操作系统第二次实例分析

第四组

CONTENTS

管道

01

操作系统：xv6

相关文件：pipe.c

管道(pipe)的数据结构

- 代码定义：

```
#define PIPESIZE 512
struct pipe {
    struct spinlock lock;     // 管道对应的自旋锁
    char data[PIPESIZE];  // 管道的数据缓冲区
    uint nread;      // 已读取的字节数
    uint nwrite;     // 已写入的字节数
    int readopen;    // 标志位：读端的文件是否打开
    int writeopen;   // 标志位：写端的文件是否打开
};
```

nread / nwrite超出缓冲区大小是否回滚

- 变量nread / nwrite记录了该管道自创建以来读取或写入的字节数，仅在创建管道时初始化为0
- 在读写管道时采用取余的方式来获取真实的缓冲区地址索引

  - 写入：

```
p->data[p->nwrite++ % PIPESIZE] = addr[i];
```

  - 读取：

```
addr[i] = p->data[p->nread++ % PIPESIZE];
```

  其中 `char *addr` 是用于保存要写入或已读出内容的缓冲区。

缓冲区判满和判空的条件

- 变量 nread 和 nwrite 分别记录了自该管道创建以来读取和写入的字节数量。
- 判定缓冲区满的条件：
  - 已写入的字节数 > 已读取字节数+管道缓冲区大小
  - `p->nwrite == p->nread + PIPESIZE`
- 判定缓冲区空的条件：
  - 已读取的字节数 == 已写入字节数
  - `p->nread == p->nwrite`
  - 在正常工作状态下，已读取的字节数永远小于或等于已写入的字节数。等于时意味着缓冲区空了。

为什么要使用nread和nwrite两个条件变量

- 读者和写者的阻塞原因不同：读者在缓冲区空时阻塞，写者在缓冲区满时阻塞
- 读者和写者需要在两个不同的条件变量上进行wait操作

- 若仅使用一个条件变量 cond，在多个读者和写者的情况下：
  - 读者A发现缓冲区空，执行 cond.signal 唤醒写者，然后执行 cond.wait 阻塞自身
  - 读者B接着被调度，发现缓冲区空，执行 cond.signal（此时A又被唤醒），然后执行cond.signal
  - 如果调度算法又调度了A（例如，读者A、B的优先级都高于写者），则重复上述操作，写者永远不会被调度，系统进入死锁状态

管道读写函数中能否用if替换while

- Pipewrite()

```
while(p->nwrite == p->nread + PIPESIZE){
      ……
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
```

- Piperead()

```
while(p->nread == p->nwrite && p->writeopen){
    ……
    sleep(&p->nread, &p->lock);
  }
  for(i = 0; i < n; i++){
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
```

- 假设每次调用piperead()或者pipewrite()至少要读或写一个字节
- While和if的区别在于每次睡眠后被唤醒时，是否还要检查缓冲区（满或者空）
- 一个读者写者：
  - 可以替换
  - 每次写者将缓冲区写满则进入休眠，只有当读者调用piperead才会唤醒写者（此时至少读取了一个字节，缓冲区非满）。读者同理。
- 多个读者写者：
  - 不能替换
  - 某个写者将缓冲区写满则进入休眠。当一个读者调用piperead时，将会唤醒队列中的所有写者（加入就绪队列）。由于有多个写者，某个写者可能将其再次写满，而之前被唤醒的写者必须要在写入前检查缓冲区是否已满，否则可能出错。

02

睡眠锁

操作系统：xv6

相关文件：sleeplock.h

sleeplock.c

Sleeplock的结构

```
struct sleeplock {
  uint locked;       // 是否被锁住
  struct spinlock lk; // 用于控制该sleeplock只能被互斥访问的自旋锁

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
```

Sleeplock的使用场景

- Sleeplock类似于研讨课上实现的互斥锁，当进程无法获得锁时就会被阻塞于该锁的睡眠队列，直到该锁被释放时才会被唤醒，而不会像spinlock一样反复测试锁（原地自旋）
- Sleeplock适用于获取锁后需要较长时间才会释放的情景，有利于防止CPU时间的大量浪费
- 在xv6中查找出的引用：
  - Buffer cache模块（bio.c）
  - 文件系统模块（fs.c）

Sleeplock的实现

- 自旋锁（spinlock）通过屏蔽中断和原子操作实现，从而避免竞争条件，保证互斥访问
  - 从获取自旋锁到释放自旋锁期间，中断都被屏蔽，也不允许抢占
  - 如果进程获取自旋锁后又要睡眠，则必须先释放该自旋锁（xv6中的sleep函数保证在睡眠前自动释放自旋锁，然后在唤醒时自动获取）
- 获取sleeplock的流程：
  - 先获取该锁中的自旋锁，以确保互斥（如果自旋锁被占用，显然会在acquire处反复测试等待）
  - 获取自旋锁后，进入临界区，检查locked变量确认该锁是否被锁住
  - 被锁住则释放对应的自旋锁并进入该锁（条件变量）的阻塞队列，若该锁被释放则会被唤醒并自动获取自旋锁
  - 若该锁可用则锁住该锁，然后设置相应的信息（pid、name等），最后释放对应的自旋锁离开临界区
- 释放sleeplock的流程：
  - 先获取对应的自旋锁并进入临界区
  - 释放锁
  - 唤醒该锁阻塞队列
  - 离开临界区，释放对应的自旋锁

为什么不需要屏蔽中断

- 在自旋锁（spinlock）中屏蔽中断，是为了防止获得锁的进程被抢占，中断处理程序或其他进程因为等待该锁而原地“自旋”，始终不会让出CPU，造成系统死锁
  - 获得自旋锁的进程在执行临界区代码时，不能因为任何原因放弃CPU
- 对于sleeplock：
  - 获得锁的进程A因为中断被抢占
  - 其他进程试图获取锁失败，进入阻塞队列让出CPU
  - A一定有机会再次获取CPU，从而运行临界区代码和释放锁
  - 系统不会死锁

Linux & pthread 读写锁

03

操作系统：Linux

相关文件：rwlock_types.h

spinlock_types.h

rwlock.h

rwsem.h

rwsem.c

Linux读写锁

- **由来：**读写文件时，读操作通常不冲突（可以多个进程一起读一个文件），而写操作冲突。如果单纯采用普通的互斥锁，效率较低
- **特性：**读写锁有解锁、读加锁和写加锁三种状态
  - 解锁状态：允许任何进程加锁
  - 读加锁状态：可以继续加读锁（所有试图对其加读锁的进程都能获取锁），不能加写锁（防止冲突）。试图加写锁的进程将被阻塞
  - 写加锁状态：不能加读锁或者写锁（试图加锁的都会被阻塞）
  - 资源被读加锁时，若有进程试图对其加写锁，除了阻塞该进程，通常还会阻塞之后请求加读锁的进程，防止资源持续被读加锁，而要写的进程持续阻塞
- **适用情形：**读操作的频率远大于写操作，此时用读写锁能够有效提高效率
- **总结：**读写锁是一种“共享-独占锁”，其基本特性是“读共享，写独占”

Pthread读写锁接口

- # include<pthread.h>
- 读写锁初始化：
  - `int pthread_rwlock_init(pthread_rwlock_t* rwlock, const pthread_rwlockattr_t* attr);`
  - 第一个参数为读写锁指针，第二个参数为读写锁属性指针。
  - 函数按读写锁属性对读写锁进行初始化。默认属性可传入NULL作为属性值。
- 加读锁：
  - `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);`
- 加写锁：
  - `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`
- 释放读写锁：
  - `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);`
- 销毁读写锁：
  - `int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);`

Thanks.

卜文添

2019/10/30