第二十二章　实验 4: 多核处理

22.1　简介

在本实验中，ChCore 将支持在多核处理器上启动（第一部分），在多个线程上并行地执行程序（第二部分），创建新进程执行给定二进制文件（第三部分），以及进程间的通信（第四部分）。

22.1.1　评分

本实验中，代码部分的总成绩为 100 分。可使用如下命令检查当前得分：

```
oslab$ make grade
…
Score: 100/100
```

本实验运行或调试某一用户程序X的方式与实验 3 相同，可以使用make run-X和make run-X-gdb等命令。

22.2　第一部分：多核支持

在本实验中，由 ChCore 运行的 CPU 核心数量将从 1 扩展为PLAT_CPU_NUM（定义在kernel/common/machine.h中，代表可用 CPU 核心的数量）。要使用这些 CPU 核心，需要添加对多核的支持。此外，随着 CPU 核心的增多，可能会遇到并发问题。因此，需要拿锁以防止出现多个 CPU 核心中运行的代码同时修改某一内核关键数据，造成数据竞争。

所有 CPU 核心在开机时会被同时启动，在引导时则会被分为两种类型。一个指定的 CPU 核心会引导整个操作系统和初始化自身，被称为**主 CPU**

(primary CPU)。其他的 CPU 核心只初始化自身即可，被称为**副 CPU** (backup CPU)。CPU 核心仅在系统引导时有所区分，在其他阶段，每个 CPU 核心都是被相同对待的

与之前的实验一样，CPU 核心执行位于`boot/start.S`中的`_start`。此时，只有主 CPU 能够开始执行`_start`中全部的代码，副 CPU 则被阻塞直到主 CPU 显式地将激活它们为止。

练习 1

阅读`boot/start.S`中的汇编代码`_start`。比较实验三和本实验之间`_start`部分代码的差异（可使用`git diff lab3`命令）。说明`_start`如何选定主 CPU，并阻塞其他副 CPU 的执行。

22.2.1 启动多核

与之前的实验一样，主 CPU 在第一次返回用户态之前会在`kernel/main.c`中执行`main()`函数，进行操作系统的初始化任务。之后，主 CPU 会执行一系列逻辑以激活各个副 CPU。

需要注意的是，编写多核操作系统时，区分每个 CPU 核心专用的本地状态和整个系统共享的全局状态非常重要。在本实验中，包含有`PLAT_CPU_NUM`个元素的数组通常用于存储每个 CPU 核心的本地状态。为了方便确定当前执行该代码的 CPU 核心 ID，我们在`kernel/common/smp.c`中提供了`smp_get_cpu_id()`函数。该函数总是返回调用它的 CPU 核心的 ID，该 ID 可用作访问上述数组的索引。

例如，`kernel/commen/smp.c`中的`cpu_status`包含用于管理副 CPU 引导的标志，将在下一部分中使用这些标志。

本实验在`main`函数中新增了激活各个副 CPU 的过程(`kernel/common/smp.c`中的`enable_smp_cores`函数)。请阅读`enable_smp_cores`和`_start`，以了解多核的启动过程。

练习 2

完善主 CPU 激活各个副 CPU 的函数：`enable_smp_cores()`和`kernel/main.c`中的`secondary_start()`。同时，请注意测试代码会要求各个副 CPU 按序被依次激活。完成该练习后应能够通过`smp`测试，并获得测试的前 5 分。

练习 3

熟悉启动副 CPU 的控制流程（同主 CPU 类似），并回答以下问题：初始化时，主 CPU 同时激活所有副 CPU 而不是依次激活每个副 CPU 的设计是否正确？换言之，并行启动副 CPU 是否会导致并发问题？

提示：检查每个 CPU 核心是否共享相同的内核堆栈以及控制流中的每个函数调用是否会导致数据竞争。

22.2.2 大内核锁

从现在开始，内核代码已经可以运行在多核上了。然而为了确保代码不会由于并发执行而引起错误，应首先解决并发问题。在本实验中，我们使用最简单的并发控制方法，即内核中的全局共享锁（大内核锁）即可达成这一目的。具体而言，在 CPU 核心以内核态访问任何数据之前，它应该首先获得大内核锁。同理，CPU 核心应当在退出内核态之前释放大内核锁。大内核锁的获取与释放，保证了同时只存在一个 CPU 核心执行内核代码、访问内核数据，因此不会产生竞争。

在文件`kernel/common/lock.c`中，提供了一个简单的排号锁，用以充当大内核锁。

练习 4

请熟悉排号锁的基本算法，并在`kernel/common/lock.c`中完成`unlock()`和`is_locked()`的代码。至此，实验代码应通过`mutex`测试，并获得对应的 5 分。

注意：本练习无需使用任何汇编代码（例如内存屏障等），且需要编写的代码少于五行。

ChCore 使用`lock_kernel()`和`unlock_kernel()`两个接口对大内核

锁进行封装。在本实验中，为了在 CPU 核心进入内核态时拿锁、离开内核态时放锁，应该在以下 6 个位置调用上述接口：

1. kernel/main.c中的main()：主 CPU 在激活副 CPU 之前需要首先获得了大内核锁。
2. kernel/main.c中的secondary_start()：在初始化完成之后且副 CPU 返回用户态之前获取大内核锁。
3. kernel/exception/exception_table.S中 的el0_syscall：在跳转到syscall_table中相应的syscall条目之前获取大内核锁（该部分汇编代码已实现完成）。
4. kernel/exception/exception.c中的handle_entry_c：在该异常处理函数的第一行获取大内核锁。因为在内核态下也可能会发生异常，所以如果异常是在内核中捕获的，则不应获取大内核锁。
5. kernel/exception/irq.c中的handle_irq：在中断处理函数的第一行获取大内核锁。与handle_entry_c类似，如果是内核异常，则不应获取该锁。
6. kernel/exception/exception_table.S中 的exception_return：在第一行中释放大内核锁。由于所有情况下，内核态返回用户态都使用exception_return，因此这是唯一需要调用unlock_kernel()的位置。

   注意：此处是本实验中唯一需要编写一行汇编代码的部分。

为简单起见，除了实验后续部分涉及的内核态空闲线程外，本实验在内核态下将禁用中断。这一功能是在硬件的帮助下实现的。当异常触发时，中断即被禁用。当汇编代码eret被调用时，中断则会被重新启用。因此，在整个实验的实现过程中，可以无需考虑时钟中断打断内核代码执行的情况，而仅需要其对用户态进程的影响。

练习 5

在kernel/common/lock.c中 实 现kernel_lock_init()、lock_kernel()和unlock_kernel()。如上所述，通过在适当的位置调用lock_kernel()和unlock_kernel()，使用大内核锁来处理可能的并发问题。至此，实验代码应通过big lock测试，并获得对应的 5 分。

练习 6

为了保护寄存器上下文，在el0_syscall调用lock_kernel()时，在栈上保存了寄存器的值。然而，在exception_return中调用unlock_kernel()时，却不需要将寄存器的值保存到栈中，试分析其原因。

22.3 第二部分：调度

截止目前，本实验的多核操作系统已经可以运行，但仍然无法对多个线程进行调度。本部分将首先实现协作式调度，从而允许当前在 CPU 核心上运行的线程退出时，CPU 核心能够切换到另一个线程继续执行。然后，将实现抢占式调度，使得内核可以在一定时间片后重新获得对 CPU 核心的控制，而无需当前运行线程的配合。最后，将扩展调度器，允许线程被调度到所有 CPU 核心上。

22.3.1 协作式调度

在本部分中，将实现一个基本的调度器，该程序调度在同一 CPU 核心上运行的线程。所有相关的数据结构都可以在kernel/sched/sched.h中找到。

current_threads是一个数组，分别指向每个 CPU 核心上运行的线程。与实验 3 不同，本实验中current_thread现在是一个表示current_threads[smp_get_cpu_id()]的宏，尽管其语义不会改变。

ready_thread_queue包含调度器的就绪线程（等待被调度的线程）列表。

sched_ops表示 ChCore 中的调度器。它存储指向不同调度操作的函数指针，以支持不同的调度策略。

sche_init()初始化调度器。sched_enqueue()将新线程添加到调度器的就绪队列中。sched_dequeue()从调度器的就绪队列中取出一个线程。当 ChCore 调度要执行的线程时，将调用sched()，它将当前线程放入调度器的就绪队列，然后选择调度器的就绪队首线程出队并进行调度。sched()使用sched_choose_thread()是sched()的辅助函数，用于选择下一个进行

调度的线程。sched_handle_timer_irq()是时间中断处理函数，会在后续部分介绍。

cur_sched_ops指向 ChCore 使用的实际sched_ops （ChCore 用在kernel/sched/sched.h中定义的静态函数封装对cur_sched_ops的调用。）。

Round Robin（时间片轮转）调度策略

我们将在kernel/sched/policy_rr.c中实现名为rr的sched_ops，以实现 Round Robin 方式调度线程。ChCore 中的 Round Robin 调度工作方式如下：

每个 CPU 核心都有自己rr_ready_queue的列表，该列表存储 CPU 核心的就绪线程。一个线程只能出现在一个 CPU 核心的rr_ready_queue中。当 CPU 核心调用rr_sched_enqueue()时，应将给定线程放入 CPU 核心自己的rr_ready_queue中，并将线程的状态设置为TS_READY。同样，当 CPU 核心调用rr_sched_dequeue()时，应该从 CPU 核心自身的rr_ready_queue中取出给定线程，并将线程状态设置为TS_INTER，代表了线程的中间状态。

一旦 CPU 核心要发起调度，它只需要调用rr_sched()，首先检查当前是否正在运行某个线程。如果是，它将调用rr_sched_enqueue()将线程添加回rr_ready_queue。然后，它调用rr_choose_thread()来选择要调度的线程。

当 CPU 核心没有要调度的线程时，它不应在内核态忙等，因为此时它持有的大内核锁这锁住整个内核。所以，我们的 Round Robin 策略为每个 CPU 核心创建一个空闲线程（kernel/sched/policy_rr.c的idle_threads）。rr_choose_thread首先检查 CPU 核心的rr_ready_queue是否为空。如果是，rr_choose_thread返回 CPU 核心自己的空闲线程。如果没有，它将选择rr_ready_queue的队首并调用rr_sched_dequeue()使该队首出队，然后返回该队首。idle_thread不应出现在rr_ready_queue中。因此，rr_sched_enqueue()和rr_sched_dequeue()都应空闲线程进行特殊处理。

rr_sched_init()用于初始化调度器。它只能在kernel/main.c的main()中被调用一次，并且主 CPU 负责初始化rr_ready_queue和idle_threads中的所有条目。

练习 7

完善kernel/sched/policy_rr.c中的调度功能。完成本练习后应能够运行cooperative测试并获得 10 分。现在，调度器应可以在一个 CPU 核心上工作。

练习 8

如果异常是从内核态捕获的，CPU 核心不会在kernel/exception/irq.c的handle_irq中获得了大内核锁。但是，有一种特殊情况，即如果空闲线程（以内核态运行）中捕获了错误，则 CPU 核心还应该获取大内核锁。否则，内核可能会被永远阻塞。请思考一下原因。

练习 9

现在，尽管调度器尚未完成，但已经可以运行一些简单的用户态程序。在syscall.c中实现系统调用sys_get_cpu_id()，它告诉用户态程序正在运行的 CPU 核心的 ID。在sched.c中实现系统调用sys_yield()，使用户态程序可以启动线程调度。完成本练习后应能够运行/yield_single.bin并获得以下输出：

```
...
Hello, I am thread 0
Hello, I am thread 1
Iteration 0, thread 0, cpu 0
Iteration 0, thread 1, cpu 0
Iteration 1, thread 0, cpu 0
Iteration 1, thread 1, cpu 0
...
```

由于测试脚本会首先运行内核测试，然后再运行用户程序测试。虽然目前可以正确地运行用户态程序，但是还无法运行make grade获得yield single测试的 5 分。

22.3.2 抢占式调度

使用刚刚实现的协作式调度器，ChCore 能够在单个 CPU 核心内部调度线程。然而，若用户线程不想放弃对 CPU 核心的占据，内核便只能让用户线

程继续执行，而无法强制用户线程中止。因此，在这一部分中，本实验将实现抢先式调度，以帮助内核定期重新获得对 CPU 核心的控制权。

时钟中断与抢占

请尝试运行`yield_spin.bin`程序。该用户程序的主线程将创建一个“自旋线程”，该线程在获得 CPU 核心的控制权后便会执行无限循环，进而导致无论是该程序的主线程还是 ChCore 内核都无法重新获得 CPU 核心的控制权。就保护系统免受用户程序中的错误或恶意代码影响而言，这一情况显然并不理想，任何用户应用线程均可以如该“自旋线程”一样，通过进入无限循环来永久“霸占”整个 CPU 核心。

处理时钟中断

为了处理“自旋线程”的问题，允许 ChCore 内核强行中断一个正在运行的线程并夺回对 CPU 核心的控制权，我们必须扩展 ChCore 以支持来自时钟硬件的外部硬件中断。

定时器中断初始化的相关代码已包含在本实验的初始代码中，并在`kernel/exception/exception.c`中的`exception_init_per_cpu()`函数内被调用。

练习 10

在`exception_init_per_cpu()`中恢复被注释代码`timer_init()`，这将在用户态下启用硬件定时器中断。内核的终端处理逻辑结束后会调度线程。此时，`yield_spin.bin`应可以正常工作：主线程应能在一定时间后重新获得对 CPU 核心的控制并正常终止。由于测试脚本的原因，在完成该练习时还无法通过`make grade`来获得`yield spin`测试的 5 分。

调度预算（Budget）

在实际的操作系统中，如果每次时钟中断都会触发调度，会导致调度时间间隔过短、增加调度开销。

对于每个线程，我们在`kernel/sched/sched.h`中维护一个`sched_cont`。实际上，这对应了每个线程的、**预算（budget）**。当处理

时钟中断时，将当前线程的预算减少一。在之前的调度策略实现sched()中，调度器应只能在某个线程预算等于零时才能调度该线程。

kernel/sched/policy_rr.c中的rr_sched_handle_timer_irq()是内核在处理时钟中断时调用的处理程序，可以在其中修改当前线程的预算。

练习 11

在kernel/sched/policy_rr.c中修改调度器逻辑，以便它可以支持预算机制。按照上述说明，实现rr_sched_handle_timer_irq()。不要忘记在kernel/sched/sched.c的sys_yield()中重置“预算”，确保sys_yield()在被调用后可以立即调度当前线程。完成本练习后应能够preemptive测试并获得 5 分。

处理器亲和性

到目前为止，已经实现了一个基本完整的调度器。但是，ChCore 中的 Round Robin 策略为每个 CPU 核心维护一个rr_ready_queue，并且无法在 CPU 核心之间调度线程。

为了解决此问题，**亲和性（Affinity）**的概念被引入，亲和性使线程可以绑定到特定的 CPU 核心。创建线程时，还应指定线程的亲和性。如果亲和性的值设置为某个 CPU 核心的 ID，则该线程将添加到该 CPU 核心的rr_ready_queue中。我们之前的 Round Robin 策略将线程绑定到创建该线程 CPU 核心的rr_ready_queue。在创建将亲和性设置为NO_AFF的线程时，这也是默认行为。

练习 12

为kernel/sched/policy_rr.c中 的rr_sched_enqueue()增 加线程的亲和性支持。如果亲和性设置为NO_AFF，则应将线程加入当前 CPU 核心的rr_ready_queue。否则，将线程排队到亲和性指定的rr_ready_queue上。完成本练习后应能够通过affinity测试并获得 5 分。

对调度器进行综合sched测试，应能够通过sched测试并获得 5 分。到目前为止，已经通过了所有本实验的内核测试。

运 行make grade， 应 能 够 通 过yield single，yield spin和yield multi测试，并获得 15 分。

为了让用户态程序可以在运行时设置线程的亲和性，系统提供了sys_set_affinity和sys_get_affinity系统调用，用于设置或获取线程的亲和性。

练习 13

在thread.c中实现sys_set_affinity和sys_get_affinity。完成本练习后应能够通过yield aff和yield mutli aff测试，并获得 10 分。

至此，实验的第二部分已全部完成，请确认通过了所有第二部分的测试。

22.4 第三部分：spawn()

ChCore 现在可以运行在kernel/main.c的main()中创建的进程。但是，更全面的操作系统应允许用户态程序执行特定二进制文件。

当操作系统执行给定的可执行二进制文件时，它应创建一个负责执行文件的新进程。为了执行一个特定的文件，Linux 首先调用fork()，然后调用exec()。其实，还有另一个叫做spawn()的接口，它可以提供与fork()和exec()的组合类似的功能。在第三部分中，我们将实现用户态的spawn()。

22.4.1 spawn() 接口

阅读user/lib/spawn.c中的spawn()的代码。它首先获取特定文件的 ELF（可执行和可链接格式），并将 ELF 传递给spawn()的核心函数。在本部分中，将实现名为launch_process_with_pmos_caps()的函数，然后使spawn()可运行。首先，我们将介绍核心功能的接口。

```
1 int launch_process_with_pmos_caps(struct user_elf
  ↪ *user_elf,
2       int *child_process_cap,
3       int *child_main_thread_cap,
4       struct pmo_map_request *pmo_map_reqs,
5       int nr_pmo_map_reqs,
6       int caps[],
7       int nr_caps,
8       s32 aff)
```

user_elf指定要执行的特定文件的 ELF 结构。
child_process_cap用于输出所创建子进程 capability。
child_main_thread_cap用于输出子进程主线程的 capability。
pmo_map_reqs用于指定父进程与子进程共享内存的映射。
nr_pmo_map_reqs指定pmo_map_reqs的数量。
caps用于指定父进程需要转移给子进程的 capability。
nr_caps指定caps的数量。
aff是子进程的主线程的亲和性。

22.4.2 基本工作流

本节将实现一个基本的spawn()。父进程可以创建执行特定文件的子进程，但是父进程不能将任何信息传递给子进程，也不能从spawn()获取任何输出。我们已提供了launch_process_with_pmos_caps()的代码框架，该部分仅需要遵循基本的工作流程并填写代码中的空白即可。

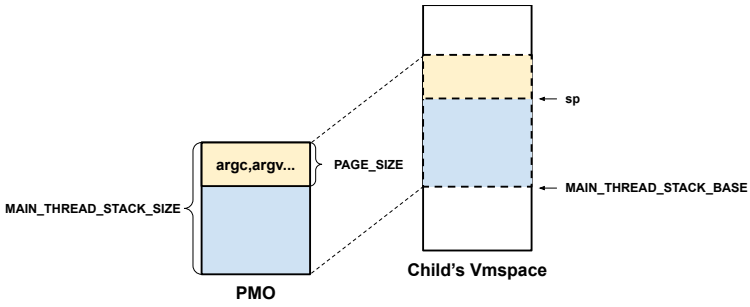

图 22.1: spawn() 中 PMO 与子进程虚拟地址空间的对应关系

基本工作流程如下：

1. 使用sys_create_process()创建一个子进程。
2. 将二进制 ELF 中的每个段映射到子进程。
3. 使用sys_create_pmo()创建一个新的物理内存对象（PMO）main_stack_pmo，用于子进程的堆栈。PMO 的大小为MAIN_THREAD_STACK_SIZE，应被立即分配。
4. 在父进程的本地内存中构造（准备）一个初始页面，该页面存储诸如argc，argv的参数。

5. 使用sys_write_pmos()将初始页面写到main_stack_pmo的顶端页。

6. 使用sys_write_pmo()将main_stack_pmo映射到子进程。

   应该将其映射到具有VM_READ和VM_WRITE权限的子进程的地址MAIN_THREAD_STACK_BASE。

7. 创建子进程的主线程。当执行主线程时，它从sp开始其线程的栈。

练习 14

在user/lib/spawn.c中实现spawn()。完成本练习后应能够通过spawn basic测试，并获得 10 分。
注意：本练习只需用正确的表达式替换所有LAB4_SPAWN_BLANK即可。

22.4.3 为 spawn() 支持传递 pmo 和 cap

在本节中，将实现完整的spawn()。父进程能够将共享内存传递给子进程，并将给定的 capability 转移给子进程。实验中需要处理的参数是pmo_map_reqs，nr_pmo_map_reqs，caps和nr_caps。

在基本工作流程的第 4 步之前，应该做以下几步：

A: 检查nr_pmo_map_reqs > 0，然后用pmo_map_reqs调用sys_map_pmos()。

B: 检查nr_caps> 0，然后用caps调用usys_transfer_caps()。

完成基本工作流程的步骤 7 之后，应该：

C: 如果不为空,则设置child_process_cap和child_main_thread_cap。

练习 15

在user/lib/spawn.c中实现spawn()。完成本练习后应能够通过spawn info测试，并获得 5 分。
注意：本练习无需大量代码，代码总数不应超过 **20 行**。

22.5 第四部分：进程间通信

尽管spawn()使父进程和子进程能够共享内存并相互通信。但是，如果两个进程没有父子关系，它们无法通过spawn()共享内存，进而相互通信。在最后一部分，我们将修改 ChCore 以支持一种更通用的方法，使进程之间可以相互传递消息。

22.5.1 ChCore 的进程间通信

ChCore 的 IPC 接口不是传统的send()/recv()接口。其更像客户端-服务器模型，其中 IPC 请求接收者是服务器，而 IPC 请求发送者是客户端。

为了处理 IPC，服务器需要先注册回调函数ipc_dispatcher()，并且为了处理 IPC，其主线程不能够退出。ChCore 使用ipc/ipc.h中的ipc_connection将客户端与服务器一一绑定。对于每个ipc_connection，内核都会创建一个用户态服务器线程，负责执行服务器的ipc_dispatcher()。线程属于服务器的进程中，因此线程与服务器共享相同的vm_sapce。

为了发起 IPC 请求，客户端应首先创建到服务器的ipc_connection，然后才能将 IPC 请求发送到服务器。当内核处理sys_ipc_call()时，它将客户端线程的执行迁移到服务器线程（我们将此过程称为线程迁移）。然后内核返回到服务器线程以执行ipc_dispatcher()回调函数。服务器线程获取结果后，调用sys_ipc_return()。在处理sys_ipc_return()时，内核将服务器线程的执行再次迁移回客户端线程，进而完成整个 IPC 的流程。

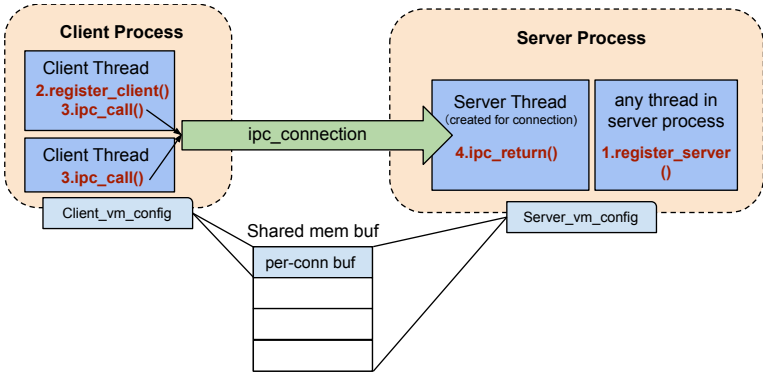

图 22.2: ChCore 进程间通信示意图

22.5.2 进程间通信的系统调用

- `sys_ipc_register_server()`用于给服务器注册 IPC 回调函数。

  它的参数是`ipc_dispatcher()`的地址，最多可服务的客户端数和`server_vm_config`。`server_vm_config`指定了服务器使用的虚拟地址配置，用于 1）`ipc_connection`线程使用的栈，以及 2）客户端和服务器之间使用的共享内存缓冲区。

- 客户端使用`sys_register_client()`创建从调用者线程到目标服务器的`ipc_connection`。

  它的参数是服务器和`client_vm_config`的 capability。`client_vm_config`指定虚拟地址（在客户端中）和客户端与服务器之间使用的共享内存缓冲区的大小。它的返回值是`ipc_connection`的 capability。

- 客户端使用`sys_ipc_call()`发出 IPC 请求。

  它的参数是`ipc_connection`和`ipc_msg`的 capability。`ipc_msg`实际存储于客户端和服务器之间的每个`ipc_connection`独有的共享缓冲区起始地址上。对于小消息，客户端可以直接使用缓冲区传输数据；对于大消息，客户端需要创建 PMO，并将 PMO 的 capability 存储在缓冲区中以传输共享内存。`kernel/ipc/ipc.h`提供了`ipc_msg`的详细信息。

- `sys_ipc_return()`由`ipc_connection`线程使用。

  它的参数是`ipc_dispatcher()`的返回 64 位值。

练习 16

在`kernel/ipc/`中实现了大多数 IPC 相关的代码，请根据注释完成其余代码。完成本练习后应能够通过`ipc data`和`ipc mem`测试，并获得 15 分。

注意：除使用正确代码逻辑替换所有`LAB4_IPC_BLANK`外，本练习有还有少量额外代码需要编写。

练习 17

熟悉 IPC 的工作流程，并实现一个简化的系统调用`sys_ipc_reg_call()`。它与`sys_ipc_call()`相似，唯一的区别是`sys_ipc_reg_call()`的第二个参数是 64 位值，而不是共享内存的地址。该参数应作为`ipc_dispatcher()`的唯一参数直接传递到服务器线程。完成本练习后应能够通过`ipc reg`测试，并获得 5 分。

22.5.3 Send/Recv IPC

完成本节的内容可以获得本实验的额外得分。

与 ChCore 不同，某些系统使用`send()`和`recv()`作为它们的 IPC 接口。请为 ChCore 设计和实现 Send/Recv IPC。接收者线程调用`recv()`，然后等待，直到任何发送者线程调用`send()`并将 IPC 消息发送到接收者线程。

额外练习

在完成所有先前的练习和问题后，创建一个名为 lab4-bonus 的新分支以完成此部分练习。然后，为 ChCore 实现`ipc_send()`和`ipc_recv()`。在新的分支中，可以根据需要修改任何代码。

以下是一些要求：

- 需要提供一个用户态程序（一个即可）来测试 Send/Recv IPC。可以现有测试（例如 `ipc reg`）实现。
- 只需要与`ipc_reg_call()`一样发送的 64-bit 的值作为消息即可，无需支持共享内存消息传递。
- 不能仅仅对`ipc_call()`和`ipc_return()`进行简单地封装。
- 如果需要，可以重用`ipc_register_server()`和`ipc_register_client()`。
- `ipc_send()`和`ipc_recv()`的返回值或参数没有限制。
- 修改内核代码的方式不受限制。

至此，**本实验的内容全部结束**。请确保通过所有测试并获得 100 分的总分。

本实验：扫码反馈