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

| 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

18373526 王勇 <672515431@qq.com> 权重：1

18373554 田震 <1030010026@qq.com> 权重：1

18373756 郝晟 <2431811460@qq.com> 权重：1

---- 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.

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

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

>> single process?

---- ALGORITHMS ----

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

>> kernel.

>> 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?

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

>> and how it interacts with process termination.

>> 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"?

>> 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?

---- 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?