实验 4：多核、多进程、调度与IPC

在本实验中，ChCore将支持在多核处理器上启动（第一部分）；实现多核调度器以调度执行多个线程（第二部分）；实现第一个用户态系统服务：进程管理器并支持 `spawn` 系统调用以启动新的进程（第三部分）；最后实现内核信号量（第四部分）。

第一部分：多核支持

在本实验第一部分中，ChCore将进入多核的世界。为了让ChCore支持多核，我们需要考虑如下问题：

- 如何启动多核，让每个核心执行初始化代码并开始执行用户代码？
- 如何区分不同核心在内核中保存的数据结构（比如状态，配置，内核对象等）？
- 如何保证内核中对象并发正确性，确保不会由于多个核心同时访问内核对象导致竞争条件？

在启动多核之前，我们先介绍ChCore如何解决第二个问题。ChCore对于内核中需要每个CPU核心单独存一份的内核对象，都根据核心数量创建了多份（即利用一个数组来保存）。ChCore支持的核心数量为 `PLAT_CPU_NUM` （该宏定义在 `kernel/common/machine.h` 中，其代表可用CPU核心的数量，根据具体平台而异）。 比如，实验使用的树莓派3平台拥有4个核心，因此该宏定义的值为4。ChCore会CPU核心的核心ID作为数组的索引，在数组中取出对应的CPU核心本地的数据。为了方便确定当前执行该代码的CPU核心ID，我们在 `kernel/arch/aarch64/machine/smp.c` 中提供了 `smp_get_cpu_id` 函数。该函数通过访问系统寄存器 `tpidr_el1` 来获取调用它的CPU核心的ID，该ID可用作访问上述数组的索引。

启动多核

在实验1中我们已经介绍，在QEMU模拟的树莓派中，所有CPU核心在开机时会被同时启动。在引导时这些核心会被分为两种类型。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**其他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。

> **思考题 1：** 阅读汇编代码 `kernel/arch/aarch64/boot/raspi3/init/start.S` 。说明ChCore是如何选定主CPU，并阻塞其他其他CPU的执行的。

在树莓派真机中，还需要主CPU手动指定每一个CPU核心的的启动地址。这些CPU核心会读取固定地址的上填写的启动地址，并跳转到该地址启动。在 `kernel/arch/aarch64/boot/raspi3/init/init_c.c` 中，我们提供了 `wakeup_other_cores` 函数用于实现该功能，并让所有的CPU核心同在QEMU一样开始执行 `_start` 函数。

与之前的实验一样，主CPU在第一次返回用户态之前会在 `kernel/arch/aarch64/main.c` 中执行 `main` 函数，进行操作系统的初始化任务。在本小节中，ChCore将执行 `enable_smp_cores` 函数激活各个其他CPU。

> **思考题2：** 阅读汇编代码 `kernel/arch/aarch64/boot/raspi3/init/start.S`, `init_c.c` 以及 `kernel/arch/aarch64/main.c` ，解释用于阻塞其他CPU核心的 `secondary_boot_flag` 是物理地址还是虚拟地址？是如何传入函数 `enable_smp_cores` 中，又该如何赋值的（考虑虚拟地址/物理地址）？

> **练习3：** 完善主CPU激活各个其他CPU的函数： `enable_smp_cores` 和 `kernel/arch/aarch64/main.c` 中的 `secondary_start` 。请注意测试代码会要求各个其他CPU按序被依次激活。在完成该部分之后，应看到如下输出：

```
[INFO] CPU 0 is active
[INFO] CPU 1 is active
[INFO] CPU 2 is active
[INFO] CPU 3 is active
[INFO] All 4 CPUs are active
```

完成该练习后应能够通过 `Boot multicore` 测试，并获得对应的5分。注意，这里由于实验后续步骤还未完成，其他核心会有对应的报错输出，导致评分脚本*可能暂时无法正确给分*。请先自行判断输出中是否包含有以上内容。随着后续实验进行，评分脚本将能够正确判断。

大内核锁

从现在开始，内核代码已经可以运行在多核上了。为了确保代码不会由于并发执行而引起错误，ChCore应首先解决并发问题。在本小节中，ChCore将使用最简单的并发控制方法，即通过一把**大内核锁**保证不会有多个核心同时访问内核的数据（即互斥访问）。具体而言，CPU核心在进入内核态执行内核代码之前，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/arch/aarch64/sync/ticket.c` 中，我们提供了一个简单的排号锁，用以充当大内核锁。

ChCore中大内核锁共有三个接口进行封装：初始化大内核锁的 `kernel_lock_init` ，上锁与放锁操作 `lock_kernel` 与 `unlock_kernel` 。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5个位置调用上述接口进行**加锁操作**：

1. `kernel/arch/aarch64/main.c` 中的 `main` ： 主CPU在初始化完成之后且其他CPU返回用户态之前获取大内核锁。
2. `kernel/arch/aarch64/main.c` 中的 `secondary_start` ： 其他CPU在初始化完成之后且返回用户态之前获取大内核锁。
3. `kernel/arch/aarch64/irq/irq_entry.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/arch/aarch64/irq/irq_entry.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/arch/aarch64/irq/irq_entry.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。

练习4： 本练习分为以下几个步骤：

1. 请熟悉排号锁的基本算法，并在 `kernel/arch/aarch64/sync/ticket.c` 中完成 `unlock` 和 `is_locked` 的代码。
2. 在 `kernel/arch/aarch64/sync/ticket.c` 中实现 `kernel_lock_init` 、 `lock_kernel` 和 `unlock_kernel` 。
3. 在适当的位置调用 `lock_kernel` 。
4. 判断什么时候需要放锁，添加 `unlock_kernel` 。（注意：由于这里需要自行判断，没有在需要添加的代码周围插入TODO注释）

至此，ChCore已经可以通过使用大内核锁来处理可能的并发问题。本练习的前2项可以通过提供的内核锁测试检查是否正确，后2项内容则需后续完成调度后再核对（ChCore应能正常运行用户态程序）。

完成本练习后应能够通过 `Mutex test` 测试，并获得对应的5分。

注意：本部分测试需要打开 `CHCORE_KERNEL_TEST` ，即在 `.config` 文件中如下行选择 `ON` 。

```
CHCORE_KERNEL_TEST:BOOL=ON
```

> **思考题5：** 在 `el0_syscall` 调用 `lock_kernel` 时，在栈上保存了寄存器的值。这是为了避免调用 `lock_kernel` 时修改这些寄存器。在 `unlock_kernel` 时，是否需要将寄存器的值保存到栈中，试分析其原因。

在课本中，我们还介绍了由于嵌套中断可能会导致死锁，其必须使用可重入锁来解决。为简单起见，ChCore实验不考虑可重入锁。在内核中，我们关闭了中断。
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。

第二部分：调度

截止目前，ChCore已经可以启动多核，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**主动退出或主动放弃CPU**时，CPU核心能够切换到另一个线程继续执行。其后，我们将支持抢占式调度，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，我们将扩展调度器，允许线程被调度到所有CPU核心上。

协作式调度

在本部分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/include/sched/sched.h` 中找到，这里简要介绍其涉及的数据结构：

- `current_threads` 是一个数组，分别指向每个CPU核心上运行的线程。而 `current_thread` 则利用 `smp_get_cpu_id` 获取当前运行核心的id，从而找到当前核心上运行的线程。
- `sched_ops` 则是用于抽象ChCore中调度器的一系列操作。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`： 进行一次调度。即将正在运行的线程放回就绪队列，然后在就绪队列中选择下一个需要执行的线程返回。
  - `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。
  - `sched_top`： 用于debug打印当前所有核心上的运行线程以及等待线程的函数。
- `cur_sched_ops` 则是一个 `sched_ops` 的实例，其在 `sched_init` 的时候初始化。ChCore用在 `kernel/include/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` 的调度策略，以实现时间片轮转方式调度线程。回顾一下时间片轮转调度：其将维护一个FIFO（先入先出）的线程就绪队列，所有可以执行的线程均放在该就绪队列中。调度器将每次将当前正在执行的线程放回到就绪队列中，并从该就绪队列中按照先入先出的顺序取一个就绪的线程执行。

具体而言，在ChCore的实现中，每个CPU核心都有自己线程就绪队列（ `rr_ready_queue_meta` ），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核心的就绪队列中。当调用 `sched_enqueue` 时，rr策略会指向 `rr_sched_enqueue` 。该函数会将传入的线程放入合适的核心的线程就绪队列中，并将线程的状态设置为 `TS_READY` 。

同样，当CPU核心调用 sched_dequeue 时，rr策略会指向 rr_sched_dequeue 。该操作将从核心自己的就绪队列中取出给定线程，并将线程状态设置为 TS_INTER ，代表了线程的中间状态。在本部分还未支持设置CPU的亲和度，因此只要求将当前的线程加入到当前的核心的等待队列中。此外，还应该更新该线程的 thread_ctx->cpuid 代表其当前所属的CPU核心。

一旦CPU核心要发起调度，它只需要调用 rr_sched 。该函数将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 将线程添加回就绪队列。然后，它调用 rr_sched_choose_thread 按照上述的时间片轮转的策略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，否则它持有的大内核锁将锁住整个内核。所以，ChCore为每个CPU核心创建一个空闲线程（即 idle_threads ）。空闲线程将不断的执行一个空循环，其是现在 kernel/arch/aarch64/sched/idle.S 。 rr_sched_choose_thread 首先检查CPU核心的就绪队列是否为空。如果是， rr_sched_choose_thread 将返回CPU核心自己的空闲线程。如果就绪队列中有其他的线程，它将选择就绪队列的队首并调用 rr_sched_dequeue 使该队首出队，然后返回该队首线程。在ChCore中， idle_threads 不应出现在就绪队列中。因此， rr_sched_enqueue 和 rr_sched_dequeue 都应空闲线程进行特殊处理。

当线程退出时（即 thread_exit_state 被设置为 TE_EXITING 时），其也会调用 rr_sched 来调度到其他线程执行。因此，当判断到当前的线程正在退出时， rr_sched 需要更新线程的状态 state 为 TS_EXIT 以及其退出状态 thread_exit_state 为 TE_EXITED 。此外，该线程也不应加入到等待队列中。

> **思考题6：** 为何 idle_threads 不会加入到等待队列中？请分析其原因？

rr_sched_init 用于初始化调度器。它只能在内核初始化流程中被调用一次。由主CPU负责初始化 rr_ready_queue_meta 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，包括 rr_sched_enqueue ， rr_sched_dequeue ， rr_sched_choose_thread 与 rr_sched ，需要填写的代码使用 LAB 4 TODO BEGIN 标出。在完成该部分后应能看到如下输出，并通过 cooperative 测试获得5分。
>
> ```
> [INFO] Pass tst_sched_cooperative!
> ```
>
> **注意：本部分测试需要打开 CHCORE_KERNEL_TEST ，即在 .config 文件中如下行选择 ON 。**
>
> ```
> CHCORE_KERNEL_TEST:BOOL=ON
> ```

> **思考题8：** 如果异常是从内核态捕获的，CPU核心不会在 kernel/arch/aarch64/irq/irq_entry.c 的 handle_irq 中获得大内核锁。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。

在本小节中，ChCore还处于协作式调度。顾名思义，协作式调度需要线程主动放弃CPU。为了实现该功能，我们提供了 sys_yield 这一个新的syscall。该syscall可以主动放弃当前CPU核心，并调用上述的 sched 接口完成调度器的调度工作。

> **练习9：** 在 kernel/sched/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。此外，ChCore还添加了一个新的系统调用 sys_get_cpu_id ，其将返回当前线程运行的CPU的核心id。请在 kernel/syscall/syscall.c 文件中实现该函数。
>
> 运行用户态程序 yield_single.bin 应能看到以下输出：

```
Hello from ChCore userland!
Hello, I am thread 0
Hello, I am thread 1
Iteration 1, thread 0, cpu 0
Iteration 1, thread 1, cpu 0
Iteration 2, thread 0, cpu 0
Iteration 2, thread 1, cpu 0
Iteration 3, thread 0, cpu 0
Iteration 3, thread 1, cpu 0
Iteration 4, thread 0, cpu 0
...
```

提示：成功运行该用户态程序同时需要上一小节在正确的地方添加加锁与放锁条件。如果无法正确运行，可以进一步检查大内核锁相关代码是否实现正确。

注意：本部分测试需要关闭 `CHCORE_KERNEL_TEST`，即在 `.config` 文件中如下行选择 `OFF`。并将对应的启动程序设置为 `yield_single.bin`

```
CHCORE_ROOT_PROGRAM:STRING=yield_single.bin
CHCORE_KERNEL_TEST:BOOL=OFF
```

评分脚本将在后续练习检查该部分正确性。

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。

练习10： 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（ `timer_init` ）。请在主CPU以及其他CPU的初始化流程中加入对该函数的调用。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。

在完成功能后，运行用户态程序 `yield_spin.bin` 应能看到以下输出：

```
Hello, I am thread 1
Successfully regain the control!
```

注意：本部分测试需要关闭 `CHCORE_KERNEL_TEST`，即在 `.config` 文件中如下行选择 `OFF`。并将对应的启动程序设置为 `yield_spin.bin`

```
CHCORE_ROOT_PROGRAM:STRING=yield_spin.bin
CHCORE_KERNEL_TEST:BOOL=OFF
```

评分脚本将在后续练习检查该部分正确性。

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/include/sched/sched.h` 中维护了一个**调度上下文** `sched_cont` 。 `sched_cont` 中存在一个成员 `budget` ，其表示该线程被调度时的“预算”。当每一次处理时钟中断时，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。通过给每个线程合适的“预算”，就可以避免过于频繁的调度。此外，我们已经提供了一个函数原型 `rr_sched_refill_budget` ，其应在当前线程预算用完之后被使用，并且给当前线程重新填满默认的预算 `DEFAULT_BUDGET` 。

练习11： 在 `kernel/sched/sched.c` 处理时钟中断的函数 `sched_handle_timer_irq` 中添加相应的代码，以便它可以支持预算机制。更新其他调度函数支持预算机制，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 `tst_sched_preemptive` 测试并获得5分。

```
[INFO] Pass tst_sched_preemptive!
```

注意：本部分测试需要打开 `CHCORE_KERNEL_TEST` ，即在 `.config` 文件中如下行选择 `ON` 。

```
CHCORE_KERNEL_TEST:BOOL=ON
```

处理器亲和性（Affinity）

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核心维护一个等待队列，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。在创建线程时，线程的亲和性（ `thread_ctx->affinity` ）被设置为 `NO_AFF` 。其代表当前核心可以被调度器放在任意核心运行。将没有亲和度设置的线程加入等待队列时，ChCore将直接加入到当前核心的等待队列，否则其应该加入到相应核心的队列。为了设置与获取线程的的亲和性，ChCore提供了两个系统调用： `sys_set_affinity` 与 `sys_get_affinity` ，其将分别用于设置与获取线程的 `thread_ctx->affinity` 。除了设置亲和性，调度器也需要支持亲和性，从而保证拥有指定亲和性的线程只能在指定核心上运行。

练习12： 在 `kernel/object/thread.c` 中实现 `sys_set_affinity` 和 `sys_get_affinity` 。完善 `kernel/sched/policy_rr.c` 中的调度功能，增加线程的亲和性支持（如入队时检查亲和度等，请自行考虑）。完成后应能看到内核输出如下内容：

```
[INFO] Pass tst_sched_affinity!
[INFO] Pass tst_sched!
```

注意：本部分测试需要打开 `CHCORE_KERNEL_TEST` ，即在 `.config` 文件中如下行选择 `ON` 。

```
CHCORE_KERNEL_TEST:BOOL=ON
```

> 完成本练习后应能够通过内核 `Affinity` 与 `Sched` 测试，获得10分。并能够通过用户态所有的调度测试，包括之前的 `Yield single` ， `Yield spin` ，以及新怎的 `Yield multi` ， `Yield aff` ，与 `Yield multi aff` 测试。总共获得25分。
> 提示：这里如果确认调度部分都完成正确，但是无法正确运行，可以查看之前是否在正确的地方释放了大内核锁。

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。

第三部分：进程管理器（Process Manager）

到目前为止，ChCore执行的进程都是由ChCore内核创建的Root thread。但是，更通用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。 为此，ChCore提供了 `spawn` 的接口实现这一功能。

在这一部分中，我们将首先实现ChCore上的第一个用户态系统服务进程：进程管理器，并且实现他的第一个功能：给定可执行二进制并创建一个新的进程执行该二进制，也即 `spawn` 。

进程管理器

在Linux中，创建新的进程并执行指定的二进制均由内核实现。ChCore作为微内核，尽可能将系统服务都移动到用户态。这里我们将实现第一个ChCore的系统服务：进程管理器。ChCore进程管理器负责创建用户态进程，管理他们之间的关系，以及在进程退出后回收进程。

在实验中，我们的进程管理器只负责其中一个功能，即创建新的线程。为了能够让用户态的进程能够使用该服务，我们首先需要支持**进程间通讯**（IPC）。本实验的第四部分就是构建ChCore的进程间通讯。我们将在后续实验再展示如何通过进程间通讯处理用户程序的系统服务请求。

Spawn功能

Spawn最终要实现功能是用户给定一个二进制的路径，其可以创建一个新的进程，执行该二进制，并返回新创建的进程的主线程cap以及进程的pid。由于到目前为止，实验中还未实现文件系统服务，进程管理器还不能直接从文件系统中读取二进制。这里采取与内核创建第一个 `Root thread` 相似的方法，将本实验需要执行的二进制一同放到 `procm.srv` 的二进制中，并使用 `extern` 符号的方法找到该二进制读出来。因此本实验的 `spawn` 暂时只支持特定二进制的启动。具体代码请阅读 `userland/servers/procm/spawn.c` 。

为了让ChCore运行的用户态程序能够连接到各系统服务器（例如本节实现的 `procm` ，以及后面实验将实现的文件系统/网络栈等），在 `spawn` 创建新的进程时，还需要将所有的系统服务器的 `cap` 均传给该进程。在配置好该进程的所有参数之后， `spawn` 将调用 `launch_process` 来创建进程。其基本工作流如下：

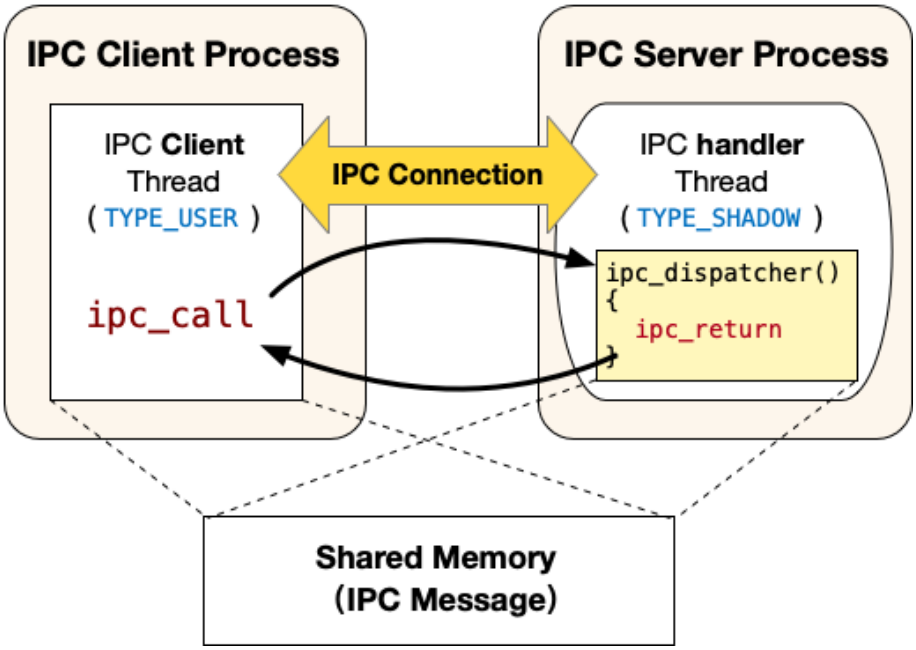
1. 使用 `__chcore_sys_create_cap_group` 创建一个子进程。
2. 如果有需要传输的初始cap（比如系统服务的cap，用于后续ipc调用系统服务），则传输这些cap。
3. 使用 `__chcore_sys_create_pmo` 来创建一个新的内存对象，用作主线程的栈，大小为 `MAIN_THREAD_STACK_SIZE` 。
4. 构建初始执行环境并写入栈顶。
5. 将二进制elf中每个段以及栈映射到相应位置。这里可以使用 `chcore_pmo_map_multi` ，其一次性把需要映射的pmo传入内核映射。除了最基本的 `cap` ， `addr` 和 `perm` ，这里还有一个 `free_cap` 。其代表是否顺带将当前进程的 `pmo` 的 `cap` 释放掉。由于是帮助新进程创建的 `cap` ，在映射后当前进程的 `cap` 无需保留，因此应该都将 `free_cap` 设置为 `1` 。
6. 创建新进程的主线程。

> **练习13**：在 `userland/servers/procm/launch.c` 中填写 `launch_process` 函数中缺损的代码，完成本练习之后应运行 `lab4.bin` 应能看到有如下输出：
>
> ```
> Hello from user.bin!
> ```
>
> **注意，本部分实验将在 `lab4.bin` 的应用程序中进行。请自行修改 `.config` 文件，将启动程序修改为 `lab4.bin` 。**
>
> ```
> CHCORE_ROOT_PROGRAM:STRING=lab4.bin
> ```
>
> 完成本练习后应能通过 `spawn` 测试，并获得相应的5分。

第四部分：进程间通讯

在本部分，我们将实现ChCore的进程间通讯，从而允许跨地址空间的两个进程可以使用IPC进行信息交换。

ChCore进程间通讯概览



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。如图所示，为了处理IPC，服务器（接收者）需要先注册回调函数 `ipc_dispatcher` 。当客户端（发送者）需要发送IPC请求的时候，其将使用 `sys_ipc_call` 切换到一个服务器专属的线程并执行该回调函数处理该请求。而在该请求处理结束之后，将会调用 `sys_ipc_return` 回到调用IPC的客户端（发送者）线程。

该服务器专属的IPC处理线程为每一个IPC连接专属，且属于一种特殊的线程类型 `TYPE_SHADOW` 。不同于正常的用户态线程 `TYPE_USER` ，这种类型的线程不再拥有**调度上下文**（Scheduling Context），也即不会主动被调度器调度到。其在客户端注册IPC连接的时候被创建，且只有在执行IPC的时候方能被调度到。为了实现该功能，该处理线程将在调用IPC的时候**继承**IPC客户端线程的调度上下文（即budget），从而能被调度器正确地调度。

ChCore IPC具体流程

为了实现ChCore IPC的功能，首先需要在Client与Server端创建起一个一对一的IPC Connection。该Connection保存了IPC Server中处理IPC请求的线程（即下图中IPC handler Thread）、Client与Server的共享内存（用于存放IPC通讯的内容）。同一时刻，一个Connection只能有一个Client接入，并使用该Connection切换到Server的处理流程。

ChCore提供了一系列机制，用于创建Connection以及创建每个Connection对应的Server处理线程。下面将以具体的IPC注册到调用的流程，详细介绍ChCore的IPC机制：

1. IPC服务器调用 `ipc_register_server` （ `libchcore/src/ipc/ipc.c` 中）来注册自己为IPC的服务器端。其主要包含以下内容：
   1. 构造IPC服务所需要的参数 `ipc_vm_config` ，其包括：
      - `stack_base_addr` ： IPC服务线程的栈基地址
      - `stack_size` ： 服务线程栈的大小
      - `buf_base_addr` ： IPC用于传输数据的共享内存基地址
      - `buf_size` ： IPC共享内存大小

      ChCore的IPC支持同时多个不同的IPC客户端连接服务器。处理这些Client的服务器线程明显不能够使用同一个栈以及共享内存。因此ChCore会赋予每一个不同的连接不同的 `conn_idx` ，并以此为索引来寻找对应的栈与共享内存。
   2. 调用ChCore提供的的系统调用： `sys_register_server` 。该系统调用需要传入三个参数，分别是用于处理IPC的例程 `server_handler` （图中对应 `ipc_dispatcher` ），支持连接的客户端最大数量以及刚才构造的IPC服务器参数。服务例程 `server_handler` 接受两个参数 `ipc_msg` 以及 `pid` 。其中 `pid` 为发起IPC的客户端的PID，其用于区分不同的进程。

      ChCore内核中将使用 `copy_from_user` 从用户态拷入的IPC服务器参数，并且构造一个bitmap用于记录空闲的 `conn_idx` （即 `conn_bmp` ）。
2. IPC客户端调用 `ipc_register_client` （ `libchcore/src/ipc/ipc.c` 中）来与给定服务器建立连接（connection）。其主要包含以下内容：
   1. 构建IPC客户端的参数 `ipc_vm_config` ，其只需要以下两个值：
      - `buf_base_addr` ： 当前客户端的连接的IPC共享内存的基地址
      - `buf_size` ： IPC共享内存的大小。

      同样，这里需要注意给不同的客户端不同的 `client_id` ，确保不会用到同一块的共享内存。
   2. 调用 `sys_register_client` 系统调用注册连接。其会创建专属于该连接的IPC处理线程，并且配置好IPC的共享内存。由于调用该接口时，服务器可能还未准备好（即执行 `register_server` ），因此可能会注册失败，此时返回 `-EIPCRETRY` 。这里使用一个重试循环将其包裹。具体而言，该系统调用将执行以下步骤：
      1. 创建一个新的连接 `connection` 。
      2. 查询bitmap分配一个新的 `conn_idx` 。
      3. 以此为index寻找合适的服务器配置（即栈与共享内存）。
      4. 创建对应的pmo，然后映射到对应的地址空间中。
      5. 将 `connection` 的 `cap` 赋予客户端与服务器。
3. IPC客户端调用 `ipc_call` 完成一次IPC通讯。

   在使用 `ipc_call` 之前，IPC客户端需要使用 `ipc_create_msg` 来创建IPC信息（填metadata），然后使用 `ipc_set_msg_data` 以及 `ipc_set_msg_cap` 来设置需要传输的数据或cap。其本质就是在之前的共享内存中找到相应位置，然后把需要传输的数据拷贝过去。 `ipc_set_msg_data` 还会传入一个 `offset` 参数，用于表示需要设置的数据在 `ipc` 数据中的偏移量。

   而 `ipc_call` 则直接调用系统调用 `sys_ipc_call` 。该系统调用将迁移到之前注册的handler线程。具体而言， `sys_ipc_call` 需要完成以下步骤。
   1. 判断是否要传递cap。如果要传递，则需要使用 `cap_copy` 传递给server并设置好对应的 `cap` 。
   2. 调用 `thread_migrate_to_server` 配置好handler thread的上下文并切换到该shadow thread执行。此外，由于当前的客户端线程需要等待IPC服务器线程处理完毕，因此需要更新其状态为 `TS_WAITING` ，且不要加入等待队列。

   `ipc_call` 默认传输两个参数给IPC服务器的处理线程，包括调用 `ipc_call` 时传入的参数以及该线程所属进程的 `pid` 。

4. IPC服务器的处理线程在处理完IPC请求之后使用 `sys_ipc_return` 系统调用返回。该系统调用也包含两个参数，一个是返回值，另一个是server传回的cap数量。与发送类似，如果有cap需要传递，则需要使用cap_copy传回client并更新对应的cap。最后调用 `thread_migrate_to_client` 回到客户端。

> **练习14：** 在 `libchcore/src/ipc/ipc.c` 与 `kernel/ipc/connection.c` 中实现了大多数IPC相关的代码，请根据注释完成其余代码。运行 `lab4.bin` 之后应能看到如下输出：
>
> ```
> Hello from ChCore Process Manager!
> Hello from user.bin!
> Hello from ipc_client.bin!
> IPC test: connect to server 2
> IPC no data test .... Passed!
> IPC transfer data test .... Passed!
> IPC transfer cap test .... Passed!
> IPC transfer large data test .... Passed!
> 20 threads concurrent IPC test .... Passed!
> ```
>
> **注意，本部分实验将在 `lab4.bin` 的应用程序中进行。请自行修改 `.config` 文件，将启动程序修改为 `lab4.bin` 。**
>
> ```
> CHCORE_ROOT_PROGRAM:STRING=lab4.bin
> ```
>
> 完成本练习后应能够通过所有ipc测试，包括 `ipc no data` 、 `ipc data` 、 `ipc cap` 、 `ipc large data` 、 `ipc multiple` ，并获得25分。

第五部分：内核信号量

本实验的最后一部分将实现一套线程等待（挂起）/唤醒机制。线程挂起需要调度器合作，因此操作系统内核需要提供新的原语辅助线程完成挂起，并在合适的时机唤醒该线程。因此，我们在ChCore内核中提供了一套信号量的机制，用于为上层应用提供挂起/唤醒的能力。

信号量回顾

信号量的本质是一个资源计数器，其提供两个原语： `wait_sem` 与 `signal_sem` 。 `wait_sem` 将消耗资源并将资源计数器减少；而 `signal_sem` 则代表生产资源，其将使资源计数器增加。如果当前计数器为0则代表没有资源可供消耗，此时如果有线程调用 `wait_sem` 操作系统应当挂起当前线程，直到有其他的线程调用了 `signal_sem` 。

内核中实现信号量

为了能够允许用户态的应用程序使用操作系统内核中的信号量，我们需要提供三个新的系统调用，分别对应信号量的创建、wait操作与signal操作。

- `sys_create_sem()` ： 要求内核创建一个新的信号量，其返回该信号量的 `cap` 。
- `sys_wait_sem(int sem_cap, int is_block)` ： 等待在某一个信号量上，需要传入等待的信号量的 `cap` ，并设定是否阻塞。如果 `is_block` 是1，则代表如果信号量不可用则挂起当前线程并等待。否则代表如果信号量无可用的资源时返回 `-EAGAIN` 。
- `sys_signal_sem(int sem_cap)` ： 唤醒等待在某一信号量上的线程。

在书中，我们介绍了如何使用条件变量实现信号量。而条件变量中最重要也是需要操作系统辅助实现的一部分是保证**原子的放锁与阻塞（挂起）**。同样的，如果我们需要在操作系统内核中实现信号量，其中最重要的一环是保证在检查发现没有可用的资源（即 `sem->count == 0` ）后并在真正挂起当前线程之前，不会有其他的线程调

用 signal_sem 。也即保证检查计数器到挂起两个操作的原子性。不过，ChCore采用了大内核锁。这代表所有的wait操作与signal操作都是互斥的，因此也保证了上述情况不会发生。

> **练习15：** ChCore在 kernel/semaphore/semaphore.h 中定义了内核信号量的结构体，并在 kernel/semaphore/semaphore.c 中提供了创建信号量 init_sem 与信号量对应syscall的处理函数。请补齐 wait_sem 操作与 signal_sem 操作。
>
> 在完成后运行 test_sem.bin 应能看到如下输出：
>
> ```
> Hello thread 2! Before delay!
> Hello thread 2! Before signal!
> Thread 2 return
> Hello thread 1! wait sem 6 return
> Thread 1 return
> ```
>
> **提示:**
>
> - 线程结构体中提供了 sem_queue_node 可以用于将该线程加入到 sem 的等待队列
> - 使用之前调度器实现的接口可以实现挂起线程的功能。
>
> **注意，本部分实验将在 test_sem.bin 的应用程序中进行。请自行修改 .config 文件，将启动程序修改为 test_sem.bin 。**
>
> ```
> CHCORE_ROOT_PROGRAM:STRING=test_sem.bin
> ```
>
> 完成本练习应能通过 sem 测试，并获得相应的5分。

生产者消费者

使用内核信号量可以用于实现生产者消费者问题，我们也将通过这个实验检测你内核信号量实现的正确与否。为了辅助实现生产者消费者问题，我们提供了线程安全的缓冲区 buf.c/h 。如教材中的例子，为了保证多生产者多消费者对缓冲操作的正确性，我们实现了 buffer_add_safe 与 buffer_remove_safe ， 其用到了在用户态的自旋锁 spin.c/h 。除此之外，我们提供了用于生产者生产新的信息的方法 produce_new ， 以及用于消费者消费新信息的方法 consume_msg 。

> **练习16：** 在 userland/apps/lab4/prodcons_impl.c 中实现 producer 和 consumer 。
>
> 完成后运行 prodcons.bin 应能看到如下输出：
>
> ```
> Progress:0%==10%==20%==30%==40%==50%==60%==70%==80%==90%==100%
> Producer/Consumer Test Finish!
> ```
>
> **注意，本部分实验将在 prodcons.bin 的应用程序中进行。请自行修改 .config 文件，将启动程序修改为 prodcons.bin 。**
>
> ```
> CHCORE_ROOT_PROGRAM:STRING=prodcons.bin
> ```
>
> 完成本练习应能通过 prodcons 测试，并获得相应的5分。

阻塞互斥锁

本实验的最后将使用**内核信号量**可以用于实现**用户态**阻塞互斥锁。阻塞互斥锁在互斥锁不空闲时会将当前线程挂起，当互斥锁可用时再由锁持有者唤醒。

> **练习17：** 请使用内核信号量实现阻塞互斥锁，在 `userland/apps/lab4/mutex.c` 中填上 `lock` 与 `unlock` 的代码。注意，这里不能使用提供的 `spinlock` 。
>
> 完成后运行 `test_mutex.bin` 应能看到如下输出：
>
> ```
> Begin Mutex Test!
> Global Count 1600
> test_mutex passed!
> ```
>
> **注意，本部分实验将在 `test_mutex.bin` 的应用程序中进行。请自行修改 `.config` 文件，将启动程序修改为 `test_mutex.bin` 。**
>
> ```
> CHCORE_ROOT_PROGRAM:STRING=test_mutex.bin
> ```
>
> 完成本练习应能通过 `mutex` 测试，并获得相应的5分。