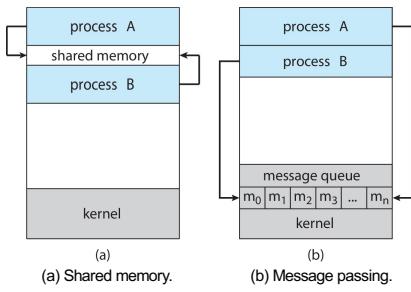


3.1.2 进程间通信(2)



进程间通信模型

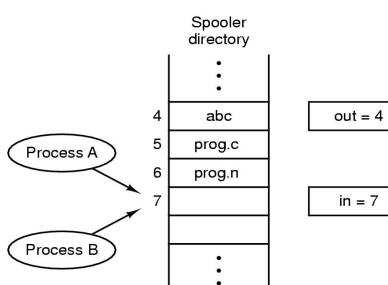
7

3.2 进程同步

- 3.2.1 临界资源与临界区
- 3.2.2 Peterson 算法
- 3.2.3 硬件同步
- 3.2.4 互斥锁
- 3.2.5 信号量
- 3.2.6 经典进程同步问题
- 3.2.7 管程
- 3.2.8 同步实例

8

3.2.1 临界资源与临界区(1)



9

3.2.1 临界资源与临界区(2)

- 临界资源
 - 一次仅允许一个进程使用的资源
- 临界区
 - 每个进程访问临界资源时必须互斥执行的程序
- 使用临界资源需遵循的原则
 - 互斥使用：不能同时有两个进程在临界区内执行
 - 有空让进：临界资源空闲时，应允许一个请求临界资源的进程进入临界区

10

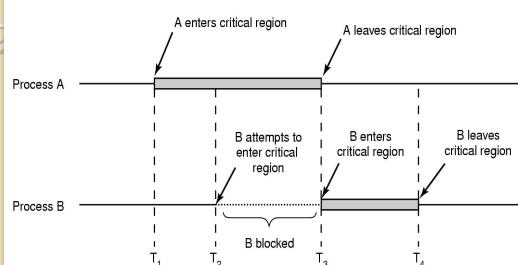
3.2.1 临界资源与临界区(3)

- 有限等待：不能使进入临界区的进程无限期地等待在临界区之外
- 让权等待：等待进入临界区的进程应释放处理器后阻塞等待

➤ 进程Pi的通用结构

```
do {  
    entry section  
    critical section  
    exit section  
    reminder section  
} while (1);
```

3.2.1 临界资源与临界区(4)



进程A和进程B互斥使用临界区

12

3.2.2 Peterson算法(1)

- 解决进程互斥进入临界区的软件方法
- 适用于两个进程交替执行临界区与剩余区
- 2个进程共享2个数据项
 - int turn;
 - 表示哪个进程可以进入临界区
 - boolean flag[2];
 - 表示哪个进程准备进入临界区

13

3.2.2 Peterson算法(2)

```
// 进程Pi  
void Pi(){  
    while(TRUE){  
        flag[i]=TRUE;  
        turn=j;  
        while(flag[j]&&(turn==j));  
        critical section  
        flag[i]=FALSE;  
        reminder section  
    }  
}  
  
// 进程Pj  
void Pj(){  
    while(TRUE){  
        flag[j]=TRUE;  
        turn=i;  
        while(flag[i]&&(turn==i));  
        critical section  
        flag[j]=FALSE;  
        reminder section  
    }  
}
```

注：退出while循环的条件是，要么另一个进程/线程不想使用临界区，要么此进程/线程拥有访问权限

14

3.2.2 Peterson算法(3)

➤ Limitation

- Peterson's solution is not guaranteed to work on modern computer architectures for the primary reason that, to improve system performance, processors and/or compilers may **reorder** read and write operations that have no dependencies.

15

3.2.2 Peterson算法(4)

➤ Example: (Reorder)

```
boolean flag=false;  
int x=0;
```

Thread1: `while (!flag);
print x;`

Thread2: `x=100;
flag=true;`

- Is the value of x outputted by Thread1 100?

16

3.2.2 Peterson算法(5)

➤ How does this affect Peterson's solution?

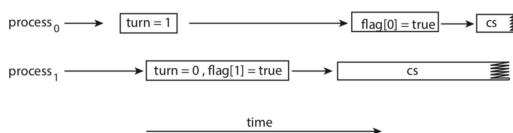


Figure 6.4 The effects of instruction reordering in Peterson's solution.

17

3.2.3 硬件同步(1)

➤ 解决进程互斥进入临界区的硬件方法

- 关中断
- Memory Barriers
- Hardware Instructions
 - Test-and-Set instruction
 - Compare-and-Swap instruction
- Atomic Variables

18

3.2.3 硬件同步(2)

➤ 关中断

- 进程刚进入临界区后立即禁止所有中断；进程要离开之前打开所有中断
- 原理：**CPU只有在发生时钟中断或其它中断时才会进行进程切换**
- 实现

```
关中断(disable)  
critical section  
开中断(enable)
```

19

3.2.3 硬件同步(3)

• 优点

- 简单

• 缺点

- 将禁止中断的权力交给用户进程是不明智的
- 在多处理器系统中，禁止中断仅对执行本指令的CPU有效，其他CPU仍将继续运行，并可访问共享资源

20

3.2.3 硬件同步(4)

➤ Memory Barriers

- How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**.
- **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.
- **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.

21

3.2.3 硬件同步(5)

- Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor.
- To address this issue, computer architectures provide **instructions** that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors. Such instructions are known as **memory barriers** or **memory fences**.

22

3.2.3 硬件同步(6)

➤ Example: (memory barriers)

```
boolean flag=false;  
int x=0;
```

```
Thread1: while ( !flag )  
          memory barrier0;  
          print x;
```

```
Thread2: x=100;  
          flag=true;
```

- The value of x outputted by Thread1 is 100.

23

3.2.3 硬件同步(7)

- The important characteristic of test and set() instruction is that it is executed **atomically**.

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

Figure 6.5 The definition of the atomic test_and_set() instruction.

24

3.2.3 硬件同步(8)

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

Figure 6.6 Mutual-exclusion implementation with test_and_set().

25

3.2.3 硬件同步(9)

- The **compare and swap() instruction (CAS)**, just like the test and set() instruction, operates on two words atomically, but uses a different mechanism that is based on swapping the content of two words.

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

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

    return temp;
}
```

Figure 6.7 The definition of the atomic compare_and_swap() instruction.

26

3.2.3 硬件同步(10)

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

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

Figure 6.8 Mutual exclusion with the compare_and_swap() instruction.

27

3.2.4 互斥锁(1)

- 基于硬件的方法复杂且程序员无法访问
- 使用更高级别的软件工具——互斥锁

• 原理

- 为临界资源设置锁位变量W
- W=0, 资源空闲可用
- W=1, 资源已被占用
- 初始化及退出临界区时, 将W置为0
- 进入临界区时, 将W置为1

k: if(W==1) goto k;
else W=1;

忙等待

28

3.2.4 互斥锁(2)

- 实现: 多个进程利用互斥锁实现互斥

```
const int n=3;          int w=0;
void P(int i){
    while(1){  TestSet(w); //加锁
                Critical Section
                w=0;           //开锁
                Remainder Section
            }
    int main() {      w=0;
                    Parbegin  P(1); P(2); P(3); Paren
    }
```

29

3.2.4 互斥锁(3)

• 自旋锁

- 利用CAS实现

• 优点

- 没有上下文切换

- 在现代多核计算系统上, 自旋锁被广泛用于许多操作系统中

```
代码片段 8-7 自旋锁
1 void lock_init(int *lock)
2 {
3     // 初始化自旋锁
4     *lock = 0;
5 }
6 void lock(int *lock)
7 {
8     while(atomic_CAS(lock, 0, 1) != 0)
9         ; // 循环忙等
10 }
11 void unlock(int *lock)
12 {
13     *lock = 0;
14 }
15
16 }
```

30

3.2.5 信号量(1)

➤ 信号量

- 进程同步工具
- 1965年由Dijkstra（荷兰）提出
- 物理意义
 - 表示资源的实体
- 结构

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

31

3.2.5 信号量(2)

- value: 表示该类资源的当前可用数量
- L: 等待使用该类资源的阻塞进程队列的队首指针
- 操作
 - 初始化: 将value初始化为该类资源的可用数量
 - 原子操作P: 申请资源
 - 原子操作V: 释放资源
 - 信号量value为负数时, 其绝对值表示在该信号量上等待的进程数目

32

3.2.5 信号量(3)

```
P(S):           //wait(S); down(S)
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block;
    }
V(S):           //signal(S); up(S)
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
```

33

3.2.5 信号量(4)

- 利用信号量实现n个进程间的互斥
 - 互斥信号量mutex, 初值为1
 - 第i个进程的执行代码

```
do {
    P(mutex)
    <critical section>
    V(mutex)
    <reminder section>
} while (1);
```

34

3.2.5 信号量(5)

➤ 利用信号量实现进程间的同步

- 示例: 利用信号量实现计算进程与打印进程之间的同步过程。假定计算进程和打印进程共同使用一个单缓冲
- 信号量
 - 计算进程: 信号量empty, 表示缓冲区是否空, 初值为1;
 - 打印进程: 信号量full, 表示缓冲区中是否有可供打印的计算结果, 初始值为0

35

3.2.5 信号量(6)

```
empty=1;full=0;
parbegin
begin      //计算进程
loop
compute next number
P(empty)
add the number to buf
V(full)
.....
end loop
end
parend
begin      //打印进程
loop
P(full)
print next number
V(empty)
add the empty to buf
.....
end loop
end
```

36

3.2.5 信号量(7)

➤ 信号量分类

- 公用信号量

- + 互斥信号量，用于解决进程之间互斥进入临界区

- 私用信号量

- + 同步信号量，用于解决异步环境下进程之间的同步

37

3.2.6 经典进程同步问题(1)

➤ 生产者—消费者问题

- 是相互合作进程关系的一种抽象
- 生产者：当进程释放一个资源时，可把它看成是该资源的生产者
- 消费者：当进程申请使用一个资源时，可把它看成该资源的消费者
- + 计算进程：打印数据的生产者；空缓冲的消费者
- + 打印进程：打印数据的消费者；空缓冲的生产者

38

3.2.6 经典进程同步问题(2)

• 问题描述：

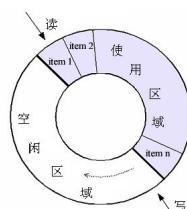
- 假定有一组生产者(M个)和一组消费者(N个)进程，通过一个有界环形缓冲区(k个缓冲块)发生联系。生产者将生产的产品放入缓冲区，消费者从缓冲区取用产品。
- 当缓冲区满时，生产者要等消费者取走产品后才能向缓冲区放下一个产品；当缓冲区空时，消费者要等生产者放一个产品入缓冲区后才能从缓冲区取下一个产品。

39

3.2.6 经典进程同步问题(3)

• 信号量

- + empty：表示空缓冲块的个数，初值为k
- + full：有数据的缓冲块个数，初值为0
- + mutex：互斥访问临界区的信号量，初值为1



40

3.2.6 经典进程同步问题(4)

```
#define N 100
#define int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        /* TRUE is the constant 1 */
        /* generate something to put in buffer */
        /* decrement empty count */
        /* enter critical region */
        /* put new item in buffer */
        /* leave critical region */
        /* increment count of full slots */
        down(&empty);
        up(&mutex);
        up(&full);
        up(&mutex);
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        /* infinite loop */
        /* decrement full count */
        /* enter critical region */
        /* take item from buffer */
        /* leave critical region */
        /* increment count of empty slots */
        /* do something with the item */
        up(&empty);
        consume_item(item);
    }
}
```

41

3.2.6 经典进程同步问题(5)

➤ 读者—写者问题

- 有一个多进程共享的数据区（可以是一个文件或者主存的一块空间），有一些只读取这个数据区的进程（reader）和一些只往数据区中写数据的进程（writer）：
 - 任意多的读进程可以同时读数据区
 - 一次只有一个写进程可以写数据区
 - 若有写进程正在写，禁止任何进程读

42

3.2.6 经典进程同步问题(6)

➤ 信号量

- 读写互斥信号量db：实现读写互斥和写写互斥访问共享文件，初值为1
- 计数器变量rc：记录同时读的读者数，初值为0
- 读计数互斥信号量mutex：使读者互斥地访问共享变量rc，初值为1

43

3.2.6 经典进程同步问题(7)

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

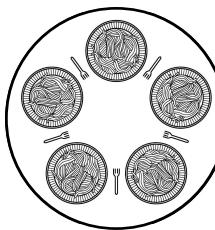
void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
```

44

3.2.6 经典进程同步问题(8)

➤ 哲学家就餐问题

- 假设有5个哲学家，花费一生的时光思考和吃饭。在桌子上放着5把叉子。一个哲学家要分两次去取其左边和右边的叉子。若得到两把叉子，就开始吃饭；吃完放下两把叉子。



45

3.2.6 经典进程同步问题(9)

```
fork: ARRAY[0-4] OF semaphore;
mutex: semaphore;
fork[0]:=fork[1]:=fork[2]:=fork[3]:=fork[4]:=l;
mutex:=1;
parbegin
P; REPEAT           /*第1个哲学家的活动情况*/
    Think FOR while; /*思考一会儿，想吃饭*/
    P(mutex);          /*申请拿叉子*/
    P(fork[i]);
    P(fork[(i+1) MOD 5]);
    V(mutex);          /*释放申请权*/
    Eat FOR WHILE;   /*吃饭*/
    V(fork[i]);
    V(fork[(i+1)MOD5]);
UNTIL false
parend
```

46

3.2.7 管程(1)

➤ 引入管程的原因

- 各个进程自备P(S)和V(S)操作，加重了用户负担
 - 大量同步操作分散在各个进程中，系统管理复杂
 - 易产生死锁
- 提出
- 1974年由Hansen和Hoare提出
- 管程的定义

47

3.2.7 管程(2)

- 一个管程调用了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据
- 管程是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块

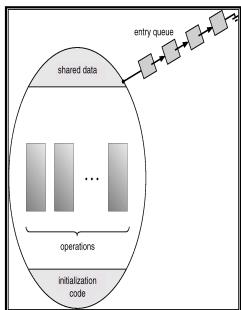
- 一次只有一个进程能在管程内活动。从而提供互斥机制，保证管程数据的一致性

48

3.2.7 管程(3)

➤ 管程的结构

- 管程名
- 局部于管程的共享变量说明
- 对该数据结构进行操作的一组过程
- 对局部于过程的数据设置初始值的语言



49

3.2.7 管程(4)

```
monitor monitor-name {  
    shared variable declarations  
    procedure body P1 (...) {  
        ...  
    }  
    procedure body P2 (...) { ...  
    }  
    procedure body Pn (...) { ...  
    }  
    {  
        initialization code  
    }  
}
```

50

3.2.7 管程(5)

➤ 管程的特点

- 局部于管程的数据结构只能被局部于管程的过程所访问，任何管程外部的过程都不能访问它
- 局部于管程的过程只能服务管程内的数据结构
- 管程每次只允许一个进程进入管程，从而实现管程的互斥

➤ 管程的同步机制

- 条件变量C

51

3.2.7 管程(6)

- 同步原语wait(c)：执行wait(c)的进程将自己阻塞在条件变量C的相应等待队列中。在阻塞前，释放管程的互斥使用权；
- 同步原语signal(c)：执行signal(c)的进程检查条件变量C的相应等待队列。如果队列为空，则执行此操作的进程继续；否则，唤醒C队列中的第一个等待者，让被唤醒者进入该管程。

52

3.2.7 管程(7)

➤ 利用管程实现临界资源的互斥使用

```
monitor mutexshow {  
    boolean busy; //临界资源状态  
    condition nonbusy; //条件变量  
    define request,release; //进程可调用过程  
    use wait,signal; //同步操作  
    Procedure request { //申请资源  
        if busy then wait(nonbusy);  
        busy=true;  
    }  
    Procedure release{ //释放资源  
        busy=false;  
        signal(nonbusy);  
    }  
    { busy=false; } //初始：资源空闲  
}
```

53

3.2.7 管程(8)

➤ 利用管程解决生产者—消费者问题

```
monitor ProducerConsumer  
    condition full,empty;  
    integer count;  
    procedure insert(item: integer);  
    begin  
        if count = N then wait(full);  
        insert_item(item);  
        count := count + 1;  
        if count = 1 then signal(empty)  
    end;  
    function remove: integer;  
    begin  
        if count = 0 then wait(empty);  
        remove = remove_item();  
        count := count - 1;  
        if count = N - 1 then signal(full)  
    end;  
    count := 0;  
end monitor;
```

```
procedure producer;  
begin  
    while true do  
        begin  
            item = produce_item;  
            ProducerConsumer.insert(item)  
        end  
end;  
procedure consumer;  
begin  
    while true do  
        begin  
            item = ProducerConsumer.remove;  
            consume_item(item)  
        end  
end;
```

54

3.2.8 同步实例(1)

➤ Linux内核同步

- 互斥访问内核共享数据结构
- 内核使用的同步技术
 - 每CPU变量
 - 把内核变量声明为每CPU变量，一个CPU不应该访问与其他CPU对应的变量
 - 为来自不同CPU的并发访问提供保护
 - 问题：异步函数、内核抢占

55

3.2.8 同步实例(2)

• 原子操作

- 优化和内存屏障

• 自旋锁

- 用于多处理机环境的一种特殊锁
- 当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。
- 读—复制—更新
- 信号量
- 禁止本地中断
- 禁止和激活可延迟函数

56

3.2.8 同步实例(3)

➤ Windows内核同步

- 在内核执行的不同阶段，必须保证每次有且仅有一个处理器在临界区执行
- 内核临界区就是修改全局数据结构的代码段
- 内核引入自旋锁实现多处理机互斥访问内核临界区
- 在Intel处理器上，自旋锁是通过“测试与设置”TestSet硬件指令实现的
- 拥有自旋锁的线程不会被剥夺处理器，其独占处理器执行

57

3.2.8 同步实例(4)

➤ Windows执行体同步

• 自旋锁的限制

- 被保护的资源必须被快速访问，并且不与其他代码进行复杂的交互
- 临界区代码不能换出内存，不能引用可分页数据，不能调用外部程序（包括系统服务），不能产生中断或异常
- 用户态同步机制
- 调度程序对象
- 内核以内核对象形式给执行体提供的用户态同步机制

58

3.2.8 同步实例(5)

- 进程、线程、事件、信号量、互斥体、可等待的定时器、I/O完成端口或文件等同步对象
- 每个同步对象有两种状态：“有信号”，“无信号”
- Win32应用程序中的一个线程可以等待一个或多个同步对象变为有信号状态实现同步
- 线程、进程终止时有信号
- Windows定义了统一的同步机制，等待调度程序对象为有信号状态
- WaitForSingleObject()

59

3.2.8 同步实例(6)

➤ Windows线程同步机制

• 事件对象

- 相当于一个“触发器”，用于通知线程某个事件是否出现，包括有信号和无信号两个状态

• 互斥体对象

- 互斥访问共享资源

• 信号量对象

- 就是资源信号量，初始值可在0到指定最大值之间设置

60