



华中科技大学

操作系统原理课程实验报告

姓 名： 徐锦慧

学 院： 计算机科学与技术

专 业： 计算机科学与技术

班 级： CS2011

学 号： U202011675

指导教师： 阳富民

分数	
教师签名	

2022 年 12 月 26 日

目 录

1 实验二·挑战一	1
1.1 实验任务	1
1.2 实验内容	1
1.3 实验调试及心得	2
2 实验三·挑战一	4
2.1 实验任务	4
2.2 实验内容	4
2.3 实验调试及心得	8

1 实验二·挑战一

1.1 实验任务

本实验为挑战实验，给定应用 `app_sum_sequence.c`，需要实现复杂缺页异常等内容，使应用能在控制台输出正确结果。基础代码将继承和使用 `lab2_3` 完成后的代码。具体要求为：

- 1) 修改进程的数据结构以对虚拟地址空间进行监控。
- 2) 修改 `kernel/strap.c` 中的异常处理函数。对于合理的缺页异常，扩大内核栈大小并为其映射物理块；对于非法地址缺页，报错并退出程序。

最终要通过修改 PKE 内核，使得对于不同情况的缺页异常进行不同的处理。文件名需要包含路径，如果是用户源程序发生的错误，路径为相对路径，如果是调用的标准库内发生的错误，路径为绝对路径。

1.2 实验内容

1. 分析给定应用

首先观察给定应用 `app_sum_sequence.c` 文件。关键代码如下：

```
uint64 sum_sequence(uint64 n, int *p) {  
    if (n == 0)  
        return 0;  
    else  
        return *p=sum_sequence( n-1, p+1 ) + n;  
}
```

程序思路基本同 `lab2_3` 一致，对给定 `n` 计算 0 到 `n` 的和，但要求将每一步递归的结果保存在数组 `ans` 中。创建数组时，我们使用了当前的 `malloc` 函数申请了一个页面（4KB）的大小，对应可以存储的个数上限为 1024。

在函数调用时，我们试图计算 1025 求和，首先由于 `n` 足够大，所以在函数递归执行时会触发用户栈的缺页，需要对其进行正确处理，确保程序正确运行；其次，1025 在最后一次计算时会访问数组越界地址，由于该处虚拟地址尚未有对应的物理地址映射，因此属于非法地址的访问，这是不被允许的，对于这种缺页异常，应该提示用户并退出程序执行。

根据输出结果可以看出：前八个缺页是由于函数递归调用引起的，而最后一个缺页是对动态申请的数组进行越界访问造成的，访问非法地址，程序报错退出。

2. 修改 handle_user_page_fault()函数

由 lab2_3 可知，产生缺页异常的本质还是应用往未被映射的内存空间“写”所导致的，通过分析代码可知，CAUSE_STORE_PAGE_FAULT 这类异常被放在 kernel/strap.c 文件中处理，因此需要修改该文件中的 handle_user_page_fault()函数。

接着我们需要通过输入的参数 stval 判断缺页异常是否合法。对于合理的缺页异常，扩大内核栈大小并为其映射物理块；对于非法地址缺页，报错并退出程序。

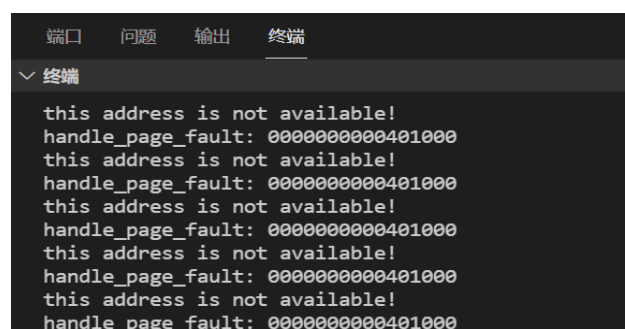
stval 存放的是发生缺页异常时程序想要访问的逻辑地址，我们用它来判断缺页的逻辑地址在用户进程逻辑地址空间中的位置，看是不是比 USER_STACK_TOP 小，且比我们预设的可能的用户栈的最小栈底指针要大。若满足，则为合法的逻辑地址，否则报错并退出。通过前面的实验可知，USER_STACK_TOP 即 0x7ffff000；又由于 sp 寄存器会存储栈顶地址，比栈顶更低的地址属于堆空间，不能越界访问，因此需要 stval 大于当前栈顶-PGSIZE。

```
if(stval < 0x7ffff000 && stval > current->trapframe->regs.sp-PGSIZE){
    void* page = alloc_page();
    map_pages((pagetable_t)current->pagetable, ROUNDDOWN(stval,PGSIZE),
PGSIZE, (uint64)page, prot_to_type(PROT_WRITE | PROT_READ, 1));
}else{
    panic("this address is not available!");
}
```

1.3 实验调试及心得

1.3.1 实验调试

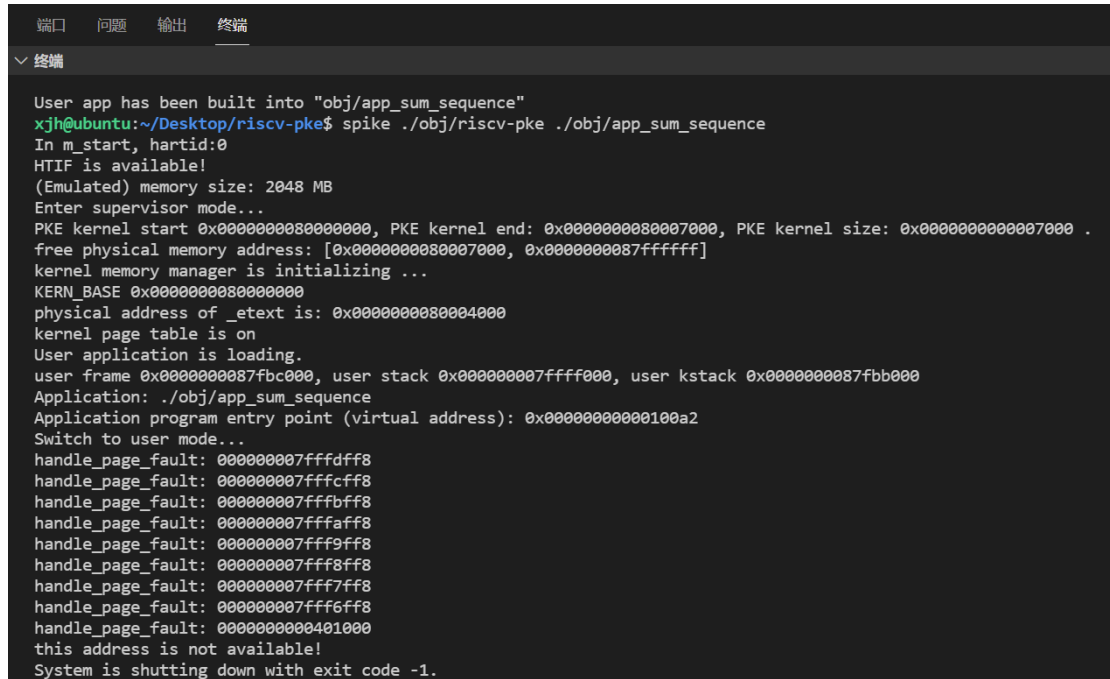
最初经过以上分析后，使用 sprint()函数输出错误信息并用 return 退出，导致出现如图 1.1 所示的报错：



The image shows a terminal window with a dark background. At the top, there are four tabs labeled '端口', '问题', '输出', and '终端'. The '终端' tab is selected and expanded, showing a list of error messages. The messages are: 'this address is not available!', 'handle_page_fault: 000000000401000', 'this address is not available!', 'handle_page_fault: 000000000401000', 'this address is not available!', 'handle_page_fault: 000000000401000', 'this address is not available!', 'handle_page_fault: 000000000401000', 'this address is not available!', and 'handle_page_fault: 000000000401000'. The messages are repeated in a loop.

图 1.1 输出错误

经过检查发现，需要使用 `panic()` 函数来完成退出。此外，还需要注意 `panic()` 函数中输出的字符。在最初修改后由于多加了换行符 “\n”，导致没有通过头歌平台的测试。删除多余字符后得到正确输出，如图 1.2 所示：



```
端口 问题 输出 终端
▼ 终端
User app has been built into "obj/app_sum_sequence"
xjh@ubuntu:~/Desktop/riscv-pke$ spike ./obj/riscv-pke ./obj/app_sum_sequence
In m_start, hartid:0
HTIF is available!
(Emulated) memory size: 2048 MB
Enter supervisor mode...
PKE kernel start 0x000000080000000, PKE kernel end: 0x000000080007000, PKE kernel size: 0x000000000007000 .
free physical memory address: [0x000000080007000, 0x000000087ffffff]
kernel memory manager is initializing ...
KERN_BASE 0x000000080000000
physical address of _etext is: 0x000000080004000
kernel page table is on
User application is loading.
user frame 0x000000087fbc000, user stack 0x00000007ffff000, user kstack 0x000000087fbb000
Application: ./obj/app_sum_sequence
Application program entry point (virtual address): 0x0000000000100a2
Switch to user mode...
handle_page_fault: 00000007ffdf8
handle_page_fault: 00000007ffcf8
handle_page_fault: 00000007ffbf8
handle_page_fault: 00000007ffaf8
handle_page_fault: 00000007ff9f8
handle_page_fault: 00000007ff8f8
handle_page_fault: 00000007ff7f8
handle_page_fault: 00000007ff6f8
handle_page_fault: 00000000401000
this address is not available!
System is shutting down with exit code -1.
```

图 1.2 正确输出

1.3.2 心得体会

通过本次实验，我更深入的认识了 PKE 操作系统内核中用户态栈空间的管理、用户进程的压栈请求，学会了如何处理不同情况的缺页异常。

此外，我也学会了如何根据给定文件分析代码及其内部关系，通过注释、参数等推断函数的作用。在今后的学习中，还需更注重细节，例如一个换行符就可能导致输出的结果无法通过测试。

2 实验三 • 挑战一

2.1 实验任务

本实验为挑战实验，给定应用 `user/app_wait.c`，需要实现进程等待和数据段复制等内容，使应用能在控制台输出正确结果。基础代码将继承和使用 `lab3_3` 完成后的代码。具体要求为：

1) 通过修改 `PKE` 内核和系统调用，为用户程序提供 `wait()` 函数的功能，`wait()` 函数接受一个参数 `pid`：

当 `pid` 为 -1 时，父进程等待任意一个子进程退出即返回子进程的 `pid`；

当 `pid` 大于 0 时，父进程等待进程号为 `pid` 的子进程退出即返回子进程的 `pid`；

如果 `pid` 不合法或 `pid` 大于 0 且 `pid` 对应的进程不是当前进程的子进程，返回 -1。

2) 补充 `do_fork()` 函数，实验 3_1 实现了代码段的复制，还需要继续实现数据段的复制并保证 `fork` 后父子进程的数据段相互独立。

最终测试程序可能和给出的用户程序不同，但都只涉及 `wait()` 函数、`fork()` 函数和全局变量读写的相关操作。

2.2 实验内容

1. 分析给定应用

首先观察给定应用 `user/app_wait.c` 文件，主要代码如下：

```
int main(void) {
    flag = 0;
    int pid = fork();
    if (pid == 0) {
        flag = 1;
        pid = fork();
        if (pid == 0) {
            flag = 2;
            printu("Grandchild process end, flag = %d.\n", flag);
        } else {
            wait(pid);
            printu("Child process end, flag = %d.\n", flag);
        }
    } else {
        wait(-1);
        printu("Parent process end, flag = %d.\n", flag);
    }
}
```

```

    }
    exit(0);
    return 0;
}

```

在以上程序中，父进程把 `flag` 变量赋值为 0，然后 `fork` 生成一个子进程，接着通过 `wait()` 函数等待子进程的退出。子进程把自己的变量 `flag` 赋值为 1，然后 `fork` 生成孙子进程，接着通过 `wait()` 函数等待孙子进程的退出。孙子进程给自己的变量 `flag` 赋值为 2 并在退出时输出信息，然后子进程退出时输出信息，最后父进程退出时输出信息。由于 `fork` 之后父子进程的数据段相互独立，子进程对全局变量的赋值不影响父进程全局变量的值，因此会输出所给结果。

2. 补充 `do_fork()` 函数

在实验 3_1 中已经实现了代码段的复制，为确保父子进程的数据段相互独立，还需要实现数据段的复制，即在 `do_fork()` 函数中添加 `case DATA_SEGMENT` 的处理。

在代码段中，我们直接在子进程的虚地址转换表中添加父进程的代码段首地址及其与实地址之间的映射，把父进程的实地址复制给子进程。但是对于数据段，父进程和子进程是不共享的，所以需要生成申请一个新的页给子进程，再通过 `map_pages()` 函数完成虚地址到实地址之间的映射。关键代码如下：

```

case DATA_SEGMENT:
    map_pages(child->pagetable, parent->mapped_info[i].va,
              parent->mapped_info[i].npages*PGSIZE, (uint64)alloc_page(),
              prot_to_type(PROT_WRITE | PROT_READ | PROT_EXEC, 1));

```

在 `map_pages()` 函数中，`child->pagetable` 为子进程所在物理页面的首地址，`(uint64)alloc_page()` 为被映射的物理地址首地址，即为子进程新申请的页面首地址，`parent->mapped_info[i].npages*PGSIZE` 为所要建立的映射区间长度，`parent->mapped_info[i].va` 为将要被映射的逻辑地址，`prot_to_type(PROT_WRITE | PROT_READ | PROT_EXEC, 1)` 为映射建立后页面访问的权限。

通过 `map_pages()` 函数完成映射后，还需要完成虚拟区域的注册，关键代码如下：

```

// after mapping, register the vm region (do not delete codes below!)
child->mapped_info[child->total_mapped_region].va =
parent->mapped_info[i].va;
child->mapped_info[child->total_mapped_region].npages =

```

```
parent->mapped_info[i].npages;
    child->mapped_info[child->total_mapped_region].seg_type = DATA_SEGMENT;
    child->total_mapped_region++;
    break;
```

3. 添加系统调用

通过观察 `user_lib.c` 文件，并和 `fork()` 函数类比，可以完成进程等待的系统调用。首先在该文件中添加 `wait()` 函数，代码如下：

```
int wait(int pid){
    return do_user_call(SYS_user_wait, pid, 0, 0, 0, 0, 0, 0);
}
```

接着在 `syscall.h` 文件中完成对 `SYS_user_wait` 的注册，代码如下：

```
// added @lab3_challenge_1
#define SYS_user_wait (SYS_user_base + 6)
```

在 `syscall.c` 文件的 `do_syscall()` 函数中添加 `SYS_user_wait` 情况的实现，代码如下：

```
case SYS_user_wait:
    return sys_user_wait(a1);
```

并在该文件中添加 `sys_user_wait()` 函数，代码如下：

```
ssize_t sys_user_wait(int pid) {
    // sprint("User call wait.\n");
    return do_wait(pid);
}
```

以上均为添加进程等待的系统调用部分，完成这些全局变量的定义后，在 `do_wait()` 函数中实现真正的进程等待功能。

4. 实现 `do_wait()` 函数

首先考虑进程间等待停止阻塞的关系如何表示。我们在 `process` 数据结构里定义一个新的元素：`blockmap`。`Blockmap` 为当前进程正在等待停止阻塞的进程号，例如当前进程正在等待 `pid=5` 的进程结束，则该进程的 `blockmap` 第 5 位为 1。通过 `blockmap` 和按位运算，我们可以很容易的表示进程间的等待停止阻塞关系： $\text{blockmap} \mid= 1 \ll \text{pid}$ 。

接下来考虑 `do_wait()` 的逻辑结构。`do_wait()` 函数传入一个参数 `pid`，当 `pid` 为 -1 时，父进程等待任意一个子进程退出；当 `pid` 大于 0 时，父进程等待进程号为 `pid` 的子进程；如果 `pid` 不合法或 `pid` 大于 0 且 `pid` 对应的进程不是当前进程

的子进程，则返回-1。经过以上分析，do_wait()函数的流程图如图 2.1 所示：

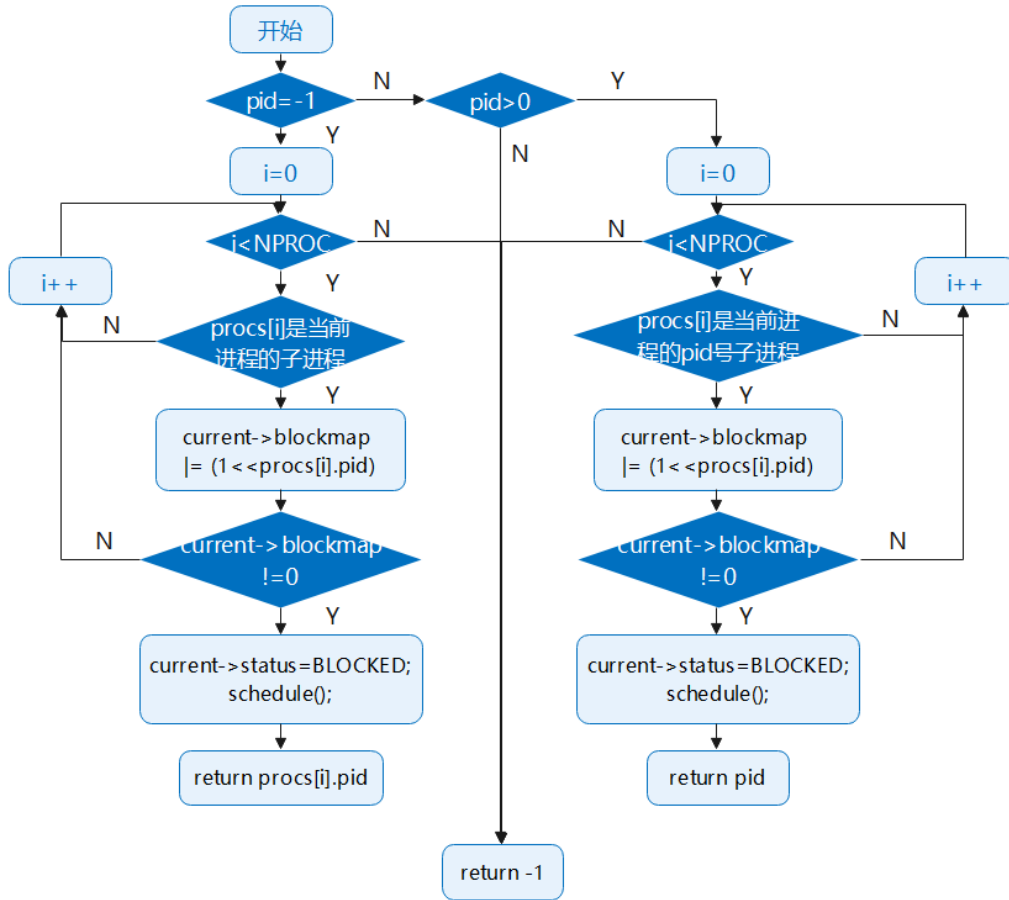


图 2.1 do_wait() 函数流程图

5. 实现 do_exit()函数

完成以上步骤后，还需要补充 syscall.c 文件中的 exit()函数。当一个进程结束时有可能会通知给其他进程，我们通过在 exit()函数中调用 do_exit()来实现。如果当前进程是某个进程的子进程，首先找出等待停止阻塞的进程号 pid，如果父进程的状态为 BLOCKED，并且当前进程的进程号也为 pid，则将父进程的状态改为 READY，并加入就绪队列。代码如下：

```

void do_exit()
{
    // It is possible to notify other processes when a process ends
    if(current->parent){
        // This process is a son of a process
        int pid=0;
        uint64 n=(current->parent->blockmap)&(1<<current->pid);
        current->parent->blockmap^=(1<<current->pid);
        for(pid=0;(n&1)==0;pid++){

```

```

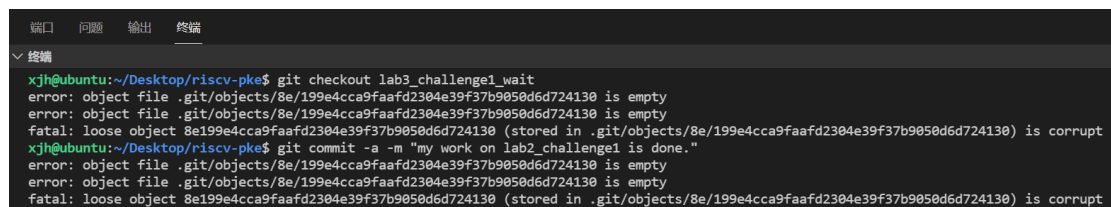
        n>>=1;
    }
    if(current->parent->status==BLOCKED && current->pid==pid){
        current->parent->status=READY;
        insert_to_ready_queue(current->parent);
    }
}
}
}

```

2.3 实验调试及心得

2.3.1 实验调试

当实验开始，通过 `git checkout` 切换分支至 `lab3_challenge1_wait` 时，终端出现如图 2.2 所示的报错：



```

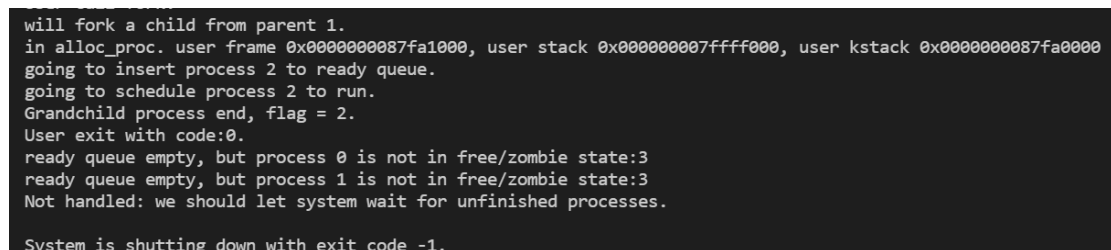
xjh@ubuntu:~/Desktop/riscv-pke$ git checkout lab3_challenge1_wait
error: object file .git/objects/8e/199e4cca9faafd2304e39f37b9050d6d724130 is empty
error: object file .git/objects/8e/199e4cca9faafd2304e39f37b9050d6d724130 is empty
fatal: loose object 8e199e4cca9faafd2304e39f37b9050d6d724130 (stored in .git/objects/8e/199e4cca9faafd2304e39f37b9050d6d724130) is corrupt
xjh@ubuntu:~/Desktop/riscv-pke$ git commit -a -m "my work on lab2_challenge1 is done."
error: object file .git/objects/8e/199e4cca9faafd2304e39f37b9050d6d724130 is empty
error: object file .git/objects/8e/199e4cca9faafd2304e39f37b9050d6d724130 is empty
fatal: loose object 8e199e4cca9faafd2304e39f37b9050d6d724130 (stored in .git/objects/8e/199e4cca9faafd2304e39f37b9050d6d724130) is corrupt

```

图 2.2 切换分支错误

经过查询相关资料发现，这是由于上一次 `git commit` 未成功导致，导致修改后的代码全部在更改暂存区。通过取消暂存区更改，重新 `git commit` 后成功解决上述问题。

接着通过 2.2 节中的分析来实现 `do_fork()`、`do_wait()`、`do_exit()` 函数，运行后部分输出错误，如图 2.3 所示：



```

will fork a child from parent 1.
in alloc_proc. user frame 0x0000000087fa1000, user stack 0x000000007ffff000, user kstack 0x0000000087fa0000
going to insert process 2 to ready queue.
going to schedule process 2 to run.
Grandchild process end, flag = 2.
User exit with code:0.
ready queue empty, but process 0 is not in free/zombie state:3
ready queue empty, but process 1 is not in free/zombie state:3
Not handled: we should let system wait for unfinished processes.

System is shutting down with exit code -1.

```

图 2.3 `do_exit()` 错误

经过分析发现，这是由于 `process.c` 中的 `do_exit()` 函数出错，调试后得到正确输出，如图 2.4 所示：

```
终端
Switch to user mode...
in alloc_proc. user frame 0x000000087fbc000, user stack 0x00000007ffff000, user kstack 0x000000087fbb000
User application is loading.
Application: ./obj/app_wait
CODE_SEGMENT added at mapped info offset:3
DATA_SEGMENT added at mapped info offset:4
Application program entry point (virtual address): 0x0000000000100b0
going to insert process 0 to ready queue.
going to schedule process 0 to run.
User call fork.
will fork a child from parent 0.
in alloc_proc. user frame 0x000000087fae000, user stack 0x00000007ffff000, user kstack 0x000000087fad000
going to insert process 1 to ready queue.
going to schedule process 1 to run.
User call fork.
will fork a child from parent 1.
in alloc_proc. user frame 0x000000087fa1000, user stack 0x00000007ffff000, user kstack 0x000000087fa0000
going to insert process 2 to ready queue.
going to schedule process 2 to run.
Grandchild process end, flag = 2.
User exit with code:0.
going to insert process 1 to ready queue.
going to schedule process 1 to run.
Child process end, flag = 1.
User exit with code:0.
going to insert process 0 to ready queue.
going to schedule process 0 to run.
Parent process end, flag = 0.
User exit with code:0.
no more ready processes, system shutdown now.
System is shutting down with exit code 0.
```

图 2.4 正确输出结果

2.3.2 心得体会

通过本次实验，我对 wait 系统调用有了更深入的认识，它是进程管理中一个非常重要的系统调用，主要的功能为收回其他进程资源、进程同步。当一个进程退出后，它所占用的资源并不一定能立即收回，需要父进程调用进程等待来收回资源。

此外，通过在 fork() 函数中实现数据段复制，我对系统内核的数据段也有了更深刻的理解。不同于代码段，数据段的父进程和子进程并不共享，它们的虚地址相通但实地址并不相通，因此要申请一个新页面给子进程来完成复制。

实验过程中也遇到了一些难题。例如使用 git 提交代码失败、do_exit() 函数出错等，通过查阅相关资料、请教老师同学，最终都顺利解决了。

总的来说，本次实验趣味性很强，通过所给函数的蛛丝马迹来实现正确输出，有种探险寻宝的感觉；所给的指导资料很详细到位，老师们也在课程群里很积极的答疑解惑，在此非常感谢课程组老师们的辛苦付出！