

# 国科大操作系统研讨课任务书

RISC-V 版本



版本 1.0

---

# 目录

---

<b>P3 进程管理、通信与多核执行</b>	<b>1</b>
1 实验说明 . . . . .	1
2 本章解读 . . . . .	2
3 简易终端的制作和基础指令的实现 . . . . .	2
3.1 一块屏幕 (screen) . . . . .	2
3.2 一个可以解析指令的进程 (shell) . . . . .	2
3.3 任务 1: 终端和终端命令的实现 . . . . .	3
3.4 进程的创建和退出 . . . . .	4
3.5 任务 1 续: spawn、kill、exit、waitpid 方法的实现 . . . . .	4
4 同步原语 . . . . .	6
4.1 信号量 (semaphores) . . . . .	7
4.2 屏障 (barriers) . . . . .	8
4.3 任务 2: 实现同步原语: semaphores、barriers . . . . .	8
4.4 mailbox 通信 . . . . .	10
4.5 任务 2 续: 进程间的通信——mailbox 实现 . . . . .	10
5 多核 CPU 的支持 . . . . .	11
5.1 多核的启动 . . . . .	11
5.2 从核的例外处理 . . . . .	12
5.3 任务 3: 开启双核并行运行 . . . . .	13
5.4 任务 4: 多核下的 mailbox 处理 . . . . .	13
5.5 任务 5: shell 命令 taskset——将进程绑定在指定的核上 . . . . .	15

---

## Project 3

# 进程管理、通信与多核执行

---

### 1 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已经通过代码调用的一些限制体现了出来，这一界限将在后续的虚存中更好的体现出来）。

我们将继续完善一些任务处理相关的功能，比如：进程的启动、杀死一个正在运行的程序、进程的正常退出，以及进程间的同步、通信等功能。更重要的是，我们将实现一个简单的终端，让我们的操作系统拥有一个简易的 UI，交互性更强。

我们的 start-code 提供了一个大家可以在自己的代码基础上打补丁的程序。大家可以复制自己的 project2 文件夹，起名为 project3，然后运行打补丁的程序。有的文件可能无法自动打补丁，因此会报错，这些文件需要同学们自己手动把 start-code 中的代码合并进来。具体的使用方法参见 p3-autoupdate-patch 文件夹的 README 文件。

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。实现一些终端命令以实现进程在终端上的启动、杀死、状态查询。

**任务二** 了解屏障 (barriers)、信号量 (semaphores)、mailbox 这三种同步方式并实现其功能。

**任务三** 实现多核的启动，并实现简单的程序多核并行执行及调度。

**任务四** 在上面 mailbox 功能的基础上，实现多核情况下的多进程同时收发信息。

**任务五** 实现动态绑核的终端命令 taskset，从而实现动态的将进程绑定在某个核上。

在上面的 5 个任务中，S-core 的同学只需要完成任务一和任务二的屏障功能，A-core 的同学需要完成任务一到任务三，C-core 的同学需要完成全部五个任务。下面将本 Project 的各个 core 需要完成的任务列成表格，请同学们在看完任务书之后再对应这个表格，确认需要完成的任务：

评分等级	需要完成的任务
S-core	shell 终端及 shell 命令, barrier
A-core	全部三种同步原语, 双核并行运行
C-core	双核下防死锁的 mailbox 和 taskset 动态绑核命令

表 P3-1: 各个等级需要完成的任务列表

## 2 本章解读

这一部分的要点是:

1. 理解 shell 的作用及其简化的实现
2. 理解实现进程通信的多种方式
3. 理解多核情况下操作系统的相关处理

这一章其实是在完善我们的小系统。在之前的章节里, 我们已经顺利地建立了进程管理和基本的锁。现在, 我们要进一步扩展之前的功能, 并允许用户进程之间进行交互。

## 3 简易终端的制作和基础指令的实现

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)。

### 3.1 一块屏幕 (screen)

所谓的一块屏幕, 我们在 Project2 的时候就已经用到过了。它实际上就是用我们之前提供给大家的 screen 库函数来进行操作的。在本次实验大家不需要考虑这部分的实现, 使用库 printf 将内容打印在屏幕上即可。当然还是要主要 cursor 的信息, 在 Project2 的时候大家可能已经注意到了, cursor 决定了输出在屏幕上的位置。

### 3.2 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. 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. 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. 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。

串口输入函数由 BBL 提供，函数名为 `sbi_console_getchar`，在 Project1 的时候大家已经用过 [1]。请大家将其封装为系统调用以后在实现 shell 的时候使用。

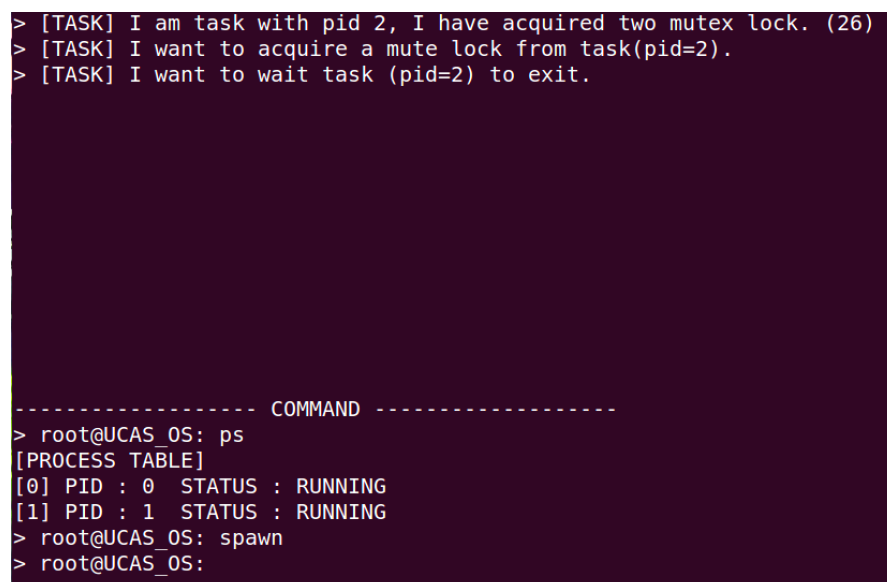
### 3.3 任务 1：终端和终端命令的实现

#### 实验要求

请实现以下功能：

- 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。
- 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。
- 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。
- 实现 `clear`，输入 `clear` 可以实现清屏。
- 实现对无法识别的命令的处理，输出 `Unknown command` 报错并返回到 shell 命令提示符

最终效果的参考图P3-1，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:
```

图 P3-1: 最终效果示例

除了上述的需要要实现的功能，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。

## 注意事项

test/test\_shell.c 文件中要实现 shell 的相关处理操作。shell 在本次实验中担任着启动所有任务的角色，因此我们应该在内核初始化之后，就把 shell 启动起来，将它作为 pid 号为 1 的第一个用户态进程。

此外，shell 输入应支持退格键，在输入错误时用退格键删除之前的输入。

在实现 getchar 功能的时候建议将字符读入到一个小的 buffer 中，等到用户按回车键时，判断 buffer 中的内容，是否是某个命令。对于每个命令分别设置相应的处理函数，实现该命令的功能。使用小 buffer 的原因是用户可能会输入退格键，直接试图根据输入的字符做个状态机解析可能会比较麻烦。等输入完了再解析稍微省事一点。解析不用很复杂，就直接判断 buffer 的开头是不是某个指令就好了。

自己测试的时候，能够看到输入的回显（这个也需要自己写），能够输入并解析些简单命令就可以进入下一节的内容了。

## 3.4 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，接口约定如表P3-2所示。

## 3.5 任务 1 续：spawn、kill、exit、waitpid 方法的实现

### 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。

### 文件介绍

本实验请大家根据自己之前的代码继续实现。

### 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，如图P3-2所示。exec 命令之后的数字可以和图中不一样，按照你实现的数字顺序即可。

test\_shell.c 中的 test\_task 数组的内容，它包含的我们 project3 的所有测试。exec 只需要启动数组下标对应位置的任务就行。

函数签名	<code>pid_t sys_spawn(task_info_t *info, void* arg, spawn_mode_t mode);</code> 返回启动的线程的 pid
参数说明	info 为任务的 task_info 结构体，提供入口地址等信息 arg 为传递给 task 的参数，task 的入口函数可以接收到该参数 mode 为 ENTER_ZOMBIE_ON_EXIT 或 AUTO_CLEANUP_ON_EXIT。前者代表退出后进入 ZOMBIE 状态，后者代表退出时自动释放包括 pcb 在内的所有资源，不进入 ZOMBIE 状态。
功能	启动一个新的线程
函数签名	<code>int sys_kill(pid_t pid)</code>
参数说明	pid 为需要被 kill 的线程的 pid。成功时返回 1，找不到对应的 pcb 返回 0。
功能	杀死制定线程，回收分配给它的内存资源
函数签名	<code>void sys_exit(void)</code>
功能	退出当前任务
函数签名	<code>int sys_waitpid(pid_t pid)</code>
参数说明	pid，被等待的线程
功能	等待 pid 等于我们传入的 pid 的那个线程执行完

表 P3-2: 接口约定

```
----- 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:
```

图 P3-2: 输入 “exec i” 启动测试数组中的第 i 个任务

3. 使用 `exec` 指令启动 `task_test_waitpid`，该任务的描述见表P3-3。
4. 实验成功的结果如图P3-3所示。

```
> [TASK] I am task with pid 3, I have acquired two mutex lock. (498)
> [TASK] I have acquired a mutex lock from task(pid=3)..
> [TASK] Task (pid=3) has exited.

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

图 P3-3: 启动后最后一个任务会等待第一个任务的退出

5. 在 `shell` 中实现 `kill [pid]` 指令的解析：允许用户终止某个线程的运行。由于锁是用户态的，所以这里内核无法替用户线程释放锁。因此，`kill` 以后如果线程持有锁，无需做额外处理。只需要释放内核管理的相关资源。

### 注意事项

- 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、pcb 等。
- `pid` 是由 `sys_spawn` 返回的，进而通过 `sys_spawn` 传递给第二个进程。因此，这里不需要特意让某个进程的 `pid` 是某个预设好的值

## 4 同步原语

在 Project2 的时候，我们实现了进程的运行和调度，但是两个进程之间如果需要进行同步，则需要比较复杂的机制。例如我们在 Project2 里面实现的锁，通过操作系统提供的锁服务，两个进程可以对某个临界区数据进行保护，并分别进行访问和操作。而在



任务	描述
ready_to_exit_task	申请两把锁，过一段时间后退出。
wait_lock_task	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.”。
wait_exit_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。 启动 ready_to_exit_task 和 wait_lock_task 两个任务。并将第一个任务的 pid 传给第二个任务
	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。

表 P3-3: task\_test\_waitpid 任务说明

这一节的内容中，我们将介绍几种常见的同步原语，并由同学们在自己的操作系统中实现它们。

值得一提的是，就做到 Project3 的现实情况而言，我们这里依然很难清楚的在操作系统里面区分进程和线程的概念。这是因为我们现在还没有实现内存空间的隔离和虚拟内存管理，进程之间使用完全隔离的内存空间这一特性很难体现。所以要强调的是，研讨课这里只是为了简化线程和进程的区别，实际上它们两者的区别还是希望大家通过理论课的学习和后续研讨课 Project4 的内容来扎实的理解。到 Project3 为止我们依然希望大家类似 Project2 的情况，通过代码调用来体现进程之间的隔离。

## 4.1 信号量 (semaphores)

### 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(信号变量)）和发送（即 V(信号变量)）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。

### 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 屏障 (barriers)

Barriers 也是同步原语，如一组进程的 Barrier 可以用来同步该程序组，只有当该程序组中所有进程到达屏障点（可称之为同步点）时，所有程序才得以继续执行。

屏障可以告诉一组进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，即一旦所有的进程都到达了屏障点，它们才能够继续执行下去，否则先到达屏障点的进程就会在此处等待其他进程的到来，因此屏障操作也是一个相当重量级的同步操作。

### 4.3 任务 2：实现同步原语：semaphores、barriers

#### 实验要求

了解操作系统内进程间同步的机制，学习和掌握信号量 (semaphores) 和壁垒 (barriers) 的原理和实现方法，做 S-core 的同学只需要完成 barriers。针对这些同步原语，我们要求实现 start-code 中的相关函数。

#### 文件介绍

请基于任务 1 的项目代码继续进行实现。

#### 实验步骤

1. 实现 semaphores、barriers 的内核代码并为之封装系统调用。
2. semaphores 测试：使用 spawn 启动 test\_semaphore 任务。该任务会启动三个三个测试进程，并等待它们退出。测试进程的行为如表P3-4所示。
3. barriers 测试：使用 exec 启动 test\_barriers 任务，该测试是三个进程不停的进入屏障，只有当 3 个进程都达到屏障后才会解除阻塞，进行下一轮的进入。如果只有两个进程达到屏障，那么它们会被阻塞，直到第三个达到屏障。**由于屏障的存在，三个进程打印出的循环次数应该是一起增长的。**如图P3-4所示。

任务入口	功能
<code>semaphore_add_task1</code>	每次对临界区的值 +1，一共进行 10 次
<code>semaphore_add_task2</code>	每次对临界区的值 +1，一共进行 20 次
<code>semaphore_add_task3</code>	每次对临界区的值 +1，一共进行 30 次

表 P3-4: task\_test\_semaphore 任务说明

```
> [TASK] Exited barrier (3).(sleep 2 s)
> [TASK] Exited barrier (3).(sleep 1 s)
> [TASK] Exited barrier (3).(sleep 2 s)

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

图 P3-4: 测试 barrier

## 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。

## 4.4 mailbox 通信

在本节，大家将实现更复杂的一种通信方式 mailbox，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。

具体来说，所谓的 mailbox 通信就是为两个任务建立一个公共的临界区，进程 1 把想发送给进程 2 的数据放到里面，进程 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？

因此在本章节，同学们将实现一个 mailbox 通信功能去实现进程间的通信，同时保证数据的一致性。当然请大家注意，这一节我们强调的是进程间的通信，也就是说我们不允许两个进程之间共享同一个变量或者内存地址。所以向其他进程的信箱发送信息或接收自己信箱的信息应该使用信箱的名字而非信箱的变量。操作系统才是那个根据信箱名字来真正投递信件或返回信件内容的实体。

## 4.5 任务 2 续：进程间的通信——mailbox 实现

### 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。表 P3-5 为接口的描述。

函数名	说明
1 mbox_open(name)	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 mbox_close(box)	关闭 mailbox，如果该 mailbox 的引用数为 0，则释放该 mailbox。
3 mbox_send(box, msg, msg_length)	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 mbox_recv(box, msg, msg_length)	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。

表 P3-5: mailbox 接口约定

## 文件介绍

请继续之前的代码继续完成。

## 实验步骤

1. 完成 IPC 相关接口实现。
2. 本次的测试任务为 `strserver_task` 和 `strgenerator_task`, `strgenerator_task` 为发送进程, 测试时会启动多个, `strserver_task` 为接收进程, 只启动一个。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。

对于本次的测试, 我们准备一个复杂的测试场景。两个或更多的进程为发送 `mail` 的进程, 一个进程为接收 `mail` 的进程。发送 `mail` 的进程不断的发出长度随机的字符串, 而接收 `mail` 的进程要不断的在 `mailbox` 已满的时候将收到的字符串长度打印出来。由于发送进程也会打印自己发送的长度, 因此接收进程收到的字符串长度应等于所有发送 `mail` 进程发出的字符串长度总和。

由于多个进程在同时向同一个进程的 `mailbox` 里面发送字符串, 因此到了一定时间之后接收进程的 `mailbox` 就会满, 此时则发送就会失败。注意发送失败的具体判断条件应为要发送的字符串长度大于 `mailbox` 的剩余空间长度。

## 5 多核 CPU 的支持

在之前的实验中, 我们使用的一直是单核的 `NutShell` 处理器, 其实我们的开发板可以切换成两个核心的 `Rocket` 处理器。现在我们需要切换到双核处理器, 把两个核的时钟中断都打开, 让两个核能同时处理不同的进程, 让进程的运行速度得到大大地提高, 让这样充分利用多核处理器的优势。显著的优势往往伴随着巨大的挑战, 在实现双核的过程中, 大家需要思考许多问题: 如何启动双核? 开启从核的时钟中断之后需要怎样对就绪队列进行管理? 是设计成双核共用一个就绪队列还是采用每个核各自管理各自的队列? 此外还需要思考双核的例外处理是否共用一套代码, 如果共用一套例外处理的代码, 那么当两个处理器核同时陷入内核时就需要保证只能有一个核能对内核进行访问, 此时需要思考应该采用什么机制来对内核进行保护。除此之外, 双核操作系统的设计还需要考虑进程间通信的问题。因此, 想要实现双核的操作系统, 有很多设计都需要重新考虑。

### 5.1 多核的启动

首先让我们切换到双核的 `Rocket` 处理器。切换的方法是把开发板上的 `sw1` 开关拨到里侧, 即开发板下侧并排的两个拨动开关里面靠近电源线的那一个。拨到里侧之后把开发板重新上电, 再启动就可以看到类似 `NutShell` 的大写 `ROCKET` 标志。这时就加载了双核的 `Rocket` 处理器。注意这是要使用 `loadbootm` 命令代替 `loadboot` 命令加载内核, 切换到了双核处理器之后不允许使用 `loadboot` 命令了。

另外，在 QEMU 上我们一直加载的都是双核处理器，只是没有启动从核。只要在加载的时候使用 `loadbootm` 命令，就可以把从核也启动起来了，后面的现象就和开发板一样了。

两个处理器核心都包含完整的一套寄存器（通用寄存器及特权寄存器）以及一级 cache。对于双核实验来说，最需要弄清楚的事情是我们如何从主核控制从核执行代码。不同的体系结构略有差别，但最核心的一点是相通的，就是利用中断机制。在前面的实验中我们直观地理解了，中断可以使处理器核跳转到某个指定的地址。因此，让从核执行代码的方式也是利用中断的这个特性。我们只需要让从核触发中断，自然可以让从核执行代码（中断处理函数就是我们的代码，只要能执行起来，后面都由我们控制）。那么怎么产生中断呢？对于多核系统，有一种特殊的中断叫做核间中断 (IPI)。核间中断就是从从一个核心发往另一个核心的中断。

对于 RISC-V 处理器来说，有一个专门的 `sbi_send_ipi(const unsigned long *hart_mask)` 可以用于触发核间中断。如果不指定 MASK 的话，这个函数会给所有的核心发一个 software interrupt 中断。

另外，RISC-V 处理器核一启动其实两个核心就都自动起来了，会自动都跳到 boot-loader 开始执行（先后顺序不一定，甚至可能是从核先执行）。在跳入到 boot-loader 时，`a0` 寄存器会存放处理器的 id 号（该 id 号可以从 csr 的 `mhartid` 寄存器中读出）。各个核心的 `mhartid` 号不同，但至少会有一个核心的 id 号为 0。你可以将这个 id 号为 0 的处理器作为主核。

唤醒从核需要注意的是，主从核都需要栈空间来运行，为了避免主从核相互影响，需要为每个核设置其独立的栈空间用于启动。

本次任务需要大家实现双核启动，需要看到两个核心都进入到了内核的 `main` 函数中。可以在 `main` 函数中用 `printk` 打印核心 id，从而判断是否两个核心都工作起来了。双核启动的相关代码从 `bootblock.S` 中就需要添加，大家可以参考 `start-code` 把需要实现的代码补完。

## 5.2 从核的例外处理

从核的例外处理流程和主核相同，因此，可以让主核和从核共用一套例外处理的代码。对于主核来说，主核负责完成操作系统的所有必要的初始化工作，之后，从核只需要做必要的设置（如中断入口、时钟中断等）即可正常工作。

由于双核是同时执行的，如果两个核心同时执行内核代码，访问共享的内核变量，就可能造成同步相关的错误。因此，需要将共享的变量或一些不能打断的过程设置为临界区，用锁保护起来。为了便于大家快速实现多核，这里建议大家使用 Linux 早期的策略：实现一个大内核锁。也就是将内核整个作为一个临界区，用一把大锁锁住。每次进入内核时立马上锁，退出内核时解锁。

大内核锁的实现依赖于前面的实验提到过的原子指令。因为双核是同时执行的，所以为了能够实现双核间的同步，只能利用原子指令提供的原子性。因此，需要利用前面提到的原子指令实现一个自旋锁，再将这个自旋锁作为大内核锁使用。

当然，除了大内核锁以外，还有多种方案可以实现多核支持。比如现在的多数操作系

统都会细粒度地对共享的数据结构加锁。我们内核中共享的会更改的数据结构并不是非常多, 如果有兴趣也可以实现成对共享数据结构细粒度加锁的模式。当然, 并不是加了锁保护了就可以直接跑起来, 因为有一些变量只有一个是不行的。比如 `current_running`, 多核情况下, 其实可以有多个正在运行的进程, 只用一个变量显然是管理不了的。此外, 调度队列要不要每个核心各有一个? 多核对于 `screen` 的输出会有何影响? 这些问题都需要考虑。有些在大内核锁的保护下, 只有一份就可以了, 有些也许需要多份。也有些用一份或者多份都可以。例如, 调度队列其实全局就一个是可以的, 每个核放一个也是可以的, 全凭自己根据自己代码的具体情况思考设计。

### 5.3 任务 3: 开启双核并行运行

此前我们已经将从核唤醒, 但此时从核只能运行一个进程, 还不能处理时钟中断, 不能进行进程调度, 所以我们需要把从核的例外处理打开, 让从核也能触发时钟中断, 进行多进程调度。

#### 实验步骤

1. 打开从核的时钟中断, 让两个核都能进行时钟中断处理和进程调度。
2. 运行 `test_multicore.c` 中的测试用例。测试效果如图 P3-5所示, 可以看到单核运行的耗时是双核耗时的 1.2 倍左右, 在对这个实验检查时, 我们希望看到单核耗时/双核耗时的比值越大越好。

#### 注意事项

1. 在开启从核的例外处理之后, 就可以去运行 `test_multicore.c` 中的测试用例, 在这个过程中, 可能会遇到单核时没有出现过的 bug, 这就说明之前的系统的实现有问题, 在双核运行时暴露出来了, 这也是实现双核的一大挑战!
2. 在用 `qemu` 进行双核调试时在可以使用 `gdb` 命令 `info threads` 来查看两个核当前运行到什么位置。
3. 由于开发板的主频比较低, 在做该实验时, 建议将时钟中断设置为每秒 100 次这个量级。

### 5.4 任务 4: 多核下的 mailbox 处理

此时我们的系统以及能正常跑在双核的板子上了, 我们可以实验一个双核版的进程间通信, 以此来检验双核下我们的同步操作是否能正常工作。另外, 我们在任务 2 中实现的 mailbox 场景还比较简单, 进程要么只发送 mail 或者只接收 mail。这里我们需要实现一个三个进程互相收发场景。从这个任务开始, 只有 C-core 的同学必须完成。

```
startlcompute,pfrom = 0, to = 2500000 Done
start compute, from = 2500000, to = 5000000 Done

single core: 13245998 ticks, result = 630
multi core: 10047581 ticks, result = 630

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

图 P3-5: 双核 vs 单核速率对比图



## 实验步骤

1. 同学自行实现三个进程互相收发 mail 的功能，发送的内容依然是长度随机的字符串，且发送目标也随机。
2. 要求三个进程行为相同，且在发送失败或接收失败的时候进入阻塞状态。但三个进程不能死锁。

## 注意事项

在双核系统中运行测试用例时，因为两个核都在向屏幕打印，所以屏幕输出可能会比较乱。这是因为 start code 给的 screen 是为单核设计的，有共享的变量记录打印位置。但实际上我们之前为了给每个任务展现一个其独享 screen 的错觉，已经在 pcb 中放置了记录光标位置的域。所以，其实完全有办法让每个核上面的进程都认为自己独占了屏幕。毕竟真正的输出是在时钟中断中完成的，两个核上的进程只是在向缓冲区里面输出东西而已。

RISC-V 上为了让 printk 可以遵循 printf 的坐标，做了部分同步。但到这个实验以后，printk 和 printf 混合输出的概率不高。用户任务全是 printf 了。所以如果 printk 中对于 current\_running 中 cursor 的一些操作影响了你，你可以考虑删除掉。

对于实验功能要求的一点提示：由于三个进程的行为相同，且发送或接收失败时均会进入阻塞，那么如果三个进程开头都在接收，或者三个 mailbox 都满了但三个进程又同时在发送 mail，就有可能导致三个进程同时进入阻塞状态，从而形成死锁。推荐的设计方案是，将发送和接收两个功能做成同一个系统调用，发送或接收任意一个可以进行就不会阻塞，用户程序通过返回值来判断发送和接收哪个完成了，从而重试另一个操作。这样的设计可以避免死锁的发生，当然我们也鼓励同学们使用其他的合理的设计来达到任务要求。

## 5.5 任务 5: shell 命令 taskset——将进程绑定在指定的核上

在实际的系统中，有时我们需要让某个进程在特定的处理器核上执行。例如一些情况下为了提升性能，会让线程固定在某个核心上，这样就不会由于在核间反复迁移导致无法充分利用处理器的 Cache 中的数据。在本任务中，我们来实现这一功能。同时，为了能够更直观地展示该功能，需要扩展 ps 命令，让它能够显示每个进程允许在哪个核上执行。对于正在运行的线程（不是 ready 的，而是正在 running 的），输出它在哪个核上。测试用例在 test\_affinity.c 中，该测试会自动启动 5 个子任务。子任务循环完成计算，每计算一部分会输出提示信息。

绑定处理器核通过 taskset 命令完成。该命令有两种形式：

**taskset mask 任务号** 启动任务并设置其允许运行的核为 mask

**taskset -p mask pid** 设置进程 pid 的允许运行的核为 mask

mask 是一个 16 进制数，每一位代表是否运行进程运行在该核心上。比如 0x1 代表只运行在核 0 上面。0x2 代表只允许运行在核 1 上面。0x3 代表允许运行在核 0 和核 1 上

面。第一种形式类似于之前实现的 `exec` 的功能，只不过可以为被启动的任务指定一个 `mask`，来说明它被允许调度到哪个核心上。第二种形式是为已经运行的进程设置 `mask` 的。

这里需要大家额外支持一个功能，就是在一个任务通过 `sys_spawn` 启动子任务时，如果没有通过显式的指定 `mask`，则自动继承父任务的 `mask`。

### 实验步骤

1. 实现 `taskset` 命令。
2. 运行给定的测试用例，效果如图P3-6所示。测试方式是先用 `taskset` 让 `test_affinity` 任务启动，`mask` 设置为 `0x1`，也就是只允许在第一个核心上面运行。之后 `ps` 看到它以及它开启的子任务确实是都在第一个核心上运行的。之后选择一个 `test_affinity` 开启的子任务，用 `taskset` 将 `mask` 设置为 `0x2`，之后再 `ps`，看到该任务在第二个核心上运行，同时该任务运行的速度比其他的任务都快（过几轮循环后应该会明显快于其他任务）。

```
start test cpu affinity, pids = {3, 4, 5, 6, 7, }
[3] integer test (24/100)
[4] integer test (15/100)
[5] integer test (18/100)
[6] integer test (15/100)
[7] integer test (18/100)

taskset 0x1 8.
exec process[8].
> root@UCAS_OS: ps
[PROCESS TABLE]
[0] PID : 1  STATUS : RUNNING MASK: 0xffffffffffffffff on Core 1
[1] PID : 2  STATUS : BLOCKED MASK: 0x1
[2] PID : 3  STATUS : READY MASK: 0x1
[3] PID : 4  STATUS : READY MASK: 0x1
[4] PID : 5  STATUS : RUNNING MASK: 0x1 on Core 0
[5] PID : 6  STATUS : READY MASK: 0x1
[6] PID : 7  STATUS : READY MASK: 0x1
> root@UCAS_OS: taskset -p 0x2 3
taskset -p 0x2 3.
> root@UCAS_OS:
```

图 P3-6: 绑核效果图

---

## 参考文献

---

- [1] UG585, “Zynq-7000 soc.” [https://www.xilinx.com/support/documentation/user\\_guides/ug585-Zynq-7000-TRM.pdf](https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf), 2018. [Online; accessed 20-September-2021].