DESIGN DOCUMENT FOR PROJECT 1: USER PROGRAMS

GROUP

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

姓名	学号	权重
杜宸洋	18373119	110
王熙林	18373491	110
任博林	18373691	110
高 锐	17351046	70

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.

None.

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?

对于参数传递问题，我们在`start_process()`函数中进行处理。为了更加便捷地处理字符串，我们新建了`command_break()`函数。为了保证使参数的顺序正确，我们从后向前倒序扫描参数字符串，因此我们得到的第一个标记是最后一个参数，我们得到的最后一个标记是第一个参数。我们可以继续减少`esp`指针以设置`argv[]`元素。代码如下所示：

```
int i=argc;
char* addr_arr[100];//存地址
while(--i>=0){
    if_.esp = if_.esp - sizeof(char)*(strlen(argv[i])+1); // "\0"
    addr_arr[i]=(char *)if_.esp;
    memcpy(if_.esp,argv[i],strlen(argv[i])+1);
}
```

在获取所有参数之后按照文档中所提示的顺序对esp栈帧以及相关数据进行操作即可，具体详见start_process()函数。

为了避免栈溢出，我们将参数数组大小限制在100。

RATIONALE

A3: Why does Pintos implement strtok_r() but not strtok()?

strtok_r()更具线程安全性。它是可重入的，以避免另一个线程获得控制权并调用strtok,

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. 在内核级别减少了不必要的工作，可以缩短运行的时间。
2. 在将可执行文件传递给内核之前检查它是否存在，以避免内核错误，保证了可靠性。

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.

```
在syscall.h中加入以下结构体
struct file_node{//文件标识符
  int fd;
  struct list_elem elem;
  struct file* file;
};

struct child_process{
  tid_t tid; // 进程标识符
  struct list_elem elem;
};

struct read_elem{//管道读元素
  tid_t tid;
  int op;
  struct list_elem elem;
  int ret_value;
};

struct wait_elem{//管道等待元素
  tid_t parent_tid;
  //等待只由父进程调用，防止出现二次调用父进程的情况
  tid_t child_tid;
  int op;
  struct list_elem elem;
  struct semaphore waitSema;
};
在struct thread中加入以下成员变量
    tid_t parent_id;              /* 父进程*/
    struct list child_list;      /* 子进程序列 */
    struct list file_list;       /* 打开文件列表*/
    int ret;                     /* 返回值*/
    struct file *exec;       /* 此当前进程运行文件 */
```

B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

每次文件被打开时，系统会新建一个文件标识符的结构体，其成员包括被打开的文件的指针，以及一个整个系统内唯一的整数值（用来标识文件）。所以我们的实现中文件标识符对于全系统来说都是唯一的，但进程也会维持一个自己打开的文件的列表。

ALGORITHMS

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

在系统调用读，写时，首先检查传入的帧的所有参数空间是否有效，将文件标识符，buffer，size提取出来后，再检查buffer以及buffer+size的指针是否都有效。如果其中有无效地址，则退出，返回值为-1，否则则继续执行。

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?

最少次数为1，最大次数为4096。当`pagedir_get_page()`没有验证指针，且所有数据都存储在单个页面上，则只需调用一次，若数据的字节分布在4096页中，则需要调用4096次。对于2 bytes 数据的情况，最少1次，最多2次。

我们目前没有什么比较好的方法来提高。

B5: Briefly describe your implementation of the "wait" system call and how it interacts with process termination.

系统调用`wait`会直接调用`process_wait`。在这方面我们参考了管道的设计，进程退出时会向一个管道内写入退出信息，当父进程需要等待时，会先检查该管道内是否有需要的子进程的信息，如果有则读取，返回，如果没有则会写入wait请求并阻塞。当一个子进程退出时，会先写入退出信息，然后检查有没有父进程的wait请求，有的话就会唤醒父进程。

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.

在执行相关操作之前，首先检查参数空间是否有效，确保该空间是用户空间（`is_user_vaddr`）且调用`pagedir_get_page()`的返回值不为空，在取出相应的参数后，再检查参数所指向的空间是否有效，如buffer的起始地址和终止地址（buffer+size）。

比如：对于write系统调用：

首先对esp+4到esp+16进行检查（检查参数位置是否有效），有效后取出参数buffer和size，检查buffer和buffer+size的地址是否有效，最后执行write操作。

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"?

我们在thread结构体中加入了`ret`变量来保存进程的返回值。子进程可以通过`parent_id`来获得父进程的id，并结合`GetThreadByTid()`可以获得父进程的访问权。

我们设计的目的是当意外发生的时候，子进程可以随时调用全局的退出函数`ExitStatus()`。因此，如果我们将它保存在子进程中，当它在父进程检查之前退出时，就无法获得返回值。

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?

在thread结构体中，我们加入了两个用来记录父子进程的变量。

```
    tid_t parent_id;              /* 父进程*/
    struct list child_list;      /* 子进程序列 */
```

在进程创建的时候，将C添加到P的子进程列表中，并将P添加到C的父进程中。这样做可以确保如果子进程首先执行完，则父进程将指导子进程等待并不会终止。否则，如果子进程仍然存在，父进程会等待子进程。当进程退出的时候，所有的资源都会被释放。

RATIONALE

B9: Why did you choose to implement access to user memory from the kernel in the way that you did?

因为这样的实现方式比较容易，因为它允许我们的系统调用实现在一个函数调用中轻松验证对用户内存的访问。

B10: What advantages or disadvantages can you see to your design for file descriptors?

优点：

1. 文件描述符十分简洁。
2. 记录了所有已经打开的文件，这样可以更加灵活地操作打开的文件。

缺点：

1. 占用内核空间，用户程序可能会打开大量文件崩溃内核。

B11: The default `tid_t` to `pid_t` mapping is the identity mapping. If you changed it, what advantages are there to your approach?

我们没有改变它。现在每个进程都对应一个线程。优点则在于如果改变了，那么一个进程就可以对应多个不同的线程。