操作系统　Operating System

第四章 进程与并发程序设计(3)
——同步与互斥

沃天宇
woty@buaa.edu.cn

2024年4月12日

几个算法的共性问题

- 无论是软件解法（如Peterson）还是硬件（如TSL或XCHG）解法都是正确的，它们都有一个共同特点：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则该进程将原地等待，直到允许为止。

1. 忙等待：浪费CPU时间

2. 优先级反转：低优先级进程先进入临界区，高优先级进程一直忙等

解决之道

- 忙等->阻塞
- Sleep
- Wakeup

- E.g.银行排队
  - 忙等：看到队很长，坚持排队
  - 阻塞：看到队很长，先回家歇会儿（sleep），有空柜台了，大堂经理（scheduler）电话通知再过来（wakeup）

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

信号量机制

- 1965年Dijkstra。P（S），V（S）操作。P、V分别是荷兰语的test(proberen)和increment(verhogen)）
- 其基本思路是使用一种新的变量类型（semaphore）
- 信号量只能通过初始化和两个标准的原语来访问，作为OS核心代码执行，不受进程调度的打断

经典信号量机制

P(S): while S<=0 do skip
S:=S-1;

V(S): S:=S+1;

计数信号量机制

```
Type semaphore = record
        value : integer;
        L : list of process;
end

Procedure P(S)
        var S : semaphore;
        begin
                S.value := S.value -1;
                if S.value<0 then block(S.L);
        end

procedure V(S)
        var S : semaphore;
        begin
                S.value := S.value + 1;
                if S.value<=0 then wakeup(S.L)
        end
```

信号量的使用：

- 必须置一次且只能置一次初值
- 只能由P、V操作来改变

物理意义

- S.value为正时表示资源的个数
- S.value为负时表示等待进程的个数
- P操作分配资源
  - 如果无法分配则阻塞（wait）
- V操作释放资源
  - 如果有等待进程则唤醒（signal）

信号量机制的实现

- 原子性问题；
  - 关中断、TS指令等
- PCB链表形式。

《现代操作系统》P74

信号量的应用

- 互斥
  - 利用信号量实现进程互斥（S=1）
- 同步
  - 利用信号量实现进程同步（S=0）
  - 例如：描述进程执行的前趋/后继关系

互斥

P(S)

临界区

V(S)

P(S)

临界区

V(S)

同步

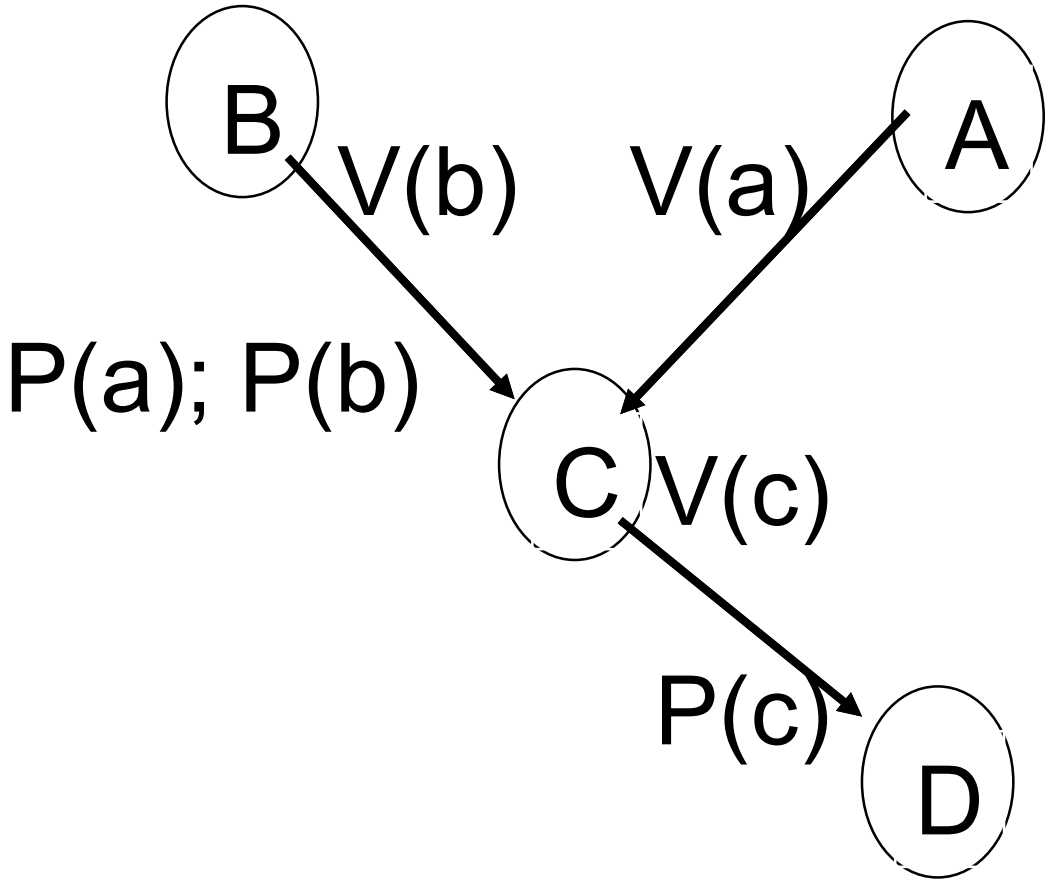
P(S)

代码

V(S)

代码

前趋关系

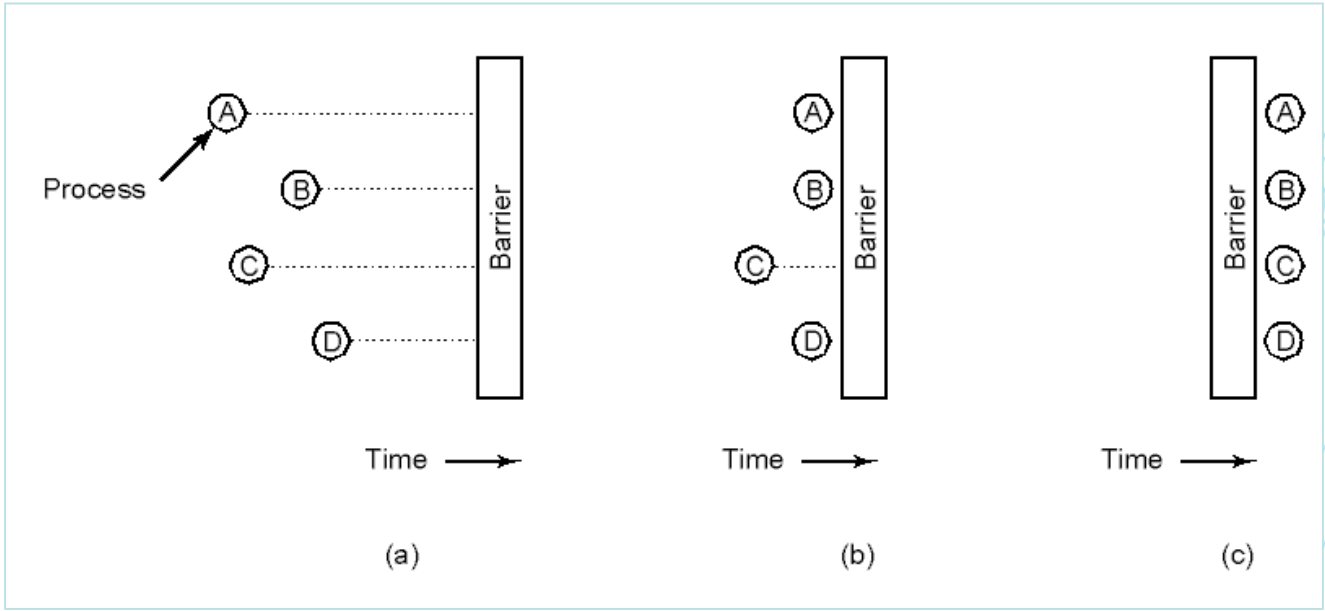


信号量在并发中的典型应用

应用	描述
互斥（Mutual exclusion）	可以用初始值为1的信号量来实现进程间的互斥。一个进程在进入临界区之前执行semWait操作，退出临界区后再执行一个semSignal操作。这是实现临界区资源互斥使用的一个二元信号量。
有限并发（Bounded concurrency）	是指有n（1≤n≤c，c是一个常量）个进程并发的执行一个函数或者一个资源。一个初始值为c的信号量可以实现这种并发。
进程同步（Synchronization）	是指当一个进程P_i想要执行一个a_i操作时，它只在进程P_j执行完a_j后，才会执行a_i操作。可以用信号量如下实现：将信号量初始为0，P_i执行a_i操作前执行一个semWait操作；而P_j执行a_j操作后，执行一个semWait操作。

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
  - 科学计算中的迭代
  - 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或进程组的同步
- 提示：

```
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```

- 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或者负值。到达后取正值。
- 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
  - mutex = Semaphore(1) //保护count
  - barrier = Semaphore(0)//线程到达之前都是0或者负值。到达后取正值

  - mutex.wait()
  - count = count + 1
  - mutex.signal()

  - if count == n: barrier.signal() # 唤醒一个线程
  - barrier.wait()
  - barrier.signal() # 一旦线程被唤醒，有责任唤醒下一个线程

“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);

“信号量集”机制

当利用信号量机制解决了单个资源的互斥访问后，我们讨论如何控制同时需要多个资源的互斥访问。信号量集是指同时需要多个资源时的信号量操作。

- “AND型”信号量集
- 一般信号量集

AND型信号量集机制

- 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。

```
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
  endfor
```

一般“信号量集”机制

- 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），占用值为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
```

一般“信号量集”机制

几个例子：

- 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

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

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，使用不当就可能导致死锁（如P、V操作的次序错误、重复或遗漏）。
- 管程：把分散的临界区集中起来，为每个可共享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

- 1972年前后，在**E.W. Dijkstra**等工作的基础上，由**Brinch Hansen**和 **Tony Hoare**所提出，**Hansen**在并发的**Pascal**语言中首先实现了管程。
- 一个管程是由过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或者软件包。
  - 局部于该管程的共享数据：资源状态
  - 局部于该管程的若干过程：对数据的操作
- 互斥：任一时刻，管程中只能有一个活跃进程
- 管程是一种语言概念，由编译器负责实现互斥

管程的实现

- 目标：为每个共享资源设立一个管程，对共享资源及其操作进行封装，从而简化对共享资源的互斥访问过程。类似于“面向对象”的观点。

- 局部控制变量（临界资源）：一组局部于管程的控制变量
- 初始化代码：对控制变量进行初始化的代码
- 操作原语（互斥）：对控制变量和临界资源进行操作的一组原语过程（程序代码），是访问该管程的唯一途径。
- 条件变量（同步）：每个独立的条件变量是和进程需要等待的某种原因相联系的，当定义一个条件变量x时，系统就建立一个相应的等待队列
  - **wait(x)**:把调用者进程放入x的等待队列
  - **signal(x)**：唤醒x等待队列中的一个进程

管程的条件变量

- 由于管程通常是用于管理资源的，因而在管程内部，应当存在某种等待机制。当进入管程的进程因资源被占用等原因不能继续运行时使其等待。为此在管程内部可以说明和使用一种特殊类型的变量----条件变量。
- 每个条件变量表示一种等待原因，并不取具体数值——相当于每个原因对应一个队列。

条件变量与信号量的区别

- 条件变量的值不可增减，P-V操作的信号量值可增减
  - wait操作一定会阻塞当前进程；但P操作只有当信号量的值小于0时才会阻塞。
  - 如果没有等待的进程，signal将丢失；而V操作增加了信号量的值，不会丢失。
- 访问条件变量必须拥有管程的锁

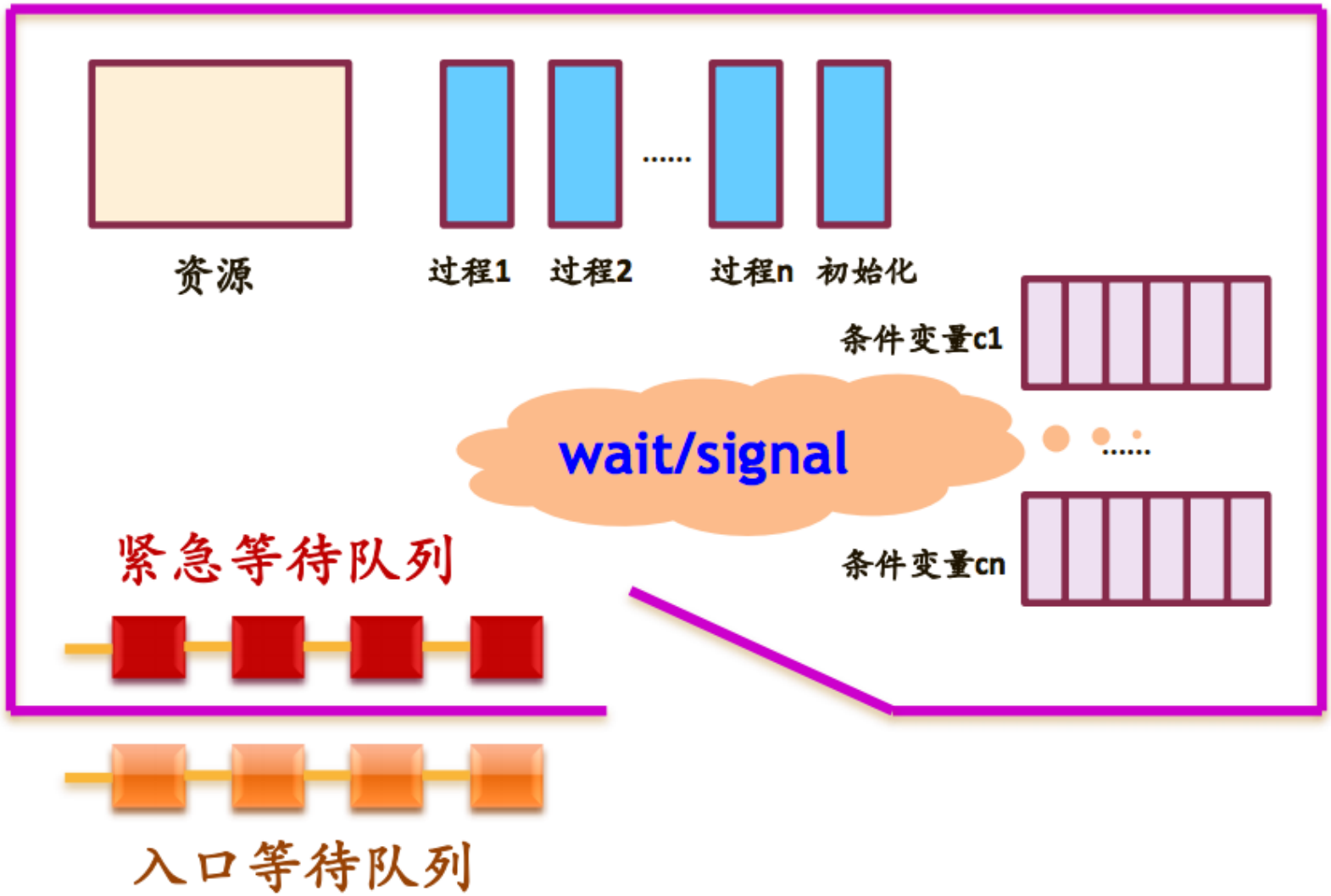
多个进程同时在管程中出现

- 场景：
  - 当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权。
  - 当后面进入管程的进程执行唤醒操作时（例如P唤醒Q），管程中便存在两个同时处于活动状态的进程。

Hoare管程与Hansen管程

- 当signal(x)时，如何避免管程中有两个活跃的进程
  - Hoare管程（阻塞式条件变量）：执行signal的进程等待，直到被释放进程退出管程或等待另一个条件
  - Mesa管程（非阻塞式条件变量）：被释放进程等待，直到执行signal的进程退出管程或者等待另一个条件
  - Hansen管程：执行signal的进程立即退出管程，即signal操作必须是管程中的过程体的最后一个操作

Hoare管程



Hoare管程

- 入口等待队列(entry queue)：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。
- 紧急等待队列：如果进程P唤醒进程Q，则P等待Q继续，如果进程Q在执行又唤醒进程R，则Q等待R继续，...，如此，在管程内部，由于执行唤醒操作，可能会出现多个等待进程（已被唤醒，但由于管程的互斥进入而等待），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。它的优先级应当高于入口等待队列的优先级。

Hoare管程的同步原语

- 同步操作原语wait和signal：针对条件变量x，x.wait()将自己阻塞在x队列中，x.signal()将x队列中的一个进程唤醒。
  - x.wait()：如果紧急等待队列非空，则唤醒第一个等待者；否则释放管程的互斥权，执行此操作的进程排入x队列尾部（紧急等待队列与x队列的关系：紧急等待队列是由于管程的互斥进入而等待的队列，而x队列是因资源被占用而等待的队列）。
  - x.signal()：如果x队列为空，则相当于空操作，执行此操作的进程继续；否则唤醒第一个等待者，执行x.signal()操作的进程排入紧急等待队列的尾部。

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(Inter-Process-Comm)

- 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：适用于分布式系统，基于共享内存的多处理机系统，单处理机系统，能够传送任意数量的数据，可以解决进程的同步问题和通信问题，主要包括三类：管道、共享内存、消息系统。

IPC概述

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

无名管道 (Pipe)

- 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
- 只能用于父子进程或者兄弟进程之间（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。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

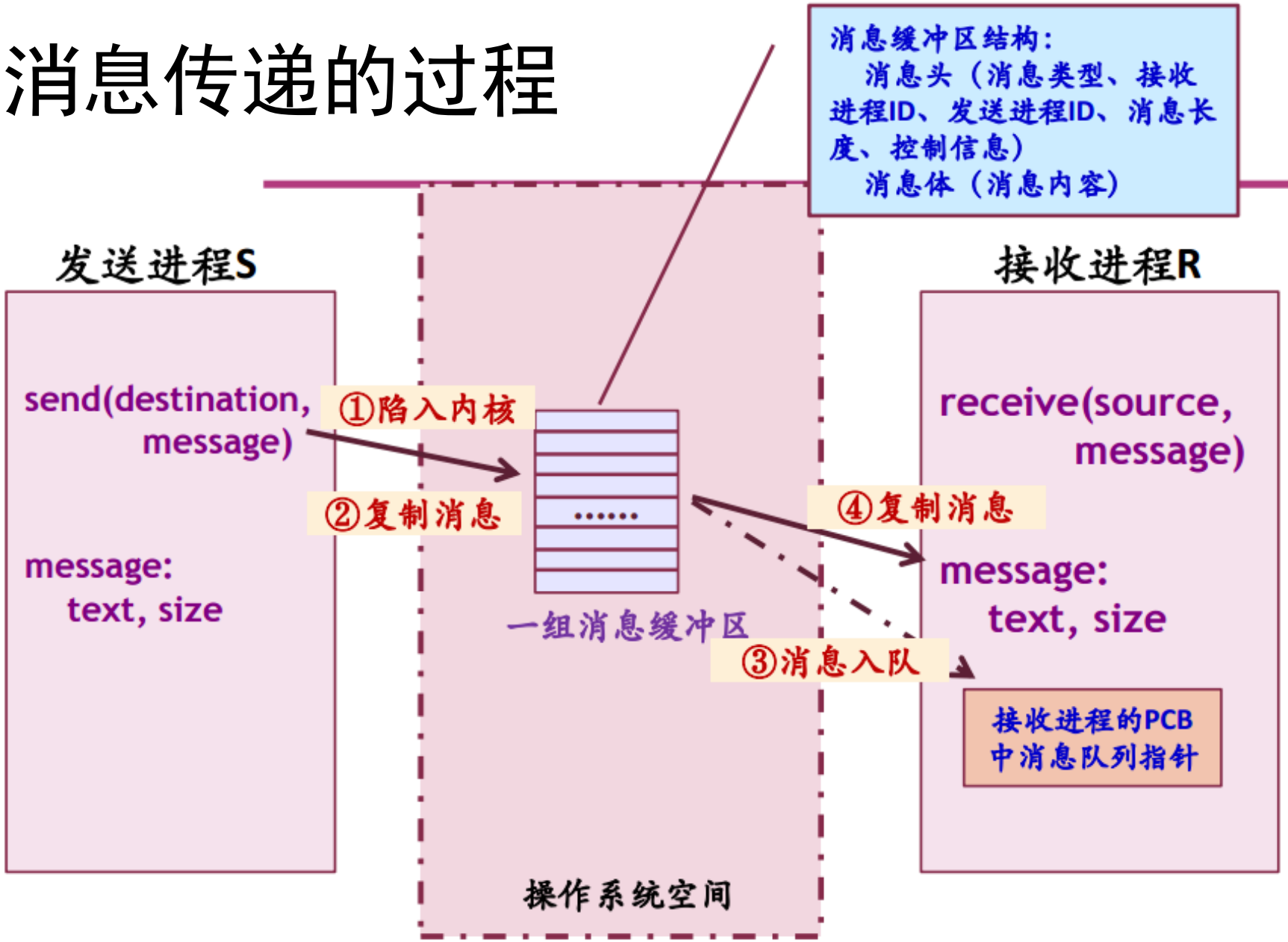
有名管道（Named Pipe或FIFO）

- 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（能够访问该路径的进程以及FIFO的创建进程之间），因此，通过FIFO不相关的进程也能交换数据。
- 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）

- 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

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，由于共享内存可以同时读但不能同时写，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（因为进程可以直接读写内存）。
- 进程之间在共享内存时，保持共享区域直到通信完毕。

共享内存机制