实验五——参数传递与系统调用

课程名称:	操作系统实践	指导教师:	张民
姓　　名:	王海生	学　　号:	10235101559
实验编号:	实验五	实验名称:	参数传递与系统调用

代码仓库：`https://github.com/Hanson-Wang-chn/ECNU-Operating-System-WHS.git`

目录

1 实验目的

1. 完成参数传递和部分系统调用（`exit` 和 `write`），使得 `make check` 通过 `args` 相关的 5 个测试。
2. 完成实验报告并提交。

2 实验内容与设计思想

2.1 参数传递

1. **字符串解析：**

   为了准确地获取启动新进程所需的各项信息，首先需要对传入的命令行字符串进行解析。这里采用了 strtok_r 函数来分离进程名称和其他参数，同时为了避免直接修改原始字符串，我们在处理前对其进行了复制。

2. **内存管理：**

   考虑到性能和安全性，所有临时使用的数据结构都应在适当的时候释放。例如，在提取进程名后立即将不再需要的副本释放；而在加载程序失败的情况下，则立即清理已分配但未使用的资源。

3. **栈布局调整：**

   为了让新创建的进程能够接收到正确的参数列表，我们需要按照特定格式将参数压入栈中。这涉及到更新 `intr_frame` 结构体中的指针，确保当模拟返回中断时，新进程可以访问到完整的环境设置。

2.2 系统调用

1. **注册机制：**

   为了支持新的系统调用，必须先定义它们的编号，并为每个调用关联一个处理函数。通过建立一个数组来存储这些处理函数的地址，我们可以根据调用号快速定位到对应的处理器。

2. **实现方法：**

   - 对于 `exit` 系统调用，实现了向内核报告退出状态的功能，允许子进程结束时给父进程反馈结果。
   - `write` 系统调用负责处理标准输出流的数据写入操作，它会检查文件描述符是否合法，并将缓冲区的内容打印出来。
   - `wait` 系统调用用于等待某个子进程终止，并获取它的退出代码。为此，我们设计了一个 `exit_info` 结构来跟踪子进程的状态，并利用信号量解决了父子进程间的同步问题。

3 使用环境

3.1 主机系统配置

本次实验的主机系统环境如下表所示：

项目名称	详细信息
操作系统	macOS Sequoia 15.2
系统类型	64 位操作系统，基于 ARM 的处理器
CPU	Apple M1 Pro
GPU	Apple M1 Pro
内存	32GB 统一内存
磁盘	512GB SSD

3.2 Docker 配置

在官网下载并安装后，Docker 容器正常运行，如下图所示：

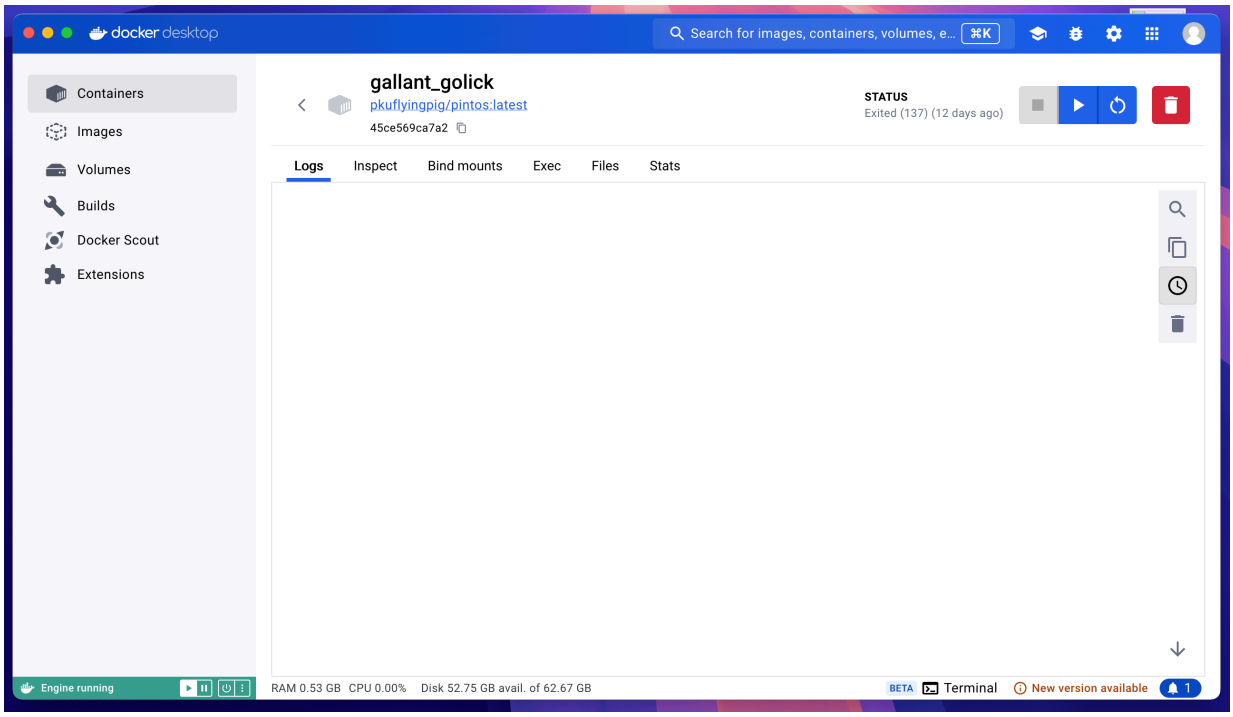

图 1: Docker 容器

接着使用下面的命令实现磁盘挂载，方便文件管理：

启动 Docker 容器并挂载文件

```
1    docker run -it --rm --name pintos --mount type=bind,source=/Users/
       wanghaisheng/Desktop/Coding/Courses/ECNU-Operating-System-WHS/pintos,
       target=/home/PKUOS/pintos pkuflyingpig/pintos bash
```

完成后如下图所示：

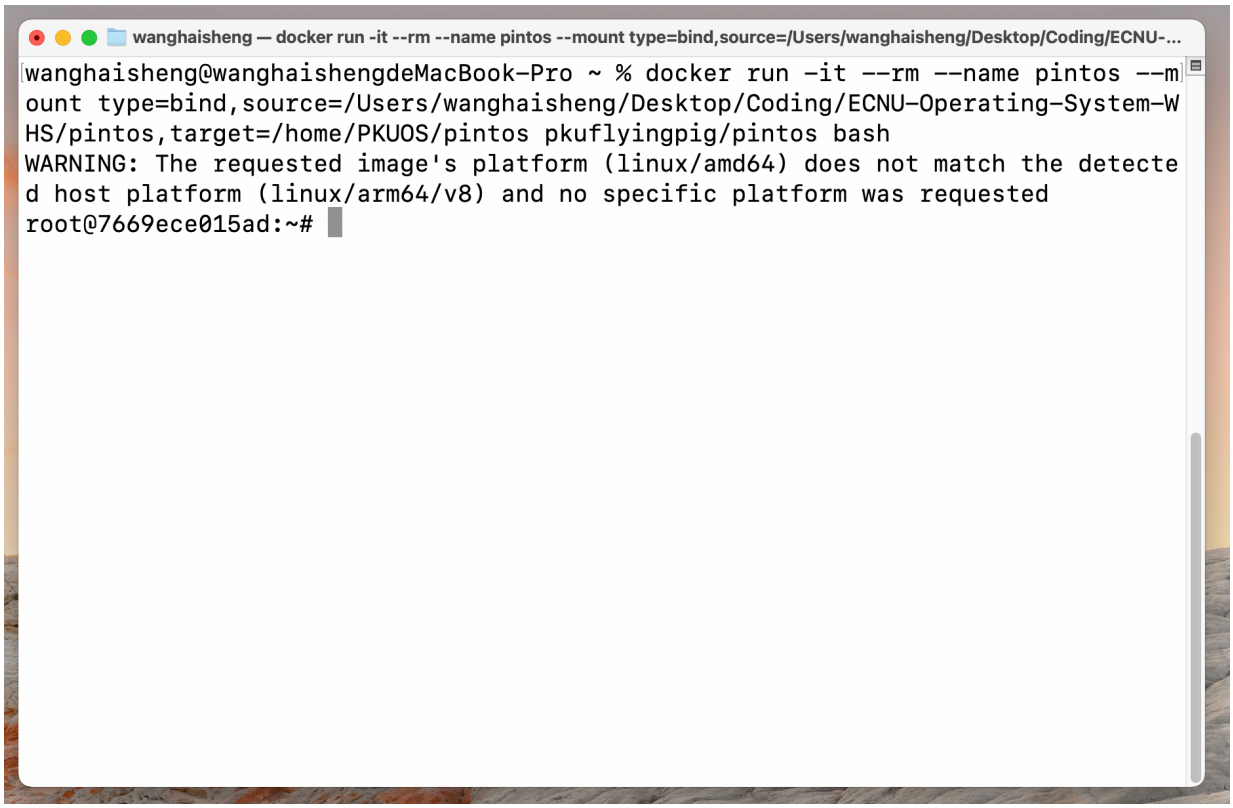

图 2: 完成环境配置

4 Argument Passing

参数分离的任务是从传入的命令字符串中解析出文件名和文件参数，因此需要对输入的字符串进行分割。在审查 process_execute 函数的过程中发现，此函数调用 thread_create 时，第一个参数代表进程的名字，而最后一个参数则是用户提供的完整字符串。

首先，应提取输入字符串的第一个子串作为进程的名字（这虽然有助于调试，但并非强制要求；也可以选择任意名称）。根据提示，采用 strtok_r 函数来实现这一操作。通过查阅该函数的文档得知，strtok_r 在处理过程中会修改原始字符串，所以应该在分割进程名称之前复制一份新的字符串以确保原始字符串保持不变。

遵循 process_execute 中的内存分配策略，为新复制的字符串分配一个页面大小的空间。考虑到该字符串的使用周期，它将在 init_thread 中被复制到进程信息结构体中，由此可以确定，在 thread_create 调用之后，该字符串将不再有其他用途，因此应在函数结束前释放该页空间。

对于 fn_copy 的生命周期管理：如果进程创建失败，则应立即释放该页；若进程创建成功，fn_copy 的所有权将转移给新进程负责，这意味着 file_name 将在 start_process 中得到适当的释放。

```
1     tid_t process_execute(const char* file_name) {
2             char* fn_copy, *name_copy; tid_t tid;
```

```
3               fn_copy = palloc_get_page(0);
4               name_copy = palloc_get_page(0);
5               if(fn_copy == NULL || name_copy == NULL) return TID_ERROR;
6               strlcpy(fn_copy, file_name, PGSIZE);
7               strlcpy(name_copy, file_name, PGSIZE);
8               char* save_ptr;
9               name_copy = strtok_r(name_copy,"",&save_ptr);
10              tid = thread_create(name_copy, PRI_DEFAULT, start_process, fn_copy);
11              palloc_free_page(name_copy);
12              if(tid == TID_ERROR) palloc_free_page(fn_copy);
13              return tid;
14      }
```

根据实验手册，进程创建的最后一步是推送一个虚拟的返回地址，具体是在 `start_process` 中通过内联汇编语句 `asm volatile("movl %0, %%esp; jmp intr_exit"::"g"(&if_):"memory");` 来实现。在此之前，需要将所有参数压入栈中，并更新 `intr_frame` 结构体。

前面提到，进程的名字不一定等同于要加载的程序名称，因此在这里需要再次复制字符串以获取实际要加载的程序名。同时需要注意的是，原始对象的生命周期在 `load` 函数执行完毕后即告终止，因此应确保释放这些资源。

在 `load` 操作完成后，实现了名为 `push_arguments` 的函数用于将参数压入栈中。该函数确保了在设置虚假返回地址之前，所有必要的参数已经被正确地放置在栈上，准备就绪以供新进程使用。

```
1       void push_argument(void** esp, char* cmd) {
2               int argc = 0, argv[64];
3               char* token, *save_ptr;
4               (*esp) = PHYS_BASE;
5               for(token = strtok_r(cmd,"␣",&save_ptr); token != NULL; token =
                    strtok_r(NULL,"␣",&save_ptr)) {
6                       size_t len = strlen(token);
7                       (*esp) -= (len + 1);
8                       memcpy((*esp), token, len + 1);
9                       argv[argc++] = (*esp);
10              }
11              (*esp) = (int)(*esp) & 0xfffffffc; // word_align
12              (*esp) -= 4, (*(int*)(*esp)) = 0; // argv[argc]
13              for(int i = argc - 1; i >= 0; i--)// argv[i];
14              (*esp) -= 4, (*(int*)(*esp)) = argv[i];
15              (*esp) -= 4, (*(int*)(*esp)) = (*esp) + 4;// argv
16              (*esp) -= 4, (*(int*)(*esp)) = argc; // argc
17              (*esp) -= 4, (*(int*)(*esp)) = 0; // return address
18      }
```

新复制的字符串同样需要严格管理其生命周期。如果 `load` 操作失败，则必须释放该字符串并终止进程，以避免内存泄漏；反之，若 `load` 成功执行，在将参数压入栈中之后应立即释放该字符串，确保资源得到妥善处理。

由于测试输出现在是通过系统调用来实现的（这与 Project 1 中直接打印的方式有所不同），当前使用 make check 进行测试将会导致所有测试结果都显示为失败。然而，根据手册指导，可以利用 hex_dump 函数来打印栈上的值，以此作为调试和验证的一种手段。

```
1     static void start_process(void* file_name_) {
2             char* file_name = file_name_;
3             struct intr_frame if_;
4             bool success;
5             /* Initialize interrupt frame and load executable.*/
6             memset(&if_, 0, sizeof if_);
7             if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
8             if_.cs = SEL_UCSEG;
9             if_.eflags = FLAG_IF | FLAG_MBS;
10            char* cmd = palloc_get_page(0);
11            strlcpy(cmd, file_name_, PGSIZE);
12            char* save_ptr;
13            file_name = strtok_r(file_name,"␣",&save_ptr);
14            success = load(file_name,&if_.eip,&if_.esp);
15            palloc_free_page(file_name);
16            if(!success) {
17                    palloc_free_page(cmd);
18                    thread_exit();
19            }
20            push_argument(&if_.esp, cmd);
21            hex_dump((uintptr_t)if_.esp, if_.esp,(PHYS_BASE) - if_.esp, true);
22            palloc_free_page(cmd);
23            /* Start the user process by simulating a return from an interrupt,
                 implemented by intr_exit(in threads/intr-stubs.S).  Because
                 intr_exit takes all of its arguments on the stack in the form of
                  a`struct intr_frame', we just point the stack pointer(%esp) to
                 our stack frame and jump to it.*/
24            asm volatile("movl␣%0,%%esp;␣jmp␣intr_exit"::"g"(&if_):"memory");
25            NOT_REACHED();
26    }
```

执行 make && pintos --filesys-size=2 -p tests/userprog/args-multiple-a args-multiple--f extract run 'args-multiple some arguments for you!' 后 hex_dump 的结果如下:

```
1     bffffffb0 00 00 00 00-05 00 00 00 c0 ff ff bf| ............|
2     bffffffc0 f2 ff ff bf ed ff ff bf-e3 ff ff bf df ff ff bf|................|
3     bffffffd0 da ff ff bf 00 00 00 00-00 00 79 6f 75 21 00 66|..........you!.f|
4     bffffffe0 6f 72 00 61 72 67 75 6d-65 6e 74 73 00 73 6f 6d|or.arguments.som|
5     bfffffff0 65 00 61 72 67 73 2d 6d-75 6c 74 69 70 6c 65 00|e.args-multiple.|
```

5 System Call

5.1 注册系统调用

系统调用号在 `lib/syscall-nr.h` 中定义。根据文档，系统调用号会被放置在栈顶，随后的参数则依次压入栈中。为了简化函数调用的过程，这里采用一个数组来存储各函数的地址。具体而言，在注册系统调用时，使用对应的系统调用号作为数组的下标，将相应的 handler 函数地址存储到该位置。这样一来，在处理系统调用时，只需要从栈中取出系统调用编号，并据此调用相应的函数即可。

这种方法不仅提高了查找效率，还使得代码结构更加清晰简洁，便于维护和扩展。

```
1     void syscall_halt(struct intr_frame* f) { }
2     void syscall_exit(struct intr_frame* f) { }
3     void syscall_exec(struct intr_frame* f) { }
4     void syscall_wait(struct intr_frame* f) { }
5     void syscall_create(struct intr_frame* f) { }
6     void syscall_remove(struct intr_frame* f) { }
7     void syscall_open(struct intr_frame* f) { }
8     void syscall_filesize(struct intr_frame* f) { }
9     void syscall_read(struct intr_frame* f) { }
10    void syscall_write(struct intr_frame* f) {
11    }
12    void syscall_seek(struct intr_frame* f) { }
13    void syscall_tell(struct intr_frame* f) { }
14    void syscall_close(struct intr_frame* f) { }
```

```
1     int (*func[20])(struct intr_frame*);
```

```
1     void syscall_init(void) {
2             intr_register_int(0x30, 3, INTR_ON, syscall_handler,"syscall");
3             func[SYS_HALT] = syscall_halt;
4             func[SYS_EXIT] = syscall_exit;
5             func[SYS_EXEC] = syscall_exec;
6             func[SYS_WAIT] = syscall_wait;
7             func[SYS_CREATE] = syscall_create;
8             func[SYS_REMOVE] = syscall_remove;
9             func[SYS_OPEN] = syscall_open;
10            func[SYS_FILESIZE] = syscall_filesize;
11            func[SYS_READ] = syscall_read;
12            func[SYS_WRITE] = syscall_write;
13            func[SYS_SEEK] = syscall_seek;
14            func[SYS_TELL] = syscall_tell;
15            func[SYS_CLOSE] = syscall_close;
16    }
```

```
1    static void syscall_handler(struct intr_frame* f UNUSED) {
2            int number = *(int*)(f->esp);
3            (func[number])(f);
4    }
```

5.2 实现系统调用

为了支持参数传递（Argument Passing）的测试，这里采用较为简单的方法来实现 SystemCall。

5.2.1 exit

根据函数声明 void exit(int status): 该函数用于终止当前进程，并向内核返回一个表示退出状态的错误码。为了终止当前进程，只能调用 thread_exit 函数，而 thread_exit 内部则会进一步调用 process_exit 函数。为了确保线程安全地将错误码返回给内核，可以将错误码存储在 thread 结构体中。请注意，在 init_thread 函数中应当初始化 t->error_code = 0。

通过全局搜索 thread_exit 可知，在任何异常退出的情况下都应设置错误码。例如，在文件 exception.c 中的 kill 函数里，应设置 thread_current()->exit_code = -1；同样地，在刚刚实现的 start_process 函数中也需进行相应的设置。为此，可以定义以下辅助函数来简化这一过程。

```
1    // thread.c
2    void error_exit() {
3            thread_current()->exit_code = -1;
4            thread_exit();
5    }
```

在系统调用中同样按照参数位置设置错误码并终止进程。

```
1    void syscall_exit(struct intr_frame* f) {
2            int exit_code = *(int*)(f->esp + 4);
3            thread_current()->exit_code = exit_code;
4            thread_exit();
5    }
```

5.2.2 write

首先就是根据函数声明 int write(int fd, const void* buffer, unsigned size) 来获取参数。fd 是文件描述符, 当 fd == 0 时是标准输入流, 不是 write 操作, 所以直接退出, 当 fd == 1 时是标准输出流, 需要把内容写到缓冲区, 本报告只实现写入缓冲区的功能。

尽管实现了上述功能，测试仍然未能通过，问题在于主进程在子进程结束之前就已经退出了。为了确保主进程正确地等待子进程完成其任务，必须在主进程中添加相应的逻辑来同步子进程的生命周期，例如使用等待（wait）系统调用来确保主进程不会提前退出。

```
void syscall_write(struct intr_frame* f) {
        int fd = *(int*)(f->esp + 4);
        char* buf = *(char**)(f->esp + 8);
        int size = *(int*)(f->esp + 12);
        if(fd == 0) error_exit();
        if(fd == 1) {
                putbuf(buf, size);
                f->eax = size;
        }
}
```

5.2.3 wait

根据函数声明 `int wait(pid_t pid)`，该函数的作用是依据传入的子进程 `pid` 来获取并返回其退出状态码。为了确保在子进程退出后仍能访问到它的相关信息，解决方案是在每个进程中创建一个名为 `exit_info` 的结构体，用于存储该进程的相关信息。具体来说，每个父进程会维护一个 `exit_info` 列表，用来记录它所有的子进程的信息。此外，为了保证父进程和子进程之间的同步，将使用信号量（semaphore）来管理。

`exit_info` 结构体应存储在堆上，并通过 `malloc` 分配内存。关于 `exit_info` 的生命周期，它从进程创建时开始，直到没有任何进程需要这段信息为止——即当父进程结束时终止。因此，`exit_info` 中还应该包含一个指向父进程的引用 `parent`，以跟踪和管理其生命周期。这确保了在适当的时候可以正确地释放与子进程相关的资源，同时避免了内存泄漏问题。

```
struct thread {
        /* Owned by thread.c.*/
        tid_t tid; /**< Thread identifier.*/
        enum thread_status status; /**< Thread state.*/
        char name[16]; /**< Name(for debugging purposes).*/
        uint8_t* stack; /**< Saved stack pointer.*/
        int priority; /**< Priority.*/
        struct list_elem allelem; /**< List element for all threads list.*/
        struct list_elem waitelem;
        int64_t wake_tick;
        /* Shared between thread.c and synch.c.*/
        struct list_elem elem; /**< List element.*/
        #ifdef USERPROG
        /* Owned by userprog/process.c.*/
        uint32_t* pagedir; /**< Page directory.*/
        uint32_t exit_code;
        struct list child_list;
        struct exit_info* linked_exit;
        struct semaphore sema_wait;
        #endif
```

```
21              /* Owned by thread.c.*/
22              unsigned magic; /**< Detects stack overflow.*/
23      };
24      struct exit_info {
25              tid_t tid;
26              struct thread* linked_thread;
27              struct thread* parent;
28              int exit_code;
29              bool is_still_alive;
30              bool is_being_waited;
31              struct list_elem child_elem;
32      };
```

在 thread_create 和 init_thread 中进行初始化。

```
1       tid_t thread_create(const char* name, int priority, thread_func* function,
            void* aux) {
2               struct thread* t;
3               struct kernel_thread_frame* kf;
4               struct switch_entry_frame* ef;
5               struct switch_threads_frame* sf;
6               tid_t tid;
7               ASSERT(function != NULL);
8               /* Allocate thread.*/
9               t = palloc_get_page(PAL_ZERO);
10              if(t == NULL) return TID_ERROR;
11              /* Initialize thread.*/
12              init_thread(t, name, priority);
13              tid = t->tid = allocate_tid();
14              t->linked_exit = malloc(sizeof(struct exit_info));
15              t->linked_exit->parent = thread_current();
16              t->linked_exit->tid = tid;
17              t->linked_exit->linked_thread = t;
18              t->linked_exit->exit_code = 0;
19              t->linked_exit->tid = tid;
20              t->linked_exit->is_still_alive = true;
21              t->linked_exit->is_being_waited = false;
22              list_push_back(&t->linked_exit->parent->child_list,&t->linked_exit->
                    child_elem);
23              /* Stack frame for kernel_thread().*/
24              kf = alloc_frame(t, sizeof*kf);
25              kf->eip = NULL;
26              kf->function = function;
27              kf->aux = aux;
28              /* Stack frame for switch_entry().*/
```

```
29              ef = alloc_frame(t, sizeof*ef);
30              ef->eip = (void(*)(void)) kernel_thread;
31              /* Stack frame for switch_threads().*/
32              sf = alloc_frame(t, sizeof*sf);
33              sf->eip = switch_entry;
34              sf->ebp = 0;
35              /* Add to run queue.*/
36              thread_unblock(t);
37              thread_yield();
38              return tid;
39      }
40      static void init_thread(struct thread* t, const char* name, int priority) {
41              enum intr_level old_level;
42              ASSERT(t != NULL);
43              ASSERT(PRI_MIN <= priority && priority <= PRI_MAX);
44              ASSERT(name != NULL);
45              memset(t, 0, sizeof*t);
46              t->status = THREAD_BLOCKED;
47              strlcpy(t->name, name, sizeof t->name);
48              t->stack = (uint8_t*) t + PGSIZE;
49              t->priority = priority;
50              t->magic = THREAD_MAGIC;
51              t->wake_tick = -1;
52              t->exit_code = 0;
53              list_init(&t->child_list);
54              sema_init(&t->sema_wait, 0);
55              old_level = intr_disable();
56              list_insert_ordered(&all_list,&t->allelem, thread_more_priority,
                    NULL);
57              // list_push_back(&all_list,&t->allelem);
58              intr_set_level(old_level);
59      }
```

在系统调用中，只需调用 process_wait 即可。根据文档描述，process_wait 的功能是查找子进程；如果找到的子进程仍在运行，则执行 P 操作以使当前进程阻塞，直至子进程结束并获取其退出码。成功返回后，还需确保从相关链表中移除已等待的子进程记录。

相应地，每当有 P 操作存在时，必然伴随有 V 操作用于解除阻塞。具体到 process_exit 函数中，它会遍历所有子进程来更新每个子进程的父进程引用 (exit_info->parent)。接着，该函数检查当前进程的退出信息 (exit_info) 生命周期是否已经完成；若确实已完成，则进行资源释放。最后，更新或清理当前进程的退出信息字段。

```
1       int process_wait(tid_t child_tid UNUSED) {
2               struct thread* cur = thread_current();
3               struct list* l = &cur->child_list;
```

```
4               struct list_elem* e = NULL;
5               struct exit_info* child;
6               for(e = list_begin(l); e != list_end(l); e = list_next(e))
7               {
8                       child = list_entry(e, struct exit_info, child_elem);
9                       if(child->tid == child_tid) break;
10              }
11              if(e == list_end(l)) return -1;
12              list_remove(e);
13              if(child->is_still_alive) {
14                      child->is_being_waited = true;
15                      sema_down(&child->linked_thread->sema_wait);
16              }
17              return child->exit_code;
18      }
19      void process_exit(void) {
20              struct thread* cur = thread_current();
21              uint32_t* pd;
22              struct list* l = &cur->child_list;
23              struct list_elem* e;
24              for(e = list_begin(l); e != list_end(l); e = list_next(e)) {
25                      struct exit_info* tmp = list_entry(e, struct exit_info,
                            child_elem);
26                      if(tmp->is_still_alive) tmp->linked_thread->linked_exit->
                            parent = NULL;
27              }
28              if(cur->linked_exit->parent == NULL) free(cur->linked_exit);
29              else {
30                      cur->linked_exit->exit_code = cur->exit_code;
31                      if(cur->linked_exit->is_being_waited) sema_up(&cur->
                            sema_wait);
32                      cur->linked_exit->is_still_alive = false;
33                      cur->linked_exit->linked_thread = NULL;
34              }
35              pd = cur->pagedir;
36              if(pd != NULL) {
37                      cur->pagedir = NULL;
38                      pagedir_activate(NULL);
39                      pagedir_destroy(pd);
40              }
41              printf("%s: exit(%d)\n", cur->name, cur->linked_exit->exit_code);
42      }
```

wait 的 Syscall 只需要调用 process_wait 函数即可。

```
1     void syscall_wait(struct intr_frame* f) {
2             int pid = *(int*)(f->esp + 4);
3             f->eax = process_wait(pid);
4     }
```

至此可以通过 `args` 相关的测试点。

```
1     pass tests/userprog/args-none
2     pass tests/userprog/args-single
3     pass tests/userprog/args-multiple
4     pass tests/userprog/args-many
5     pass tests/userprog/args-dbl-space
```

至此，本实验相关的测试点全部通过.

6 实验总结

通过本次实验，我对操作系统中的参数传递和系统调用有了更深入的理解与掌握。在参数传递部分，学会了如何准确地解析命令行字符串，合理管理内存以及正确调整栈布局，这使我明白了进程创建过程中数据处理的重要性和复杂性。在系统调用方面，成功实现了 `exit`、`write` 和 `wait` 等系统调用，理解了系统调用的注册机制和实现方法，并且通过信号量解决了父子进程间的同步问题，这让我对操作系统的进程管理有了更清晰的认识。