2020 年秋操作系统 xv6 源码阅读报告 4

锁

黎善达

18000012961@pku.edu.cn

2020 年 11 月 15 日

1 关键代码阅读与分析

xv6 源码中，涉及锁的机制的主要文件包括 `spinlock.h`，`spinlock.c`，`sleeplock.h`，`sleeplock.c`，该部分将重点阅读、分析这些代码。

1.1 spinlock.h

该文件，连同 `spinlock.c`，实现了**自旋锁**的机制。所谓自旋锁，即通过忙等待实现的锁。

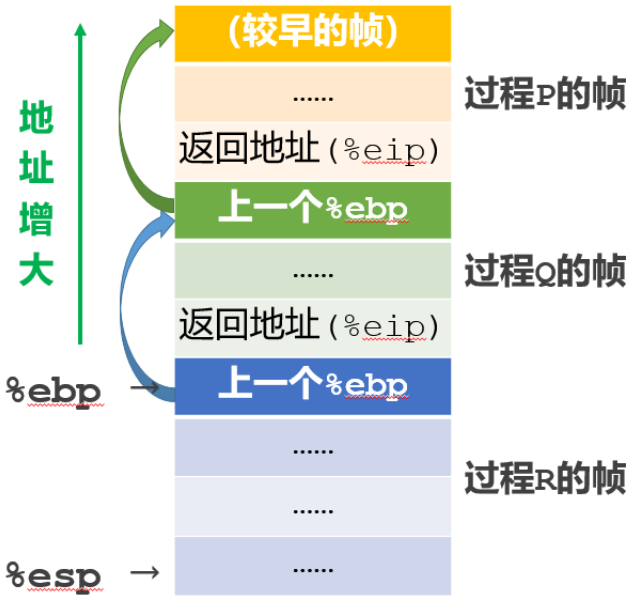
该文件的主要内容是自旋锁结构 `spinlock` 的定义。该结构十分简单，最关键的变量只有一个，即 `locked`，用于表示该锁是否被其他进程获得；此外还包括了用于调试的信息，即锁的名称、持有锁的 CPU、锁的调用栈。

1.2 spinlock.c

该文件包含了与自旋锁相关的基本操作。这里的分析并不按照文件中函数出现的顺序进行，而是先分析辅助函数 (helper function)，再分析核心功能。核心功能中涉及对辅助函数的调用，因此先理解辅助函数有助于理解核心功能。

1.2.1 void getcallerpcs(void *v, uint pcs[])

该函数用于获得（锁的）调用栈。为了说明这一程序的机理，我们将运行时栈的内容作图如下：



获得调用栈并存储于 pcs[] 数组中的代码可以理解为（略去了个别细节）：

```
1  for(i = 0; i < 10; i++){
2    //... (Some necessary check for ebp)
3    //... (If check fails, break from the loop)
4
5    pcs[i] = ebp[1]; // saved %eip
6    ebp = (uint*)ebp[0]; // saved %ebp
7  }
8  for(; i < 10; i++)  pcs[i] = 0;
```

在执行循环体前，该函数会根据传入参数，计算出当前栈帧的帧指针%ebp。注意观察运行时栈的结构可以看出，ebp[1] 即当前函数的返回地址，故将其存入 pcs[] 数组中（对应上述代码第 4 行）。同时，在运行时栈中，帧指针所指向的内存总是包含着指向上一个栈帧的指针，因此通过执行上述代码第 5 行，ebp 被更新为上一个栈帧的帧指针。通过重复执行循环体，该程序在 pcs[] 数组中逐个存入一列程序计数器，对应着运行时栈上函数调用发生的位置。

由于 pcs[] 数组大小的限制，该函数仅向上回溯至多 10 次。上述代码中第 2 行略去了一些对 ebp 的判断，这些判断用于保证 ebp 确实对应一个正确的帧指针，如果这些判断未被通过，就将从第一个循环体退出，通过第二个循环体将 pcs[] 数组的剩余位置置 0。

综上，该函数将在 pcs[] 数组中保存最近 10 次过程调用相关的程序计数器；若执行的过程调用不足 10 次，则以 0 补齐。

1.2.2 int holding(struct spinlock *lock)

该函数是一个简单的判断，用于检查当前 CPU 是否持有参数所指明的锁。若锁 lock 的 locked 变量为 1，且 cpu 变量指向当前的 CPU，则返回 1；否则返回 0。

1.2.3 void pushcli(void) 和 void popcli(void)

这两个函数用于实现在允许多层开关中断的情境下中断使能的维护。在关于进程模型的代码阅读中，我们曾注意到，描述 CPU 信息的 cpu 结构体中有一个 ncli 变量，该变量事实上就记录了该 CPU 关中断的层数。

这两个函数的实现使得多层开关中断满足如下特点：

- 关中断层数为非负数。当因为调用 popcli() 而导致 ncli 变量变为负数时，操作系统将调用 panic() 函数，报告这一情况并崩溃。
- 调用 pushcli() 后，关中断层数增加 1；调用 popcli() 后，关中断层数减少 1。
- 当关中断层数不小于 1 时，当前 CPU 关中断；当关中断层数为 0 时，当前 CPU 可以接收中断。pushcli() 和 popcli() 函数通过对 ncli 变量是否为 0 进行特判以实现这一点。若调用 popcli() 时，系统发现此时 CPU 允许中断，则将调用 panic() 函数，报告这一情况并崩溃。

在涉及锁的代码中正确处理中断是重要的。在开中断的条件下，考虑这样的情境：一个进程持有某个锁 L，且在未释放 L 时因为中断而进入了中断处理程序 P。若此时程序 P 恰好也需要获得锁 L，则将由于锁 L 未被释放而一直忙等待；且由于中断处理程序无法正常返回，该进程将始终无法执行释放锁 L 的指令——这就造成了死锁。

为了避免这种情况，当中断处理程序会使用某个锁时，处理器就不能在允许中断发生时持有这把锁。xv6 采用了一种更简单直接的处理方式：开中断条件下不能持有任何自旋锁。接下来将会分析的获得锁、释放锁函数（即 acquire()、release()）将通过 pushcli() 和 popcli() 函数保证这一点。

1.2.4 void initlock(struct spinlock *lk, char *name)

该函数进行自旋锁的初始化，在参数 lk 指向的内存空间中写入相应的初始信息。自然地，初始化时，锁的 locked 变量被置 0，即没有进程持有这把锁。

1.2.5 void acquire(struct spinlock *lk)

该函数将使得调用它的进程获得参数中所指的锁。如果参数中所指的锁已被占用，当前进程就会忙等待：重复执行一个 while 循环直至能够获得锁。获得锁后，该函数还会维护锁结构体中存储的用于调试的响应信息，即通过 mycpu() 函数和前文所述的 getcallerpcs() 函数维护持有锁的 CPU、相应的调用栈。

该函数中，有若干要点值得注意：

（1）该函数的第一条语句就是 pushcli()，确保在关中断条件下获得锁。这里，必须使用 pushcli() 而不宜使用 cli()：因为在一个进程同时持有多把锁时，应多次调用 pushcli()（从而在当前 CPU 对应结构体中的 ncli 变量中记录关中断的层数），每释放一把锁时调用一次 popcli()；只有在所有的锁都被释放时，先前调用的 pushcli() 才会被全部抵消，从而恢复接收中断。如此做确保了在允许接收中断时不持有任何锁。若只是简单地调用 cli() 关中断，则无法记录持有多把锁带来的多次关中断的情况，也就无法确保仅在所有锁都被释放时才接收中断。

（2）该函数会检查当前进程在调用前是否已经持有了参数中所指的锁。若已持有，操作系统将调用 panic() 函数，报告这一情况并崩溃。

（3）该函数采用 x86 提供的原子化的指令 xchg 来实现忙等待和获得锁。逻辑上，忙等待和获得锁的代码应等同于：

```
1  for(;;) {
2    if(!lk->locked) {
3      lk->locked = 1;
4      break;
5    }
6  }
```

但上面的循环体不是原子化的，故可能存在两个进程同时经过第 2 行的判断发现 lk->locked 为零，然后都将其置 1 并认为自己获得了锁 lk，这与一把锁至多只能被一个 CPU 拥有的原则相违。为避免这一点，该函数采用了如下的实现：

```
1  while(xchg(&lk->locked, 1) != 0)
2    ;
```

采用 xchg 指令将内存中的 lk->locked 与数字 1 交换，并检查内存中 lk->locked 的值。若原先内存中 lk->locked 的值是 1，则 xchg(&lk->locked, 1) 并不会改变它，并且该函数再一次执行判断；若原先内存中 lk->locked 的值是 0，则 xchg(&lk->locked, 1) 会将之置 1，且跳出 while 循环（即不再忙等待）并获得锁 lk。

（4）在上述忙等待的代码与获得锁后维护锁的相关信息的代码之间，有如下一行代码：

```
1  __sync_synchronize();
```

编译器在编译过程中，可能会通过交换部分语句的顺序以优化性能；硬件在执行指令时也可能如此做。若维护锁的相关信息的指令被移到了获得锁之前执行，则可能出现错误。这里，上述的代码如同一道栅栏，提示编译器和硬件进行优化时不能改变这一条语句前后的指令的顺序，从而确保锁确实发挥其

相应功能。

1.2.6 void release(struct spinlock *lk)

该函数将使得调用它的进程释放参数中所指的锁。释放锁前，该函数会先清除锁结构体中存储的用于调试的响应信息，即持有锁的 CPU、相应的调用栈。

该函数中，有若干要点值得注意：

（1）该函数会检查当前进程在调用前是否已经持有了参数中所指的锁。若未持有，操作系统将调用 panic() 函数，报告这一情况并崩溃。

（2）该函数中也调用了 __sync_synchronize() 函数，调用的时机恰好是将 lk->locked 置 0 前。这将确保经过编译器和硬件的优化后，所有需要保护的指令仍会在将 lk->locked 置 0 前执行完，从而确保锁确实发挥其相应功能。

（3）该函数采用 x86 提供的原子化的指令 movl 将 lk->locked 置 0，其意图与 acquire() 函数使用 xchg 指令是一样的。同时，注释指出其他的操作系统会使用 C 语言提供的原子语句来实现，而不是直接使用汇编语言。

1.3 sleeplock.h

该文件，连同 sleeplock.c，实现了**睡眠锁**的功能。尽管自旋锁已经能够解决同步互斥问题，但睡眠锁依旧十分必要，小结部分将具体指出这一点。

该文件的主要内容是睡眠锁结构。该结构十分简单，最关键的变量有两个，即 locked 和 lk。前者用于表示该锁是否被其他进程获得；后者是一个 spinlock 类型的结构体，用于处理维护睡眠锁时会遇到的互斥问题。此外还包括了用于调试的信息，即锁的名称和持有锁的进程号。

1.4 sleeplock.c

该文件包含了与睡眠锁相关的基本操作。由于已经有自旋锁功能的存在，这些操作中需要解决互斥问题的临界区代码都由睡眠锁结构中的自旋锁来保护。

1.4.1 void initsleeplock(struct sleeplock *lk, char *name)

该函数进行自旋锁的初始化，在参数 lk 指向的内存空间中写入相应的初始信息。自然地，初始化时，锁的 locked 变量和 pid 变量被置 0，即没有进程持有这把锁。该函数还需要调用 spinlock.c 文件中的 initlock() 函数，初始化睡眠锁中的自旋锁。

1.4.2 void acquiresleep(struct sleeplock *lk)

该函数将使得调用它的进程获得参数中所指的睡眠锁。如果参数中所指的锁已被占用，当前进程就会忙睡眠，直至能够获得参数中的锁。获得锁后，该函数将锁的 locked 变量置 1，维护锁结构体中存储的用于调试的响应信息。

该函数中，有若干要点值得注意：

（1）由于整个函数都涉及对睡眠锁的访问，因此整个函数的执行过程都由需要睡眠锁对应的自旋锁保护以实现互斥。

（2）该函数采用如下循环检查 lk->lock 变量是否为 1：

```
1    while (lk->locked) {
2      sleep(lk, &lk->lk);
3    }
```

这里，采用 while 语句是必要的，因为一个睡眠中的进程可能会在它所等待的条件未满足时就被唤醒，采用 while 语句使得进程每次被唤醒时都能检查它所等待的条件是否满足。如果采用的是 if 语句，则有可能造成错误。

该函数还调用了 sleep() 函数，由于 sleep() 函数在进程模型部分已经进行了分析，此处不再重复。

1.4.3 void releasesleep(struct sleeplock *lk)

该函数将使得调用它的进程释放参数中所指的睡眠锁。释放锁时，该函数会将 locked 和 lk 变量置 0，并唤醒在 lk 上睡眠的进程（从而使得因为调用 acquiresleep() 而进入睡眠状态的锁被唤醒以获得锁）。

与上文类似，该函数也通过使用自旋锁来确保对 lk 访问的互斥。

1.4.4 int holdingsleep(struct sleeplock *lk)

该函数是一个简单的判断，用于检查参数所指明的锁是否被持有。因此，该函数本质上只需要将 lk->locked 的值直接返回即可。但由于涉及共享资源 lk 的访问，因此该函数使用了睡眠锁 lk 中的自旋锁对访问进行保护。

2 小结

xv6 提供了两种锁的机制，这两种机制的实现都十分注意系统运行时的问题，并通过各种措施（例如使用多重关中断、汇编指令等）避免了潜在的问题。下面从几个角度对其予以评价：

2.1 自旋锁、睡眠锁的特点

自旋锁的机制足以处理大部分同步互斥问题。但是自旋锁采用忙等待实现，带来效率上的不足。xv6 在自旋锁的基础上实现了**自旋锁**。睡眠锁的引入是十分必要的，它使得一个暂时不能够获得相应锁的进程进入睡眠态并在合适的时机唤醒，而不是进行浪费 CPU 资源的忙等待。尤其是当一个进程需要长期持有锁时，就应选择睡眠锁而不宜选择自旋锁。

2.2 死锁问题的避免

前文的分析表明，为了避免死锁，当中断处理程序会使用某个锁时，处理器就不能在允许中断发生时持有这把锁。在 xv6 中，开中断条件下不能持有任何自旋锁。相应地，由于持有睡眠锁并不需要关中断，因此**中断处理程序不能尝试获取任何睡眠锁**。

除此之外，由于调用 `acquiresleep()` 函数可能会使当前进程睡眠以让出 CPU，所以**在持有任何自旋锁时都不能调用** `acquiresleep()`。否则，调度程序可能会调度一个试图获取已被持有的锁的进程，从而带来死锁。

2.3 锁的使用粒度

在使用锁时，一个重要的折中权衡是锁的粒度的把握。如果为了实现上的简捷而采用“大内核锁”，则代码将大为简化；但这样就牺牲了并发性，因为同一时间只有至多一个 CPU 可以运行在内核上。但如果将锁的粒度设计得过细，又会导致需要的锁的数量大幅增加，进而导致代码实现和维护的难度的增大。

以先前分析过的进程模型、进程调度为例。xv6 用一把锁，即 `ptable.lock`，处理整个进程表 `ptable` 相关的互斥问题。可以看出，如果希望进一步提高并行性，就应该采用更精细的方法，使用更多自旋锁保护 `ptable` 中不同的变量；但 xv6 作为一个教学操作系统，通过只使用一把锁的方式使得代码清晰易读，也是合理的。