

同步 (1)

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组



复习：互斥

- 我们学习了互斥和互斥的实现
- `lock(&lk)` / `unlock(&lk)`
- 回到求素数个数的例子
 - 如果有多个线程计算素数，每个都可能`sum++`
 - 如何计算出1-n中包含的素数总数？
 - 系统中有若干线程(worker threads)实现计算
 - 如何等待这些线程计算结束？

同步

同步

- 指两个或两个以上随时间变化的量在变化过程中保持一定的相对关系
 - iTunes同步(手机 vs 电脑)
 - 变速箱同步器(合并快慢速齿轮)
 - 同步电机(转子与磁场速度一致)
 - 同步电路(由时钟驱动)
 - 线程同步(在某个时间点共同达到一致的状态)
- 异步 = 不同步
 - 上述很多例子都有异步版本(异步电机、异步电路、异步线程)

在素数计数中的同步

- 有若干线程
 - 线程t通过`done[t] = 1`;表示自己已经完成
 - 主线程逐个等待

```
for (int i = 0; i < n; i++) {  
    while (!done[i]) ;  
}
```

- 在`done[t] = 1`和while循环结束之间建立了顺序关系(完成了同步)
 - Cool! 我们终于能更好地利用多处理器资源为我们做事了!
 - wait.....还记得-O0 -O1 -O2的故事吗.....?

避免自旋

- while (!done[i]) 和自旋锁很类似
 - 忙等待某个条件发生，浪费处理器资源
- 为了更好地实现同步，就需要同步API
- 条件变量(Condition Variables)
 - “等待一件事发生”

条件变量API

- **wait(c)**
 - 等待c上的事件发生
- **signal/notify(c)**
 - 报告c上的事件发生
 - 如果有线程正在等待c，则唤醒其中一个线程
- **broadcast/notifyAll(c)**
 - 报告c上的事件发生
 - 唤醒全部正在等待c的线程

条件变量：例子

- main通过wait等待
- worker线程signal完毕

```
void worker_thread(int t) {  
    ...  
    cond_signal(&c[t]);  
}  
  
void main_thread() {  
    // create workers  
    for (int i = 0; i < n; i++) {  
        cond_wait(&c[i]);  
    }  
    printf("sum = %d\n", sum);  
}
```




条件变量：解决“先signal后wait”

- 把done加回来

```
void worker_thread(int t) {  
    ...  
    done[t] = 1;  
    cond_signal(&c[t]);  
}  
  
void main_thread() {  
    // create workers  
    for (int i = 0; i < n; i++) {  
        if (!done[t]) {  
            cond_wait(&c[i]);  
        }  
    }  
    printf("sum = %d\n", sum);  
}
```

条件变量：修复done的数据竞争

- 我们希望读取done[i]读取和cond_wait中间不被打断，因此使用互斥锁
- 思考题1: cond_wait带着锁睡眠，是否会导致问题？
- 思考题2: cond_signal是否需要上锁？

```
void worker_thread(int t) {  
    ...  
    lock(&mutex);  
    done[t] = 1;  
    cond_signal(&c[t]);  
    unlock(&mutex);  
}  
  
void main_thread() {  
    // create workers  
    for (int i = 0; i < n; i++) {  
        lock(&mutex);  
        if (!done[t]) {  
            cond_wait(&c[i]); // <-- ??  
        }  
        unlock(&mutex);  
    }  
    printf("sum = %d\n", sum);  
}
```

理解并发程序的执行

- 理解事件发生可能的先后顺序及它们的后果
- 在图中画出这些先后关系
- 仔细检查是否把事情做对

wait & signal原语

- wait(&cond, &mutex)
 - 释放mutex
 - 把当前线程和mutex放入等待队列中
 - 睡眠当前线程
- signal(&cond)
 - 否则，唤如果没有在cond上睡眠的线程，返回
 - 醒在cond上睡眠的线程(唤醒的线程先试图获取mutex)

多线程：打开了一扇门

- 有了同步和互斥，就能实现各种共享内存并发程序了
 - 多线程计算素数个数(主线程wait, 任务线程signal)
- 算法并行化：充分利用电脑中的多个共享内存的处理器
 - 笔记本：2C4T；台式机：4C8T；组里的服务器：24C/24T.....
 - 以往的算法一下就能快几十倍了，好心动.....
- 挑战题：如何并行化Dijkstra算法？

生产者-消费者问题



有若干个生产者/消费者

- 生产者(Producer)线程：不断产生entity
- 消费者(Consumer)线程：不断消费entity

```
Entity *produce();  
void consume(Entity *entity);
```

- 有一个**大小固定的缓冲区**存放entities
 - 如果缓冲区满，则不能再生产
 - 如果缓冲区空，则不能再消费

生产者/消费者问题的应用

- 大家将来遇到的很多实际系统都是生产者/消费者的模型(或变体)
 - 管道：写进程是生产者；读进程是消费者
 - 所以操作系统内核里就要正确处理生产者/消费者问题
 - Web/数据库服务器：多个线程在接收请求，同时有多个线程处理请求
 - JavaScript事件：多个生产者(鼠标、键盘、时钟、callback.....)，一个消费者(event loop thread)

生产者消费者问题：描述

- 问题1：实现线程安全的push和pop
 - 假设缓冲区大小为N
 - 使用循环队列实现
- 保证：
 - pop()执行时缓冲区非空
 - push()执行时缓冲区不会溢出

```
void push(Entity *);  
Entity *pop();  
  
void producer_thread() {  
    while (1) {  
        Entity *e = produce();  
          
        push(e);  
          
    }  
}  
  
void consumer_thread() {  
    while (1) {  
          
        Entity *e = pop();  
          
        consume(e);  
    }  
}
```

生产者消费者问题：另一种描述


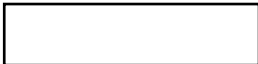
- 有两种线程：
 - `while (1) printf("(");`
 - `while (1) printf(")");`
- 加入适当的同步操作，使得生成的括号序列：
 - 一定是某个合法括号序列的前缀
 - 并且括号嵌套的深度不超过N
 - 例如 $N = 3$, `((()))((`合法, `(((((`不合法

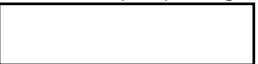
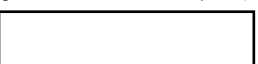


先不考虑缓冲区大小限制

- 用count表示左括号嵌套的深度
 - 即缓冲区中的数量
- Producer只需要count++
- Consumer
 - 当count == 0时需要等待
 - 怎样使用wait/signal?

```
void push(Entity *);  
Entity *pop();
```

```
void producer_thread() {  
    while (1) {  
          
        putchar('(');  
          
    }  
}
```

```
void consumer_thread() {  
    while (1) {  
          
        putchar(')');  
          
    }  
}
```



考虑缓冲区大小: $N = 1$

- 缓冲区只有一个单元
- 这个程序是否正确?

```
void producer_thread() {
    for (int i = 0; i < loops; i++) {
        lock(&mutex);
        if (count == 1)
            wait(&cond, &mutex);
        printf("("); count++;
        signal(&cond);
        unlock(&mutex);
    }
}

void consumer_thread() {
    for (int i = 0; i < loops; i++) {
        lock(&mutex);
        if (count == 0)
            wait(&cond, &mutex);
        printf(")"); count--;
        signal(cond);
        unlock(&mutex);
    }
}
```

Trick: 用两个条件变量

- empty
 - 可以填入一个(
 - 缓冲区有一个空位
- fill
 - 可以填入一个)
 - 缓冲区有一个数值
- 所以在count不满足要求时
 - producer等待empty
 - consumer等待fill

```
void producer_thread() {  
    for (int i = 0; i < loops; i++) {  
        lock(&mutex);  
        while (count == N)  
            wait(&empty, &mutex);  
        printf("("); count++;  
        signal(&fill);  
        unlock(&mutex);  
    }  
}  
  
void consumer_thread() {  
    for (int i = 0; i < loops; i++) {  
        lock(&mutex);  
        while (count == 0)  
            wait(&fill, &mutex);  
        printf(")"); count--;  
        signal(&empty);  
        unlock(&mutex);  
    }  
}
```