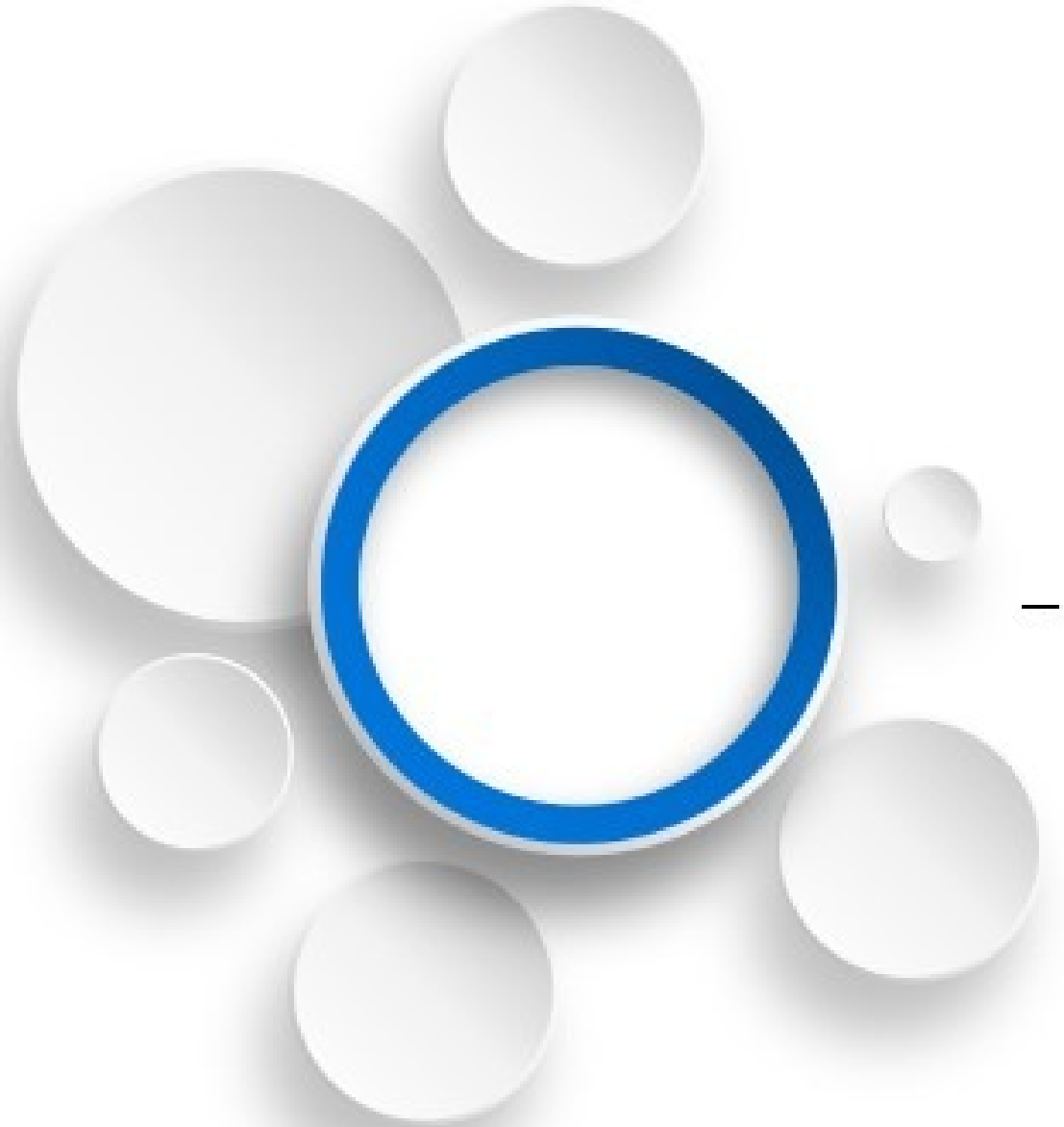


互斥和同步(2)

大纲

01 管程

02 消息传递

03 经典IPC问题

04 其他IPC机制

01

管程

信号量的缺点

- 用信号量可实现进程间的同步和互斥，但由于信号量的控制分布在整个程序中，其正确性分析很困难
  - 同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁(如P、V操作的次序错误、重复或遗漏)
  - 易读性差：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统中并发执行的各个程序
  - 不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局
  - 正确性难以保证：操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误

管程的引入

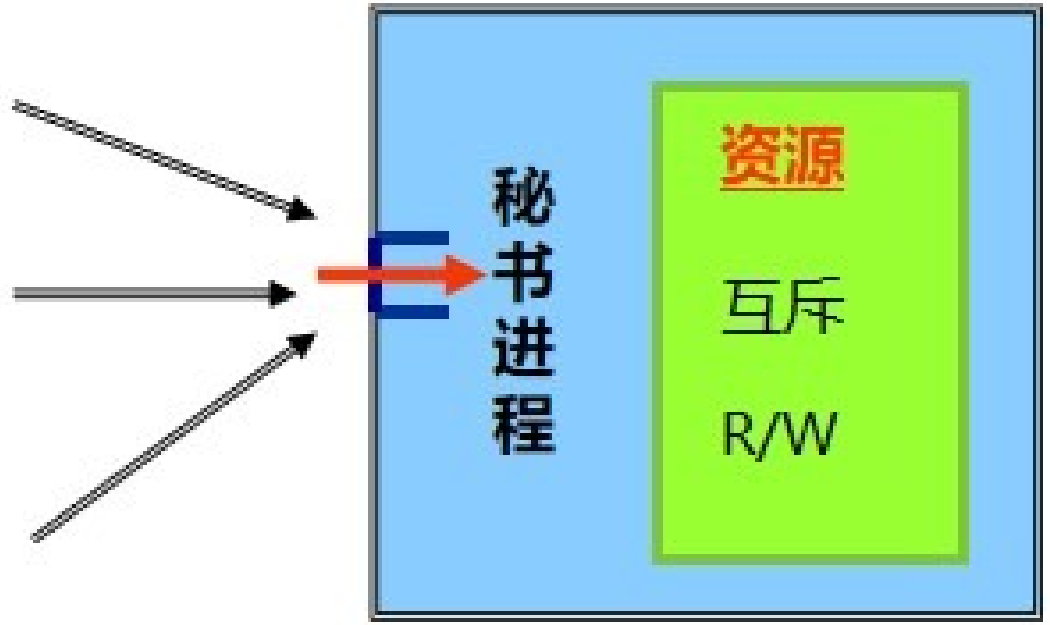
- 1971年，Dijkstra提出“秘书”进程的思想
- “秘书”每次仅让一个进程来访，这样既便于对共享资源的管理，又实现了互斥访问

进程1请求使用共享资源

进程2请求使用共享资源

.....

进程n请求使用共享资源



管程的引入

- 1973年，Hansen和Hoare把Dijkstra的“秘书”进程的思想推广为“管程”
- 管程的基本思想是把信号量及其操作原语封装在一个对象内部。即：将共享变量以及对共享变量能够进行的所有操作集中在一个模块中
- 管程的定义：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
- 引入管程可提高代码的可读性，便于修改和维护，正确性易于保证

管程的主要特性

- 模块化：一个管程是一个基本程序单位，可以单独编译
- 抽象数据类型：管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码
- 信息封装：进程可调用管程中实现的某些过程(函数)，至于这些过程是怎样实现的，在其外部则是不可见的
- 管程中的共享变量在管程外部是不可见的，外部只能通过调用管程中所说明的过程(函数)来间接地访问管程中的共享变量

管程的格式

- 类Pascal语言描述的管程

管程名：example
共享变量：i
条件变量：c
管程过程：producer()、comsumer()

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

管程互斥进入

- 管程是编程语言的组成部分，编译器采用与其他过程调用不同的方法处理对管程的调用
- 典型的处理方法：当一个进程调用管程过程时，该过程将检查在管程中是否有其他活跃进程。如果有，调用进程将被阻塞，直到另一个进程离开管程而将其唤醒；如果没有，则该调用进程可以进入
- 因为管程是互斥进入的，所以当一个进程试图进入一个已被占用的管程时它应当在管程的入口处等待，因而在管程的入口处应当有一个进程等待队列，称作入口等待队列

资源等待队列

- 管程是用于管理资源的，当进入管程的进程因资源被占用等原因不能继续运行时应当释放管程的互斥权，即将等待资源的进程加入资源等待队列
- 资源等待队列可以由多个，每种资源一个队列
- 资源等待队列由条件变量维护
- 条件变量(condition variables)是管程内的一种数据结构，且只有在管程中才能被访问，它对管程内的所有过程是全局的，只能通过wait和signal两个原语操作来控制

条件变量

- wait(c)：调用的进程阻塞，并移入与条件变量c相关的等待队列中，并释放管程，直到另一个进程在该条件变量c上执行signal( )唤醒等待进程并将其移出条件变量c队列
- signal(c)：如果c链为空，则相当于空操作，执行此操作的进程继续；否则唤醒c链中的第一个等待者

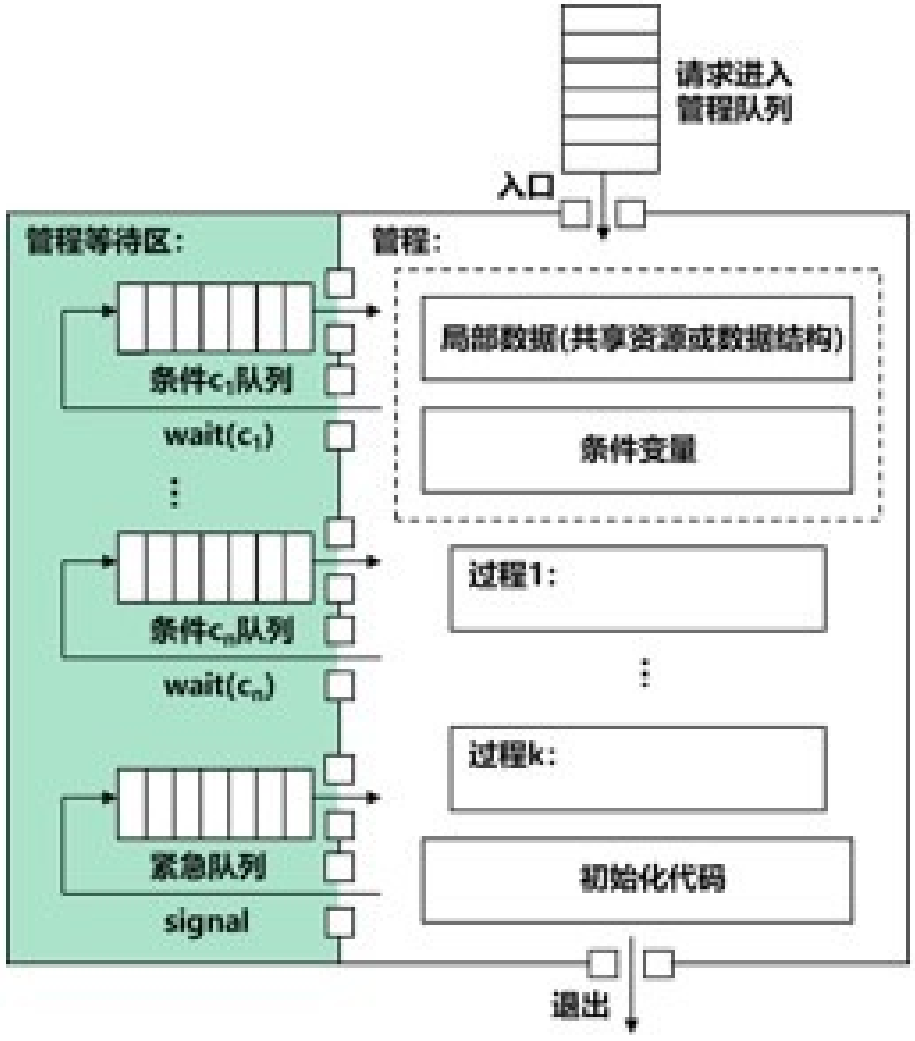
紧急等待队列

- 当一个进入管程的进程执行唤醒操作时(如P唤醒Q)，管程中便存在两个同时处于活动状态的进程
- 处理方法有两种：
  - P等待，Q继续，直到Q退出或等待(Hoare)
  - 规定唤醒为管程中最后一个可执行的操作(Hansen)

紧急等待队列

- 假设进程P唤醒进程Q，则P等待Q继续。如果进程Q在执行时又唤醒进程R，则Q等待R继续，……，于是，在管程内部，由于执行唤醒操作，可能会出现多个等待进程，因而还需要有一个进程等待队列，这个等待队列被称为紧急等待队列

管程的结构


请求进入
管程队列
入口
管程等待区:
管程:
局部数据(共享资源或数据结构)
条件变量
条件c_1队列
wait(c_1)
过程1:
条件c_n队列
wait(c_n)
过程k:
紧急队列
初始化代码
signal
退出

管程的缺点

- 管程是一个编程语言概念，编译器必须要识别管程并用某种方式对其互斥做出安排
- Java、Pascal 、 Modula-2等语言支持管程
- C、C++及多数程序设计语言都不支持管程
- C、C++及多数程序设计语言也没有信号量，但是支持信号量十分容易——编译器甚至不需知道它们的存在，信号量是OS机制，系统只需提供相应系统调用即可

用管程实现生产者-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;
```

02

消息传递

消息传递

- 信号量和管程都是设计用来解决访问公共内存的一个或多个CPU上的互斥问题的
- 对于分布式系统(每个CPU都有自己的私有内存)，这些机制将失效
- 对于不同机器间的进程通信，必须引入其他方法——例如消息传递(message passing)等高级通信机制
  - 套接字（socket）
  - 远程过程调用和远程方法调用

- 消息传递原语

  send(destination, &message)

  receive(source, &message)

- 它们是系统调用而不是语言成分

用消息传递实现生产者-消费者问题

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

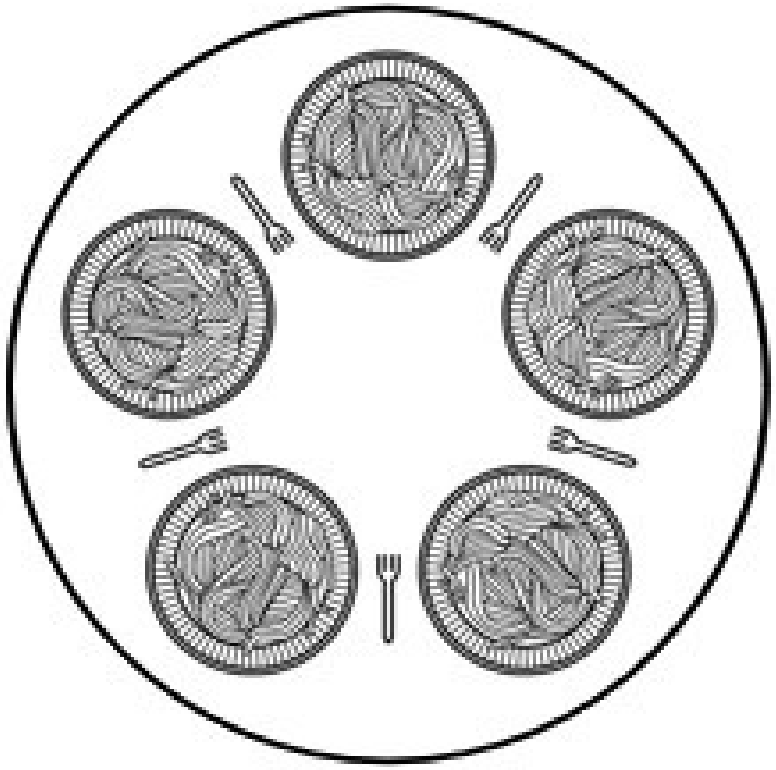
消息传递系统的设计问题

1. 不可靠的消息传递中的成功通信问题
2. 进程命名问题
3. 身份认证问题
4. 性能问题

03

经典IPC问题

- 哲学家进餐问题(the dining philosophers problem) 由Dijkstra首先提出并解决
- 5个哲学家围绕一张圆桌而坐，桌子上放着5把叉子，每两个哲学家之间放一支；哲学家的动作包括思考和进餐，进餐时需要同时拿起他左边和右边的两把叉子，思考时则同时将两把叉子放回原处。如何保证哲学家们的动作有序进行?



一种错误算法

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

正确算法

```
#define N                5                    /* number of philosophers */
#define LEFT             (i+N-1)%N            /* number of i's left neighbor */
#define RIGHT            (i+1)%N              /* number of i's right neighbor */
#define THINKING         0                    /* philosopher is thinking */
#define HUNGRY           1                    /* philosopher is trying to get forks */
#define EATING           2                    /* philosopher is eating */
typedef int semaphore;                        /* semaphores are a special kind of int */
int state[N];                                 /* array to keep track of everyone's state */
semaphore mutex = 1;                          /* mutual exclusion for critical regions */
semaphore s[N];                               /* one semaphore per philosopher */

void philosopher(int i)                       /* i: philosopher number, from 0 to N-1 */
{
     while (TRUE) {                           /* repeat forever */
          think( );                           /* philosopher is thinking */
          take_forks(i);                      /* acquire two forks or block */
          eat( );                             /* yum-yum, spaghetti */
          put_forks(i);                       /* put both forks back on table */
     }
}
```

正确算法(续)

```
void take_forks(int i)                    /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                     /* enter critical region */
        state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
        test(i);                          /* try to acquire 2 forks */
        up(&mutex);                       /* exit critical region */
        down(&s[i]);                      /* block if forks were not acquired */
}
void put_forks(i)                         /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                     /* enter critical region */
        state[i] = THINKING;              /* philosopher has finished eating */
        test(LEFT);                       /* see if left neighbor can now eat */
        test(RIGHT);                      /* see if right neighbor can now eat */
        up(&mutex);                       /* exit critical region */
}

void test(i)                              /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

- 读者-写者问题(the readers-writers problem)
- 有两组并发进程：读者和写者，共享一组数据区，要求：
  - 允许多个读者同时执行读操作
  - 不允许读者、写者同时操作
  - 不允许多个写者同时操作

读者-写者问题

```
int rc = 0; // 正在读或想要读的读者数量
semaphore mutex = 1;     // 访问rc的信号量
semaphore db = 1;    // 访问数据库的信号量

void reader() {
    wait(&mutex); // 要访问rc，加锁
    rc = rc + 1; // 读者数量+1
    if(rc == 1) wait(&db); // 首个读者来了，给数据库加锁
    signal(&mutex); // rc访问结束，解锁
    read_database(); // 读数据库
    wait(&mutex); // 要访问rc，加锁
    rc = rc - 1; // 读者数量-1
    if(rc == 0) signal(&db); // 最后的读者走了，给数据库解锁
    signal(&mutex); // rc访问结束，解锁
    else_action();
}

void writer() {
    else_action();
    wait(&db); // 要写数据库，给数据库加锁
    write_database(); // 写数据
    signal(&db); // 离开数据库，给数据库解锁
}
```

04

其他IPC机制

共享存储器系统

- 共享存储器系统(shared-memory system)中，相互通信的进程共享某些数据结构或存储区，进程之间能够通过这些空间进行通信。
- 基于共享数据结构的通信方式
- 基于共享存储区的通信方式

基于共享数据结构的通信方式

- 要求各进程共享某些数据结构，以实现进程间的信息交换
- 如在生产者-消费者问题中的有界缓冲区
- 由程序员负责对共享数据结构进行设置和对进程间同步进行处理
- 仅适用于传送相对较少量的数据，通信效率低下
- 低级进程通信

基于共享存储区的通信方式

- 在内存中划出了一块共享存储区，各进程可通过对该共享存储区的读/写来交换信息、实现通信
- 数据的形式和位置(甚至访问)均由进程负责控制
- 在通信前，进程先向系统申请获得共享存储区中的一个分区，并将其附加到自己的地址空间中，进而便可对其中的数据进行正常的读/写
- 高级进程通信

管道通信系统

- “管道”(pipe)，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件
- 首创于UNIX系统
- 能有效地传送大量数据

管道通信系统

- 写进程：向管道输入的发送进程
- 读进程：接收管道输出的接收进程
- 管道机制必须提供3方面的协调能力
  - 互斥：当一个进程正在对管道执行读/写操作时，其他进程必须等待
  - 同步：当写进程把一定数量的数据写入管道后，便去睡眠，直到读进程取走数据；当读进程读一空管道时，也应睡眠，直至写进程将数据写入管道。
  - 确定对方是否存在，只有确定了对方已存在时，才能进行通信

大纲

01 管程

02 消息传递

03 经典IPC问题

04 其他IPC机制