+---------------------------+

| CS 140 |

| PROJECT 2: USER PROGRAMS |

| DESIGN DOCUMENT |

+---------------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

18373117 吴朝旭 <1205672770@qq.com> 权重：1

主要负责参数传递、系统调用handler、错误处理、最大线程限制；

18373526 王勇 <672515431@qq.com> 权重：1

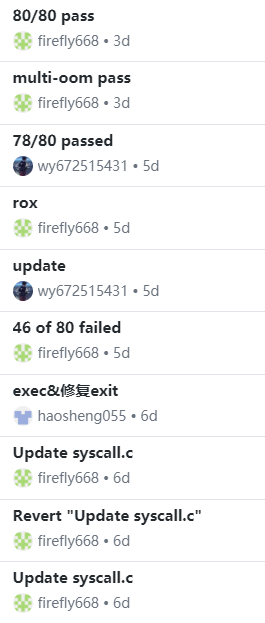
主要负责read write open close filesize，实验1与实验2衔接；

18373554 田震 <1030010026@qq.com> 权重：1

主要负责create remove seek tell；

18373756 郝晟 <2431811460@qq.com> 权重：1

主要负责halt exit exec wait。



（hefun—田震，firefly668—吴朝旭，haosheng055—郝晟，wy672515431—王勇）

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while

>> preparing your submission, other than the Pintos documentation, course

>> text, lecture notes, and course staff.

ARGUMENT PASSING

================

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

Add in struct thread:

Int ret /\*退出信息\*/

---- ALGORITHMS ----

>> A2: Briefly describe how you implemented argument parsing. How do

>> you arrange for the elements of argv[] to be in the right order?

>> How do you avoid overflowing the stack page?

Pintos原本的代码实现了无参数时的版本。所有进程又内核执行process\_execute(const char \* file\_name)，参数为执行文件名。之后创建一个新的线程执行start\_process，start\_process调用load加载可执行文件，load调用set\_stack设置栈的内容，然后返回。要想完成参数传递，传递给process\_execute的就是文件名和参数列表组成的字符串，因此涉及到了字符串拆解，这部分工作由strtok\_r()完成。参数传递的主要工作就是将参数压入栈，因此我们将工作集中到set\_stack中完成。完全按照pintos官方文档中述所的压栈方式即可，但注意申请空间要使用malloc，并且在用完之后free掉，不然会有bug。压栈的具体方式（栈指针向下增长）：

1. 从右往左压入参数；
2. 4字节对齐；
3. 压入argv[argc]=0;
4. 从右往左压入参数在栈中的地址 ；
5. 压入argv，即argv[0]的地址；
6. 压入argc；
7. 压入返回值0。

---- RATIONALE ----

>> A3: Why does Pintos implement strtok\_r() but not strtok()?

strtok\_r()是strtok()的线程安全版本，strtok()使用了全局数据，对于线程而言不安全。为了保证线程安全而选择使用strtok\_r();

>> A4: In Pintos, the kernel separates commands into a executable name

>> and arguments. In Unix-like systems, the shell does this

>> separation. Identify at least two advantages of the Unix approach.

1. 用非内核部分进行参数分离，能够避免发生内核错误而导致内核崩溃；
2. 简化了内核的运行流程，保证了内核的快速运行。

SYSTEM CALLS

============

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

Add in struct thread:

struct thread\* parent\_process; /\* 父进程 \*/

struct list child\_process;   /\* 子进程列表 \*/

struct semaphore wait\_load; /\*等待exec子进程load的信号量 \*/

bool is\_load\_success;     /\* exec的子进程是否load成功\*/

struct semaphore wait\_child\_exit;/\*等待子进程exit的信号量\*/

tid\_t thread\_wait\_for\_exit;  /\* 等待exit的子进程tid \*/

struct list set\_of\_file\_descriptors /\*该线程打开的文件list\*/

add struct:

struct child

{

   tid\_t tid; /\* 子进程的tid \*/

   struct list\_elem elem;

   int exit\_status; /\* 退出状态值 \*/

   bool is\_existed;/\* true表示子进程退出，false仍在运行 \*/

};

struct file\_plus

{

int fd; /\*文件描述符\*/

struct file\*file; /\*与文件描述相对应的文件 \*/

struct list\_elem elem1; /\*列表元素 \*/

}；

增加这个结构体的原因是：根据Pintos文档系统调用函数的说明，许多系统调用例如:filesize(int fd);read(int fd,void \*buffer,unsigned size)等都要通过文件描述符fd来对文件进行操作，但是在实现这些系统调用的时候，需要用到pintos提供的filesys.c中的函数，这些函数是通过文件指针来完成对文件的操作的，所以我们需要将文件描述符和文件本身联系起来，构造了这个结构体。通过这个结构体，我们可以根据文件描述符来找到对应的文件。

struct lock filesystem\_lock;

增加这一个锁的原因是:根据pintos文档，用于对于任意的系统调用，保证在同一时间只有一个用户执行。尤其是对于某些‘filesys’文件夹下的文件系统的代码，一次有多个线程访问是不安全的。所以说，我们将文件系统部分的代码视为临界区，对这一部分的代码使用锁来保证一次只有一个线程访问。

>> B2: Describe how file descriptors are associated with open files.

>> Are file descriptors unique within the entire OS or just within a

>> single process?

我们定义了一个结构体file\_plus，其有两个成员文件描述符fd,和文件本身file。我们在打开文件的时候，通过filesys\_open()函数得到文件本身，然后将这个文件和我们自己生成的文件描述符作为成员存储到一个新生成的结构体中，这样就将他们联系起来了。

在我们的实现方法里，我们是给每个线程设置了自己的文件描述符队列，也就是说文件描述符仅在单个线程里是独一无二的。

---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the

>> kernel.

首先检测缓冲区buffer是否为有效的用户虚拟空间（使用is\_valid\_buffer()函数，其实现方式是，检测buffer的起始和终止地址是否是空指针、用户虚拟地址、未被映射的用户虚拟内存，这三个判断通过is\_valid\_addr()函数实现）。

之后，由于需要使用文件系统的代码，所以获取锁来保证同步。

然后根据文件描述符做以下判断：

如果文件描述符为1或小于0，释放锁，并调用exit(-1)退出。

如果文件描述符为0，说明是从标准输入读入，使用input\_getc()获取输入，之后释放锁，返回读到内容的字节数。

如果文件描述符大于1，则说明是线程打开的文件，这是需要遍历线程打开的文件列表set\_of\_file\_descriptors,从中找到文件描述符对应的file\_plus结构体，从而找到对应的打开的文件，调用file\_read读取文件并获取读取到的文件的长度，之后释放锁，返回读取到的文件的长度。

如果没有在上面返回，则出现异常，调用exit(-1)退出。

我们在内核里是通过write函数向用户数据进行写操作的。

首先我们先通过调用is\_valid\_buffer()函数来检测我们是否向有效的用户内存进行写操作。然后我们上锁，保证该操作的原子性。根据文档，当我们尝试向标准输入即文件描述符等于0的文件进行写操作时，我们直接将进程exit(-1）。

同理，由于文件描述符始终大于等于0，如果传进来的参数fd小于0，直接将进程exit(-1).根据文档，如果fd为1，即向标准输出写入东西时，我们直接调用putbuf()函数，进行写操作。否则，我们我们调用file\_write()函数来进行写操作。由于file\_write的参数为文件，write的参数为fd，所以我们需要遍历该程序的file\_plus列表，找到文件描述符fd对应的文件。在执行写操作之后，我们释放锁，并且返回实际写入的字节数。

>> B4: Suppose a system call causes a full page (4,096 bytes) of data

>> to be copied from user space into the kernel. What is the least

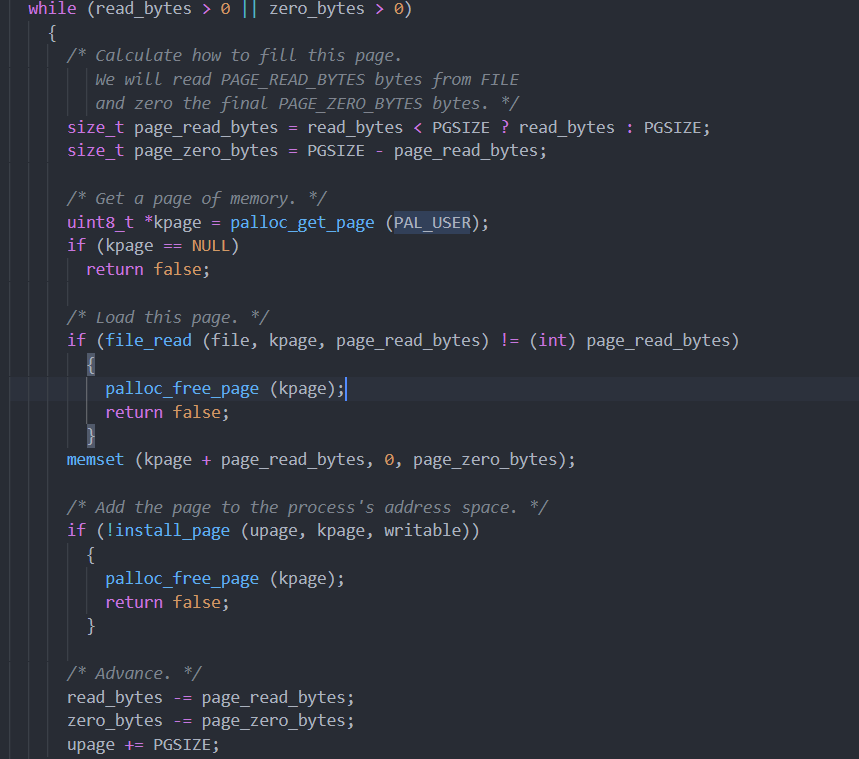
>> and the greatest possible number of inspections of the page table

>> (e.g. calls to pagedir\_get\_page()) that might result? What about

>> for a system call that only copies 2 bytes of data? Is there room

>> for improvement in these numbers, and how much?

我们拿向内核中加载可执行文件为例。



根据pintos文档可知，一页最多包含4kb。如果我们通过palloc\_get\_page函数得到的内核虚拟内存地址其恰是其所对应页能够存放4kb，那么只需一次检查（执行一次palloc\_get\_page）。

如果得到的内核虚拟内存地址其恰是其所对应页只能存放1byte，那么需要执行4096次检查。故最小值为1次，最大值为4096次。

同理，最小值为1次，最大值为2次。

>> B5: Briefly describe your implementation of the "wait" system call

>> and how it interacts with process termination.

1.使用信号量来进行同步操作。首先调用wait的父进程会在它的子进程list中找到与child\_tid对应的子进程；然后通过struct child数据结构的is\_existed成员来判断子进程退出状态；如果没退出则使用信号量的P操作阻塞当前进程，直到子进程调用V操作后退出，否则直接根据struct child的exit\_status返回子进程的退出状态。

2.子进程退出时会在在父进程的子进程list里添加它的退出信息；然后通过struct thread的thread\_wait\_for\_exit成员判断父进程是否在等子进程退出；如果是则使用V操作唤醒，否则正常退出。

>> B6: Any access to user program memory at a user-specified address

>> can fail due to a bad pointer value. Such accesses must cause the

>> process to be terminated. System calls are fraught with such

>> accesses, e.g. a "write" system call requires reading the system

>> call number from the user stack, then each of the call's three

>> arguments, then an arbitrary amount of user memory, and any of

>> these can fail at any point. This poses a design and

>> error-handling problem: how do you best avoid obscuring the primary

>> function of code in a morass of error-handling? Furthermore, when

>> an error is detected, how do you ensure that all temporarily

>> allocated resources (locks, buffers, etc.) are freed? In a few

>> paragraphs, describe the strategy or strategies you adopted for

>> managing these issues. Give an example.

---- SYNCHRONIZATION ----

>> B7: The "exec" system call returns -1 if loading the new executable

>> fails, so it cannot return before the new executable has completed

>> loading. How does your code ensure this? How is the load

>> success/failure status passed back to the thread that calls "exec"?

1.同样使用信号量来实现同步。调用exec的进程会阻塞在wait\_load这个信号量上（sema\_down），直到load返回时调用sema\_up唤醒。

2.在调用exec创建一个进程时，子进程的parent\_process属性会保存是哪个进程调用了exec；load返回时，通过parent\_process成员找到调用了exec的进程，将load返回结果保存到该进程的is\_load\_success属性。

>> B8: Consider parent process P with child process C. How do you

>> ensure proper synchronization and avoid race conditions when P

>> calls wait(C) before C exits? After C exits? How do you ensure

>> that all resources are freed in each case? How about when P

>> terminates without waiting, before C exits? After C exits? Are

>> there any special cases?

1.如B5所述，这种情况process P会因为信号量的P操作阻塞，直到process C退出时执行V操作。

2.在B1设计的数据结构中，parent\_process和child\_process可以在父进程中保存子进程的退出信息，这样无论是子进程退出后调用wait还是退出前调用wait，process P都可以访问到process C的退出信息，避免了资源竞争。

---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the

>> kernel in the way that you did?

>> B10: What advantages or disadvantages can you see to your design

>> for file descriptors?

>> B11: The default tid\_t to pid\_t mapping is the identity mapping.

>> If you changed it, what advantages are there to your approach?

SURVEY QUESTIONS

================

Answering these questions is optional, but it will help us improve the

course in future quarters. Feel free to tell us anything you

want--these questions are just to spur your thoughts. You may also

choose to respond anonymously in the course evaluations at the end of

the quarter.

>> In your opinion, was this assignment, or any one of the three problems

>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in

>> future quarters to help them solve the problems? Conversely, did you

>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist

>> students, either for future quarters or the remaining projects?

>> Any other comments?