CS 140
PROJECT 2: USER PROGRAMS
DESIGN DOCUMENT

GROUP

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

刘禹辰 18373276

何辰之 18373277

王旭琳 18373246

马瀚元 18373337

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.

在`thread.h`中定义了`struct pcb`，具体定义如下：

```
struct pcb {
    char *exec_name;
    char *cmd_line;
};
```

定义目的：创建结构体来存储文件的完整命令行指令和可执行文件名，在`process_execute()`和`start_process()`中进行参数传递。

---- 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?

实现参数传递过程的方法是`process.c`文件中的`process_execute()`。

首先需要修改该方法调用`thread_create()`创建新线程时传递的可执行文件名。原参数`file_name`包含完整的命令行指令。新创建`pcb`结构体变量`_pcb`，用`strtok_r`分离出可执行文件名，并将其与原完整命令行指令存入`_pcb`中。将`_pcb`作为`thread_create()`创建新线程后，方法`start_process()`的执行参数传入。

在`start_thread()`将命令行参数分离并传入栈`struct intr_frame if_`中，具体步骤为：

- 将`load()`方法的第一个参数改为`_pcb->exec_name`，即可执行文件名。

- 按照文档3.5.1部分程序栈的约定顺序依次入栈。入栈的顺序为：
    - 将从命令行指令分离出的参数依次入栈，包含可执行文件名。
    - 填入0使栈与4的倍数对齐。
    - 按参数在命令行中从右向左的顺序，依次入栈该参数所对应的地址。在此之前，还需先入栈一个空地址（全0）作为参数的结尾。
    - 入栈第一个参数地址所对应的栈顶地址，即argv。
    - 入栈参数个数argc以及返回地址。

参数数组`argv[]`的顺序由存放各`argv[]`元素的地址按序入栈来保证。因此`argv[]`的值入栈可以不按照顺序入栈。我们在分离参数时按顺序分离、入栈并记录该参数入栈后的栈顶指针值，并将这些记录的指针按序入栈，来保证在栈中可以按顺序找到对应参数的地址。

我们设定了指令的长度最大为4k，设置了指令的最大个数为128，在`start_process()`方法中最多只会向栈中传入128个参数。以上限制可以保证不会发生栈溢出。

---- RATIONALE ----

> 3: Why does Pintos implement strtok_r() but not strtok()?

`strtok()`方法不包含`save_ptr`参数，分离字符串时只能分离出第一个token，不便于循环分离出接下来的token。

而`strtok_r()`方法中，`save_ptr`参数可以用于保存上一次迭代后指向的位置，便于继续迭代分离出后续的token。

> 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.

pintos中将参数分离工作放在内核中执行，而类Unix系统则在Shell中执行该部分工作。

在Shell中分离命令行参数有以下好处：

1. 降低内核负担。在内核中执行分离参数操作会使系统调用的运行时间加长，影响系统调用的效率。
2. 减少内存占用。在内核中执行分离参数需要对命令行指令进行拷贝。在命令行指令长时，会占用大量的内核内存。

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`thread.h`中的`struct thread`补充定义了如下成员变量：

`int ret_val;`

定义目的：在系统调用`exit()`中使用，用于保存该进程的退出状态。

`struct semaphore wait_child;`

定义目的：在系统调用`execute()`中使用，用于保证父进程在子进程初始化加载结束后才执行`process_execute()`的返回。

`struct thread *parent_thread;`

定义目的：在系统调用`wait()`和`execute()`中使用，用于子进程能够访问到创建自己的父进程的信息。

`struct list child_list;`

定义目的：在系统调用`wait()`中使用，用于父进程保存自己创建的所有子进程的简略信息（详见下方`saved_child`结构体定义），从而使得父进程能够判断进程是否为自己的子进程以及保存子进程的退出状态。

`struct list opened_files;`

定义目的：在与文件系统相关的系统调用中使用，用于保存当前进程所打开的全部文件信息（详见下方`opened_file`结构体定义），从而使得进程能够判断进行操作的文件是否已经被打开以及保存该文件的指针。

`int cur_fd;`

定义目的：在系统调用`open()`中使用，用于分配当前进程打开文件的文件标识符`fd`。

`struct file *this_file;`

定义目的：保存当前进程执行文件指针，用于在执行当前进程时，禁止其他进程修改该进程文件。

`bool load_result;`

定义目的：该状态为`false`时，表示本次系统调用`exec()`调用失败，即用户进程空间已满，无法继续分配资源。

在`thread.h`中定义了`struct saved_child`，具体定义如下：

```
struct saved_child
{
   tid_t tid;
   int ret_val;
   struct list_elem elem;
   struct semaphore sema;
};
```

定义目的：用于父进程保存子进程的必要信息，以防止子进程结束后其内存空间被释放。保存的信息包括进程号、返回值、信号量（用于在父进程调用`wait()`时阻塞父进程、实现同步）。

在`thread.h`中定义了`struct opened_file`，具体定义如下：

```
struct opened_file
{
   struct file *f;
   int fd;
   struct list_elem elem;
};
```

定义目的：用于进程保存打开的文件的必要信息，将进程与文件建立联系。保存的信息包括文件的指针、文件描述符。

在`thread.c`里定义了`static struct lock filesys_lock;`

定义目的：用于保证所有对文件系统操作的原子性。

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

文件与进程之间的联系通过上文中提到的进程成员变量`struct list opened_files`建立。每个进程维护一个当前进程打开的文件列表。其中，文件描述符只用于当前进程找到对应的文件。不同的进程会使用相同的文件描述符，但对应的文件可能不同；同一个文件在不同的进程中可能对应不同的文件描述符。

---- ALGORITHMS ----

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

`read()`操作首先判断文件描述符：

- 若是标准输入，则调用方法`input_getc()`从键盘读入。
- 若是标准输出，则直接返回，无法从标准输出中读入内容。
- 其他情况下，判断当前文件描述符文件在当前进程中是否被打开（或是否存在），如是，调用`file_read()`方法读入；如不是，则返回0，表示未读入任何内容。

`write()`操作首先判断文件描述符：

- 若是标准输出，则调用方法`putbuf()`将内容写入控制台。
- 若是标准输出，则直接返回，无法向标准输入中写入内容。
- 其他情况下，判断当前文件描述符文件在当前进程中是否被打开（或是否存在），如是，调用`file_write()`向文件中写入；如不是，则返回0，表示未写入任何内容。

> 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?

在以上两种情况下，最大检查可能次数为2，最小检查次数为1。因为无论是4096字节还是2字节，最多只可能被分配到两页地址空间上，最小分配到一页地址空间。

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

在线程的结构体中加入了`struct thread *parent_thread`和`struct list child_list`成员变量，该变量用于保存创建的子进程的信息。

在进程被创建时，初始化自己进程的`parent_thread`变量指向父进程，并在父进程的`child_list`中添加自己的基本信息。在基本信息结构体`saved_child`中，填入`tid`，初始化信号量为0，并且将`ret_val`初始化为-1，因为子进程在异常退出时不会修改该值，而异常退出返回值为-1。

在进程退出时，子进程通过`parent_thread`找到当前父进程，并修改父进程的`child_list`中，当前子进程所对应信息的`ret_val`值，并且将信号量加1。

父进程在调用`wait()`后，会调用`process_wait()`方法。在该方法中，父进程通过`child_list`判断传入tid是否为自己的子进程。如不是，直接返回-1；如是，则对对应子进程的信号量进行-1操作，如果子进程尚在执行，则父进程阻塞，直到子进程执行完毕对信号量+1，然后将该子进程对应信息从`child_list`中移除，以此实现当再次调用对同一子进程的`wait()`时直接返回-1。最后取保存的子进程返回值作为`wait()`的返回值。

> 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.

我们在系统调用取参数时，首先会在`getargu()`中调用`check_valid_addr()`对每个参数所在的地址（即esp指针指向位置）进行判断，先确保其地址在用户进程空间内，接着判断其所在页是否已被加载，为保证安全，在检验时会测试该指针下一位置的合法性，避免出现该参数地址起始位置合法，但是余下字节位置不合法的情况（即某参数地址部分合法的情况）。接下来对于是指针的参数，我们会在调用`check_valid_argu()`对以该指针所指向的地址为起始地址的字符串中每个字符的地址进行判断，具体判断同样是调用`check_valid_addr()`函数，这样做是为确保传入的字符串参数起始地址合法而余下字节所在位置不合法的情况可以被检测到（或字符串所在地址部分合法，部分不合法的情况能够被检测到）。

---- 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"?

在`thread`结构体中添加了初始值为0的信号量。当父进程创建子进程之后，调用`sema_down()`方法阻塞本进程。当子进程加载成功或失败后，都会调用`sema_up()`方法，以此来唤醒父进程。父进程在唤醒后才判断子进程是否被加载成功，并根据情况返回`exec()`方法。

> 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. 在调用`wait()`时，首先会检查子进程是否在父进程的列表`child_list`中，若不在则返回-1；其次若子进程已经退出（指针为NULL）或者子进程状态为`THREAD_DYING`也会立刻返回-1。因此可以保证在两种情况下`wait()`都有正确的执行。
2. 当子进程退出时，它将在`process_exit()`中关闭所有已打开的文件（记录在`opened_files`）中，并将自己的进程文件关闭（`this_file`），这保证了各种情况下都能正常释放资源。
3. 子进程通过信号量与父进程保持wait时的同步。但当父进程P先退出时，不会对子进程产生任何影响。

特殊情况：父进程重复`wait()`了同一个子进程

处理方法：父进程被子进程唤醒后会将该子进程从列表`child_list`中移除，再次`wait()`该子进程时会直接返回-1

---- RATIONALE ----

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

我们使用的是第一种接入用户内存空间的方式。采用这种方式需要在每次使用指针时调用`is_user_vaddr()`和`pagedir_get_page()`方法判断该地址是否合法。采用这种方式判断地址更加直接，实现更为简单。

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

我们的实现方式是对于每个进程维护一个打开文件的列表，列表中为每一个打开的文件分配一个文件描述符以及一个存储文件的指针。其中文件描述符的分配只与当前进程有关。

这种方法的优势是允许多个进程同时打开同一个文件。不同进程对于同一文件的文件描述符并不相同，并且同一进程可以打开同一文件多次。缺点则是该方法会额外占用一定的用户内存空间。

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

更改进程id以及线程id映射的好处有：

1. 解除进程id和线程id的绑定关系后，可以实现多线程进程，即一个进程id对应多个线程id；
2. 用户只能访问进程id，而线程id则对用户透明，只能通过对应进程控制线程。