进程同步-2.1

教师：姜博
E-Mail: gongbell@gmail.com

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 经典的进程进程通信的主要方法
- 同步与互斥问题

信号量机制

- 1965年Dijkstra提出了新的变量类型Semaphore（信号量）
- S >= 0 表示当前可用的资源的数目
- 对信号量P（S）& V（S）操作原语。P、V分别是荷兰语的test(Proberen)和increment(Verhogen)）
- P: down V： up
- S的初值
  - (S=1) 实现互斥：二元信号量(等于mutex)
  - (S>1) 实现同步：通用信号量

信号量的使用：

- 必须置一次且只能置一次初值（代表资源的个数）
- 只能由P、V操作来改变
- P/V操作是原语：原子操作不会被打断
  - 原子操作：一组相关的操作要么都执行要么都不执行
  - 例如，A---B的转账操作
  - 数据库的事务

物理意义

- P（down）操作分配资源：检查信号量初值是否大于0，如果大于0，减1，继续执行；如果等于0，进程被直接阻塞（将当前进程从运行队列移动到信号量的队列），减1的操作暂时不做。
  - P是原子操作：一组操作，要么都执行，要么都不执行
  - 当S=1，A和B同时调用P(S),只有一个进程能够完成P(S)
  - 实现时，操作前关闭中断，操作后打开中断。
- 当进程执行P操作阻塞到某个信号量，进程不在运行态，所以在唤醒前都不会占用CPU资源。

物理意义

- V（up）操作释放资源：首先将信号量S增加1 (原子操作)。但是如果有一个或者多个进程在信号量的队列睡眠（这时S=1），就会随机唤醒一个进程（将进程从信号量的队列移入就绪队列），并使得其运行后能完成P操作的减1，所以这时S还是0。
  - V(S)原子操作
  - 实现时，操作前关闭中断，操作后打开中断。
  - 注意“V(S) != S=S+1
  - 如果S=6，进程A，B任意顺序调用V(S), 那么结果一定是8

信号量用于互斥

P(S)
临界区
V(S)

P(S)
临界区
V(S)

信号量同步

sem.wait() 代码A	代码B sem.signal()
P(S) 代码A	代码B V(S)

信号量同步

sem.wait() 代码A	代码B sem.signal()
P(S) 代码A	代码B V(S)

初值S=0; 代码B---代码A

使用信号量实现汇合 (Rendezvous)

- 使用信号量实现线程A和线程B的汇合(Rendezvous)。使得a1永远在b2之前，而b1永远在a2之前。
  - a1和b1的次序不加限制
- 基本的同步模式：使得两个线程在执行过程中一点汇合，直到两者都到后才能继续执行。

Thread A

```
1 statement a1
2 statement a2
```

Thread B

```
1 statement b1
2 statement b2
```

使用信号量实现会和 (Rendezvous)

- 使用信号量实现线程A和线程B的会和(Rendezvous)。使得a1永远在b2之前，而b1永远在a2之前。
- 提示：定义两个信号量，aArrived, bArrived,并且初始化为0，表示a和b是否执行到汇合点。

Thread A

```
1 statement a1
2 statement a2
```

Thread B

```
1 statement b1
2 statement b2
```

使用信号量实现会和 (Rendezvous)

- 使用信号量实现线程A和线程B的会和(Rendezvous)。使得a1永远在b2之前，而b1永远在a2之前。
- 定义两个信号量，aArrived, bArrived,并且初始化为0，表示a和b是否执行到汇合点。

Thread A

```
1 statement a1
2 aArrived.signal()
3 bArrived.wait()
4 statement a2
```

Thread B

```
1 statement b1
2 bArrived.signal()
3 aArrived.wait()
4 statement b2
```

使用信号量实现多路复用 (Multiplex)

- 实现mutex的泛化，使得n个线程能够同时运行在临界区？

使用信号量实现多路复用 (Multiplex)

- 实现mutex的泛化，使得n个线程能够同时运行在临界区？有时候也称为限流阀。
- 设置信号量multiplex=n

Multiplex solution

```
1 multiplex.wait()
2     critical section
3 multiplex.signal()
```

使用信号量实现多路复用 (Multiplex)

- 实现mutex的泛化，使得n个线程能够同时运行在临界区？
- 设置信号量multiplex=n

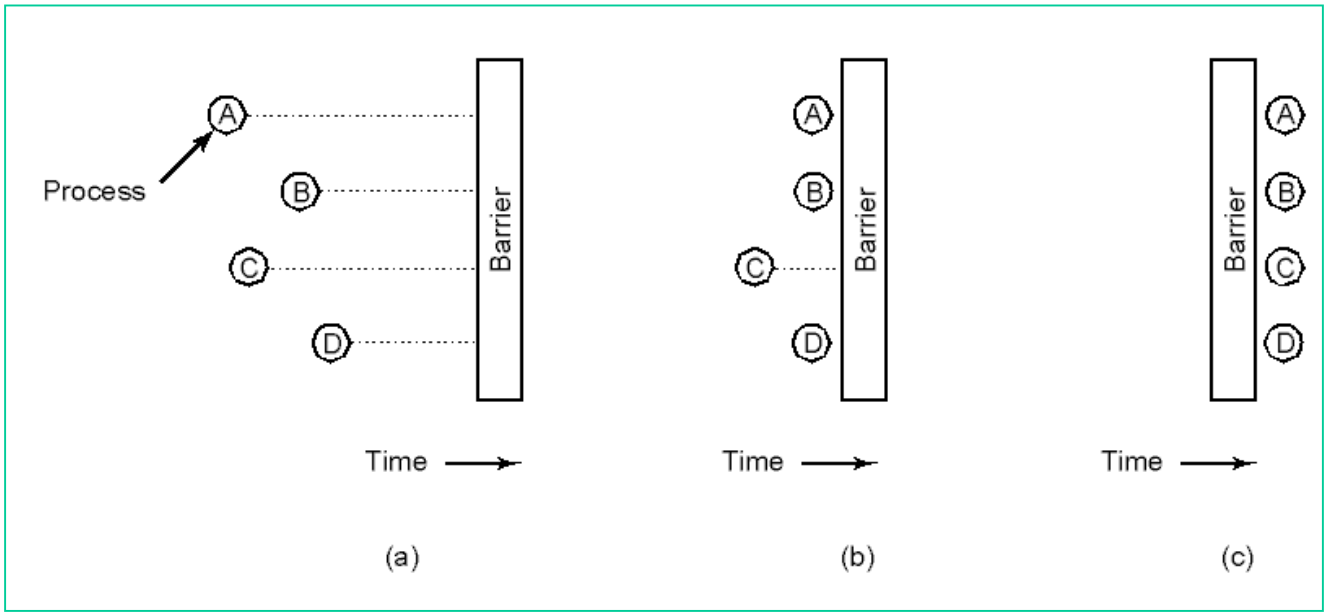
Multiplex solution

```
multiplex.wait()
    critical section
multiplex.signal()
```

- 比喻：将信号量看做n个令牌（钥匙），当线程wait时候，拿到一个令牌，signal时候释放一个令牌（钥匙）。
- 排队发号，每个号可以买一张票。

多进程同步原语：屏障Barriers

- 用于进程组的同步



- 思考：如何使用信号量实现Barrier?

多进程同步原语：屏障Barriers

- 对rendezvous进行泛化，使其能够用于多个线程，用于进程组的同步
  - 深度学习的卷进神经网络迭代
  - GPU编程中的渲染算法迭代

- 思考：如何使用信号量实现Barrier?

多进程同步原语：屏障Barriers

- 对rendezvous进行泛化，使其能够用于多个线程，用于进程组的同步
- 提示：

```
n = the number of threads
count = 0 //到达汇合点的线程数
mutex = Semaphore(1) //保护count
barrier = Semaphore(0)//线程到达之前是0，用于前n-1个线程排队
```

- Count记录到达汇合点的线程数。mutex保护count，barrier在当所有线程到达之前都是0，用于排队。
- 思考：如何使用信号量实现Barrier?

一种低级通信原语：屏障Barriers

- 思考：如何使用PV操作实现Barrier?
  - n = the number of threads
  - count = 0 //到达汇合点的线程数
  - semaphore mutex = 1 //保护count
  - semaphore barrier = 0//线程到达之前都是0或者负值。到达后取正值

  - P(mutex)
  - count = count + 1
  - V(mutex)

  - if count == n: V(barrier) # 第n个进程到来，唤醒一个线程，触发。
  - P(barrier) # 前n-1个进程在此排队
  - V(barrier) # 一旦线程被唤醒，有责任唤醒下一个线程

一种低级通信原语：屏障Barriers

- 思考：如果只有两个进程？是否还值得使用barrier对他们进行同步？

一种低级通信原语：屏障Barriers

- 思考：如果只有两个进程？是否还值得使用barrier对他们进行同步？
- 如果进程是分阶段运行，并且两个进程在都完成一个阶段执行之前，都不能进入下一段执行，那么使用barrier是非常合适的。

进程同步/互斥类问题的解答

- 解题步骤：
  - 分析问题，确定哪些操作是并发的。在并发的操作中，哪些是互斥的，哪些是同步的。
    - 多个进程操作同一个临界资源就是互斥
    - 多个进程要按一定的顺序执行就是同步
  - 根据同步和互斥规则设置信号量，说明其含义和初值；
  - 用P、V操作写出程序描述。

“信号量集”机制

Process A:
P(Dmutex);
P(Emutex);

Process B:
P(Emutex);
P(Dmutex);

Dmutex, Emutex = 1;

Process A: P(Dmutex);
Process B: P(Emutex);
Process A: P(Emutex);
Process B: P(Dmutex);

需要同时获取两个或多个临界资源时，
就可能出现由于各进程分别获得部分临界资源
并等待其余的临界资源的死锁局面

AND型信号量集机制

- AND型信号量集是指同时需要多个资源且每种占用一个资源时的信号量操作
- 基本思想：将进程需要的所有共享资源一次全部分配给它；待该进程使用完后再一起释放。
- 我们称**AND型信号量集P原语为SP，V原语为SV**。
- 在**SP**时，各个信号量的次序并不重要。
  - 虽然会影响进程归入哪个阻塞队列，但是因为是对资源全部分配或不分配，所以总有进程获得全部资源并在推进之后释放资源，因此不会死锁。

```
SP(S1, S2, … ,Sn)
  if S1>=1 and … and Sn>=1 then
     for i :=1 to n do
        Si := Si - 1;
     endfor
  else
        wait in Si;
  endif
```

```
SV(S1, S2, … ,Sn)
   for I :=1 to n do
        Si := Si + 1;
        wake waited process on Si
   endfor
```

将进程调度到第一个小于1的信号量Si的等待队列

一般“信号量集”机制

- 一般“信号量集”是指同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的信号量处理。
- 一次需要N个某类临界资源时，就要进行N次wait操作——低效且容易死锁
- 基本思想：在AND型信号量集的基础上进行扩充：进程对信号量Si的测试值为ti（用于信号量的判断，即Si >= ti，表示资源数量低于ti时，便不予分配），资源的申请量为di（用于信号量的增减，即Si = Si - di和Si = Si +di）

```
SP(S1, t1, d1, … , Sn, tn, dn)
   if S1>=t1 and … and Sn>=tn then
      for I :=1 to n do
         Si := Si - di;
      endfor
  else
   wait in Si;
  endif
```

```
SV(S1, d1, … ,Sn, dn)
   for I :=1 to n do
         Si := Si + di;
         wake waited process
   endfor
```

- 原语：
- SP(S1, t1, d1; ...; Sn, tn, dn);
- SV(S1, d1; ...; Sn, dn);

特殊情况：

- SP(S, d, d)
  - 表示每次申请**d**个资源，当资源数量少于**d**个时，便不予分配
- SP(S, 1, 1)
  - 表示互斥信号量
- SP(S, 1, 0)
  - 可作为一个可控开关**(**当**S≥1**时，允许多个进程进入临界区；当**S=0**时禁止任何进程进入临界区**)**

P.V操作的优缺点

- 优点：
  - 简单，而且表达能力强（用P.V操作可解决任何同步互斥问题）
- 缺点：
  - 不够安全；P.V操作使用不当会出现死锁；遇到复杂同步互斥问题时实现复杂

进程同步-2.2

教师：姜博
E-Mail: gongbell@gmail.com

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 经典的进程间通信的主要方法
- 同步与互斥问题

互斥量 (Mutex)

- 信号量值设置为1就是互斥量。
- 用于实现对共享资源和代码的互斥访问。
- 0 解锁状态，1 加锁状态
- 如果mutex已经加锁，调用mutex_lock会阻塞
- 很方便在用户态线程库实现

互斥量 (Mutex)

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

Pthreds提供的互斥量 (Mutex)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Pthreds提供的互斥量 (trylock)

Trylock使得进程可以不会因为等待信号量而阻塞。

```
int result = pthread_mutex_trylock(&mutex);

if(result==0)
{
    sleep(5);
    printf(" i is %d\n",i);

    pthread_mutex_unlock(&mutex);
}
else
    if (result == EBUSY)
            printf("thread busy\n");
}
```

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 经典的进程间通信的主要方法
- 同步与互斥问题

管程 (Monitor)

- 用信号量可实现进程间的同步，但：(1)加重了编程的负担；(2)同步操作分散在各个进程中，使用不当可能导致死锁或逻辑错误（如P/V操作次序错误、重复、遗漏）
- 管程：把分散的临界区集中起来，为每个临界资源设计一个专门机构来统一管理各进程对该资源的访问，这个专门机构称为管程。
- 管程可以函数库的形式实现。相比之下，管程比信号量好控制。
- 管程是一种高级同步原语。

管程的引入

- 1973年，Hoare和Hanson所提出
- 一个管程定义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。
- 为每个临界资源设立一个管程，由用户编写，对共享变量的访问通过其公有接口实现
- 类似“面向对象”的观点,封装了共享资源的使用
- 通过编译器支持，和特定语言相关
  - C，pascal不支持
  - Java，Objective C支持

管程的实现

- 对于临界资源，管程需要提供互斥的访问和同步的机制
- 管程的机制保证只有一个进程在管程内执行
- 同步机制使用
  - wait和signal，两个同步变量
- 如何防止signal后两个进程同时执行?
  - Hoare，让新唤醒进程执行
  - Hanse，让调用signal的进程立刻退出，signal是最后一个语句
  - 让signal进程继续执行，退出后执行唤醒的进程

管程的实现

```
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;
```

管程的实现

```
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;
```

管程的实现

- 使用wake和signal
  - 不同于sleep和wakeup的实现版本
  - 管程的函数会被自动化保证互斥访问
  - 检查缓冲区操作和wait操作会保证原子性

管程的实现-JAVA

```
public class ProducerConsumer {
        static final int N = 100;       // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );      // instantiate a new monitor

        public static void main(String args[ ]) {
            p.start( );       // start the producer thread
            c.start( );       // start the consumer thread
        }

        static class producer extends Thread {
            public void run( )  { // run method contains the thread code
                int item;
                while (true) {      // producer loop
                    item = produce_item( );
                    mon.insert(item);
                }
            }
            private int produce_item( ) { ... }      // actually produce
        }
```

管程的实现

```
static class consumer extends Thread {
    public void run( )  { run method contains the thread code
        int item;
        while (true) {      // consumer loop
            item = mon.remove( );
            consume_item (item);
        }
    }
    private void consume_item(int item) { ... }// actually consume
}
```

管程的实现

```
static class our_monitor {   // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0;   // counters and indices

    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
        buffer [hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N;          // slot to place next item in
        count = count + 1;          // one more item in the buffer now
        if (count == 1) notify( );          // if consumer was sleeping, wake it up
    }
```

JAVA语言保证一旦对象的**同步方法**被一个线程开始执行，其他线程不能执行这个对象的任何同步方法.

管程的实现

```
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );      // if the buffer is empty, go to sleep
        val = buffer [lo]; // fetch an item from the buffer
        lo = (lo + 1) % N;          // slot to fetch next item from
        count = count – 1;          // one few items in the buffer
        if (count == N – 1) notify( ); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
```

管程的优缺点

- 依赖于编译器支持，和特定语言相关
  - C，pascal不支持
  - Java，Objective C支持
- 不适用于分布式系统
  - 仅仅适用于单核或者多核的共享内存系统

内容提要

- 同步与互斥问题
- 基于忙等待的互斥方法
- 基于信号量的方法
- 基于管程的同步与互斥
- 进程通信的主要方法
- 经典的进程同步与互斥问题

进程间通信

- 低级通信：只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量和管程机制。缺点：
  - 传送信息量小：效率低，每次通信传递的信息量固定，若传递较多信息则需要进行多次通信。
  - 编程复杂：用户直接实现通信的细节，编程复杂，容易出错。
- 高级通信：能够传送任意数量的数据，包括三类：管道、共享内存、消息系统等。

IPC概述

- IPC历史
  - AT&T的贝尔实验室： System V IPC，通信进程局限在单个计算机内。(管道，信号，消息队列，信号量，共享内存区)
  - BSD的加州大学伯克利分校的伯克利软件发布中心：基于套接字（Socket）的进程间通信机制。
  - 电子电气工程协会（IEEE）： POSIX
- SolarisIPC
  - Solaris则把两者（SYSTEM V和BSD）都继承了下来，并用于不同场合。POSIX放在了函数库中。
  - 都有轻微变动和增加。
- WindowsIPC

IPC概述

- 管道（Pipe）及命名管道（Named pipe或FIFO）
- 信号（Signal）
- 消息队列（Message）
- 共享内存（Shared memory）
- 信号量（Semaphore）
- 套接字（Socket）

IPC概述

- 管道（Pipe）及命名管道（Named pipe或FIFO）
  - 管道是一种半双工的通信方式，数据只能单向流动。
- 信号（signal）
  - 信号是比较复杂的通信方式，用于通知接受进程有某种事件发生
- 消息队列（Message）
  - 消息队列是消息的链接表，有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。
  - 消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

IPC概述

- 共享内存（Shared memory）
  - 使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。
  - 需要结合其他通信机制（如信号量）保证安全
- 信号量（Semaphore）
  - 主要作为进程间以及同一进程不同线程间的同步手段。
- 套接字（Socket）
  - 更为一般的进程间通信机制，可以通过网络接口用于不同机器之间的进程间通信。
  - 也可以用于本机进程间的通讯

管道通信(Pipe)

- 管道是用于连接读进程和写进程以实现两个进程通信的共享文件，又称管道文件。
- Unix系统中，管道分为有名管道和无名管道。
  - 无名管道： sort < file1 |grep sth_to_search
  - 有名管道： mkfifo()函数创建

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
int mknod(const char *filename, mode_t mode | S_IFIFO, (dev_t)0);

mkfifo fifotest
cat < fifotest
另一个shell进程内： echo “hello fifo” > fifotest
```

管道通信(Pipe)-无名管道

- 无名管道-杀死一个叫conky的进程

ps aux | grep conky| grep -v grep | awk '{print $2}' | xargs kill

ps aux

```
$ ps aux
 rahmu     1925  0.0  0.1 129328  6112 ?        S    11:55   0:06 tint2
 rahmu     1931  0.0  0.3 154992 12108 ?        S    11:55   0:00 volumeicon
 rahmu     1933  0.1  0.2 134716  9460 ?        S    11:55   0:24 parcellite
 rahmu     1940  0.0  0.0  30416  3008 ?        S    11:55   0:10 xcompmgr -cC -t-5 -l-5 -r4
 rahmu     1941  0.0  0.2 160336  8928 ?        Ss   11:55   0:00 xfce4-power-manager
 rahmu     1943  0.0  0.0  32792  1964 ?        S    11:55   0:00 /usr/lib/xfconf/xfconfd
 rahmu     1945  0.0  0.0  17584  1292 ?        S    11:55   0:00 /usr/lib/gamin/gam_server
 rahmu     1946  0.0  0.5 203016 19552 ?        S    11:55   0:00 python /usr/bin/system-cor
 rahmu     1947  0.0  0.3 171840 12872 ?        S    11:55   0:00 nm-applet --sm-disable
 rahmu     1948  0.2  0.0 276000  3564 ?        Sl   11:55   0:38 conky -q
```

grep conky

```
$ ps aux | grep conky
 rahmu     1948  0.2  0.0 276000  3564 ?        Sl   11:55   0:39 conky -q
 rahmu     3233  0.0  0.0   7592   840 pts/1    S+   16:55   0:00 grep conky
```

管道通信(Pipe)-无名管道

- 无名管道-杀死一个叫conky的进程

ps aux | grep conky| grep -v grep | awk '{print $2}' | xargs kill

```
$ ps aux | grep conky | grep -v grep
 rahmu     1948  0.2  0.0 276000  3564 ?        Sl   11:55   0:39 conky -q
```

awk '{print $2}'

```
$ ps aux | grep conky | grep -v grep | awk '{print $2}'
 1948
```

xargs kill : kill <items> : kill 1948

$ ps aux | grep conky | grep -v grep | awk '{print $2}' | xargs kill

无名管道 (Pipe)

- 管道是半双工的，数据只能向一个方向流动；需要双向通信时，需建立起两个管道；
- 只能用于父子进程或者兄弟进程间（具有亲缘关系的进程）；
- 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在在内存中。
- 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。先进先出（first in first out）。

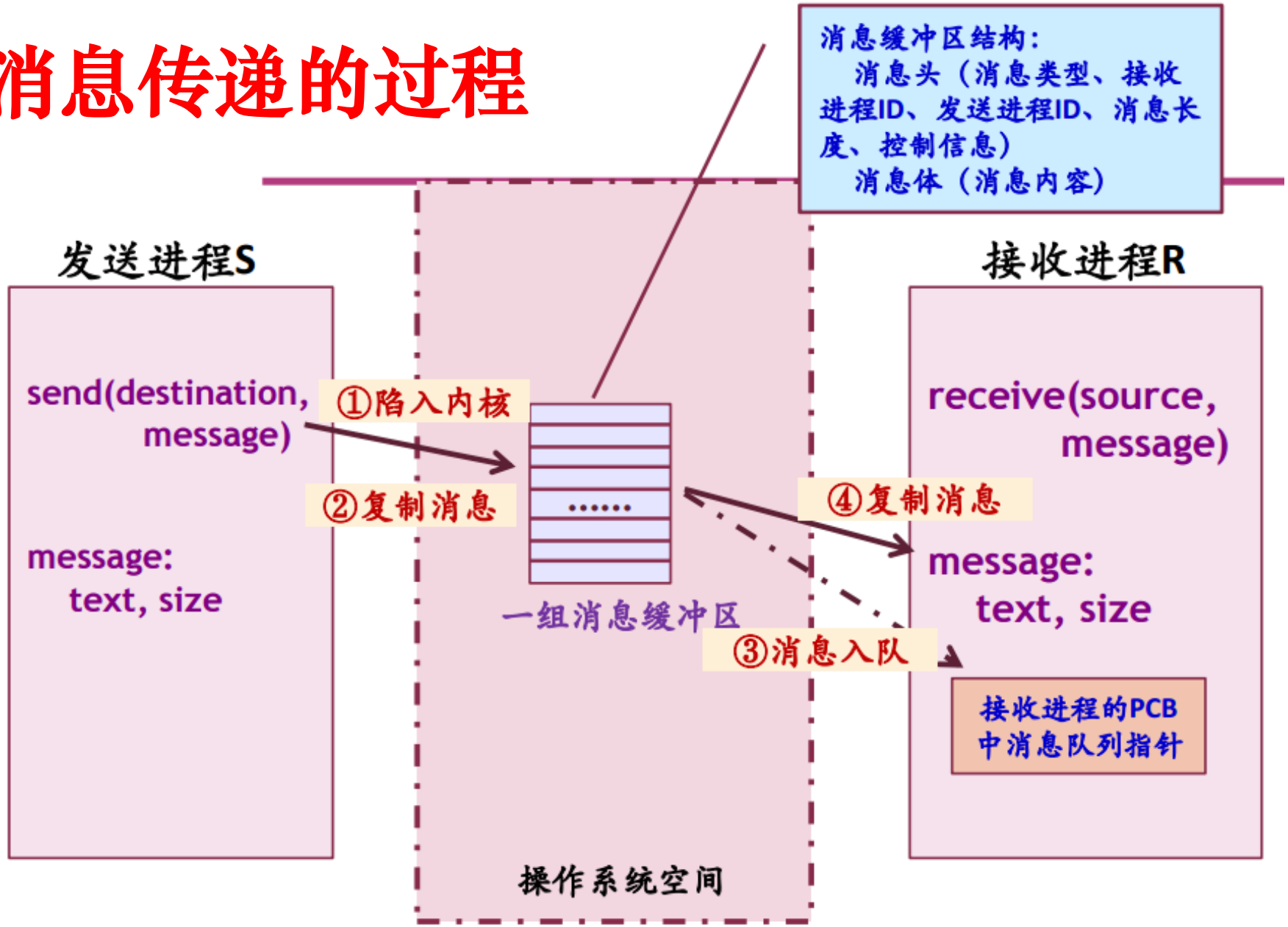
有名管道（Named Pipe或FIFO）

- 无名管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道提出后，该限制得到了克服。
- FIFO不同于管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信。
- 需注意的是，FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。

消息传递（message passing）

- 管程：过度依赖编译器；适用于单机环境。
- 消息传递——两个通信原语（OS系统调用）
  - send (destination, &message)
  - receive(source, &message)
- 调用方式
  - 阻塞调用
  - 非阻塞调用
- 主要问题：
  - 解决消息丢失、延迟问题（TCP协议）
  - 编址问题：mailbox

消息传递的过程


消息缓冲区结构：
消息头（消息类型、接收进程ID、发送进程ID、消息长度、控制信息）
消息体（消息内容）
发送进程S
接收进程R
send(destination, message)
①陷入内核
②复制消息
message:
text, size
一组消息缓冲区
④复制消息
receive(source, message)
message:
text, size
③消息入队
接收进程的PCB中消息队列指针
操作系统空间

用P-V操作实现Send原语

```
Send（destination, message）
{
 根据destination找接收进程;
 如果未找到，出错返回;

 申请空缓冲区P(buf-empty);
   P(mutex1);
     取空缓冲区;
   V(mutex1);

把消息从message处复制到空缓
冲区;
   P(mutex2);
    把消息缓冲区挂到接收进程
    的消息队列;
   V(mutex2);

   V(buf-full);
}
```

receive原语的实现?

信号量:
buf-empty初值为N
buf-full初值为0
mutex1初值为1
mutex2初值为1

共享内存

- 共享内存是最有用的进程间通信方式，也是最快的IPC形式（因为它避免了其它形式的IPC必须执行的开销巨大的缓冲复制）。
- 两个不同进程A、B共享内存的意义是，同一块物理内存被映射到进程A、B各自的进程地址空间。
- 当多个进程共享同一块内存区域，则需要同步机制约束（互斥锁和信号量都可以）。
- 共享内存通信的效率高。

POSIX

Semaphores	Message Queues	Shared Memory
sem_open	mq_open	shm_open
sem_close	mq_close	shm_unlink
sem_unlink	mq_unlink	
sem_init	mq_getattr	
sem_destroy	mq_setattr	
sem_wait	mq_send	
sem_trywait	mq_receive	
sem_post	mq_notify	
sem_getvalue	mq_getvalue	