# Pintos project2 user programs实验讲解

[Pintos project2 user programs实验讲解 1](#_Toc81853301)

[实验的意义探究 1](#_Toc81853302)

[实验思路讲解 2](#_Toc81853303)

[前置任务：打印进程退出信息 4](#_Toc81853304)

[任务1：进程参数 4](#_Toc81853305)

[任务2：系统调用 6](#_Toc81853306)

[2.1 调用参数所处内存地址 6](#_Toc81853307)

[2.2打开关闭文件 7](#_Toc81853308)

[2.3创建移除文件 8](#_Toc81853309)

[2.4读写文件 9](#_Toc81853310)

[2.5执行与等待 10](#_Toc81853311)

[2.6 multi 11](#_Toc81853312)

[2.7 rox 12](#_Toc81853313)

[2.8 bad 13](#_Toc81853314)

[2.9 资源释放 14](#_Toc81853315)

[2.10 文件系统 15](#_Toc81853316)

## 实验的意义探究

我们知道，pintos实验旨在通过补充代码，实现一个小型的操作系统。那么这个操作系统，到底是什么级别的系统呢？

这个需要从intel公司的80x86系列处理器说起。通过之前计组和系统编程已经学习到的知识，大家应该都知道8086处理器是一个16位处理器，系统总线20根，可寻址最大内存空间达1MB，范围是00000H-FFFFFH，但其内部的寄存器和ALU都只有16位，所以在寻址时，需要将段地址乘以16，再加上偏移地址，以此来实现20位的寻址。这都是后话，最主要的是，在8086时代，用户可以直接寻址到内存空间的任何位置，这就有可能导致用户错误地修改某些地址值，导致系统崩溃。

由于技术的进步以及人们对多用户操作同一台电脑的需求，在80386时代，终于将系统的工作方式给分离了，80386具有实模式、保护模式和v68模式这三种工作模式。其中，实模式其实算是8086体系的延续，在该模式下的寻址方式和8086无异，这个模式也是为了兼容16位体系下的程序。而用户一般所处的状态就是保护模式，在这个模式下可以使用386系统的32根主线，寻址模式发生根本性的变化，最主要的是引入了特权级的概念，由于我们知道在8086时代，随意的内存修改可能导致系统崩溃，而想要避免这种崩溃，就需要把系统内核给保护起来，为此，386有个4个特权级r0-3，我们的系统内核在最高级别r0中，特权级所处的内存空间分置，允许高特权级访问低特权级内存空间，而不允许反向地访问。那么这样一来，确实避免了随意对内存空间的修改，可又引出了一些新的问题，比如系统对外设端口的调用，是通过中断的方式进行的；比如创建新的进程等，这些函数都被内核给把持着，当一个用户想要手动开启这些功能的时候，就遇到了无法访问的问题，这个时候，就需要让低特权级向高特权级发起一个建议，这个建议我们把他叫做系统调用，使用系统调用，就可以让用户在低特权级下使用高特权级的功能。

而这，就是我们第二次实验所要完成的事情。

## 实验思路讲解

大部分思路可以参照官方文档<http://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html#SEC40>中的3.2章节。

在本次实验中，我们需要完成10余个系统调用的实现。首先系统调用本质上会落实到函数中，函数需要有参数，而通过计组的学习，我们已经知道，这些参数被放在栈中，当eip执行到call指令时，会将当前栈顶地址压栈，再将栈底地址赋值给栈顶，而这个时候，参数就被存放在栈底之下。我们的pintos实际上也是类似的操作。

为了让参数的传递不发生错误，我们首先需要在process.c中修改setup\_stack函数中\*esp = PHYS\_BASE为\*esp = PHYS\_BASE-12 。这一步修改的原因会在后续步骤中讲到。

本实验目标是实现系统调用，所以大部分代码都完成在syscall函数中，可以看到源代码给了我们一个syscall\_handler的框架，我们可以在这里将官方文档3.3.4中提及的系统调用函数框架搭在这里。

要想执行系统调用，肯定需要参数，前面已经提及，参数被存放在了栈底之下，而在pintos中，这个存放的位置在指针esp下，我们通过syscall的参数f这个栈帧，可以取得对应esp指针的值。栈帧f是一个名为intr\_frame的结构，可以在threads/interrupt.h文件中找到，从名字我们就可以看出，这是中断时保存的现场，其内部定义了32位机器常用的8大寄存器的值。esp就是其中之一。Pintos文档约定参数最多不超过3个，所以由此，我们就可以写出从esp位置，获取到参数的函数。

获取完参数，我们就可以正式开始实现系统调用。

从官方文档3.3.4中，我们可以看到需要完成的所有系统调用，以及其需要的参数格式，包括其实现方式。

例如第一个halt()系统调用，起终止pintos系统的功能，只需要调用threads/init.h中的shutdown\_power\_off函数，实现模拟断电即可。

第二个函数exit(int status)，需要一个返回码的参数，我们都知道，返回码标志着函数进程的退出状态，是否正常退出，这个值最终由父进程回收。我们实现时，只需要从esp附近取得这个值，并传递给处理函数即可。

为了验证我们是否成功完成这一系列操作，下一步可以先行实现write函数的一部分，即向控制台输出内容，还是根据3.3.4的文件内容，我们可以了解到调用putbuf函数，即可将内容写入缓冲区，最终输出到控制台，该函数被定义在lib/kernel/console.c中，同时当参数fd为1时，表示需要向控制台输出。Write函数与前两者不一样的是，它需要一个返回值来标志写入的个数，我们需要设计函数的返回值，并将其写入栈帧的eax中。

到这里，我们就可以开始尝试过点了，但为了防止pintos的主进程在测试进程结束前就关机，（因为还没有实现wait做到父进程等待子进程）所以我们可以在userprog/process.c中，将process\_wait添加一个sleep语句，以保证子进程顺利结束。

完成了这些，就可以运行make check，检查第一个测试用例是否有输出，如果有的话，说明前期的准备都做对了，可以开始一个一个实现系统调用了。

在过点的过程中，大家可以尝试查看测试用例的源代码，以此来获取这些测试用例分别是对应什么系统实现。

我们开始分任务讲解。

**开始前的准备**

按照实验思路的顺序，

1.首先需要在process.c中修改setup\_stack函数中\*esp = PHYS\_BASE为\*esp = PHYS\_BASE-12 。这一步是为了人为给系统调用参数留下空间，防止访问越界。

2.其次是搭建框架，在sys call.c函数中，我们可以为10余个系统调用搭建框架，以便于以后直接修改函数就能完成。

3.获取参数，我们在系统调用时，需要取得用户传入的参数，我们通过syscall的参数f这个栈帧，可以取得对应esp指针的值。参数就存放在这里，同时数量也可以被设置为一个参数，幸运的是，pintos系统调用约定参数不超过3个。

4.实现简单的exit，为了能让进程顺利结束，肯定需要先完成退出，我们在syscall.c中，只需要调用thread\_exit(); 就可以实现简单的退出。

5.向console打印信息，测试点的输出结果是靠用户调用wirte这一系统调用完成的，所以想要成功获得过点情况，就需要先实现一部分write调用。根据3.3.4的文件内容，我们可以了解到调用putbuf函数，即可将内容写入缓冲区，最终输出到控制台，该函数被定义在lib/kernel/console.c中，同时当参数fd为1时，表示需要向控制台输出。Write函数与前两者不一样的是，它需要一个返回值来标志写入的个数，我们需要设计函数的返回值，并将其写入栈帧的eax中。

6.防止wait的立即结束。为了防止pintos的主进程在测试进程结束前就关机，（因为还没有实现wait做到父进程等待子进程）所以我们可以在userprog/process.c中，将process\_wait添加一个sleep语句，以保证子进程顺利结束。

### 前置任务：打印进程退出信息

Pintos要求每一次退出时都打印退出代码，没有这个打印行为，几乎所有的点都无法通过，但其本身没有点。首先我们需要在threads/thread.h函数中修改thread结构体，为其添加退出码这个值，其次我们在系统调用exit时，需要将退出码存储在结构体中，最后在userprog/process.c中process\_exit（）函数中按文档所给的格式（见3.3.2）打印退出码。（这里需要解释一点，pintos实验做到这里时，仍没有区分线程和进程的概念，一定程度上可以混用）。

### 任务1：进程参数

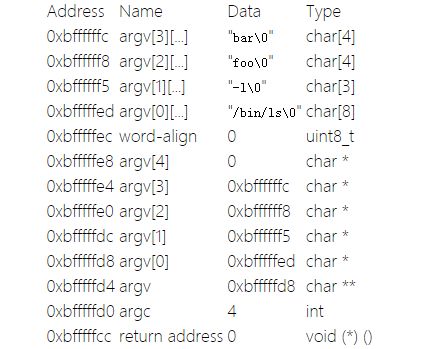
|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 1 | args-none | 命令没有任何的参数，检查第 0 个参数是不是文件名，运行命令为args-none。 |
| 2 | args-single | 有一个参数，检查参数数量和内容是否正确，运行命令为args-single onearg。 |
| 3 | args-multiple | 一共有 4 个参数，运行命令为args-multiple some arguments for you!。 |
| 4 | args-many | 一共有 22 各参数，运行命令为args-many a b c d e f g h i j k l m n o p q r s t u v。 |
| 5 | args-dbl-space | 有参数用两个空格间隔，确保要识别正确，运行命令为args-dbl-space two spaces!。不需要特别在意这个。 |

上表是该任务对应的点以及分别在考察什么。

第一个任务，实际上较为复杂，我们需要观察进程的创建过程，在process.c的process\_execute即为我们的线程创建函数。其中file\_name就是我们传递进来的参数，但是它不仅包含了我们要执行的函数，也包括了后面的参数，所以这里需要进行字符串分割。幸运的是，pintos已经为我们提供了一个字符串分割的函数，声明在lib/string.h中。

分割完参数后，可以看到进程拉起了一个线程，我们需要在线程的创建函数中start\_process中，将参数放置于esp的对应位置。堆栈结构请参照3.5.1

这里给出一张截图



同学们可参照压栈图，将分割出的参数压栈到中断现场的合适位置。

Tip 1：如果在实现该部分过程中，传递的参数始终有问题，请注意分割fn\_copy的位置和传递给start\_process的顺序，或者每次使用时，都重新memcpy拷贝一份，以防止修改到原始的数据。

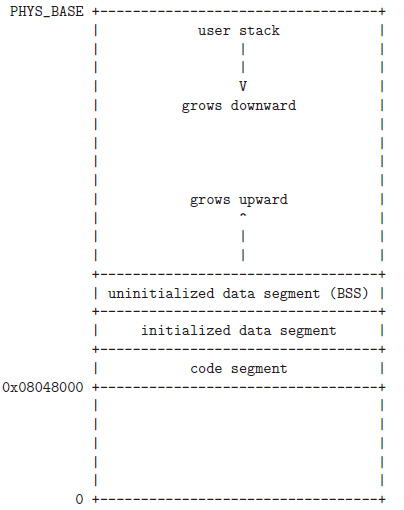
做到这里，还记得我们前面的\*esp = PHYS\_BASE-12;吗？这个时候已经处理好压栈，就可以删去-12了。现在明白为什么要-12了吗？是为了防止在为传递参数的时候，错误地移动指针，访问到不应该访问或修改的内存值。

### 任务2：系统调用

#### 2.1 调用参数所处内存地址

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 6 | sc-bad-sp | 将 esp 指向非法的地方之后触发系统调用，正常情况下应该 exit(-1)。测试用例是利用内联汇编来实现的。 |
| 7 | sc-bad-arg | 将 esp 指向了栈顶下 4 字节（刚好放进了 exit 的系统调用号），试图在获取系统调用的参数时访问非法的内存区域，正常情况下应该以 exit(-1) 退出。 |
| 8 | sc-boundary | 让 exit 的系统调用号和参数正好放在页边界，观察退出状态值是否正常。不需要特别在意这个。 |
| 9 | sc-boundary-2 | 让 exit 的参数的前三个字节和最后一个字节存放在页边界，观察退出状态值是否正常。不需要特别在意这个。 |

该任务的目的是，检查我们传递的参数的内存地址，正如我们在实验目的里阐述的，使用特权级的目的是保护内存，这里也是一样，pintos将kernel和用户内存空间分成了两部分，如下图所示，BASE以上是内存空间，08048000-BASE之间，才是用户进程空间，如果一个指针，指向了没有权限的位置，就应该直接以-1退出。文档的3.1.5部分给我们提供了两种思路，一种是userprog/pagedir.c 和 in threads/vaddr.h中的相关函数pagedir\_get\_page，来验证地址范围，一种是通过访问该地址，来造成page\_fault，在page\_fault处理函数中，再退出。



Tip 2：这里一定要记得，参数的地址不一定是连续的，而且每个参数的地址都是4个byte，注意有可能一个参数地址一半跨越界限，一半不跨越的情况。

#### 2.2打开关闭文件

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 19 | open-normal | 正常打开一个文件，如果 fd > 2 则 pass。 |
| 20 | open-missing | 打开不存在的文件，fd 应为 -1。 |
| 21 | open-empty | 文件名为空字符串，fd 应为 -1。 |
| 22 | open-boundary | 文件名跨页，看是否能正常打开。不需要特别在意这个。 |
| 23 | open-null | 文件名使用NULL，应该以 exit(-1) 退出。 |
| 24 | open-bad-ptr | 文件名指针处于用户空间，但是指向了未分配的区域，应该以 exit(-1) 退出。 |
| 25 | open-twice | 打开同一个文件两次，观察是否能正常打开且 fd 是不是不一样（要求不一样）。 |
| 26 | close-normal | 正常关闭一个文件，看程序是否正常退出。 |
| 27 | close-twice | 关闭两次同一个 fd，正常退出或者以 -1 退出都可以。 |
| 28 | close-stdin | 关闭标准输入流，正常退出或者以 -1 退出都可以。 |
| 29 | close-stdout | 关闭标准输出流，正常退出或者以 -1 退出都可以。 |
| 30 | close-bad-fd | 关闭不存在的 fd，正常退出或者以 -1 退出都可以。 |

这一块是过点的集中地，很容易写完后过许多点，收获做实验的满足感。

相信做过p1的同学们对pintos中的链表结构都不陌生了，我们可以使用链表（如果愿意的话你也可以使用数组）来维护打开的文件列表，同时按我们已有的知识，，每一个打开的文件都需要有一个文件描述符fd，且fd在不同进程中不共享，所以可见是由进程维护描述符，同时，0，1，2三个描述符已经默认分配，所以我们必须从3开始为新打开的文件分配描述符。

Tip 3：实验过点的文件不会打开很多，所以不需要考虑关闭文件后，描述符的重新利用。

同理，打开一个文件需要文件地址，这个地址，也是需要我们进行地址判断，防止其越界。到了这里，我们就可以使用filesys/filesys.c中定义的filesys\_open来完成open的系统调用了。同时也要在进程中维护打开的文件列表。

关闭也是同理，我们需要将维护的文件及其描述符从进程的文件列表中移除，并使用file\_close关闭该文件。

Tip 4：在维护打开文件元素时，使用了malloc的话，记得一定要在关闭时释放，这一点对于后面的某个点至关重要。同时注意，在一个进程退出时，也应该关闭其打开的所有文件。

#### 2.3创建移除文件

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 12 | create-normal | 正常创建一个文件，然后用 open 系统调用看是否能打开。 |
| 13 | create-empty | 创建文件名为空的文件，返回创建失败或直接以 -1 退出。 |
| 14 | create-null | 创建的文件名为 NULL，要求以 -1 退出。 |
| 15 | create-bad-ptr | 创建的文件名指向未分配空间，返回创建失败或直接以 -1 退出。 |
| 16 | create-long | 创建一个文件名很长的文件。不用特别在意他。 |
| 17 | create-exists | 创建重名的文件，返回创建失败即可。 |
| 18 | create-bound | 创建的文件名跨页。不用特别在意他。 |

这一部分较为简单，创建文件调用filesys/filesys.c中定义的filesys\_create即可，移除调用filesys/filesys.c中定义的filesys\_remove。

#### 2.4读写文件

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 31 | read-normal | 正常读取文件，比对读取的内容是否正确。 |
| 32 | read-bad-ptr | 存放读取内容的 buffer 指向了没被映射的地址，返回 0 或者以 -1 退出都可以。 |
| 33 | read-boundary | buffer 跨页，不需要特别在意他。 |
| 34 | read-zero | 读取 size = 0 的内容。 |
| 35 | read-stdout | 试图读取标准输出流，返回 0 或者以 -1 退出都可以。 |
| 36 | read-bad-fd | 读取不存在的 fd，返回 0 或者以 -1 退出都可以。 |
| 37 | write-normal | 正常写入文件，判断返回值是否正确。 |
| 38 | write-bad-ptr | 写入读取内容的 buffer 指向了没被映射的地址，返回 0 或者以 -1 退出都可以。 |
| 39 | write-boundary | buffer 跨页，不需要特别在意他。 |
| 40 | write-zero | 写入 0 byte 的内容。 |
| 41 | write-stdin | 试图写入标准输入流，返回 0 或者以 -1 退出都可以。 |
| 42 | write-bad-fd | 写入不存在的 fd，返回 0 或者以 -1 退出都可以。 |

和创建移除一样，比较简单，参照3.3.4的参数格式，同时对于write，我们已经实现了向终端写入。Read会用到input\_getc()与file\_read()，write会用到file\_write()与putbuf()。

Tip 5：这里除了注意参数是否和法外，还需要判断缓冲区是否合法。

#### 2.5执行与等待

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 43 | exec-once | 创建一个子进程，观察子进程退出代码（81），以及父进程退出代码是否正确（0）。 |
| 44 | exec-arg | 执行的子进程有多个参数，观察参数是否正确。 |
| 45 | exec-multiple | wait(exec(…))五次，观察退出代码以及退出顺序。 |
| 46 | exec-missing | 执行不存在的文件，观察 exec 返回值（-1）。 |
| 47 | exec-bad-ptr | 执行的命令指向未被分配的内存区域，返回 0 或者以 -1 退出都可以。 |
| 48 | wait-simple | wait 一个子进程，观察返回值是否和子进程的退出代码一致（81）。 |
| 49 | wait-twice | 等待两次同一个子进程，第二次等待应该返回 -1。 |
| 50 | wait-killed | 等待一个子进程，子进程刚执行就被 kill 杀死（应该以 -1 退出），观察子进程退出代码以及父进程在 wait 后的返回值。 |
| 51 | wait-bad-pid | wait 一个不存在的 pid。 |

该部分为创建子进程和父进程等待子进程并回收，也是一个难点。

这里我们还需要提到前面讲过的一点，pintos在这里还没有特别区分进程和线程的概念，可以理解为一个线程就是一个进程，所以其id可以一一映射。该部分的重点在于父进程需要维护创建的子进程列表。子进程的创建过程其实就是我们前面修改过的process\_execute()函数。注意到在该函数的thread\_create函数执行后，其实子进程就已经转移到start\_process中开始运行或者加入就绪队列了。这里我们要保证子进程一定成功创建，就需要实现一个同步锁，来保证子进程load成功才接着执行父进程，子进程一旦创建失败，说明该该调用失败了。

而对于wait操作，很明显也需要一个锁，保证父进程在子进程执行期间无法进行任何操作，等待子进程退出后，父进程获取子进程退出码，并回收资源。这里的锁的设计非常精妙，要保证父进程wait时，无法执行任何操作，子进程退出时，需要立刻通知父进程，但不能直接销毁，而要等待父进程来回收资源获取返回码等，然后才可以正常销毁。需要几个锁来完成这个操作，同学们可以好好考虑。

#### 2.6 multi

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 52 | multi-recurse | 递归创建子进程，共 16 个，观察进入和退出的顺序。 |
| 53 | multi-child-fd | 子进程尝试关闭一个父进程打开的文件（应该要失败），之后看父进程还能不能正常访问这个文件。 |

这一部分几乎没有需要多写的，只需要完成filesize这个系统调用。

Filesize在文档中的声明为：int filesize (int fd);

所以需要我们用fd去寻找打开的文件。实现时调用file\_length就行。

#### 2.7 rox

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

这一部分旨在防止进程修改正在运行的可执行文件，详见文档中的3.3.5

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

同时这一块需要实现seek调用。需要用到file\_seek（）函数。

#### 2.8 bad

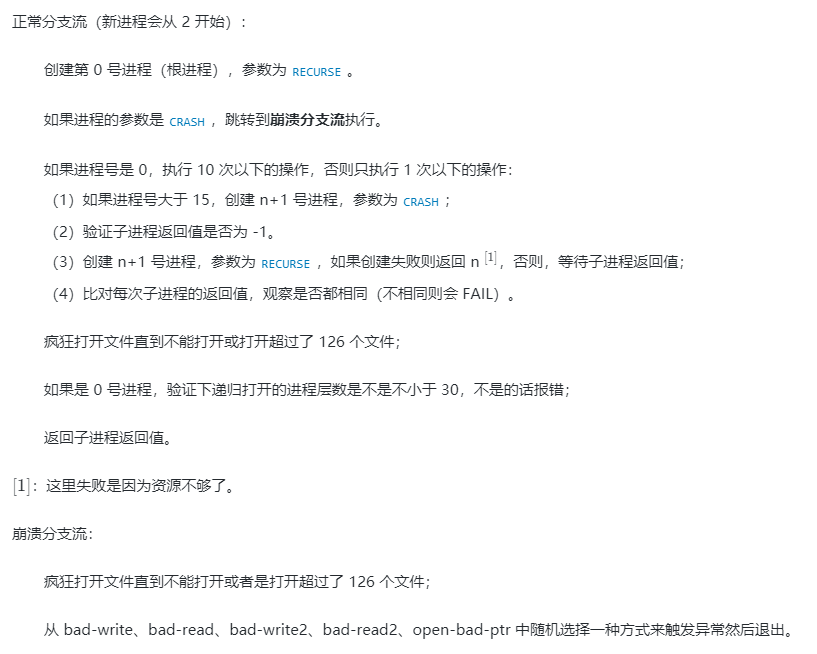
|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 57 | bad-read | 读取NULL指针的内容。 |
| 58 | bad-write | 给NULL指针的位置赋值。 |
| 59 | bad-read2 | 读取内核空间的内容。 |
| 60 | bad-write2 | 往内核空间赋值。 |
| 61 | bad-jump | 把NULL伪装成函数后调用。 |
| 62 | bad-jump2 | 把内核空间地址伪装成函数后调用。 |

这一部分，理论上要是做到了前面的每个地址访问都验证地址合法性，是可以直接通过的。如果有不通过，除了补充前面的不足外，由于访问非法地址会引起page\_fault，所以我们可以在exception.c中对page\_fault（）进行修改，在kill时结束进程，返回-1.

#### 2.9 资源释放

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

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



#### 2.10 文件系统

|  |  |  |
| --- | --- | --- |
| 序号 | 名称 | 测试点 |
| 64 | lg-create | 创建一个大的空白文件，检查文件大小与内容是否是合格的。 |
| 65 | lg-full | 创建一个大的文件，在里面填满内容，检查文件的大小与内容是否和期望的一致。 |
| 66 | lg-random | 创建一个大的文件，往里边多次随机写入内容，然后读取内容比对。 |
| 67 | lg-seq-block | 创建一个大的文件，以 block\_size 为单位多次写入内容，然后读取内容进行比对。 |
| 68 | lg-seq-random | 与上一个相比，block\_size 变为了随机数。 |
| 69 | sm-create | 创建一个小的空白文件，检查文件大小与内容是否是合格的。 |
| 70 | sm-full | 创建一个小的文件，在里面填满内容，检查文件的大小与内容是否和期望的一致。 |
| 71 | sm-random | 创建一个小的文件，往里边多次随机写入内容，然后读取内容比对。 |
| 72 | sm-seq-block | 创建一个小的文件，以 block\_size 为单位多次写入内容，然后读取内容进行比对。 |
| 73 | sm-seq-random | 与上一个相比，block\_size 变为了随机数。 |
| 74 | syn-read | 创建 10 个进程，打开同一个文件并一个字节一个字节地比对内容，保证中途没有异常发生（打开文件失败等情况）。 |
| 75 | syn-remove | 打开一个文件后移除文件，确认文件是否仍然可读写。 |
| 76 | syn-write | 比起 syn-read，读变成了写，但只写一次。 |

其中74-76涉及到了文件的锁，我们需要对文件系统设计一个锁，在所有用到的操作中，操作前都上锁，操作后都去除锁，基本上就可以实现。这一部分的修改在filesys.h和filesys.c中完成。