并发控制基础

目录

互斥与Peterson算法

失败的尝试

```
int locked = UNLOCK;

void critical_section() {
retry:
  if (locked != UNLOCK) {
    goto retry;
  }
  locked = LOCK;

  // critical section

  locked = UNLOCK;
}
```

- 并发程序不能保证 `load + store` 的原子性。

由于上述程序中无法保证lock/unlock的原子性，我们需要一定的假设(硬件/软件帮助)。

假设：内存的读/写可以保证顺序、原子完成

- ```
  val = atomic_load(ptr)
  ```
  - 对应往某个地方“贴一张纸条”（必须闭眼盲贴）
  - 贴完一瞬间就可能被别人覆盖
- ```
  atomic_store(ptr, val)
  ```
  - 看一眼某个地方的字条（只能看到瞬间的字）
  - 刚看完就可能被改掉

正确性不明的奇怪尝试（Peterson 算法）

A 和 B 争用厕所的包厢

- 想进入包厢之前，A/B 都首先举起自己的旗子；
  - A 往厕所门上贴上“B 正在使用”的标签；
  - B 往厕所门上贴上“A 正在使用”的标签；
- 然后，如果对方举着旗，且门上的名字是对方，等待；
  - 否则可以进入包厢；
- 出包厢后，放下自己的旗子（完全不管门上的标签）；

```
turn = 0
flags = {False, False}

def enter_region(process: int)-> None:
    other = 1 - process
    flags[process] = True
    turn = process

    while turn == process and flags[other] == True:
        pass
     return leave_region(process)

def leave_region(process: int) -> None:
    flags[process] = False
```

从模型回到现实……

回到我们的假设（体现在模型）

- Atomic load & store
  - 读/写单个全局变量是“原子不可分割”的
  - 但这个假设在现代多处理器上并不成立
- 所以实际上按照模型直接写 Peterson 算法应该是错的?

“实现正确的 Peterson 算法”是合理需求，它一定能实现

- Compiler barrier/volatile 保证不被优化的前提下
  - 处理器提供特殊指令保证可见性
  - 编译器提供 `__sync_synchronize()` 函数
    - x86: `mfence`; ARM: `dmb ish`; RISC-V: `fence rw, rw`
    - 同时含有一个 compiler barrier

原子指令

并发编程困难的解决

普通的变量读写在编译器 + 处理器的双重优化下行为变得复杂

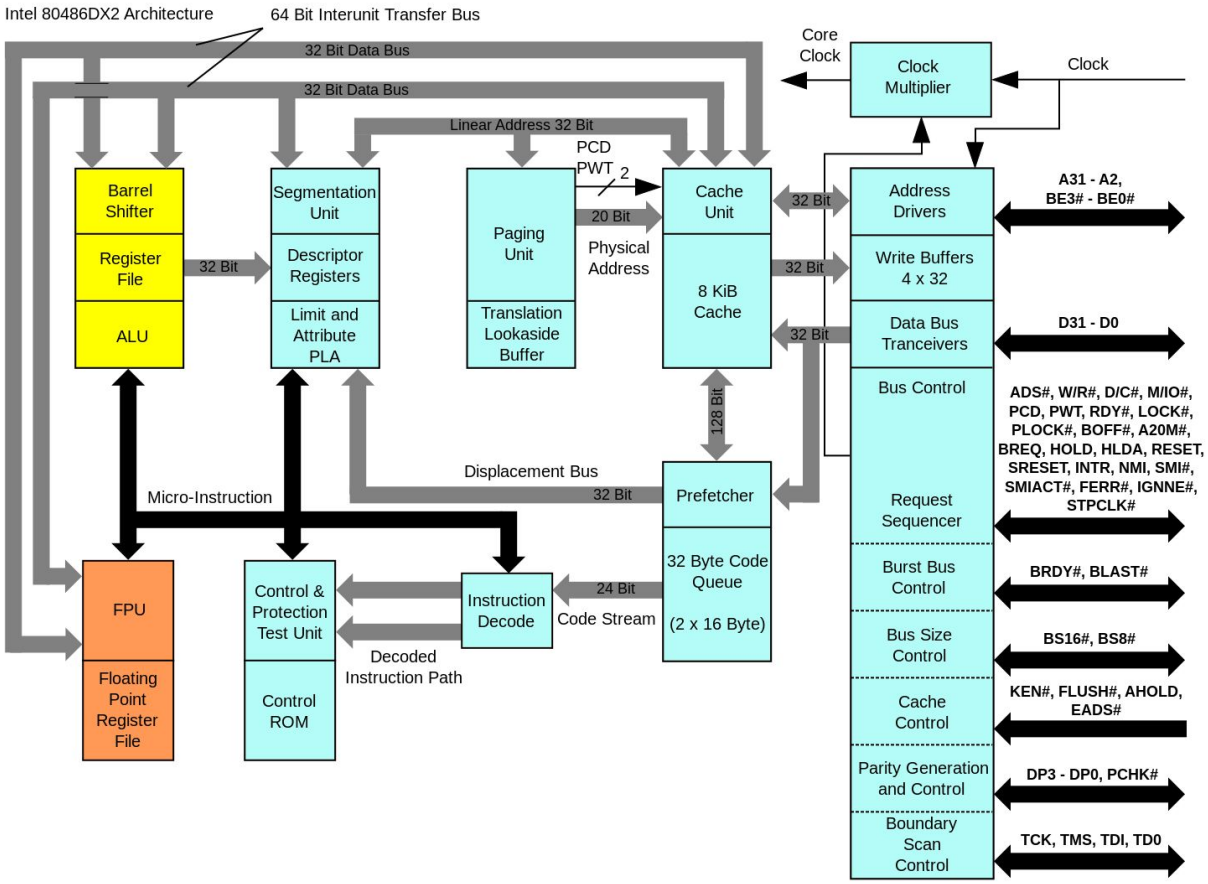
```
retry:
  if (locked != UNLOCK) {
    goto retry;
  }
  locked = LOCK;
```

解决方法：编译器和硬件共同提供不可优化、不可打断的指令

- “原子指令” + compiler barrier

```
for (int i = 0; i < N; i++)
  asm volatile("lock incq %0" : "+m"(sum));
```

“Bus lock”—从 80386 开始引入（bus control signal）



屏蔽中断（maskable）

在单处理器系统中，最简单的方法是使每个进程在刚刚进入临界区后立即屏蔽所有中断，并在就要离开之前再打开中断。屏蔽中断后，时钟中断也被屏蔽。CPU只有发生时钟中断或其他中断时才会进行进程切换，这样，在屏蔽中断之后CPU将不会被切换到其他进程。于是，一旦某个进程屏蔽中断之后，它就可以检查和修改共享内存，而不必担心其他进程介入。

对于用户程序，这个方案并不好，因为把屏蔽中断的权力交给用户进程是不明智的。设想一下，若一个进程屏蔽中断后不再打开中断，其结果将会如何？整个系统可能会因此终止。而且，如果系统是多处理器（有两个或可能更多的处理器），则屏蔽中断仅仅对执行disable指令的那个CPU有效。其他CPU仍将继续运行。并可以访问共享内存。

另一方面，对内核来说，当它在更新变量或者说数据结构的几条指令期间将中断屏蔽是很方便的。

所以结论是：屏蔽中断对于操作系统本身而言是一项很有用的技术，但对于用户进程则不是一种合适的通用互斥机制。但是，对目前比较普遍的多核心CPU来讲，屏蔽中断这一方式的有效性越来越低了。

自旋锁：实现

执行TSL/XCHG指令的CPU将锁住内存总线，以禁止其他CPU在本指令结束之前访问内存。锁住总线是一种在多处理器上的变相屏蔽中断的方案。

然而，这种方式也会逐渐失效。因为，我们现代的CPU还有cache。

```
int xchg(volatile int *addr, int newval) {
  int result;
  asm volatile ("lock xchg %0, %1"
    : "+m"(*addr), "=a"(result) : "1"(newval));
  return result;
}
```

```
int locked = 0;
void lock() {
    while (xchg(&locked, 1));
}
void unlock() {
    xchg(&locked, 0);
}
```

Take-away Messages

并发编程“很难”：想要完全理解并发程序的行为，是非常困难的——我们甚至可以利用一个“万能”的调度器去帮助我们求解 NP-完全问题。因此，人类应对这种复杂性的方法就是退回到不并发。通过互斥实现 stop/resume the world，我们就可以使并发程序的执行变得更容易理解——而只要程序中“能并行”的部分足够多，串行化一小部分也并不会对性能带来致命的影响。

课后习题/编程作业

1. 阅读材料

继续阅读教科书 Operating Systems: Three Easy Pieces 第 5 次课布置的阅读材料：

- 第 25 章 - Dialogue on Concurrency
- 第 26 章 - Concurrency and Threads
- 第 27 章 - Thread API

2. 编程实践

确保你运行、理解课堂上的示例代码。完成编程实验。