课程名称：操作系统实践	年级：2023级	上机实践成绩：
指导教师：张民	姓名：李彤	
上机实践名称：参数传递和系统调用	学号：10235101500	上机实践日期：2024.12.02 & 2024.12.17
上机实践编号：	组号：	上机实践时间：

参数传递

问题分析

现有的操作系统在运行 `pintos -- -q run 'echo x'`时，`'echo x'`被看作一个整体，无法实现参数分离。

```
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo x':
Execution of 'echo x' complete.
Timer: 52 ticks
Thread: 0 idle ticks, 52 kernel ticks, 0 user ticks
hdb1 (filesys): 2 reads, 0 writes
Console: 612 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
```

根据init.c中的代码我们可以发现，`run_task`函数的首尾两条`printf`语句都正常输出，因此问题应该出在`process_wait`语句，而process_wait语句目前只是返回-1，所以可以进一步推出问题就出在`process_wait`调用的`process_execute`。

```
int
process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

```
/** Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

代码实现

process_execute

该函数首先复制传入的文件名，然后创建一个新线程来执行该进程。

- 如果线程创建成功，函数会等待该线程的初始化完成，并返回线程ID。
- 如果线程创建失败，返回错误代码TID_ERROR。

与原函数相比，修改后的process_execute函数不仅能够根据可执行文件的大小动态分配内存，还能利用`strtok_r`函数将文件名与参数分割开。同时，函数还利用信号量实现了线程之间的同步，防止父线程在子线程未初始化时就开始执行后续逻辑。

```
tid_t process_execute (const char *file_name)
{
  tid_t tid;  // 线程ID

  // 动态分配内存以复制文件名
  char *fn_copy = malloc(strlen(file_name) + 1);  // 为文件名分配内存
  char *fn_copy2 = malloc(strlen(file_name) + 1); // 创建另一个副本来解析文件名

  if (fn_copy == NULL || fn_copy2 == NULL)
    return TID_ERROR;

  strlcpy(fn_copy, file_name, strlen(file_name) + 1);
  strlcpy(fn_copy2, file_name, strlen(file_name) + 1);

  char *save_ptr;
  fn_copy2 = strtok_r(fn_copy2, " ", &save_ptr);  // 使用 strtok_r 分割文件名以提取可执行文件名
  tid = thread_create(fn_copy2, PRI_DEFAULT, start_process, fn_copy);  // 创建新线程执行 start_process

  free(fn_copy2);  // 释放用于存储分割文件名的内存

  // 检查线程创建是否成功，如果创建失败，就释放文件名副本，并返回错误代码
  if(tid == TID_ERROR){
    free(fn_copy);
    return tid;
  }

  sema_down(&thread_current()->sema);  // 等待子线程初始化完成

  // 返回新线程的ID
  return tid;
}
```

start_process

`start_process`函数主要实现了以下修改： 创建一个文件副本，便于获取文件名和分割参数并将其存储到argv数组中。

```
// 添加代码（1）：复制文件名以便后续使用
char *fn_copy = malloc(strlen(file_name) + 1);
strlcpy(fn_copy, file_name, strlen(file_name) + 1);
```

```
// 修改代码（2）：程序的文件名
char *token, *save_ptr;
file_name = strtok_r(file_name, " ", &save_ptr);
```

```
// 修改代码（3）：处理加载成功和失败的情况
// 如果加载成功，就循环分割参数并将其压入栈中
if(success){
  int argc = 0;
  int argv[50];
  for (token = strtok_r(fn_copy, " ", &save_ptr); token != NULL;
       token = strtok_r(NULL, " ", &save_ptr)){
    if_.esp -= strlen(token) + 1;
    memcpy(if_.esp, token, strlen(token) + 1);
    argv[argc++] = (int)if_.esp;
  }

  // 将数组中分割好的参数压入栈中
  push_argument(&if_.esp, argc, argv);
```

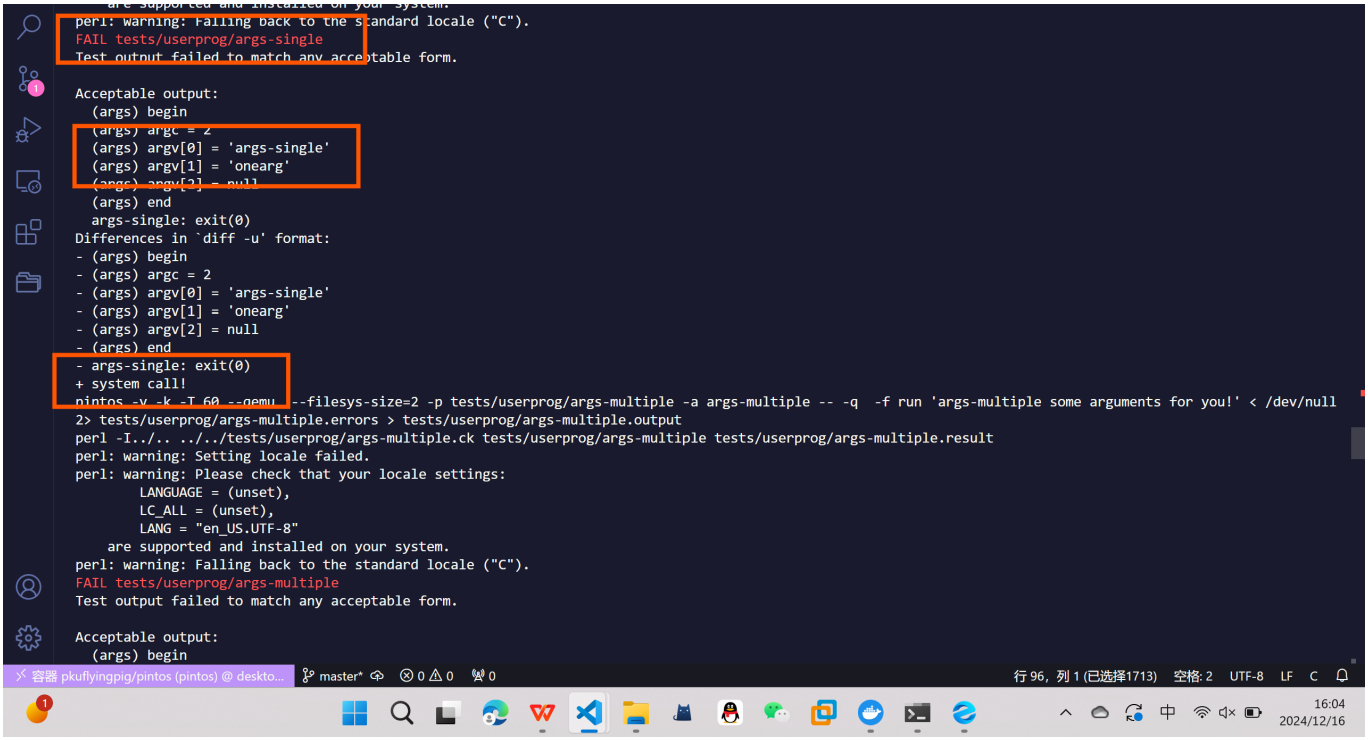
push_argument

在实现push_argument函数时要注意，因为在调用时我们是按照原本参数的输入顺序去调用的，但是考虑到栈结构FILO的性质，我们肯定要把存储的那些参数进行反转，所以在这里，我们循环从参数数组的末尾开始，从后往前推进，最后依次将指向参数地址和参数个数的指针压入栈中。为了划分边界，这里我们还在栈的两端各压入了一个0，用来标识参数的开始和结束。

```
// 将命令行参数压入栈中，准备传递给用户进程。
// 参数：esp - 栈指针，argc - 参数个数，argv - 参数数组
void push_argument(void **esp, int argc, int argv[]){
  *esp = (int) *esp & 0xfffffffc;  // 将栈指针对齐到4字节边界

  // 将NULL放入栈中，作为参数的结束标志
  *esp -= 4;
  *(int *) *esp = 0;

  // 从后向前将参数地址推入栈中
  for (int i = argc - 1; i >= 0; --i)
  {
    *esp -= 4;
    *(int *) *esp = argv[i];
  }

  // 压入指向参数数组的指针
  *esp -= 4;
  *(int *) *esp = (int) *esp + 4;

  // 将参数个数压入栈中
  *esp -= 4;
  *(int *) *esp = argc;

  // ? 分割?
  *esp -= 4;
  *(int *) *esp = 0;
}
```

测试

完成以上代码修改之后，我们在usrprog目录下先后执行 **make、make check** 指令，并观察结果。



系统调用

问题分析

可以发现，虽然我们已经成功分离了命令和参数，但是测试还是FAIL了，其原因就在于我们还没有实现系统调用这一功能，从输出结果我们也能看出这一点。

因此，接下来我们的任务就是修改系统调用。

代码实现

添加结构体定义

在thread.h文件中，我们主要添加了以下内容：

- 新增 thread_file 保存文件相关信息；
- 在 thread 结构体中增加一些参数。

```
int st_exit;                            // 退出状态
struct thread* parent;                  // 当前线程的父线程

struct list files;                      // 打开的文件列表
int file_fd;                            // 文件描述符
struct file * file_owned;               // 当前线程打开的文件
```

```
// 线程打开的文件
struct thread_file
{
  int fd;                               // 文件描述符
  struct file* file;                    // 文件指针
  struct list_elem file_elem;           // 文件列表中的元素
};
```

一些辅助函数

为了实现系统调用，我们还需要定义一些辅助函数：

- exit_special —— 异常退出函数，退出当前线程，并设置退出状态为 -1；
- check_ptr —— 检查用户传入的地址是否有效；
- is_valid_pointer —— 检查给定的栈指针及参数是否有效，确保它们指向用户空间且在有效的页面中；
- find_file_id —— 根据文件描述符（file_id）查找当前线程打开的文件；
- get_user —— 从用户空间地址读取一个字节，并返回其值。

以上这些函数都是定义在syscall.c文件中。

```
void exit_special (void)
{
  thread_current()->st_exit = -1;    // 退出状态为 -1，表示异常退出
  thread_exit ();
}
```

```c
void * check_ptr(const void *vaddr)
{
  // 如果地址不是用户空间地址，则调用 exit_special() 退出当前线程
  if (!is_user_vaddr(vaddr))
    exit_special ();

  // 获取 vaddr 所在的页，如果该页不存在，则退出
  void *ptr = pagedir_get_page (thread_current()->pagedir, vaddr);
  if (!ptr)
    exit_special ();

  // 检查 vaddr 地址的前四个字节是否可访问
  uint8_t *check_byteptr = (uint8_t *) vaddr;
  for (uint8_t i = 0; i < 4; i++)
  {
    if (get_user(check_byteptr + i) == -1)
      exit_special();
  }

  return ptr;          // 返回有效的页面地址
}
```

```c
struct thread_file * find_file_id (int file_id)
{
  struct list_elem *e;
  struct thread_file *thread_file_temp = NULL;
  struct list *files = &thread_current()->files;

  // 遍历当前线程打开的文件列表
  for (e = list_begin(files); e != list_end(files); e = list_next(e)) {
    thread_file_temp = list_entry(e, struct thread_file, file_elem);      // 获取当前元素的文件信息

    if (file_id == thread_file_temp->fd)      // 文件描述符匹配，返回该文件
      return thread_file_temp;
  }

  return false;         // 未找到对应的文件描述符
}
```

```c
static int get_user(const uint8_t *uaddr)
{
  int result;
  // 使用汇编指令读取用户空间地址的数据，并将其存入 result
  asm ("movl $1f, %0; movzbl %1, %0; 1:" : "=&a" (result) : "m" (*uaddr));
  return result;
}
```

系统调用处理函数

syscall_init 函数将系统调用的中断处理程序注册到中断向量表，并将系统调用编号与具体的处理函数关联起来。

而 **syscall_handler** 在用户程序发起系统调用后，根据系统调用的类型（即系统调用编号）来选择并执行具体的系统调用处理函数。

```c
void syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");

  syscalls[SYS_EXIT] = &sys_exit;
  syscalls[SYS_WRITE] = &sys_write;

}
```

```c
static void syscall_handler (struct intr_frame *f UNUSED)
{
  int * p = f->esp;               // 获取栈指针指向的位置（即系统调用参数的地址）
  check_ptr (p + 1);              // 检查传入的参数指针是否有效，避免访问非法内存
  int type = * (int *)f->esp;     // 从栈中读取系统调用的类型（即系统调用的编号）

  if(type <= 0 || type >= max_syscall)  // max_syscall设置为20
    exit_special ();

  if(type == SYS_EXIT || type == SYS_WAIT)
    syscalls[type](f);                // 调用对应的处理函数

  else thread_exit();
}
```

exit和write系统调用的实现

sys_exit —— 退出当前进程，并将进程的退出状态保存到当前线程中。

```c
void sys_exit (struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;  // 获取用户栈指针
  check_ptr (user_ptr + 1);     // 检查栈中是否有有效的参数
  *user_ptr++;                  // 增加指针，跳过程序名参数

  thread_current()->st_exit = *user_ptr;  // 将退出状态保存在当前线程的退出状态字段中
  thread_exit ();
}
```

sys_write —— 向文件或标准输出写入数据

在实现与文件相关的系统调用时，我们需要在 thread.c 中增加一个全局锁，对文件操作时 通过加锁操作，确保文件操作的线程安全性，避免并发访问引发的问题。

```c
static struct lock lock_f;

void acquire_lock_f()
{
  lock_acquire(&lock_f);
}

void release_lock_f()
{
  lock_release(&lock_f);
}
```

sys_write 函数实现了向标准输出或文件写入数据。它首先获取用户栈中的参数，判断文件描述符。若为标准输出，调用putbuf输出数据；否则，查找对应文件描述符，调用file_write写入文件，并返回写入的字节数。

```c
void sys_write(struct intr_frame* f)
{
  uint32_t *user_ptr = f->esp;
  check_ptr(user_ptr + 7);
  check_ptr(*(user_ptr + 6));
  *user_ptr++;

  int temp = *user_ptr;
  const char *buffer = (const char *)*(user_ptr+1);  // 获取数据缓冲区
  off_t size = *(user_ptr+2);                        // 获取写入数据的大小

  // 如果是标准输出（fd == 1），使用 putbuf 写入
  if (temp == 1) {

    // putbuf 函数在文件lib/kernel/console.c 中定义，用于将数据输出到屏幕
    putbuf(buffer, size);

    f->eax = size;          // 返回写入的字节数
  }

  else {
    struct thread_file *thread_file_temp = find_file_id(*user_ptr);  // 查找文件描述符对应的文件

    if (thread_file_temp) {
      acquire_lock_f();
      f->eax = file_write(thread_file_temp->file, buffer, size);  // 写入文件
      release_lock_f();
    }

    else f->eax = 0;  // 文件描述符无效，返回0
  }
}
```

收尾工作

完成这些系统调用即相关辅助函数的实现后，再将各种声明添加到文件头中，并 #inlcude 所需文件，下面是thread.c中添加的代码，其他文件也一样。

```
#include "filesys/file.h"
#include "process.h"
#include "pagedir.h"
#include "threads/vaddr.h"
#include "filesys/filesys.h"

# define max_syscall 20
# define USER_VADDR_BOUND (void*) 0x08048000
static void (*syscalls[max_syscall])(struct intr_frame *);

void sys_exit(struct intr_frame* f);
void sys_write(struct intr_frame* f);

static void syscall_handler (struct intr_frame *);
struct thread_file * find_file_id(int fd);
```

测试

先后执行make、make check 指令，我们可以看到与args相关的5个测试用例都通过了。

~~因为 make check 运行速度比较慢，所以这里我直接截取过程中打印出的结果，没有等其他测试用例的输出。~~

```
root@7e6bb3228e11:~/pintos/src/userprog# make check
cd build && make check
make[1]: Entering directory '/home/PKUOS/pintos/src/userprog/build'
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/args-none -a args-none -- -q  -f run args-none < /dev/null 2> t
/userprog/args-none.errors > tests/userprog/args-none.output
perl -I../.. ../../tests/userprog/args-none.ck tests/userprog/args-none tests/userprog/args-none.result
pass tests/userprog/args-none
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/args-single -a args-single -- -q  -f run 'args-single onearg' -
v/null 2> tests/userprog/args-single.errors > tests/userprog/args-single.output
perl -I../.. ../../tests/userprog/args-single.ck tests/userprog/args-single tests/userprog/args-single.result
pass tests/userprog/args-single
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/args-multiple -a args-multiple -- -q  -f run 'args-multiple so
guments for you!' < /dev/null 2> tests/userprog/args-multiple.errors > tests/userprog/args-multiple.output
perl -I../.. ../../tests/userprog/args-multiple.ck tests/userprog/args-multiple tests/userprog/args-multiple.result
pass tests/userprog/args-multiple
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/args-many -a args-many -- -q  -f run 'args-many a b c d e f g
 k l m n o p q r s t u v' < /dev/null 2> tests/userprog/args-many.errors > tests/userprog/args-many.output
perl -I../.. ../../tests/userprog/args-many.ck tests/userprog/args-many tests/userprog/args-many.result
pass tests/userprog/args-many
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/args-dbl-space -a args-dbl-space -- -q  -f run 'args-dbl-space
 spaces!' < /dev/null 2> tests/userprog/args-dbl-space.errors > tests/userprog/args-dbl-space.output
perl -I../.. ../../tests/userprog/args-dbl-space.ck tests/userprog/args-dbl-space tests/userprog/args-dbl-space.result
pass tests/userprog/args-dbl-space
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/sc-bad-sp -a sc-bad-sp -- -q  -f run sc-bad-sp < /dev/null 2> t
/userprog/sc-bad-sp.errors > tests/userprog/sc-bad-sp.output
perl -I../.. ../../tests/userprog/sc-bad-sp.ck tests/userprog/sc-bad-sp tests/userprog/sc-bad-sp.result
```

心得体会

实践课第二大任务主要让我们实现参数分离和系统调用这两大任务。

参数分离需要对pintos的栈机制有一定的了解：用户空间是从高向低生长的，因此在参数压栈时要遵循“反压”的规则。　　而系统调用则主要是实现一个系统调用表，通过系统调用号来调用对应的系统调用函数。