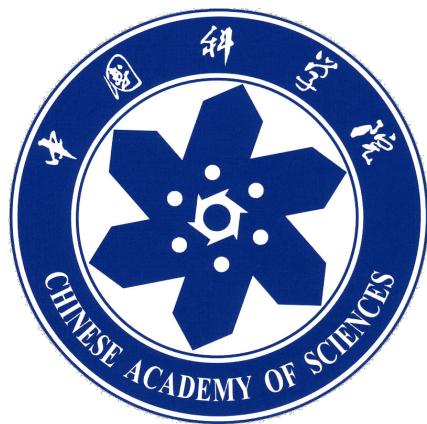


国科大操作系统研讨课任务书

Project 3



版本：3.0

一、实验说明	3
二、简易终端的制作.....	4
2.1 一块屏幕（screen）	4
2.2 一个可以解析指令的进程（shell）	4
2.3 任务 1：终端的实现.....	4
三、进程的创建和退出	6
3.1 spawn(task_t)	6
3.2 kill(pid_t)	6
3.3 exit(void).....	6
3.4 wait(pid_t)	6
3.5 任务 2：spawn、kill、exit、waitpid 方法的实现	6
四、同步原语	10
4.1 信号量（semaphores）	10
4.2 条件变量（condition variables）	10
4.3 屏障（barriers）	10
4.4 任务 3：实现同步原语：semaphores、condition variables、barriers	11
五、进程间通信	14
5.1 生产者——消费者模型.....	14
5.2 任务 4：进程间的通信——mailbox 实现	14

一、实验说明

通过之前的实验，我们的操作系统已经能够具备任务的调度、例外的处理，并且在用户态运行的进程已经可以通过系统调用接口去调用内核的代码（我们现在用户态和内核态的界限尚不明显，但这一条界限将在后续的虚存中划分出来）。

我们将继续完善一些任务处理相关的功能，比如：任务的启动、杀死一个正在运行的任务、任务的正常退出，以及任务间的同步、通信等功能。本次的实验内容如下：

任务一：实现一个简易的终端，实现 OS 和用户的简单交互。

任务二：了解 `spawn`、`kill`、`exit`、`waitpid` 的原理，实现其功能并做成系统调用接口。

任务三：了解条件变量（`condition variables`）、信号量（`semaphores`）、屏障（`barriers`），这三类同步原语的原理，并实现其功能并做成系统调用接口。

任务四：基于之前实现的同步原语实现进程间的通信（`IPC`）。

在本次实验中，尚不涉及内核模块的增添，主要完成的是几个系统调用。但随着内核功能的增多，大家可能会发现自己之前设计的一些不合理处，希望大家可以通过本次实验完善自己的代码。

二、简易终端的制作

为了方便我们接下来的任务运行和测试，我们要求同学们在本次任务完成一个小型终端的制作。这个终端可以实现用户指令的输入，用户指令的解析，最终根据用户指令的内容，调用内核相关的系统调用。

对于一个终端的定义，在传统的操作系统还是比较复杂。但考虑到任务的难度，我们在这里简化了终端的定义，大家可以简单的认为，我们这里要实现的终端就是一块屏幕（screen）、一个可以解析指令的进程（shell）。

2.1 一块屏幕（screen）

所谓的一块屏幕，实际上就是我们之前提供给大家的 screen 库，在本次实验大家不需要考虑这部分的实现，使用我们提供的现成的库 printf 即可。

2.2 一个可以解析指令的进程（shell）

shell 本质上其实是一个进程，只不过它的功能比较特殊，它负责用户和内核的交互，再具体的说 shell 负责从用户读取指令，将指令传递给内核执行，并将结果反馈给用户。我们从指令的执行流程来分析 shell 的功能：

(1) 功能 1：从输入流读取数据，这里的数据流实际上就是串口输入，关于串口输入的读取，我们在本次实验中已经给大家提供了现成的函数，大家直接用就可以了。

(2) 功能 2：读取到了指令后，shell 要做的就是解析，根据用户的输入判断用户要执行什么指令，如果输入有效就调用内核相关的系统调用。

(3) 功能 3：最后，用户输入了一串字符串，shell 进程虽然读到并解析了，但是并没有打印到屏幕上，用户也看不见，因此 shell 需要将读取到的字符串打印到屏幕上。

2.3 任务 1：终端的实现

2.3.1 实验要求

功能 1: 可以实现用户命令输入的读取、解析、显示，根据用户输入调用相关系统调用。

功能 2: 实现分屏功能，该终端启动的任务输出和用户的指令输入分开。

功能 3: 实现 ps (process show) 指令，实现系统调用 sys_ps，在终端输入 ps 可以打印出正在运行的任务列表。

功能 4: 实现 clear，输入 clear 可以实现清屏。

以下为最终效果的参考图，下半屏幕用于 shell 的输入输出，上半屏幕用于测试用例的输出：

```
> [TASK] I am task with pid 2, I have acquired two mutex lock. (26)
> [TASK] I want to acquire a mute lock from task(pid=2).
> [TASK] I want to wait task (pid=2) to exit.

----- COMMAND -----
> root@UCAS_OS: ps
[PROCESS TABLE]
[0] PID : 0 STATUS : RUNNING
[1] PID : 1 STATUS : RUNNING
> root@UCAS_OS: spawn
> root@UCAS_OS:
```

除了上述的需要实现的功能，同学们还可以根据自己的理解去完善，比如实现滚屏等额外的功能。

2.3.2 文件介绍

在 project2 的基础上，我们需要添加以下的文件：

	文件名	说明
1	test/project3/test3.h	project3 的测试头文件
2	test/test_shell.c	任务一的相关框架，补充其内容完成任务一
3	test/project3/test_kill.c	测试任务二的完成，测试 spawn、exit、kill、wait
4	test/project3/test_semaphore.c	测试任务三的完成，测试信号量 (semaphore)
5	test/project3/test_condition.c	测试任务三的完成，测试条件变量 (condition variables)
6	test/project3/test_barrier.c	测试任务三的完成，测试屏障 (barrier)
7	test/project3/test_sanguo.c	测试任务四的完成，测试进程通信
8	include/os/cond.h kernel/locking/cond.c	任务三的条件变量实现
9	include/os/sem.h kernel/locking/sem.c	任务三的信号量实现
10	include/os/barrier.h kernel/locking/barrier.c	任务三的屏障实现
11	lib/mailbox.c lib/mailbox.h	任务四的进程通信实现
12	drivers/screen.c drivers/screen.h	虚拟屏幕的实现

2.3.3 注意事项

大家可以发现，我们对于输入流的读取是采用轮询的方式进行的，shell 这个任务会不停的从串口输入中获取数据，有的时候可以拿到数据，有的时候拿不到。但实际上，对于串口这种设备的访问，我们通常是采取中断的方式，每来一个数据，外部设备就会产生一个中断，内核就会响应，去处理。这部分内容我们留给后续的实验完善，在本次实验中我们还是采用较为直接简单的方法：轮询。

刚才也说过，一个终端的完整实现远远比我们本次要求的复杂，本次实验也只是希望大家完成一个简单的具备交互功能的终端，关于它的进一步完善，比如：底层设备的管理，更复杂的字符串处理等功能，将会随着我们实验的推进一步完善。

此外，shell 在本次实验中担任着启动所有任务的角色，因此我们在内核初始化的时候，就把 shell 启动起来，将它作为 pid 号为 1 的第一个用户态进程。

三、进程的创建和退出

在之前实现中，我们对一个任务的创建是放在 `init_pcb` 中，也就是初始化 PCB 里，并没有单独的系统调用去完成一个任务的创建，并且我们的任务一旦运行起来是无法正常退出的。因此，在本次实验我们将完成任务的创建方法（`spawn`）、退出方法（`exit`），以及杀死一个任务的方法（`kill`），等待一个任务的方法（`waitpid`），并为之封装系统调用。

3.1 spawn(task_t)

传入参数: 要启动任务的相关信息（入口地址、任务类型、任务名等）

功能: 为该任务分配 PCB 及运行需要的资源，并加入调度队列。

3.2 kill(pid_t)

传入参数: pid 号

功能: 杀死该 pid 号对应的任务并释放相应的资源。

3.3 exit(void)

传入参数: 无

功能: 使得一个任务正常退出，释放其获取的资源。

3.4 wait(pid_t)

传入参数: pid 号

功能: 调用该函数的任务自身会被阻塞，直到该 pid 对应的任务退出后，才会阻塞继续运行。

3.5 任务 2: spawn、kill、exit、waitpid 方法的实现

3.5.1 实验要求

完成 `spawn`、`kill`、`exit`、`wait` 的系统调用，通过我们给定的测试集。

3.5.2 文件介绍

本次实验请大家根据自己之前任务一代码继续实现。

3.5.3 实验步骤

- (1) 实现上述的四种内核方法：`spawn`、`kill`、`exit`、`wait`，并为其封装系统调用后方可进行测试集的运行。
- (2) 在 `shell` 中实现 `exec [id]` 指令的解析：比如用户输入“`exec 2`”，那么就会启动 `test_tasks` 测试集中的第 2 个任务，`test_tasks` 数组是我们提供给大家的，里面包含了 `project3` 的所有测试，在本次实验的所有任务中，我们使用这种方法启动我们的测试任务，如下：

```
----- COMMAND -----
> root@UCAS_OS: exec 0
exec process[0].
> root@UCAS_OS: exec 1
exec process[1].
> root@UCAS_OS: exec 2
exec process[2].
> root@UCAS_OS:
```

输入“`exec {i}`”启动测试数组中的第 *i* 个任务

下图为 `test_task` 数组的内容，可以看出它包含的我们 `project3` 的所有测试：

```

struct task_info task1 = {"task1", (uint32_t)&ready_to_exit_task, USER_PROCESS};
struct task_info task2 = {"task2", (uint32_t)&wait_lock_task, USER_PROCESS};
struct task_info task3 = {"task3", (uint32_t)&wait_exit_task, USER_PROCESS};

struct task_info task4 = {"task4", (uint32_t)&semaphore_add_task1, USER_PROCESS};
struct task_info task5 = {"task5", (uint32_t)&semaphore_add_task2, USER_PROCESS};
struct task_info task6 = {"task6", (uint32_t)&semaphore_add_task3, USER_PROCESS};

struct task_info task7 = {"task7", (uint32_t)&producer_task, USER_PROCESS};
struct task_info task8 = {"task8", (uint32_t)&consumer_task1, USER_PROCESS};
struct task_info task9 = {"task9", (uint32_t)&consumer_task2, USER_PROCESS};

struct task_info task10 = {"task10", (uint32_t)&barrier_task1, USER_PROCESS};
struct task_info task11 = {"task11", (uint32_t)&barrier_task2, USER_PROCESS};
struct task_info task12 = {"task12", (uint32_t)&barrier_task3, USER_PROCESS};

struct task_info task13 = {"SunQuan", (uint32_t)&SunQuan, USER_PROCESS};
struct task_info task14 = {"LiuBei", (uint32_t)&LiuBei, USER_PROCESS};
struct task_info task15 = {"CaoCao", (uint32_t)&CaoCao, USER_PROCESS};

static struct task_info *test_tasks[16] = {&task1, &task2, &task3,
                                         &task4, &task5, &task6,
                                         &task7, &task8, &task9,
                                         &task10, &task11, &task12,
                                         &task13, &task14, &task15};

static int num_test_tasks = 15;

```

Project3 中需要用到的测试集（位于/test/test_shell.c 中）

(4) 使用 exec 指令启动 task1、task2、task3，关于任务的内容，阐述如下：

task1	申请两把锁，过一段时间后退出。
task2	请求 task1 获得的一把锁，但是由于 task1 占用该锁，task2 会被阻塞并打印出“I want to acquire a mutex lock from task.”。 直到 task1 退出任务后释放该锁，task2 会继续执行，打印出“I have acquire a mutex lock from task”后退出。
task3	调用 waitpid 系统调用函数，等待 task1 退出。 task1 退出后会继续执行，打印出“task has exited”后退出。

(5) 下图为实验成功的打印结果：

```
> [TASK] I am task with pid 2, I have acquired two mutex lock. (144)
> [TASK] I want to acquire a mutex lock from task(pid=2).
> [TASK] I want to wait task (pid=2) to exit.
```

```
----- COMMAND -----
> root@UCAS_OS: exec 0
exec process[0].
> root@UCAS_OS: exec 1
exec process[1].
> root@UCAS_OS: exec 2
exec process[2].
```

启动时候 task2 和 task3 都在等待 task1 的退出

```
> [TASK] I am task with pid 2, I have acquired two mutex lock. (498)
> [TASK] I have acquired a mutex lock from task(pid=2)..
> [TASK] Task (pid=2) has exited.
```

```
----- COMMAND -----
> root@UCAS_OS: exec 0
exec process[0].
> root@UCAS_OS: exec 1
exec process[1].
> root@UCAS_OS: exec 2
exec process[2].
```

task1 退出后，task2、task3 打印出相关信息后退出

(6) 在 shell 中实现 **kill [pid]** 指令的解析: 在上述过程中, task1 是正常通过 `exit` 退出的, 在这里我们需要实现 `kill` 指令的解析, 让用户手动杀死 task1, 如下:

```
----- COMMAND -----
> root@UCAS_OS: exec 0
exec process[0].
> root@UCAS_OS: exec 1
exec process[1].
> root@UCAS_OS: exec 2
exec process[2].
> root@UCAS_OS: kill 2
kill process pid=2
```

杀死一个任务后的结果和 task1 自动退出的结果是一样的，task2 依然可以获得锁，task3 依然可以解除阻塞继续运行。

3.5.4 注意事项

(1) 本次任务，虽然看起来比较直观，但是需要注意的细节很多，特别是在一个任务退出时需要将其占有的资源全部释放，比如：栈空间、锁。此外，还要恢复因为该任务而被挂起的任务。

(2) 在 task2 中 sys_waitpid 的参数传递为 2，这个 pid 对应的是 task1 的 pid，如果你在启动 task1 的时候，它的 pid 不是 2，请自行修改测试用例中的 pid 号，完成实验。

四、同步原语

4.1 信号量 (semaphores)

4.1.1 为什么使用信号量

为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，在任一时刻只能有一个执行线程访问代码的临界区域。我们把对共享内存进行访问的程序片段称为临界区域或临界区，而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用来协调进程对共享资源的访问的。

4.1.2 信号量的定义

信号量的本质是一种数据操作锁，它本身不具有数据交换的功能，而是通过控制其他的通信资源（文件，外部设备）来实现进程间通信，它本身只是一种外部资源的标识。信号量在此过程中负责数据操作的互斥、同步等功能。

当请求一个使用信号量来表示的资源时，进程需要先读取信号量的值来判断资源是否可用。大于 0，资源可以请求，等于 0，无资源可用，进程会进入睡眠状态（进程挂起等待）直至资源可用。当进程不再使用一个信号量控制的共享资源时，信号量的值+1（信号量的值大于 0），对信号量的值进行的增减操作均为原子操作，这是由于信号量主要的作用是维护资源的互斥或多进程的同步访问。而在信号量的创建及初始化上，不能保证操作均为原子性。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即 P(信号量)）和发送（即 V(信号量)）信息操作。最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。

4.1.3 信号量的使用方法

由于信号量只能进行两种操作等待和发送信号，即 P(sv) 和 V(sv)，他们的行为是这样的：

P(sv): 如果 sv 的值大于零，就给它减 1；如果它的值为零，就挂起该进程的执行

V(sv): 如果有其他进程因等待 sv 而被挂起，就让它恢复运行，如果没有进程因等待 sv 而挂起，就给它加 1。

举个例子，就是两个进程共享信号量 sv，一旦其中一个进程执行了 P(sv) 操作，它将得到信号量，并可以进入临界区，使 sv 减 1。而第二个进程将被阻止进入临界区，因为当它试图执行 P(sv) 时，sv 为 0，它会被挂起以等待第一个进程离开临界区域并执行 V(sv) 释放信号量，这时第二个进程就可以恢复执行。

4.2 条件变量 (condition variables)

条件变量是一种同步机制，允许线程挂起，直到共享数据上的某些条件得到满足，用于多线程之间的通信。条件变量上的基本操作有：触发条件（当条件变为 true 时）；等待条件，挂起线程直到其他线程触发条件。

条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

4.3 屏障 (barriers)

Barriers 是一种同步的手段，也是一种线程同步原语，如一组线程/进程的 Barrier 可以用来同步该线程/进程组，只有当该线程/进程组中所有线程到达屏障点（可称之为同步点）时，整个程序才得以继续执行。

屏障可以告诉一组线程在什么时候完成了各自的任务可以接下来进行其他的工作，即一旦所有的线程都到达了屏障点，它们才能够继续执行下去，否则先到达屏障点的线程就会在此处等待其他线程的到来，因此屏障操作也是一个相当重量级的同步操作。

4.4 任务 3：实现同步原语：semaphores、condition variables、barriers

4.4.1 实验要求

了解操作系统内线程、进程间同步的机制，学习和掌握信号量（semaphores）、条件变量（condition variables）、壁垒（barriers）的原理和实现方法。针对这些同步原语，我们要求实现以下函数，并为其封装系统调用：

		函数名称	说明
1	信号量 semaphore	do_semaphore_init(value)	初始化 semaphore 的数据，初始化值为 value。 value 是一个非负整数
		do_semaphore_up(sem)	semaphore 增加 1
		do_semaphore_down(sem)	先减 1，然后如果 semaphore 是负数，则阻塞
2	条件变量 condition variables	do_condition_init(void)	条件变量初始化
		do_condition_wait(lock,cond)	阻塞任务，释放锁，直到被唤醒，获得锁。 调用 task 首先需要获得锁。
		do_condition_signal(cond)	释放 condition_wait 第一个阻塞的 task， 如果没有 task 被阻塞，则什么都不做。
		do_condition_broadcast(cond)	释放 condition_wait 阻塞的所有 tasks。 释放顺序可以不考虑。
3	屏障 barrier	do_barrier_init(bar,n)	初始化 barrier 有 n 个 task
		do_barrier_wait(bar)	一直阻塞到有 n 个 task 调用 barrier_wait(bar)。 task 被阻塞的顺序不需要考虑。在第 n 个 task 调用之后立即返回，然后开始等待新的 n 个 task。

关于这些函数的申明，我们在相应的文件里已经提供给大家了，在 kernel/locking 文件夹下，对于实现和系统调用的封装请同学们自行完成。

4.4.2 文件介绍

请基于任务 2 的项目代码继续进行实现。

4.4.3 实验步骤

- (1) 实现 semaphores、condition variables、barriers 的内核代码并为之封装系统调用。
- (2) **semaphores 测试：** 使用 spawn 启动 task4、task5、task6 任务，关于测试内容的阐述如下：

task4	task4 每次对临界区的值+1，一共进行 10 次
task5	task5 每次对临界区的值+1，一共进行 20 次
task6	task6 每次对临界区的值+1，一共进行 30 次

上述三个任务同时对一个临界值进行加操作，最终，当三个任务运行完成退出后，如果实现正确，这个临界区的值应该为 60，如下图所示：

```
> [TASK] current global value 27. (10)
> [TASK] current global value 49. (20)
> [TASK] current global value 60. (30)
```

```
----- COMMAND -----
> root@UCAS_OS: exec 3
exec process[3].
> root@UCAS_OS: exec 4
exec process[4].
> root@UCAS_OS: exec 5
exec process[5].
```

可以看到，最终所有退出后，global value 的值正好是 $10+20+30=60$ ，由此可以确定，我们由信号量保证的临界区是安全的，没有存在冲突。

(3) condition variables 测试: 使用 spawn 启动 task7、task8、task9 任务，这个测试是一个基于条件变量实现的简单而经典的生产者消费者模型，感兴趣的同学可以看代码了解一下实现。其中 task7 作为生产者负责生产产品，task8、task9 作为消费者负责消费产品。当然，如果你的条件变量实现如果正确，那么生产者产生的商品总数是和消费商品的商品总数是一样的，如下图：

```
> [TASK] Total produced 150 products.
> [TASK] Total consumed 98 products.
> [TASK] Total consumed 52 products.
```

```
----- COMMAND -----
> root@UCAS_OS: exec 6
exec process[6].
> root@UCAS_OS: exec 7
> root@UCAS_OS: exec 7
exec process[7].
> root@UCAS_OS: exec 8
exec process[8].
```

可以看到，task7 一共生产了 150 个产品，task8 和 task9 分别消费了 98、52 个产品，生产和消费的产品数量是一致的，因此本次实验结果正确。

(4) **barriers** 测试：使用 exec 启动 task10、task10、task10 任务，该测试是三个任务不停的进入屏障，只有当 3 个任务都达到屏障后才会解除阻塞，进行下一轮的进入。如果只有两个任务达到屏障，那么它们会被阻塞，直到第三个任务达到屏障。由于屏障的存在，三个任务打印出的循环次数应该是一起增长的。如下图：

```
> [TASK] Exited barrier (8).
> [TASK] Ready to enter the barrier.(9)
> [TASK] Ready to enter the barrier.(9)
```

```
----- COMMAND -----
> root@UCAS_OS: exec 9
exec process[9].
> root@UCAS_OS: exec 10
exec process[10].
> root@UCAS_OS: exec 11
exec process[11].
```

```
> [TASK] Exited barrier (113).
> [TASK] Exited barrier (113).
> [TASK] Exited barrier (113).
```

```
----- COMMAND -----
> root@UCAS_OS: exec 9
exec process[9].
> root@UCAS_OS: exec 10
exec process[10].
> root@UCAS_OS: exec 11
exec process[11].
```

4.4.4 注意事项

同步原语是操作系统中的一个重要部分。针对不同的场景，可以选择合适的同步原语进行使用，它们的本质都是对临界区的保护，实现进程间的同步。

本次任务，我们给出的测试都是在用户态运行的任务，因此需要同学们完成内核里的同步原语实现用后，为其封装系统调用，方可进行测试，否则会出现问题。

五、进程间通信

5.1 生产者——消费者模型

在本节，大家将使用之前实现的同步原语实现进程间通信（IPC），所谓的进程通信通俗的而言就是一个类似于信箱的东西，发送方将信息存到信箱里，接收方从信箱里取出信息，这样就实现了进程间的通信，如下图：



可以看出，所谓的进程通信，就是为两个任务建立一个公共的临界区，任务 1 把想发送给任务 2 的数据放到里面，任务 2 再从里面拿出来，这样就完成了一次进程的通信。虽然听起来似乎很简单，但其实想实现出来也需要考虑很多问题，比如：如果一个任务想往临界区放数据，但是临界区写满了怎么办？如果一个任务想从临界区读数据，但是临界区空了怎么办？临界区肯定存在同时被多个进程访问的情况，如何保证访问的原子性？

其实聪明的同学想必已经有了自己的思路，上述的模型实际上就是一个稍复杂的生产者消费者模型，说它复杂是因为 TASK1 和 TASK2 在通信的过程中既会担任生产者的角色，也会担任消费者的的角色。既然是这种模型，那么我们更实现的同步原语就可以去解决我们上述提到了问题了。

因此在本章节，同学们将使用同步原语实现一个具备进程通信功能的库，去实现进程间的通信，同时保证数据的一致性。

5.2 任务 4：进程间的通信——mailbox 实现

5.5.1 实验要求

实现进程间的通信，使用之前实现的同步原语去保证临界区的正常访问，并通过我们给出的测试集，关于进程通信的接口申明，已经写在了`/libs/mailbox.h` 中，请同学们根据接口进行实现。下表为接口的描述：

	函数名	说明
1	<code>mbox_open(name)</code>	根据名字返回一个对应的 mailbox，如果不存在对应名字的 mailbox，则返回一个新的 mailbox。
2	<code>mbox_close(box)</code>	关闭 mailbox，如果该 mailbox 的引用数为 0，则释放该 mailbox。
3	<code>mbox_send(box, msg, msg_length)</code>	向一个 mailbox 发送数据，如果 mailbox 满了，则产生一个阻塞，直到把数据放进去。
4	<code>mbox_recv(box, msg, msg_length)</code>	从一个 mailbox 接收数据，如果 mailbox 空了，则产生一个阻塞，直到读到数据。

5.5.2 文件介绍

请继续之前的代码继续完成。

5.5.3 实验步骤

- (1) 在进行测试时请确保之前实验的系统调用都已经实现了，包括：`sys_sleep`、`sys_spawn`、`sys_kill`、`sys_waitpid`。
- (2) 完成 IPC 相关接口实现。
- (3) 本次的测试任务为 `task13`, `task14`, `task15`, 和之前的实验一样，我们会通过在终端中使用 `exec` 指令将它们一一启动。
- (4) 对于本次的测试，我们制定了一个小场景方便大家更好的去理解：

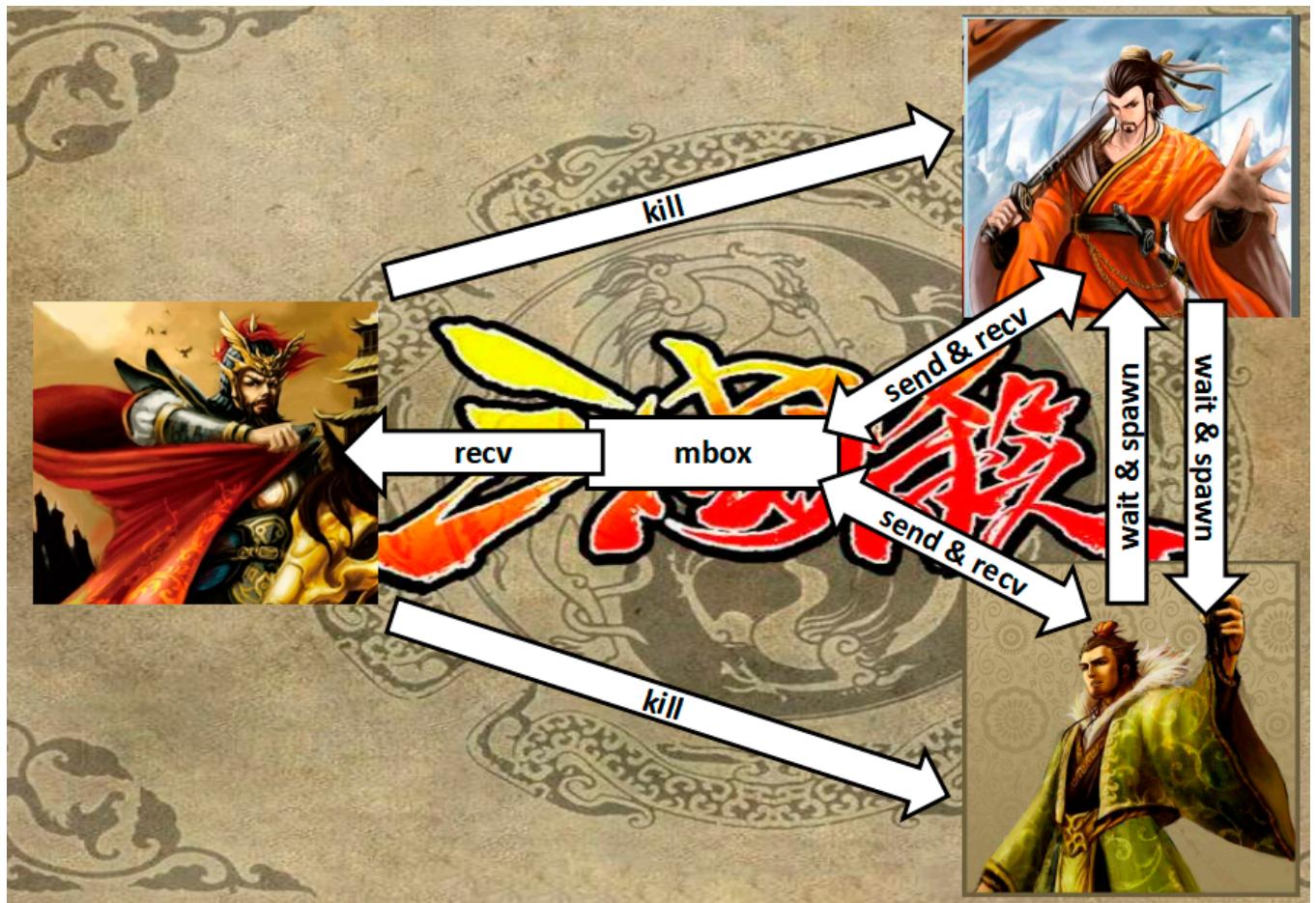
现在有三个人（进程），它们分别是刘备、孙权、曹操。



曹操是个大 BOSS，它会干掉（kill）任何他看到的人（进程），每一次行动，曹操都会选择刘备或者孙权中的一个去下手（对其执行 kill）。而刘备和孙权为了防止被曹操一个一个干掉，达成了一个合作条约：它们相互监（waitpid）听对方的状态，一旦发现对方有被曹操干掉，马上前去救助（spawn）。

如此一来，曹操每次干掉一个人，另外一个人会马上前去救助（spawn），这样场面上一直会是“三足鼎立”的局面。

刘备和孙权每次被干掉后救起，都会重新发送（mbox_send）自己的信息（pid）给对方，对方也会接收（mbox_recv）到对方的新信息，当然，这个信息也会被曹操所窃取，曹操会根据信息执行下一次的计划（kill）。



如果“三足鼎立”局面成立，你会发现无论曹操干掉刘备孙权中的哪一个，另外一个都会大喊着“**XXX I'm coming to save you!**”去把被干掉的人救（spawn）起来，每个人名字的后面会打印出自己的 pid 号，如果运行正确，刘备和孙权的 pid 会不断的上升，如下图所示：

```
[SunQuan](5): I'm waiting for LiuBei (6)
[LiuBei](6): I'm coming to save you, SunQuan!
[CaoCao](4): I am working... muahaha
[CaoCao](4): I have my decision!
[CaoCao](4): I will kill SunQuan (5)!
[CaoCao]biu biu biu ~~~~~~ AAAAAAAA SunQuan is dead QAQ.
[CaoCao](4): Oops! LiuBei(6) is alive again!
```

```
----- COMMAND -----
> root@UCAS_OS: exec 12
exec process[12].
> root@UCAS_OS: exec 13
exec process[13].
> root@UCAS_OS: exec 14
exec process[14].
```

当然，上述的描述只是对这个复杂的测试的一个好懂的解释，上图也仅为示意图，请同学们通过阅读测试代码具体了解测试的内容和逻辑。

5.5.4 注意事项

- (1) 本次实验我们实现的进程通信函数属于用户态的库函数，但是需要系统调用使用同步原语去保证。
- (2) 本次实验涉及到的内容比较多，需要用到之前实现的很多系统调用，如果同学们在测试的时候出现 bug，请认真查漏补缺和助教老师沟通和交流。