Lab4 多核与锁

本次实验的内容包括：

- 多核启动
- 加锁，处理并发问题

首先，合并新的代码：

```
git fetch --all
git checkout lab4
git merge lab3 # Or lab3-dev, or else
# Handle Marge Conflicts
```

4.0 习题部分

4.0.1 多核启动

本次实验中，`src/start.S` 新增了一部分代码。请阅读 `src/start.S` 从 `_start` 开始的代码、和下面阅读部分的 4.1~4.3 ，从以下几个角度分析多核的启动流程：

- 各 CPU 的初始 PC
- 各 CPU 的执行路径（按执行顺序，经过了 `_start, entry, el3, el2, el1` 中的哪些）
- 各 CPU 的栈指针和页表的初始值，以及在执行中是如何变化的

本部分不需要添加任何代码。

4.0.2 加锁

> 共享；可变资源；并发访问；至少有一个写操作

请阅读 `src/common/spinlock.c` 和 `src/common/spinlock.h` 。

我们给出了 `spinlock` 的简单实现：

```
typedef struct {
    volatile bool locked;
    struct cpu *cpu; // 哪个 CPU 持有了锁，用于 debug
    char *name; // 锁的名字，用于 debug
} SpinLock;
```

为了加锁，你需要调用以下两个函数：

- `acquire_spinlock(SpinLock *lock)` 加锁
- `release_spinlock(SpinLock *lock)` 释放锁

在 `sched_simple.c` 中，`ptable.lock` 是调度器的私有资源。如果你想在 `proc.c` 等其他文件里申请 `ptable.lock`，请使用 `sched_simple.c` 中提供的其他接口 `acquire_ptable_lock()` 和 `release_ptable_lock()` 。

本实验需要用到的锁，都已经定义好并进行了初始化。如果你想加新的锁，需要先定义锁的变量，再调用 `init_spinlock(SpinLock *lock, char *name)` 进行初始化，在 `name` 参数上给它一个名字。

我们已经完成的实验里，用到的共享可变资源是物理内存数据结构 `freelist` 和进程表 `ptable`（TIPS：如果你使用了全局变量，这个变量就很可能需要用锁来保护）。在涉及到这两个资源的访问时，需要加锁。

大部分锁的 `acquire` 和 `release` 都是在同一函数的执行流里完成的：

```
void foo() {
    acquire_spinlock(&a);

    // ... 中间的代码及调用的函数不会对 a 进行 acquire 或 release

    release_spinlock(&a);
}
```

但在 `ptable.lock` 里，可能出现这种情况：

```
void bar() {
    acquire_spinlock(&b);

    // ...

    swtch();

    // ...

    release_spinlock(&b);
}
```

在调用 `swtch()` 时，程序执行流转移到另一函数（可能是那个函数的 critical section），这就需要进行一定的分析。

以下是几点提示：

- `scheduler_simple()` 需要检查 `p->state`，所以在遍历 ptable 前需要加锁
- 对于 `spawn_init_process` 来说，`swtch` 是转移到 `forkret`，在这之后都不会使用 ptable 的资源，所以 `forkret` 会调用 `release_ptable_lock()`。
- `sched()` 的调用者需要持有 `ptable.lock`。该函数在 `sched()` 返回后再释放 `ptable.lock`。（lab3 那张执行图的一般路径）

本实验的测试和 lab3 一致，通过测试的现象也与 lab3 一致。

```
sys_exec: executing /init
sys_exit: in exit
```

以下为阅读部分。

4.1 SMP 架构

树莓派以及大部分笔记本都是 SMP（symmetric multiprocessing） 架构的，即所有的 CPU 拥有各自的寄存器（包括通用寄存器和系统寄存器），但共享相同的内存和 MMIO。尽管所有 CPU 的功能都是一样的，我们通常会把第一个启动的、运行 BIOS 初始化代码的 CPU 称为 BSP（ bootstrap processor），其他的 CPU 称为 APs（application processors）。但树莓派比较奇葩，是从 GPU 启动的，即 GPU 完成了类似 BIOS 的硬件初始化过程，然后才进行 CPU 的执行。

由于操作系统中有时候需要知道当前运行在哪个 CPU 上，ARMv8 用 MPIDR_EL1 寄存器来提供每个 CPU 自己的 ID。获取当前 CPU 的 ID 的方法见 `inc/arm.h:cpuid`。

更详细的内容请参考 ARM Cortex-A Series Programmer's Guide for ARMv8-A Chapter 14。

4.2 APs 的初始状态

qemu 启动时，会把 armstub8.S 编译后的二进制直接拷贝到物理地址为 0 的地方，然后所有 CPU 均以 PC 为 0 开始（但有些 CPU 会被 armstub8.S 停住，只有一个 CPU 会跳到 0x80000，具体见下文）。

而在真实的树莓派 3 上，armstub8.S 编译后的可执行代码已经被集成在了 start.elf 或 bootcode.bin 中，上板子时，我们需要把这两个文件拷贝到 SD 卡的第一个分区（必须是 FAT32），这样启动后 GPU 就会把 armstub8.S 的二进制代码从中抽取出来并直接拷贝到物理地址为 0 的地方，然后所有 CPU 均以 PC 为 0 开始。

armstub8.S 的伪代码如下。简言之，cpuid 为零 的 CPU（相当于 BSP）将直接跳转到 0x80000 的地方，而 cpuid 非零的 CPU（相当于 APs）将进行死循环直到各自的入口值非零，每个 CPU 的入口值在内存中的地址为 `0xd8 + cpuid()`。

```
extern int cpuid();

void boot_kernel(void (*entry)())
{
    entry();
}

void secondary_spin()
{
    void *entry;
    while (entry = *(void **)(0xd8 + (cpuid() << 3)) == 0)
        asm volatile("wfe");
    boot_kernel(entry);
}

void main()
{
    if (cpuid() == 0) boot_kernel(0x80000);
    else secondary_spin();
}
```

4.3 开启 APs

至于如何开启 APs，我们的 BSP 在 `start.S:_start` 中修改了 APs 的入口值并用 `sev` 指令唤醒它们。

为了提高代码的复用性，入口值都被修改成了 `start.S:entry`，相当于所有 CPU 都会从这里开始并行执行类似 Lab1 启动 中的过程，只不过每个 CPU 都有不同的栈指针。

从本次实验开始，开启 APs 之后的所有代码都会被所有 CPU 并行执行，**修改或读取全局变量时记得加锁。**

4.4 同步和锁

在 SMP 架构下，我们通常需要对内存中某个位置上的互斥访问。ARMv8 为我们提供了一些简单的原子指令如读互斥 LDXR 和写互斥 STXR 来完成此类操作，但这些指令只能被用于被设定为 inner or outer sharable, inner and outer write-back, read and write allocate, non-transient 的 Normal Memory 中。而我们确实在 `aarch64/mmu.h:PTE_NORMAL` 和 `start.S:TCR_VALUE` 中进行了相应的配置，于是这类原子指令可以正常使用。

我们在 `common/spinlock.c` 中用 GCC 提供的 atomic 宏实现了一种最简单的锁——自旋锁，相应的汇编代码可在导出的 `build/src/kernel8.asm` 中找到。当然你也可以尝试着实现 rwlock, mcs, mutex 等较为复杂的锁。

4.4.1 原子操作

下面列出了几种常见的原子操作的伪代码

Test-and-set (atomic exchange)

```
int test_and_set(int *p, int new)
{
    int old = *p;
    *p = new;
    return old;
}
```

Fetch-and-add (FAA)

```
int fetch_and_add(int *p, int inc)
{
    int old = *p;
    *p += inc;
    return old;
}
```

Compare-and-swap (CAS)

```
int compare_and_swap(int *p, int cmp, int new)
{
    int old = *p;
    if (old == cmp)
        *p = new;
    return old;
}
```

4.4.2 自旋锁

```
struct spinlock {
    int locked;
};
void spin_lock(struct spinlock *lk)
{
    while (test_and_set(lk->locked, 1)) ;
}
void spin_unlock(struct spinlock *lk)
{
    test_and_set(lk->locked, 0);
}
```

4.4.3 Ticket lock

```
struct ticketlock {
    int ticket, turn;
};
void ticket_lock(struct ticketlock *lk)
{
    int t = fetch_and_add(lk->ticket, 1);
    while (lk->turn != t) ;
}
void ticket_unlock(struct ticketlock *lk)
{
    lk-turn++;
}
```

Ticket lock 中等待的 CPU 都在不停地访问同一个变量 `lk->turn`，一旦锁被释放并修改 `lk->turn`，则需要修改所有等待中的 CPU 的 cache，当 CPU 数较多时，性能较差

4.4.4 MCS 锁

这是一种基于队列的锁，对多核的缓存比较友好，并用 test-and-set 和 compare-and-swap 提高了效率，以下实现参考自 CS140-Synchronization2

```
struct mcslock {
    struct mcslock *next;
    int locked;
};
void mcs_lock(struct mcslock *lk, struct mcslock *i)
{
    i->next = i->locked = 0;
    struct mcslock *pre = test_and_set(&lk->next, i);
    if (pre) {
        i->locked = 1;
        pre->next = i;
    }
    while (i->locked) ;
}
void mcs_unlock(struct mcslock *lk, struct mcslock *i)
{
    if (i->next == 0)
        if (compare_and_swap(&lk->next, i, 0) == i)
            return;
    while (i->next == 0) ;
```

```
    i->next->locked = 0;
}
```

4.4.5 读写锁

下面用两个自旋锁（或支持睡眠的 mutex）来实现读优先的读写锁

```
struct rwlock {
    struct spinlock r, w; /* Or mutex. */
    int cnt; /* Number of readers. */
};
void read_lock(struct rwlock *lk)
{
    acquire(&lk->r);
    if (lk->cnt++ == 0)
        acquire(&lk->w);
    release(&lk->r);
}
void read_unlock(struct rwlock *lk)
{
    acquire(&lk->r);
    if (--lk->cnt == 0)
        release(&lk-w);
    release(&lk->r);
}
void write_lock(struct rwlock *lk)
{
    acquire(&lk->w);
}
void write_unlock(struct rwlock *lk)
{
    release(&lk->w);
}
```

4.4.6 信号量

信号量（Semaphore）适用于控制一个仅支持有限个用户的共享资源，可以保证同一时刻最多有 cnt 个该资源被占用（cnt 的定义如下）。当 cnt 等于 1 时就是互斥锁（Mutex）。

```
struct semaphore {
    /* If cnt < 0, -cnt indicates the number of waiters. */
    int cnt;
    struct spinlock lk;
};
void sem_init(struct semaphore *s, int cnt)
{
    s->cnt = cnt;
}
void sem_wait(struct semaphore *s)
{
    acquire(&s->lk);
    if (--s->cnt < 0)
        sleep(s, &s->lk); /* Sleep on s. */
    release(&s->lk);
}
void sem_signal(struct semaphore *s)
```

```
{
    acquire(&s->lk);
    if (s->cnt++ < 0)
        wakeup1(s); /* Wake up one process sleeping on s. */
    release(&s->lk);
}
```

4.5 内存序

现代计算机体系对于内存访问进行了玩命优化，于是有了如下两个层面的优化

- 编译器优化，可能会导致编译产生的汇编代码的乱序
- 处理器优化，包括指令乱序执行、cache 一致性

可以看出这两者优化都是基于乱序，但完全乱序显然不行，于是为了便于编程并保证正确性，大部分编译器和处理器会遵循一定的原则，即「不能修改单线程的行为」（Thou shalt not modify the behavior of a single-threaded program.）。但在多线程的程序中会有一些问题，例如下述程序执行后 r1、r2 可能同时为 0。

```
X = 0, Y = 0;

Thread 1:
X = 1;
r1 = Y;

Thread 2:
Y = 1;
r2 = X;
```

于是我们需要显式的告诉编译器（C++ 内存模型，最终也是通过处理器提供的指令来实现）或处理器（指令屏障和内存屏障）一些额外的信息，因为编译器和处理器并不知道哪些数据是在线程间共享的，这是需要我们手动控制的。

屏障

在大多数弱内存模型（简称WMO，Weak Memory Ordering）的架构上，可通过汇编指令来告诉处理器，我们期望的内存访问读取的顺序。ARMv8 上用屏障来实现，有如下三种屏障相关的指令：

- Instruction Synchronization Barrier (ISB)：清空处理器中的指令流水，可确保在 ISB 指令完成后，才从 cache 或内存中读取位于 ISB 指令后的其他所有指令，通常作为修改系统寄存器（如 MMU）时的屏障
- Data Memory Barrier (DMB)：在当前 shareability domain 内，该指令之前的所有内存指令一定先于其后的内存指令，但对非内存指令没有要求
- Data Synchronization Barrier (DSB)：在当前 shareability domain 内，该指令之前的所有内存指令一定先于其后的内存指令，其后的所有指令均需要等到 DSB 指令完成后才能执行

One-way barriers

ARMv8 还提供两种内存操作中常用的单向屏障（One-way barriers）如下，单向屏障的性能优于之前的 DMB，且天然适合锁的实现。

- Load-acquire (LDAR): All loads and stores that are after an LDAR in program order, and that match the shareability domain of the target address, must be observed after the LDAR.

- Store-release (STLR): All loads and stores preceding an STLR that match the shareability domain of the target address, must be observed before the STLR.

其中 program order 就是我们所写的汇编代码的顺序（而扔到处理器上执行的顺序是乱序后的），shareability domain 是需要我们告诉 MMU 的（见 `start.S` 中对于 TCR 的配置，把所有内存都统一配置成了 inner shareable domain，即所有核）

C++ 内存模型

术语

sequence-before：同一个线程中的多个语句之间就是sequenced-before关系，如下 ① sequenced-before ②：

```
int i = 7; // ①
i++;       // ②
```

happens-before：顾名思义，全局事件发生的顺序关系，即多线程版的 sequence-before

Sequentially Consistent

对应 `std::memory_order_seq_cst`，是最严格的内存模型，C++ 原子操作默认采用这种模型，保证了

- 程序指令与源码顺序一致
- 所有线程的所有操作存在一个全局的顺序，即前面的原子操作 happens-before 后面的原子操作

如下例子的 assert 一定不会失败

```
Thread 1:
y.store(20);
x.store(10);

Thread 2:
if (x.load() == 10) {
    assert(y.load() == 20);
    y.store(10);
}

Thread 3:
if (y.load() == 10) {
    assert(x.load() == 10);
}
```

Relaxed

对应 `std::memory_order_relaxed`，最宽松的内存模型，除了保证之前提到的「不修改单线程的行为」原则外无任何约束，编译器可以尽情的优化

Acquire/Release

第三种方案间于前两种之间，类似 Sequentially Consistent 模式，保证了

- 同一个对象上的原子操作不允许被乱序
- Release 操作禁止了所有在它之前的读写操作与在它之后的写操作乱序
- Acquire 操作禁止了所有在它之前的读操作与在它之后的读写操作乱序

```
x = 0, y = 0;

Thread 1:
y.store(20, memory_order_release);

Thread 2:
x.store(10, memory_order_release);

Thread 3:
assert(y.load (memory_order_acquire) == 20 && x.load (memory_order_acquire) ==
0)

Thread 4:
assert(y.load (memory_order_acquire) == 0 && x.load (memory_order_acquire) ==
10)
```

上述两个 assert 可能都成功，但若采用 Sequentially Consistent 模式的话，至多一个成功

参考资料

1. C++ 内存模型
2. The C++11 Memory Model and GCC 关于 `__sync` 和 `__atomic` 宏
3. ARM Cortex-A Series Programmer's Guide for ARMv8-A Chapter 13 Memory Order
4. ARM Exclusive - 互斥锁与读写一致性的底层实现原理

Exercise

截止时间：`2021-10-29 15:24:59`。

提交方式：将实验报告提交到 `elearning` 上，文件名：`学号-lab4.pdf`。