操作系统

Operating System

黄绵秋

Undergraduate, Fudan University

E-mail: hmqluther@gmail.com

目录

PART

I

同步

1 背景

1.1 并发问题

并发需要解决以下问题

- 进程间相互交流
- 资源的共享和竞争
- 多个进程之间活动同步
- 处理器时间分配

并发需要注意:

- 共享资源是危险的
- 很难以最佳方式管理资源的分配
- 难以定位程序错误所在

本章将主要讨论资源“竞争问题”——两个或更多进程在执行时访问相同资源，并且在这些进程之间不存在信息交换。这可能导致三种问题：(1)Mutual Exclusion(互斥);(2)Deadlock(死锁);(3)Starvation(饿死)。

2 临界区问题

2.1 临界资源定义

一次只能允许一个进程使用的资源称为**临界资源 (critical resources)**。每个进程中，访问临界资源的那段代码称为**临界区 (critical section)**。访问临界资源时必须互斥地进行，以维护数据的可靠性。可以把临界资源的访问过程分成 4 个部分:

(1) 进入区。为了进入临界区使用临界资源，在进入区检查可否进入临界区，若能进入临界区，则应设置正在访问临界区的标志，以阻止其他进程同时进入临界区。

(2) 临界区。进程中访问临界资源的那段代码。

(3) 退出区。将正在访问临界区的标志清除。

(4) 剩余区。代码的其他部分。

临界区问题指设计一个协议以便协作进程，解决方案应该满足如下三条要求:

- 互斥 (mutual exclusion)
  如果进程 P_i 在其临界区内执行，那么其他进程都不能在其临界区内执行
- 进步 (progress)
  如果没有进程在其临界区内执行，并且有进程需要进入临界区，那么只有那些不在剩余区内执行的进程可以参加选择，以便确定谁能下次进入临界区，而且这种选择不能无限推迟。

- 有限等待 (bounded waiting)
  从一个进程做出进入临界区的请求直到这个请求允许为止，其他进程允许进入临界区的次数有上限。

2.2 实现临界区互斥的方法

2.2.1 软件实现法

Dekker's Algorithm

算法一：单标志法 该算法设置一个公用整型变量 turn，用于指示被允许进入临界区的进程编号，即若`turn=0`，则允许 P_0 进程进入临界区。该算法可确保每次只允许一个进程进入临界区。但两个进程必须交替进入临界区。

```
P_0进程:                P_1进程:
while(turn != 0) ;      while(turn != 1) ;      // 进入区
critical section;       critical section;       // 临界区
turn = 1;               turn = 0;               // 退出区
remainder section;      remainder section;      // 剩余区
```

问题在于，若某个进程不再进入临界区，则另一个进程也将无法进入临界区，这样很容易造成资源利用不充分。

算法二：双标志先检查 该算法的基本思想是在每个进程访问临界区资源之前，先查看临界资源是否被正确访问。若正被访问，该进程需要等待；否则，进程才能进入自己的临界区。为此，设置数据 flag[i]，若第 i 个元素值为`false`，表示 P_i 进程未进入临界区。

```
P_i进程:                    P_j进程:
while(flag[j]) ; (1)        while(flag[i]) ; (2)
flag[i] = true;  (3)        flag[j] = true;  (4)
critical section;           critical section;
flag[i] = false;            flag[j] = false;
remainder section;          remainder section;
```

优点：不用交替进入，可连续使用。缺点：P_i 和 P_j 可能同时进入临界区，如按照 (1)(2)(3)(4) 执行时。

算法三：双标志后检查 与算法二相比先设置自己的标志再检测对方状态标志。

```
    P_i进程:                    P_j进程:
    flag[i] = true;             flag[j] = true;
    while(flag[j]) ;            while(flag[i]) ;
    critical section;           critical section;
    flag[i] = false;            flag[j] = false;
    remainder section;          remainder section;
```

相类似的，会产生 starvation(饥饿) 现象。

Peterson's Algorithm 为了防止两个进程为进入临界区而无限等待，又设置变量 turn，每个进程在先设置自己的标志后再设置 turn 标志。这时，再同时检测另一个进程状态标志和允许进入状态，以便保证两个进程同时要求进入临界区时，只允许一个进程进入临界区。

```
P_i进程:
flag[i] = true; turn = j;
while(flag[j] && turn == j) ;
critical section;
flag[i] = false;
remainder section;
```

本算法的基本思想是算法一和算法三的结合。利用 flag 解决临界资源的互斥访问，而利用 turn 解决“饥饿”现象。具体如下：
考虑进程 P_i，一旦设置 flag[i] = true，就表明它想要进入临界区，同时 turn = j，此时若进程 P_j 已在临界区，符合进程 P_i 中的 while 循环条件，则 P_i 不能进入临界区。若 P_j 不想进入临界区，即 flag[j] = false，循环条件不符合，则 P_j 可以进入。

2.2.2 硬件实现方法

基于软件的解决方案并不保证在现代计算机体系结构上正确工作。计算机提供了特殊的硬件指令，用于检测和修改字的内容，或用于原子地交换两个字（作为不可中断的指令）。通过硬件支持实现临界问题的方法称为低级方法或元方法。

中断屏蔽方法 禁止一切中断发生，或称之为屏蔽中断、关中断。因为 CPU 只在发生中断时引起进程切换，因此屏蔽中断能够保证当前运行的进程让临界区代码顺利地执行完，进而保证互斥的正确实现，然后执行开中断。

这种方法限制了处理机交替执行程序的能力，因此执行的效率会明显降低。然而多处理器的中断禁止会很耗时，因为消息要传递到所有处理器。消息传递会延迟进入临界区，并降低系统效率。对内核来说，在它执行更新变量或列表的几条指令期间，中断禁止是很方便的，但将中断禁止的权力交给用户则很不明智，若一个进程中断禁止后不再取消中断禁止，则系统可能因此终止。

硬件指令方法 `test_and_set()`为原子操作。其功能是读出指定标志后把该标志设置为真。可以通过声明一个`bool`变量`lock`并初始化为`false`，结构如下：

```
bool test_and_set(bool* target) {
    bool rv = *target;
    *target = true;
    return rv;
}
```

```
do {
    while (test_and_set(&lock)) ; // do nothing
    /* critical section */
    lock = false;
    /* remainder section */
} while(true)
```

`compare_and_swap()`声明全局`bool`变量`lock`，并初始化为 0，第一个调用`compare_and_swap()`的进程将`lock`设置为 1。然后它会进入它的临界区，随后调用的`compare_and_swap()`则不会成功。

```
int compare_and_swap(int* value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0) ; // do nothing
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```

硬件实现优缺点

优点

(1) 适用于任意数量进程，且无论单处理机还是多处理机

(2) 简单，容易验证正确性

(3) 可应用于多临界区情况

缺点

(1) busy-waiting 浪费处理器周期

(2) 可能存在 starvation

3 信号量定义与使用

3.1 信号量定义

信号量 (semaphore)S 是个 int 型变量，它只能通过三种方式进行操作:

- initialize
  信号量 S 一般初始化为大于 0 的正整数

```
struct semaphore {
    int count;
    queueType queue;
}
```

- wait wait(S) 操作会减少信号量 S，如果 S.count≤ 0，则会将 S 阻塞 (block)

```
void wait(semaphore S) {
    S.count--;
    if (S.count < 0) {
        place this process in S.queue;
        block this process;
    }
}
```

- signal signal(S) 操作会增加信号量 S，如果 S.count= 0，则会解除 S 的阻塞

```
void signal(semaphore S) {
    S.count++;
    if (S.count <= 0) {
        remove a process P from S.queue;
        place process P on ready list;
    }
}
```

当 S.count≥ 0 时，S.count 为不需要阻塞即可直接执行的最多进程数。
当 S.count< 0 时，S.count 为被阻塞在 S.queue 中的进程数量。

3.2 强信号量和弱信号量

对于普通信号量和二进制信号量而言，会用一个队列来管理等待信号量的进程。而按照其中根据何种顺序来确定进程的去除顺序，课分为强信号量和弱信号量。

- 强信号量 (Strong Semaphore)：被阻塞最久的进程最先从队列中释放。
- 弱信号量 (Weak Semaphore)：不另外规定进程从队列中被释放的顺序
  可能出现 starvation

4 经典同步问题

4.1 生产者-消费者问题

问题描述：一组生产者进程和一组消费者进程共享一个初始为空、大小为 n 的缓冲区，只有缓冲区没满时，生产者才能把消息放入缓冲区，否则必须等待；只有缓冲区不空时，消费者才能从中取出消息，否则必须等待。由于缓冲区是临界资源，它只允许一个生产者放入消息，或一个消费者从中取出消息。

```
semaphore mutex = 1;                    // 临界区互斥信号量
semaphore empty = n;                    // 空闲缓冲区
semaphore full = 0;                     // 缓冲区初始化为空
producer() {
    while(1) {
        /* produce an item in next_produced */
        wait(empty);                    // 获取空缓冲区单元
        wait(mutex);                    // 进入临界区

        /* add next_produced to the buffer */
        signal(mutex);                  // 离开临界区，释放互斥信号量
        signal(full);                   // 满缓冲区数+1
    }
}
comsumer() {
    while(1) {
        wait(full);                     // 获取满缓冲区单元
        wait(mutex);                    // 进入临界区
        /* remove an item frm buffer to next_consumed */
        signal(mutex);                  // 离开临界区，释放互斥信号量
        signal(empty);                  // 空缓冲区数量+1
        /* consume the item in next_consumed   */
    }
}
```

4.2 读者-写者问题

问题描述：有读者和写者两组并发进程，共享一个文件，当两个或以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：(1) 允许多个读者可以同时对文件执行读操作；(2) 只允许一个写者往文件中写信息；(3) 任一写者在完成写操作之前不允许其他读者或写者工作；(4) 写者执行写操作前，应让已有的读者和写者全部退出。

```
semaphore rw_mutex = 1;                 // 用于保证读者和写者互斥地访问文件
semaphore mutex = 1;                    // 用于保护更新read_count时的互斥
int read_count = 0;                     // 用于记录当前的读者数量
writer() {
    while(1) {
        wait(rw_mutex);                 // 互斥访问共享文件
```

```
        /* writing is performed */
        signal(rw_mutex);                    // 释放共享文件
    }
}
reader() {
    while (1) {
        wait(mutex);                         // 互斥访问read_count变量
        read_count++;
        if (read_count == 1)
            wait(rw_mutex);                  // 阻止写进程写
        signal(mutex);                       // 释放互斥变量read_count
        /* reading is performed */
        wait(mutex);
        read_count--;
        if (read_count == 0)
            signal(rw_mutex);                // 允许写进程写
        signal(mutex);
    }
}
```

类似的为**独木桥问题**，如下

例子: 一座桥最多容纳 4 辆车，车可以从西往东也可以从东往西开，同一时间桥上只能有一个方向的车

```
int carcount = 4, status = 1; // carcount为车的容量，不需要等待时status为1
int wemutex = 1, ewmutex = 1; // 两个方向的互斥锁
int wecount = 0, ewcount = 0; // 分别表示两个方向上等待的车的数量
void west_to_east() {
    wait(wemutex);   // 修改wecount值
    wecount++;
    if (wecount == 1) wait(status); // 尝试让等待的车上桥
    signal(wemutex);
    wait(carcount);  // 等待桥上有位置，直到桥上的车数量少于4
    drive_throught();
    signal(carcount); // 离开桥，重新
    wait(wemutex);
    wecount--;
    if (wecount == 0) signal(status);
    signal(wecount);
}
void east_to_west() {
    wait(ewmutex);
    ewcount++;
    if (ewcount == 1) wait(status);
    signal(ewmutex);
    wait(carcount);
    drive_throught();
    signal(carcount);
    wait(ewmutex);
    ewcount--;
    if (ewcount == 0) signal(status);
    signal(ewcount);
}
```

4.3 哲学家进餐问题

5 管程

管程内只有一个进程正在执行。