`esp`’ member of the `struct intr_frame` passed to it. (`struct intr_frame` is on the kernel stack.)

传递个 `syscall_handler` 的 `intr_frame` 结构中的 `esp` 指针指向了一个整形数据，这个数据就是当前系统的中断值。每当一个系统中断发生的时候，就会占据 `esp` 指针下的 4 字节的空间。当处理完系统终端后，需要移除这部分的内存以表示处理完成，也能够让其他执行继续访问该内存。

因为涉及到系统调用，实验手册也要求对于用户程序要处理非法内存访问的问题。当出现以下情况：

- 访问空地址
- 访问内核部分的地址
- 访问无效的地址（如没有被映射到虚拟内存的内存区域）

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above PHYS\_BASE). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

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：

- `pagedir_get_page()`

```
1  /* Looks up the physical address that corresponds to user virtual
2   address UADDR in PD. Returns the kernel virtual address
3   corresponding to that physical address, or a null pointer if
4   UADDR is unmapped. */
5  void *
6  pagedir_get_page (uint32_t *pd, const void *uaddr)
7  {
8      uint32_t *pte;
9
10     ASSERT (is_user_vaddr (uaddr));
11
12     pte = lookup_page (pd, uaddr, false);
13     if (pte != NULL && (*pte & PTE_P) != 0)
14         return pte_get_page (*pte) + pg_ofs (uaddr);
15     else
16         return NULL;
17 }
```

- `is_user_vaddr()`

```
1  /* Returns true if VADDR is a user virtual address. */
2  static inline bool
3  is_user_vaddr (const void *vaddr)
4  {
5      return vaddr < PHYS_BASE;
6 }
```

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

除此之外，为了实现 `wait`，`exec` 这样涉及到子进程的调用和 `read`，`write` 等涉及到文件操作的系统调用，现有的 `thread` 的数据结构是不够用的。在后续分析到具体的系统调用时会尝试实现。

## Solution

在深入具体的系统调用之前，先处理非法内存访问的问题。在 `syscall.c` 中定义：

```

1 void *
2 is_valid_addr(const void *vaddr)
3 {
4     void *page_ptr = NULL;
5     if (!is_user_vaddr(vaddr) || !(page_ptr =
6         pagedir_get_page(thread_current()>pagedir, vaddr)))
7     {
8         exit_process(-1);
9     }
10    return page_ptr;
11 }

```

如果地址无效则以异常状态终止进程，有效则返回物理地址。

再编写一个 `pop_stack` 函数，用于从栈中取得元素，方便取得参数：

```

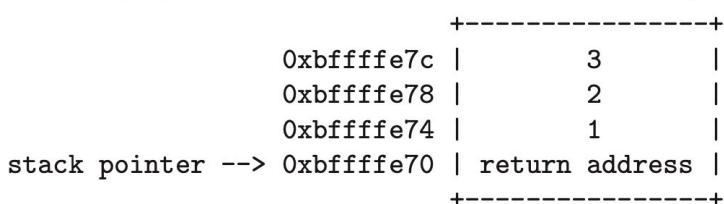
1 void pop_stack(int *esp, int *a, int offset){
2     int *tmp_esp = esp;
3     *a = *((int *)is_valid_addr(tmp_esp + offset));
4 }

```

稍后用到的所有弹出操作都需要用到这个函数。对于如下的栈结构，由于 pintos 使用的 4 字节对齐，函数后面的 `offset` 参数可以使用整数来方便地取到参数。

#### 6. The caller pops the arguments off the stack.

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:



接下来开始逐个完善系统调用。对于每一个系统调用的原型和介绍，在官方的实验手册上都有完整说明（[https://www.scs.stanford.edu/10wi-cs140/pintos/pintos\\_3.html](https://www.scs.stanford.edu/10wi-cs140/pintos/pintos_3.html)）这里为了避免实验报告篇幅过长，不再引用原文。

#### void halt (void)

简单地调用 `devices/shutdown.c` 下的 `shutdown_power_off()` 函数：

```

1 void syscall_halt(void){
2     shutdown_power_off();
3 }

```

```
pid_t exec (const char *cmd_line)
```

为了顺利取得子进程的状态以返回子进程的 pid, 我们需要在 `thread` 结构体加入如下定义:

```
1  bool load_success; //if the child process is loaded successfully
2  struct semaphore load_sema; // semaphore to keep the thread waiting
3  until it makes sure whether the child process if successfully loaded.
4
5  struct list children_list;
6  struct thread* parent;
7  struct child_process * waiting_child; //pid of the child process it
8  is currently waiting
```

其中, `child_process` 是自行定义的一个用于存储父进程正在等待的子进程信息的结构体。同样在 `thread.h` 中定义如下:

```
1  /* ++ 2 System Call */
2  struct child_process {
3      int tid;
4      struct list_elem child_elem; // element of itself point to its
5      parent's child_list
6      int exit_status; //store its exit status to pass it to its parent
7      bool if_waited; // whether the child process has been waited()
8      struct semaphore wait_sema;
9  };
```

相应的在 `init_thread` 中对其初始化:

```
1  /* ++ 2 */
2  list_init (&t->children_list);
3  t->parent = running_thread();
4  t->exit_status = INIT_EXIT_STAT;
5  sema_init(&t->load_sema, 0);
6  t->waiting_child=NULL;
7  t->self=NULL;
```

其中 `children_list` 和 `parent` 对于后面 `wait` 的系统调用也很重要, 这两个数据成员的初始化保证了等待系统调用的正确执行。

当 `syscall_exec(file_name)` 被调用的时候。首先应当确认传递来的参数 `file_name` 是否为有效的。如果有效, 则执行 `process_execute()`。

先简单编写系统调用函数如下:

```
1 int
2 syscall_exec(struct intr_frame *f)
3 {
4     char *file_name = NULL;
5     pop_stack(f->esp, &file_name, 1);
6     if (!is_valid_addr(file_name))
7         return -1;
8
9     return exec_process(file_name);
10 }
```

子函数 `exec_process()` 用于解决该调用的核心流程。

整个执行过程都是通过之前已经定义的 `filesys_lock` 来实现互斥访问的。由于 Pintos 的文件系统内不是线程安全的，所以为了保证不并行执行多个文件访问操作，需要使用这个锁。具体的代码如下：

```
1 int
2 exec_process(char *file_name)
3 {
4     int tid;
5     lock_acquire(&filesys_lock);
6     char * name_tmp = malloc (strlen(file_name)+1);
7     strlcpy(name_tmp, file_name, strlen(file_name) + 1);
8
9     char *tmp_ptr;
10    name_tmp = strtok_r(name_tmp, " ", &tmp_ptr);
11
12    struct file *f = filesys_open(name_tmp); // check whether the file
exists. critical to test case "exec-missing"
13
14    if (f == NULL)
15    {
16        lock_release(&filesys_lock);
17        tid = -1;
18    }
19    else
20    {
21        file_close(f);
22        lock_release(&filesys_lock);
23        tid = process_execute(file_name);
24    }
25    return tid;
26 }
```

因为进程可能创建失败，`tid` 可能会是失败的。所以父进程应当等待子进程的创建来确保其是否创建成功。所以这里使用的是定义的另一个信号量 `load_sema` 来实现。当子进程开始创建时，会对自己的 `load_sema` 进行 P 操作，使得父进程等待。当子进程成功创建后，会对 `load_sema` 进行 V 操作，父进程停止等待。

所以对于 `process.c` 还需要做如下修改：

在 `process_execute` 中加入 P 操作：

```
1  tid = thread_create (thread_name, PRI_DEFAULT, start_process,
2  fn_copy);
3  free(thread_name); //free the file name created by malloc mannually
4  if (tid == TID_ERROR)
5      palloc_free_page (fn_copy);
6  else
7  {
8      sema_down(&current_thread->load_sema); //keep the thread waiting
9      until start_process() exits.
10     if (!current_thread->load_success) //if the child process is not
11         loaded successfully
12         return -1;
13 }
```

在 `process_start` 中加入 V 操作：

```
1  struct thread * current_thread = thread_current();
2  current_thread->parent->load_success = success;
3
4  if (!success) {
5      ASSERT(current_thread->parent->exit_status==INIT_EXIT_STAT)
6      /* exit_status now should be INIT_EXIT_STAT handle later,
7      becasuse process start fail, and exit_status init value is
8      INIT_EXIT_STAT. */
9      thread_exit();
10     sema_up(&current_thread->parent->load_sema);
```

`int wait (pid t pid)`

当进程系统调用 `wait` 时，将会等待子进程退出并且返回子进程的 `pid`，取得子进程的退出状态。所以在该调用中，需要首先获取子进程的 `tid`：

```

1 int
2 syscall_wait(struct intr_frame *f)
3 {
4     tid_t child_tid;
5     pop_stack(f->esp, &child_tid, 1);
6     return process_wait(child_tid);
7 }

```

接着阻塞在 `process_wait`。首先判断 `children_list` 是否为空。如果为空则无需等待进程并直接返回。这里同样使用了一个 `sema_down` 信号量用于实现子进程与父进程之间的同步问题。相应的，需要在 `thread_exit` 函数中，即进程成功退出了之后，进行对 `wait_sema` 的 V 操作：

```

1 /* Deschedules the current thread and destroys it. Never
2      returns to the caller. */
3 void
4 thread_exit (void)
5 {
6     ASSERT (!intr_context ());
7     /* ++2 System Call */
8     enum intr_level old_level = intr_disable ();
9     if (thread_current ()->parent->waiting_child != NULL)
10    {
11        if (thread_current ()->parent->waiting_child->tid ==
12            thread_current ()->tid)
13            sema_up (&thread_current ()->parent->waiting_child->wait_sema);
14    }
15    intr_set_level (old_level);
16    .....

```

完善后的 `process_wait()` 函数应当如下，包含了对 `wait_sema` 的 P 操作：

```

1 int
2 process_wait (tid_t child_tid)
3 {
4     struct thread *current_thread = thread_current ();
5
6     enum intr_level old_level = intr_disable ();
7     struct list_elem *tmp_e = find_child_proc (child_tid);
8     struct child_process *ch = list_entry (tmp_e, struct child_process,
9                                           child_elem);
10    intr_set_level (old_level);
11    if (!ch || !tmp_e)

```

```

12     return -1;
13
14     current_thread->waiting_child = ch;
15     //current_thread->waiting_child = ch;
16
17     if(!ch->if_waited){
18         sema_down(&ch->wait_sema);
19
20         //ch->if_waited=true;
21     }
22     // else    //if the child process has been waited
23     //    return -1;
24
25     list_remove(tmp_e);
26
27     return ch->exit_status;
28 }
```

其中，用于寻找子进程的 `find_child_proc(child_tid)` 函数也应当自行编写并定义在 `thread.c` 中，并将函数原型加入 `thread.h`：

```

1  /* ++2 System Call */
2  struct list_elem *
3  find_child_proc(tid_t child_tid)
4  {
5      ASSERT (intr_get_level () == INTR_OFF);
6
7      struct list_elem *tmp_e;
8
9      for (tmp_e = list_begin (&thread_current()->children_list); tmp_e
10          != list_end (&thread_current()->children_list);
11          tmp_e = list_next (tmp_e))
12      {
13          struct child_process *f = list_entry (tmp_e, struct
14          child_process, child_elem);
15          if(f->tid == child_tid)
16          {
17              return tmp_e;
18          }
19      }
20      return NULL;
21 }
```

对于 `process.c` 中的函数 `process_exit()`，需要增加以下语句：

```

1  printf("%s: exit(%d)\n", current_thread->name, exit_status);
2
3  if (lock_held_by_current_thread(&filesys_lock))
4  {
5      lock_release(&filesys_lock);
6  }
7
8  lock_acquire(&filesys_lock);
9  clean_all_files(&current_thread->opened_files);
10
11 file_close(current_thread->self);
12
13
14 struct list_elem *elem_pop;
15 while(!list_empty(&thread_current()->children_list))
16 {
17     elem_pop = list_pop_front(&thread_current()->children_list);
18     struct process_file *f = list_entry(elem_pop, struct
19         child_process, child_elem);
20     free(f);
21 }
22 lock_release(&filesys_lock);

```

在终止的时候释放所有锁，并且调用 `clean_all_files()` 来关闭开启进程的所有文件。函数 `clean_all_files` 应当编写在 `syscall.c` 下，定义如下：

```

1 void
2 clean_all_files(struct list* files)
3 {
4     struct process_file *proc_f;
5     while(!list_empty(files))
6     {
7         proc_f = list_entry(list_pop_front(files), struct process_file,
8             elem);
9         file_close(proc_f->ptr);
10        list_remove(&proc_f->elem);
11        free(proc_f);
12    }
13 }

```

`void exit (int status)`

终止进程，从栈顶获取进程终止状态，终止进程：

```

1 void
2 syscall_exit(struct intr_frame *f)
3 {
4     int status;
5     pop_stack(f->esp, &status, 1);
6     exit_process(status);
7 }

```

子函数 `exit_process` 的定义如下。将当前进程的状态设置成 `status`，然后依次遍历子进程，更新子进程和父进程的执行状态，最后调用 `thread_exit()`。之所以调用 `thread_exit` 而不直接调用 `process_exit` 是因为前者会进行一些对信号量的同步操作，并且不仅可以用于终止用户进程，也可以以终止系统程序。如果调用者是用户程序，也会直接调用 `process_exit()`。

```

1 void
2 exit_process(int status)
3 {
4     struct child_process *cp;
5     struct thread *cur_thread = thread_current();
6
7     enum intr_level old_level = intr_disable();
8     for (struct list_elem *e = list_begin(&cur_thread->parent-
9         >children_list); e != list_end(&cur_thread->parent->children_list); e =
10        = list_next(e))
11     {
12         cp = list_entry(e, struct child_process, child_elem);
13         if (cp->tid == cur_thread->tid)
14         {
15             cp->if_waited = true;
16             cp->exit_status = status;
17         }
18     }
19     cur_thread->exit_status = status;
20     intr_set_level(old_level);
21 }

```

`bool create (const char *file, unsigned initial_size)`

接下来的一些 System Call 都设计到了文件系统的操作，因此之前的数据结构又不够用了。所以为了维护每个进程打开了的文件列表，统计每个进程打开了的文件数目，需要再次完善 `thread` 的结构体中的数据成员。在 `thread` 结构体中加入如下定义：

```
1 struct list opened_files;      //all the opened files
2 int fd_count;
```

相应地在 `thread_init` 中初始化：

```
1 list_init (&t->opened_files);
2 t->fd_count=2;
```

除此之外，为了能够储存正在处理的文件的文件指针和文件描述符，以及文件在 `opened_files` 列表中的储存位置，需要在 `syscall.h` 中新建一个 `process_file` 结构体指针：

```
1 struct process_file {
2     struct file* ptr;
3     int fd;
4     struct list_elem elem;
5 }
```

有了上面的结构体，可以编写 `create` 的系统调用如下。在取得调用参数中的初始化文件大小和文件名字之后，将会调用 `filesys.c` 中的 `filesys_create` 创建文件，在创建过程中使用 `filesys_lock` 来保证同步操作。创建文件的调用不包含打开文件。打开文件的调用将会由单独的系统内调用实现：

```
1 int
2 syscall_create(struct intr_frame *f)
3 {
4     int ret;
5     off_t initial_size;
6     char *name;
7
8     pop_stack(f->esp, &initial_size, 5);
9     pop_stack(f->esp, &name, 4);
10    if (!is_valid_addr(name))
11        ret = -1;
12
13    lock_acquire(&filesys_lock);
14    ret = filesystem_create(name, initial_size);
15    lock_release(&filesys_lock);
16    return ret;
17 }
```

`bool remove (const char *file)`

用于删除文件。

无论文件是打开状态还是关闭状态，都会直接删除文件，并且在删除打开的文件的时候不会先关闭。

```
1 int
2 syscall_remove(struct intr_frame *f)
3 {
4     int ret;
5     char *name;
6
7     pop_stack(f->esp, &name, 1);
8     if (!is_valid_addr(name))
9         ret = -1;
10
11    lock_acquire(&filesys_lock);
12    if (filesys_remove(name) == NULL)
13        ret = false;
14    else
15        ret = true;
16    lock_release(&filesys_lock);
17
18    return ret;
19 }
```

同样是在操作过程中将 `filesys_remove` 嵌套在对 `filesys_lock` 的锁访问中，在执行操作之前先检测文件是否指向一个有效地址，如果无效则直接退出。

`int open (const char *file)`

打开一个文件，并返回一个文件描述符 `fd`。

保证互斥的情况下调用 `filesys_open`，然后将取得的文件和文件指针存储到之前定义的 `process_file` 结构体中，添加到进程的文件打开列表内。

具体的实现如下：

```
1 int
2 syscall_open(struct intr_frame *f)
3 {
4     int ret;
5     char *name;
6
7     pop_stack(f->esp, &name, 1);
8
9     /* ++2 Hack open-empty */
10    if (!is_valid_addr(name) || strlen(name) == 0){
```

```
11     ret = -1;
12     return ret;
13 }
14
15
16 lock_acquire(&filesys_lock);
17 struct file *fptr = filesys_open(name);
18 lock_release(&filesys_lock);
19
20 if (fptr == NULL)
21     ret = -1;
22 else
23 {
24     struct process_file *pfile = malloc(sizeof(*pfile));
25     pfile->ptr = fptr;
26     pfile->fd = thread_current()->fd_count;
27     thread_current()->fd_count++;
28     list_push_back(&thread_current()->opened_files, &pfile->elem);
29     ret = pfile->fd;
30 }
31 return ret;
32 }
33 }
```

文件描述符 0 用于表示 `STDOUT_FILENO`，1 表示 `STDIN_FILENO`，即标准输入流和标准输出流。文件描述符的返回值永远不会返回这两个数值，而且这两个文件描述符始终为所有线程打开，所以之前在初始化线程的打开文件的数量的时候也是初始化为 2。

`int filesize (int fd)`

返回文件的大小。

```
1 int
2 syscall_filesize(struct intr_frame *f)
3 {
4     int ret;
5     int fd;
6     pop_stack(f->esp, &fd, 1);
7
8     lock_acquire(&filesys_lock);
9     ret = file_length(search_fd(&thread_current()->opened_files, fd)->ptr);
10    lock_release(&filesys_lock);
11
12    return ret;
13 }
```

子函数 `search_fd` 用于根据文件描述符来返回一个 `process_file` 的结构体，可以用这个结构体来获取到文件的内存地址。该函数实现如下：

```
1 struct process_file *
2 search_fd(struct list* files, int fd)
3 {
4     struct process_file *proc_f;
5     for (struct list_elem *e = list_begin(files); e != list_end(files);
6          e = list_next(e))
6     {
7         proc_f = list_entry(e, struct process_file, elem);
8         if (proc_f->fd == fd)
9             return proc_f;
10    }
11    return NULL;
12 }
```

`int read (int fd, void *buffer, unsigned size)`

根据文件描述符 `fd` 将制定的 `size` 字节数据读入到缓存当中，返回实际读取到的字节数或者 -1（当文件无法读取的时候）。

```
1 int
2 syscall_read(struct intr_frame *f)
3 {
4     int ret;
5     int size;
6     void *buffer;
7     int fd;
```

```

8
9     pop_stack(f->esp, &size, 7);
10    pop_stack(f->esp, &buffer, 6);
11    pop_stack(f->esp, &fd, 5);
12
13    if (!is_valid_addr(buffer))
14        ret = -1;
15
16    if (fd == 0)
17    {
18        int i;
19        uint8_t *buffer = buffer;
20        for (i = 0; i < size; i++)
21            buffer[i] = input_getc();
22        ret = size;
23    }
24    else
25    {
26        struct process_file *pf = search_fd(&thread_current()-
27>opened_files, fd);
27        if (pf == NULL)
28            ret = -1;
29        else
30        {
31            lock_acquire(&filesys_lock);
32            ret = file_read(pf->ptr, buffer, size);
33            lock_release(&filesys_lock);
34        }
35    }
36
37    return ret;
38 }

```

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

根据文件描述符 `fd` 从指定的缓存中写入特定字节大小到文件中。返回实际写入的字节数。

需要注意的是如果写过了 `EOF`，通常的解决方案是要扩大文件。但目前文件系统的实现是将文件大小固定了的，所以我们的实现应当是尽可能多地写入字节。

完整的代码如下：

```

1  int
2  syscall_write(struct intr_frame *f)

```

```

3  {
4      int ret;
5      int size;
6      void *buffer;
7      int fd;
8
9      pop_stack(f->esp, &size, 7);
10     pop_stack(f->esp, &buffer, 6);
11     pop_stack(f->esp, &fd, 5);
12
13     if (!is_valid_addr(buffer))
14         ret = -1;
15
16     if (fd == 1)
17     {
18         putbuf(buffer, size);
19         ret = size;
20     }
21     else
22     {
23         enum intr_level old_level = intr_disable();
24         struct process_file *pf = search_fd(&thread_current()-
>opened_files, fd);
25         intr_set_level (old_level);
26
27         if (pf == NULL)
28             ret = -1;
29         else
30         {
31             lock_acquire(&filesys_lock);
32             ret = file_write(pf->ptr, buffer, size);
33             lock_release(&filesys_lock);
34         }
35     }
36
37     return ret;
38 }
```

当 `fd==1` 时，写入标准输出流。及调用 pintos 预制的 `putbuf()` 函数实现，否则则在保证互斥的情况下调用 `filesys_write` 实现。

`void seek (int fd, unsigned position)`

查找调用，即改变下一个要进行读 / 写的字节位置。

当读的位置超过了 `EOF` 时不能报错，而是读取到 0 字节，以表示已经到达了文件结尾。而写的操作超过了 `EOF` 时将会抛出错误。（由 `file_seek()` 已经实现）

完整的代码如下：

```
1 void
2 syscall_seek(struct intr_frame *f)
3 {
4     int fd;
5     int pos;
6     pop_stack(f->esp, &fd, 5);
7     pop_stack(f->esp, &pos, 4);
8
9     lock_acquire(&filesys_lock);
10    file_seek(search_fd(&thread_current()->opened_files, pos)->ptr,
11               fd);
12    lock_release(&filesys_lock);
13 }
```

`unsigned tell (int fd)`

取得文件的读取位置，返回值是下一个将被读取或者写入的字节位置。

```
1 int
2 syscall_tell(struct intr_frame *f)
3 {
4     int ret;
5     int fd;
6     pop_stack(f->esp, &fd, 1);
7
8     lock_acquire(&filesys_lock);
9     ret = file_tell(search_fd(&thread_current()->opened_files, fd)-
10                  >ptr);
11    lock_release(&filesys_lock);
12
13    return ret;
14 }
```

`void close (int fd)`

最后一个系统调用是要实现文件的关闭操作。该调用将会在任意进程终止的时候依次调用。

```
1 void
2 syscall_close(struct intr_frame *f)
```

```

3  {
4      int fd;
5      pop_stack(f->esp, &fd, 1);
6
7      /* ++2 Hack close-stdout & close-badfd */
8      if(fd == 1 || fd == 0x20101234){
9          exit_process(-1);
10     }
11
12     lock_acquire(&filesys_lock);
13     clean_single_file(&thread_current()->opened_files, fd);
14     lock_release(&filesys_lock);
15 }
```

嵌套的子函数 `clean_single_file` 用于关闭一个指定的文件，不需要再重复用 `filesys_lock` 处理互斥问题：

```

1 void
2 clean_single_file(struct list* files, int fd)
3 {
4     struct process_file *proc_f = search_fd(files, fd);
5     if (proc_f != NULL){
6         file_close(proc_f->ptr);
7         list_remove(&proc_f->elem);
8         free(proc_f);
9     }
10 }
```

## Structure

实现了要求的所有系统调用之后，在 `handler` 中使用 `switch` 语句处理这些中断信号：

```

1 static void
2 syscall_handler (struct intr_frame *f UNUSED)
3 {
4     int *p = f->esp;
5     is_valid_addr(p);
6
7     int system_call = *p;
8     switch (system_call)
9     {
10     case SYS_HALT: syscall_halt(); break;
11     case SYS_EXIT: syscall_exit(f); break;
12     case SYS_EXEC: f->eax = syscall_exec(f); break;
13     case SYS_WAIT: f->eax = syscall_wait(f); break;
```

```

14     case SYS_CREATE: f->eax = syscall_create(f); break;
15     case SYS_REMOVE: f->eax = syscall_remove(f); break;
16     case SYS_OPEN: f->eax = syscall_open(f); break;
17     case SYS_FILESIZE: f->eax = syscall_filesize(f); break;
18     case SYS_READ: f->eax = syscall_read(f); break;
19     case SYS_WRITE: f->eax = syscall_write(f); break;
20     case SYS_SEEK: syscall_seek(f); break;
21     case SYS_TELL: f->eax = syscall_tell(f); break;
22     case SYS_CLOSE: syscall_close(f); break;
23
24     default:
25         printf("Default %d\n", *p);
26     }
27 }
```

并将刚刚实现的函数原型定义在 `syscall.c` 中：

```

1 int exec_process(char *file_name);
2 void exit_process(int status);
3 void * is_valid_addr(const void *vaddr);
4 struct process_file* search_fd(struct list* files, int fd);
5 void clean_single_file(struct list* files, int fd);
6 void syscall_exit(struct intr_frame *f);
7 int syscall_exec(struct intr_frame *f);
8 int syscall_wait(struct intr_frame *f);
9 int syscall_create(struct intr_frame *f);
10 int syscall_remove(struct intr_frame *f);
11 int syscall_open(struct intr_frame *f);
12 int syscall_filesize(struct intr_frame *f);
13 int syscall_read(struct intr_frame *f);
14 int syscall_write(struct intr_frame *f);
15 void syscall_seek(struct intr_frame *f);
16 int syscall_tell(struct intr_frame *f);
17 void syscall_close(struct intr_frame *f);
18 void syscall_halt(void);
```

## Mission 4 Denying Writes to Executables (拒接写可执行文件)

### Requirements

Add code to deny writes to files in use as executables. Many OSes do this because of the unpredictable results if a process tried to run code that was in the midst of being changed on disk. This is especially important once virtual memory is implemented in project 3, but it can't hurt even now.

You can use `file_deny_write()` to prevent writes to an open file. Calling `file_allow_write()` on the file will re-enable them (unless the file is denied writes by another opener).

最后一个要求是不能允许进程写正在执行的进程文件。

## Analysis

在进程创建的时候只要调用 `file_deny_write()` 使该文件无法被写，在进程结束的时候调用 `file_allow_write()` 使该文件重新允许被写即可。

这两个函数已经在 `filesys` 中实现。

## Solution

在 `process.c` 的 `load` 函数中加入语句：

```
1  /* ++2 Deny Write */
2  file_deny_write(file);
```

即可禁止其他进程对该正在运行的文件进行写操作。

在 `process.c` 的 `process_exit()` 函数中加入语句：

```
1  /* ++2 Allow write */
2  file_allow_write(current_thread->self);
```

## Result

至此，系统调用部分已经全部实现完成。再次运行

```
1  make SIMULATOR=--bochs check
```

得到如下运行结果：

```
1  pass tests/filesys/base/syn-write
2  pass tests/userprog/args-none
3  pass tests/userprog/args-single
4  pass tests/userprog/args-multiple
5  pass tests/userprog/args-many
6  pass tests/userprog/args-dbl-space
7  pass tests/userprog/sc-bad-sp
8  pass tests/userprog/sc-bad-arg
9  pass tests/userprog/sc-boundary
```

```
10  pass tests/userprog/sc-boundary-2
11  pass tests/userprog/halt
12  pass tests/userprog/exit
13  pass tests/userprog/create-normal
14  pass tests/userprog/create-empty
15  pass tests/userprog/create-null
16  pass tests/userprog/create-bad-ptr
17  pass tests/userprog/create-long
18  pass tests/userprog/create-exists
19  pass tests/userprog/create-bound
20  pass tests/userprog/open-normal
21  pass tests/userprog/open-missing
22  pass tests/userprog/open-boundary
23  pass tests/userprog/open-empty
24  pass tests/userprog/open-null
25  pass tests/userprog/open-bad-ptr
26  pass tests/userprog/open-twice
27  pass tests/userprog/close-normal
28  pass tests/userprog/close-twice
29  pass tests/userprog/close-stdin
30  pass tests/userprog/close-stdout
31  pass tests/userprog/close-bad-fd
32  pass tests/userprog/read-normal
33  pass tests/userprog/read-bad-ptr
34  pass tests/userprog/read-boundary
35  pass tests/userprog/read-zero
36  pass tests/userprog/read-stdout
37  pass tests/userprog/read-bad-fd
38  pass tests/userprog/write-normal
39  pass tests/userprog/write-bad-ptr
40  pass tests/userprog/write-boundary
41  pass tests/userprog/write-zero
42  pass tests/userprog/write-stdin
43  pass tests/userprog/write-bad-fd
44  pass tests/userprog/exec-once
45  pass tests/userprog/exec-arg
46  pass tests/userprog/exec-multiple
47  pass tests/userprog/exec-missing
48  pass tests/userprog/exec-bad-ptr
49  pass tests/userprog/wait-simple
50  pass tests/userprog/wait-twice
51  pass tests/userprog/wait-killed
52  pass tests/userprog/wait-bad-pid
53  FAIL tests/userprog/multi-recuse
54  pass tests/userprog/multi-child-fd
```

```
55  pass tests/userprog/rox-simple
56  pass tests/userprog/rox-child
57  pass tests/userprog/rox-multichild
58  pass tests/userprog/bad-read
59  pass tests/userprog/bad-write
60  pass tests/userprog/bad-read2
61  pass tests/userprog/bad-write2
62  pass tests/userprog/bad-jump
63  pass tests/userprog/bad-jump2
64  FAIL tests/userprog/no-vm/multi-oom
65  pass tests/filesys/base/lg-create
66  pass tests/filesys/base/lg-full
67  pass tests/filesys/base/lg-random
68  pass tests/filesys/base/lg-seq-block
69  pass tests/filesys/base/lg-seq-random
70  pass tests/filesys/base/sm-create
71  pass tests/filesys/base/sm-full
72  pass tests/filesys/base/sm-random
73  pass tests/filesys/base/sm-seq-block
74  pass tests/filesys/base/sm-seq-random
75  pass tests/filesys/base/syn-read
76  pass tests/filesys/base/syn-remove
77  pass tests/filesys/base/syn-write
78  2 of 76 tests failed.
```

终端命令行截图如下：

```
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
2 of 76 tests failed.
make[1]: *** [check] Error 1
make: *** [check] Error 2
✖ billchen@Bills-MacBook-Pro-2018 ~/OneDrive/Workspace/Pintos/src/userprog ↵ master •
```

76 个测试中，只有 2 个测试 `multi-oom` 和 `multi-recurse` 未通过。

至此，Project 2 完成。

## Remark

Project 2 也算是完成了。相比 Project 1 的在原系统基础上的小修小改，Project 2 自由度更高，代码量也更大，大多数地方都需要自己从 0 开始手动实现。这对我来说是一个更大的挑战。在实现的过程中查阅了无数资料，官方的实验文档和实验指导也翻来覆去看了好几遍，花费了无数的心血和时间。但看到 `make check` 时屏幕上快速跳动的 `pass` 时，熬过的夜晚都变得值得了。

通过实现参数传递，我对程序运行时的内存结构有了更深刻的理解。从细节入手，深究栈在程序运行时的构建过程，使得堆栈的结构不再停留在书本中，而是需要自己在代码中一个指针一个指针地拼凑出来，这个过程的意义是非常重大的。

而系统调用部分的实现我尝试面向测试来做，一个一个测试地查看其中的问题所在，能够发现很多第一次写的调用函数的不完善的地方。例如对无效指针的引用，对标准输入输出的读写，以及重复执行相同操作的返回值等。尤其是为了解决同步问题，贯穿整个 Project 2 的几个信号量的使用，让我形象地体会到了 P / V 操作的实用性与必要性。通过一步一步地完善代码，让程序变得更健壮，能适应更多的情况，我也对操作系统的高稳定要求有了更深刻的理解。

我深知 Pintos 只是一个简单的操作系统，在真实的生产环境中一定还有更复杂、更困难的挑战要去面对。但通过这第一次和操作系统内核的深度接触，我对操作系统的面貌有了大概的了解，希望在未来的学习与工作中能够继续深入自己的知识水平，能够更加得心应手地应对各种问题。

---

2020.1.2 (Thu.) 2:36 AM

*To Whom Reading This: Happy New Decade :-)*