

XV6 源码第四次阅读报告

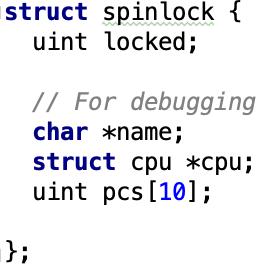
同步机制部分

XV6 源代码阅读——同步机制与虚存管理

、 阅读代码:

锁部分 spinlock.h spinlock.c 以及相关其他 件代码

1. spinlock.h：对 spinlock 结构体进 了定义。



Spinlock 结构体由 locked 变量表 锁是否 效。以下内容都是为了 debug：

name 为锁的名字，cpu 为在上锁的 cpu 状态，pcs 记录函数调 栈。

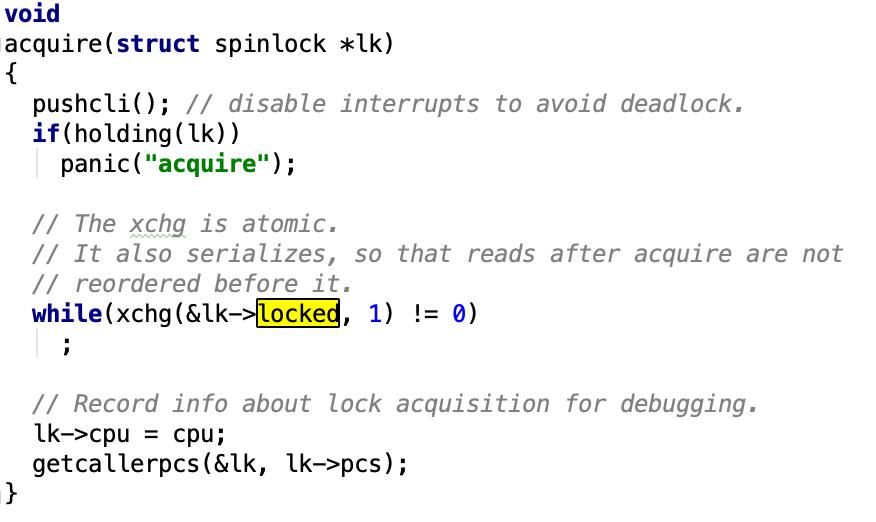
1. spinlock.c：initlock（）对锁进 了初始化。

holding（） 来判断当前 cpu 是否在上锁。

getcallerpcs（） 于记录当前函数调 栈。

pushcli（）是关闭中断，popcli（）是打开中断。

acquire（）是上锁，release（）是释放锁。上锁的步骤如下。 先关闭中断。xv6 采 了 386 硬件上的 条特殊指令 xchg。在这个原 操作中，xchg 交换了内存中的 个字和 个寄存器的值。函数 acquire（）在循环中反复使 xchg；每 次都读取 lk->locked 然后设置为 1。如果锁已经被持有了， lk->locked 就已经为 1 了，故 xchg 会返回 1 然后继续循环。如果 xchg 返回 0，但是 acquire 已经成功获得了锁，即 locked 已经从 0 变为了 1，这时循环可以停 了。 旦锁被获得了，acquire 会记录获得锁的 CPU 和栈信息，以便调试。函数 release（）则做了相反的事：清除调试信息并释放锁。



二、 问题讨论

请 家围绕如下 些问题阐述原理课的相关内容，以及 XV6 中是如何实现的。

1. 什么是临界区?什么是同步和互斥?什么是竞争状态?临界区操作时中断是否应该开启?中断会有什么影响?XV6 的锁是如何实现的，有什么操作?xchg 是什么指令，该指令有何特性？

临界区指的是 个访问共 资源（例如：共 设备或是共 存储器）的程序 段，这些共 资源有着 法被多个线程访问的特性。当有线程进 临界区段时，其他线程或是进程必须等待，有 些同步的机制必须在临界区段的进 点与离开点实现，以确保这些共 资源是被互斥获得使 。

同步亦称直接制约关系，它是指为完成某种任务 建 的两个或多个进程，这些进程因为需要在某些位置上协调它们的 作次序 等待、传递信息所

产 的制约关系。进程间的直接制约关系就是源于它们之间的相互合作。互斥亦称间接制约关系。当 个进程进 临界区使 临界资源时，另 个进程必须等待, 当占 临界资源的进程退出临界区后，另 进程才允许去访问此临界资源。

竞争状态：如果两个或者多个线程在没有互相同步的情况下，访问某个共享的资源，并试图同时读和写这个资源，就处于相互竞争的状态。这种情况被称作竞争状态。要想解决竞争状态，对 个共享资源的读和写操作必须是原 化的。也就是说，同 时刻只能有 个线程对共享资源进 读和写操作。

临界区操作时，中断不应该开启。即使在单个处理器上，中断也可能导致并发：在允许中断时，内核代码可能在任何时候停下来，然后执 中断处理程序。假设 iderw 持有 idelock，然后中断发 ，开始运 ideintr。ideintr 会试图获得 idelock，但却发现 idelock 已经被获得了，于是就等着它被释放。这样， idelock 就永远不会被释放了，只有 iderw 能释放它，但又只有让 ideintr 返回

iderw 才能继续运 ，这样处理器、整个系统都会死锁。为了避免这种情况，当

中断处理程序会使 某个锁时，处理器就不能在允许中断发 时持有锁。xv6

做得更决绝：允许中断时不能持有任何锁。xv6 锁的具体实现步骤在代码阅读

部分已经阐明。

xchg 是原 操作，这是这条指令的最重要特性。交换指令 xchg 是两个寄存

器，寄存器和内存变量之间内容的交换指令。xchg()函数使 GCC 的内联汇编

语句，该函数中通过 xchg 原 性交换 spinlock.locked 和 newval，并返回

spinlock.locked 原来的值。当返回值为 1 时，说明其他线程占 了该锁，继续循

环等待；当返回值为 0 时，说明其他地 没有占 该锁，同时 locked 本设置成

* 了，所以该锁被此处占 。

1. 基于 XV6 的 spinlock，请给出实现信号量、读写锁、信号机制的设计 案(三选 ，请写出相应的伪代码)

这 选择实现信号量和信号机制的设计 案。

struct semaphore {

int value;

struct spinlock lock;

struct proc \*queue[NPROC]; // 进程等待队列,这是 个循环队列

int end; // 队尾

int start; // 队头

};

* 初始化信号量

void sem\_init(struct semaphore \*s, int value) {

s->value = value;

initlock(&s->lock, "semaphore\_lock");

end = start = 0;

}

void sem\_wait(struct semaphore \*s) {

acquire(&s->lock); // 竞争锁,如果竞争不到进 旋

s->value--;

if (s->value < 0) {

s->queue[s->end] = myproc(); // myproc()获取当前进程, 放 队尾

s->end = (s->end + 1) % NPROC; // 循环队列计算新的队尾

* 1. 释放锁(下 个 sem\_wait 的进程才能进 acquire),
* 2. 然后进 睡眠等待, 被唤醒时重新竞争锁

sleep(myproc(), &s->lock);

}

release(&s->lock);

}

void sem\_signal(struct semaphore \*s) {

acquire(&s->lock); // 竞争锁

s->value++;

if (s->value <= 0) {

wakeup(s->queue[s->start]); // 唤醒循环队列头的进程

s->queue[s->start] = 0;

s->start = (s->start + 1) % NPROC; // 重新计算队头

}

release(&s->lock);

}

* proc.h
* Per-process state

struct proc {

uint sz; // Size of process memory (bytes)

pde\_t\* pgdir; // Page table

char \*kstack; // Bottom of kernel stack for this process

enum procstate state; // Process state

volatile int pid; // Process ID

struct proc \*parent; // Parent process

struct trapframe \*tf; // Trap frame for current syscall

struct context \*context; // swtch() here to run process

void \*chan; // If non-zero, sleeping on chan

int killed; // If non-zero, have been killed

struct file \*ofile[NOFILE]; // Open files

struct inode \*cwd; // Current directory

char name[16]; // Process name (debugging)

};