内容导航：

第2章 进程的描述与控制

进程通信概念

进程通信是指进程之间的信息交换(各进程为了保持联系而交换信息)

低级进程通信：**所交换的信息量相对较少，例如：**进程的同步和互斥

- 效率低
- 通信对用户不透明

高级进程通信：**用户可直接利用OS所提供的一组通信命令，高效地传送大量数据的一种通信方式**

- 使用方便
- 高效地传送大量数据

进程通信的类型(1)

共享存储器系统

- 基于共享数据结构的通信方式（效率低）
- 基于共享存储区的通信方式（高级）

管道通信

- 管道：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。
- 管道机制的协调能力：互斥、同步、对方是否存在

进程通信的类型(2)

消息传递系统

- 直接通信方式
- 间接通信方式（通过邮箱）

客户机-服务器系统

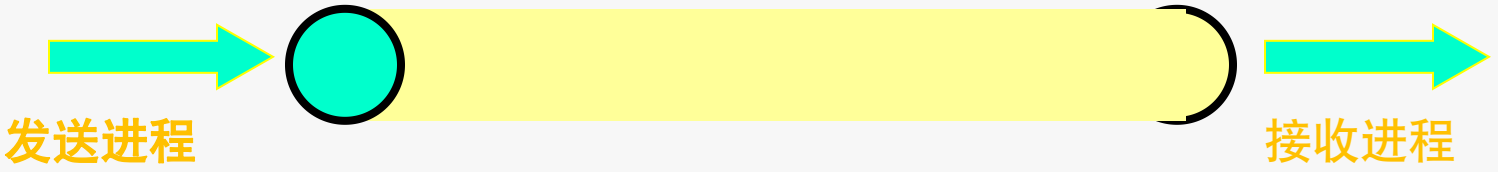
- 套接字（Socket）
- 远程过程调用（RPC）和远程方法调用（RMI，Java）

进程通信（Inter-Process Communication）

1. 共享存储器（shared-memory）方法的主要思想是：进程之间通过共享某些数据结构或存储区实现信息传送。
2. 有两种类型：
   - 基于共享数据结构的通信方式 （例：生产者—消费者问题中的有界缓冲区，属于低级通信方式）
   - 基于共享存储区的通信方式：在存储器中划出一块共享存储区，诸进程通过对共享存储区中数据的读或写实现通信（传输大量数据，属于高级通信）

进程通信（Inter-Process Communication）

1. 所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件
2. 向管道提供输入的发送进程(即写进程)，以字符流形式将大量的数据送人管道；而接收进程(即读进程)，可从管道中接收数据
3. 这种方式首创于UNIX系统，因它能传送大量的数据，且很有效，故又被引入到许多其它操作系统中



字符流方式写入读出
先进先出顺序

进程通信（Inter-Process Communication）

1. 互斥：即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。
2. 同步：指当写(输入)进程把一定数量的数据写入pipe，便去睡眠等待， 直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。
3. 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。

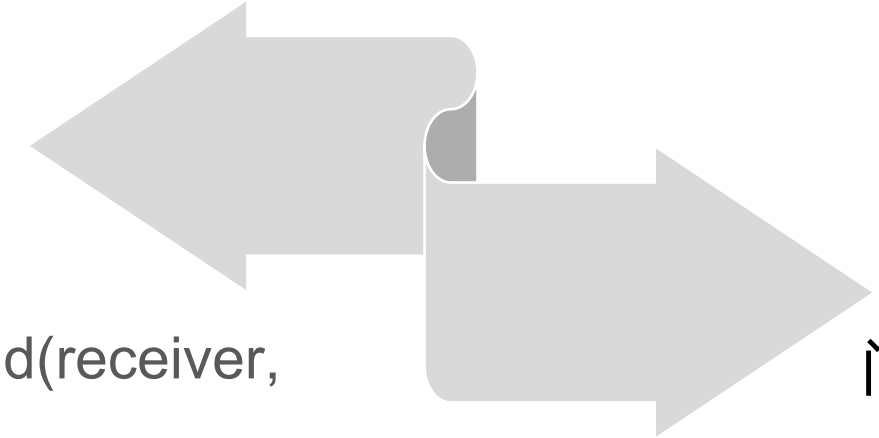
进程通信（Inter-Process Communication）

1. 当前应用**最广泛**的进程间通信机制
2. 系统提供**发送消息**Send(message)与**接收消息**Receive(message)两个操作，进程间则通过使用这两个操作进行通讯，无需共享任何变量
3. 实现方式的不同分成**直接通信方式**和**间接通信方式**两种
4. 进程间的数据交换，以**格式化的消息**为单位。所谓“信息”通常由消息头和消息正文构成
5. 实现了大量数据的传递，隐藏了实现细节，简化了程序编制复杂性，使用最广泛

消息传递通信的实现方式

直接通信方式

- 发送原语：send(receiver, message)
- 接收原语：receive(sender, message)



间接通信方式：通过信箱来完成

- 信箱结构
- 消息的发送和接收
  - 发送原语：send(mailbox, message)
  - 接收原语：receive(mailbox, message)
- 信箱类型
  - 私用邮箱，公共邮箱，共享邮箱

进程通信（Inter-Process Communication）

1. 直接通信原语
2. 消息的格式
3. 进程同步方式
4. 通信链路

- 发送进程利用OS所提供的发送命令，直接把消息发送给目标进程
- 要求发送进程和接收进程都以显式方式提供对方的标识符
- 系统提供下述两条通信命令(原语)：
  - Send(Receiver, message)
  - Receive(Sender, message)
- 例： Send(P2, m1)， Receive(P1，m1)

- 某些情况下，接收进程可与多个发送进程通信，因此，它不可能事先指定发送进程。
- 例如，用于提供打印服务的进程，它可以接收来自任何一个进程的“打印请求”消息。
- 对于这样的应用，在接收进程接收消息的原语中的源进程参数，是完成通信后的返回值，接收原语可表示为： Receive (id, message)

进程通信（Inter-Process Communication）

1. 直接通信原语
2. 消息的格式
3. 进程同步方式
4. 通信链路

- 通常，可把一个消息分成**消息头和消息正文**两部分。
- **消息头**包括消息传输时所需的**控制信息**，如发送进程名、接收进程名、消息长度、消息类型、消息编号及消息的发送日期和时间；
- **消息正文**则是发送进程实际上所发送的数据。

```
type Msg = record
        msgsender   消息发送者
        msgnext     下一个信息，链指针
        msgsize     整个消息的字节数
        msgtext     消息正文
    end
```

消息头

消息正文

进程通信（Inter-Process Communication）

1. 直接通信原语
2. 消息的格式
3. 进程同步方式
4. 通信链路

在进程之间进行通信时，往往也需要辅以进程同步，以使它们能协调通信

- 发送进程阻塞、 接收进程阻塞
- 发送进程不阻塞、 接收进程阻塞
- 发送进程和接收进程均不阻塞

进程通信（Inter-Process Communication）

1. 直接通信原语
2. 消息的格式
3. 进程同步方式
4. 通信链路

为了在发送进程和接收进程之间能进行通信，必须在它们之间建立一条通信链路。有两种方式建立通信链路：

1. 显式建立链路。由发送进程在通信之前，用“建立连接”命令(原语)请求系统为之建立一条通信链路；在链路使用完后，也应用显式方式拆除链路。这种方式主要用于计算机网络中
2. 隐式建立链路。发送进程无需明确提出建立链路的请求，只需利用系统提供的发送命令(原语)，系统会自动地为之建立一条链路。这种方式主要用于单机系统中

进程通信（Inter-Process Communication）

1	直接通信原语
2	消息的格式
3	进程同步方式
4	通信链路

1. 根据**通信方式**的不同，又可以把链路分为：
   - **单向通信链路**：只允许发送进程向接受者进程发送消息
   - **双向链路**：允许由进程A向进程B发送消息，也允许进程B同时向进程A发送消息
2. 根据**通信链路的容量**的不同也可以把链路分成：
   - **无容量通信链路**：在这种通信链路上没有缓冲区，因而不能暂存任何消息
   - **有容量通信链路**：在这种通信链路上设置了缓冲区，因而能暂存消息，缓冲区数目愈多，通信链路的容量愈大

进程通信（Inter-Process Communication）

共享存储器系统 | 管道通信 | 消息传递系统

直接消息传递系统 | 信箱通信（间接消息传递系统）

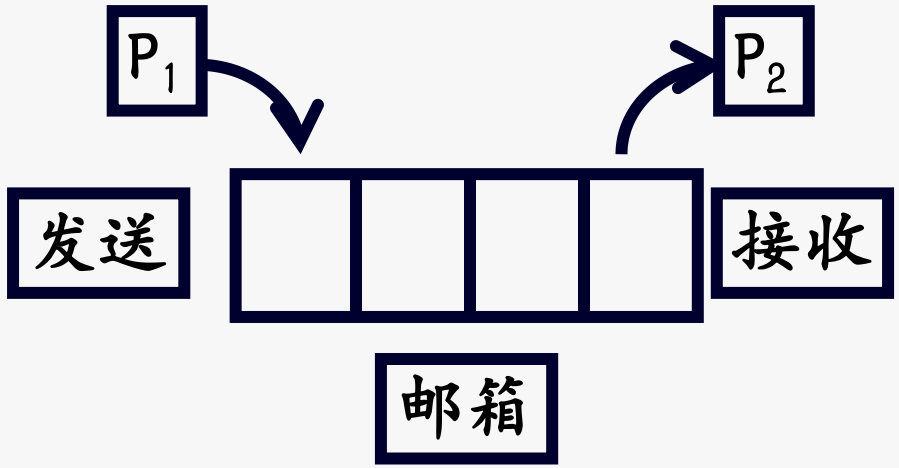
1. 直接通信原语
2. 消息的格式
3. 进程同步方式
4. 通信链路

```
repeat
    ...
    produce an item in nextp;
    ...
    send(consumer, nextp);
  until false;
repeat
    receive(producer, nextc);
    ...
    consume the item in nextc;
  until false;
```

- 利用直接通信原语，来解决生产者-消费者问题
- 当生产者生产出一个产品(消息)后，便用Send原语将消息发送给消费者进程；而消费者进程则利用Receive原语来得到一个消息。
- 如果消息尚未生产出来，消费者必须等待，直至生产者进程将消息发送过来。

进程通信（Inter-Process Communication）

1. 进程不把消息直接发给接收者进程，而把消息放在**某个双方共知的信箱中**
2. 既可以实现实时通信，又可以**实现非实时通信**
3. 系统提供若干原语，分别用于信箱的**创建**、**撤销**和消息的**发送**、**接收**



进程通信（Inter-Process Communication）

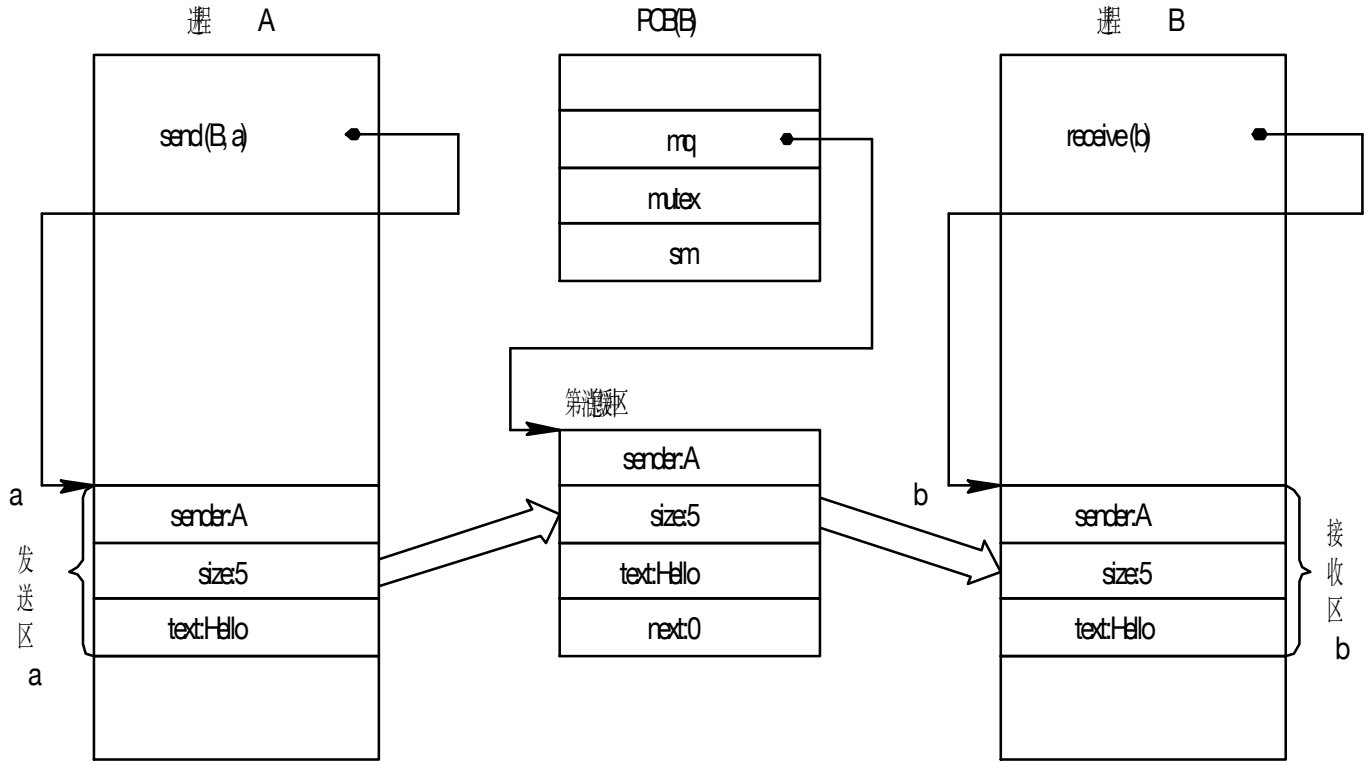
- 信箱分为以下三类
  - 私用信箱：用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中
  - 公用信箱：它由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息
  - 共享信箱：它由某进程创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息

进程通信（Inter-Process Communication）

1. 消息缓冲队列通信机制
2. 首先由美国的Hansen提出，并在RC4000系统上实现，后来被广泛应用于本地进程之间的通信中。
3. 基本思想：发送进程利用send原语，将消息发给接收进程；接收进程则利用Receive原语，接收消息。

进程通信（Inter-Process Communication）

共享存储器系统 | 管道通信 | 消息传递系统



① 设置发送区

② 填入发送消息相关信息

③ 调用发送原语

④ 发送区信息复制到缓冲区

⑤ 将缓冲区挂到目标进程的消息队列中

⑥ 调用接收原语

⑦ 将待接收消息复制到指定消息接收区

发送原语

```
procedure send(receiver, a)
   begin
     getbuf(a.size,i);                    根据a.size申请缓冲区；
     i.sender：   =a.sender;  将发送区a中的信息复制到消息缓冲区之中；
     i.size：    =a.size;
     i.text：    =a.text;
     i.next：    =0;
    getid(PCB set, receiver.j);   获得接收进程内部标识符；
    wait(j.mutex);
    insert(j.mq, i);   将消息缓冲区插入消息队列；
    signal(j.mutex);
    signal(j.sm);
  end
```

接收原语

```
procedure receive(b)
  begin
   j： =internal name;      j为接收进程内部的标识符；
   wait(j.sm);
   wait(j.mutex);
   remove(j.mq, i);         将消息队列中第一个消息移出；
   signal(j.mutex);
   b.sender：=i.sender;   将消息缓冲区i中的信息复制到接收区b;
    b.size：=i.size;
   b.text：=i.text;
   releasebuf(i)
 end
```

Linux进程通信方式

无名管道实例

```
#define INPUT 0
#define OUTPUT 1
void main() {
    int file_descriptors[2];
    pid_t pid;                    /*定义子进程号*/
    char buf[256];
    int returned_count;
    pipe(file_descriptors);    /*创建无名管道*/
    if((pid = fork()) == -1) {   /*创建子进程*/
        printf("Error in fork/n");
        exit(1);
    }
    if(pid == 0) {                /*执行子进程*/
        printf("in the spawned (child) process.../n");
        /*子进程向父进程写数据，关闭管道的读端*/
        close(file_descriptors[INPUT]);
        write(file_descriptors[OUTPUT], "test data",
                                    strlen("test data"));
        exit(0);
    }
    else {            /*执行父进程*/
        printf("in the spawning (parent) process.../n");
        /*父进程从管道读取子进程写的数据，关闭管道的写端*/
        close(file_descriptors[OUTPUT]);
        returned_count = read(file_descriptors[INPUT],
                                    buf, sizeof(buf));
        printf("%d bytes of data received from spawned process:
        %s/n",returned_count, buf); } }
```

有名管道实例

```
#include <stdio.h>
#include <unistd.h>
void main() {
FILE * in_file, *out_file;
int count = 1;
char buf[80];
in_file = fopen("mypipe", "r");     /*读有名管道*/
if (in_file == NULL) {
printf("Error in fdopen./n");
exit(1);
}
while ((count = fread(buf, 1, 80, in_file)) > 0)
printf("received from pipe: %s/n", buf);
fclose(in_file);
```

```
out_file = fopen("mypipe", "w");    /*写有名管道*/
if (out_file == NULL) {
printf("Error in fdopen./n");
exit(1);
}
sprintf(buf,"this is test data for the named pipe
example./n");
fwrite(buf, 1, 80, out_file);
fclose(out_file);
}
```

内容导航：

第2章 进程的描述与控制

线程的概念

时间

- 60年代中期：提出进程概念
- 80年代中期：提出线程概念
- 90年代后：多处理机系统引入线程

引入进程的目的

- 使多个程序并发执行，提高资源利用率及系统吞吐量

进程的2个基本属性：

- 进程是一个可拥有资源的独立单位；
- 进程是一个可独立调度和分派的基本单位。

线程的概念

提出线程的目的

- 减少程序在并发执行时所付出的时空开销
- 使OS具有更好的并发性
- 适用于SMP结构的计算机系统

进程是拥有资源的基本单位（传统进程称为重型进程）

线程作为调度和分派的基本单位（又称为轻型进程）

线程和进程的比较(1)

01 调度的基本单位

- 在传统的OS中，拥有资源、独立调度和分派的基本单位都是进程；
- 在引入线程的OS中，线程作为调度和分派的基本单位，进程作为资源拥有的基本单位；
- 在同一进程中，线程的切换不会引起进程切换，在由一个进程中的线程切换到另一个进程中的线程时，将会引起进程切换。

02 并发性

- 在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间，也可并发执行。

线程和进程的比较(2)

03 拥有资源

- 进程是系统中拥有资源的一个基本单位，它可以拥有资源。
- 线程本身不拥有系统资源，仅有一点保证独立运行的资源。
- 允许多个线程共享其隶属进程所拥有的资源。

04 独立性

- 同一进程中的不同线程之间的独立性要比不同进程之间的独立性低得多。

线程和进程的比较(2)

05 系统开销

- 在创建或撤消进程时，OS所付出的开销将显著大于创建或撤消线程时的开销。
- 线程切换的代价远低于进程切换的代价。
- 同一进程中的多个线程之间的同步和通信也比进程的简单。

06 支持多处理机系统

线程的状态和线程控制块

线程状态

- 执行态、就绪态、阻塞态
- 线程状态转换与进程状态转换一样

线程控制块（thread control block，TCB）

- 线程标识符、一组寄存器、线程运行状态、优先级、线程专有存储区、信号屏蔽、堆栈指针

内容导航：

第2章 进程的描述与控制

线程的实现

实现方式：

- 内核支持线程KST
- 用户级线程ULT
- 组合方式

具体实现（不同系统实现不同）：

- 内核支持线程的实现（利用系统调用）
- 用户级线程的实现（借助中间系统）

内核支持线程KST

在内核空间实现

- 无论用户进程中的线程，还是系统进程中的线程，其创建、撤销和切换都依靠内核，在内核空间通过系统调用实现
- 在内核空间为每个线程设置了一个线程控制块，内核根据该控制块感知该线程存在，并加以控制

优点：

- 在多处理机系统中，内核可同时调度同一进程的多个线程
- 如一个线程阻塞了，内核可调度其他线程(同一或其他进程)。
- 线程的切换比较快，开销小。
- 内核本身可采用多线程技术，提高执行速度和效率。

缺点：

- 对用户线程切换，开销较大。

用户级线程ULT

在用户空间实现

- 用户级线程仅在用户空间中，与内核无关，所有对线程的操作(创建、撤销、同步、互斥等)也在用户空间中由线程库的函数（过程）完成，而无需内核帮助

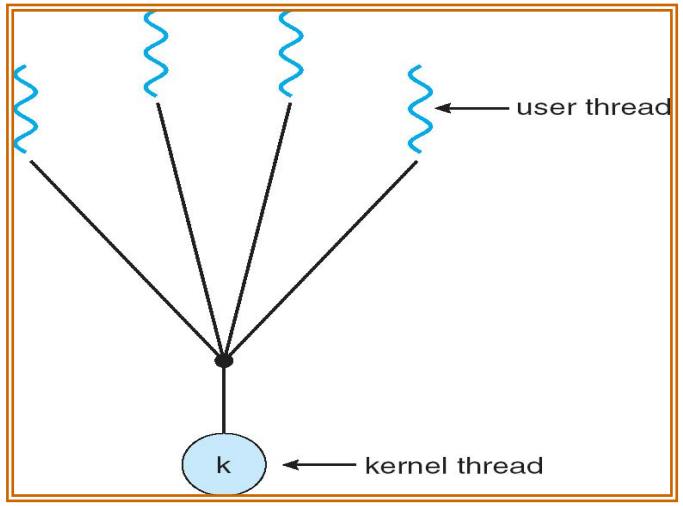
优点：

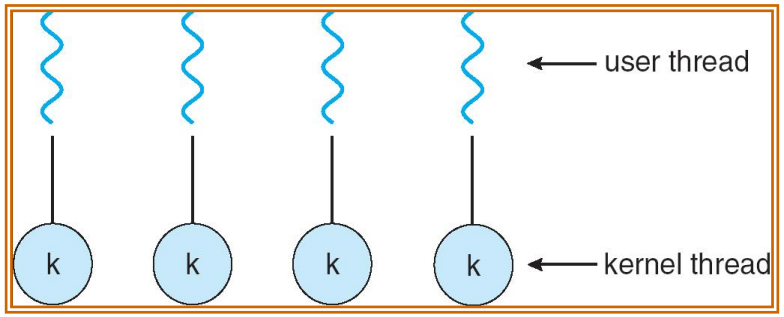
- 线程切换不需要转换到内核空间。
- 调度算法可以是进程专用的。
- 线程的实现与OS平台无关。

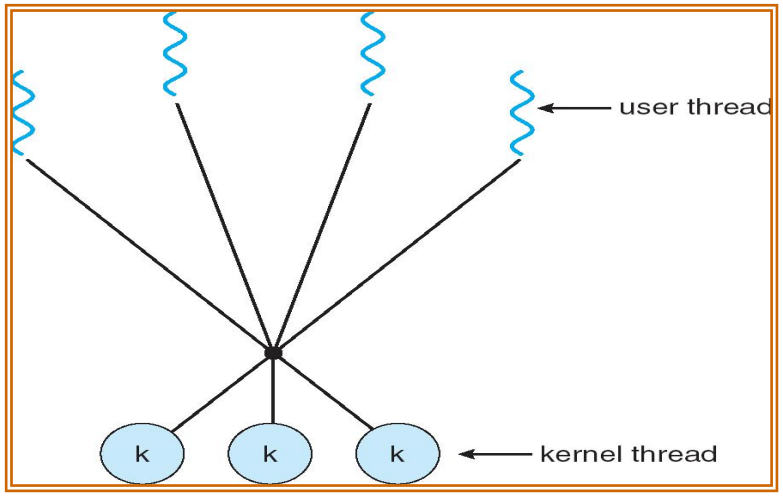
缺点：

- 系统调用的阻塞问题。
- 多线程应用不能利用多处理机进行多重处理的优点。

ULT与KST组合方式







- 多对一模型
- 一对一模型
- 多对多模型

多对一模型

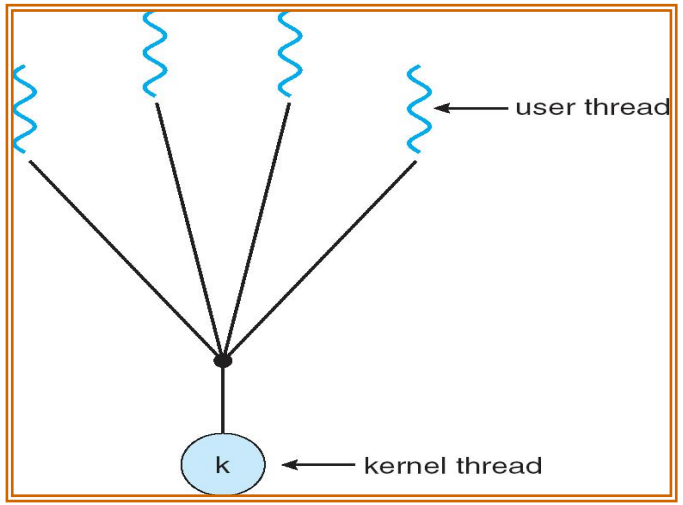
01 多个用户级线程映射到一个内核线程。

02 多个线程不能并行运行在多个处理器上。

03 线程管理在用户态执行，因此是高效的，但一个线程的阻塞系统调用会导致整个进程的阻塞 。

04 用于不支持内核线程的系统中。

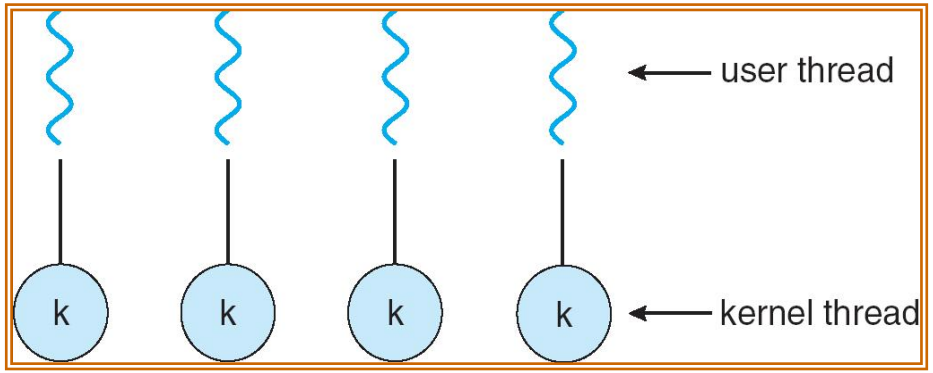
05 例子:
- Solaris Green Threads
- GNU Portable Threads



一对一模型

01 每个用户级线程映射到一个内核线程。

02 比多对一模型有更好的并发性。



03 允许多个线程并行运行在多个处理器上。

04 创建一个ULT需要创建一个KLT，效率较差。

05 例子：

- Windows 95/98/NT/XP/2000
- Linux
- Solaris 9 and later
- OS/2

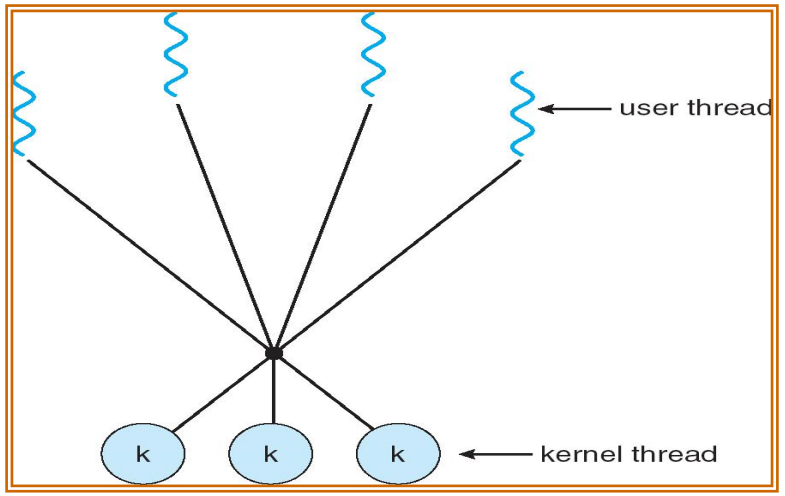
多对多模型

多个用户级线程映射为相等或小于数目的内核线程。

允许操作系统创建足够多的KLT。

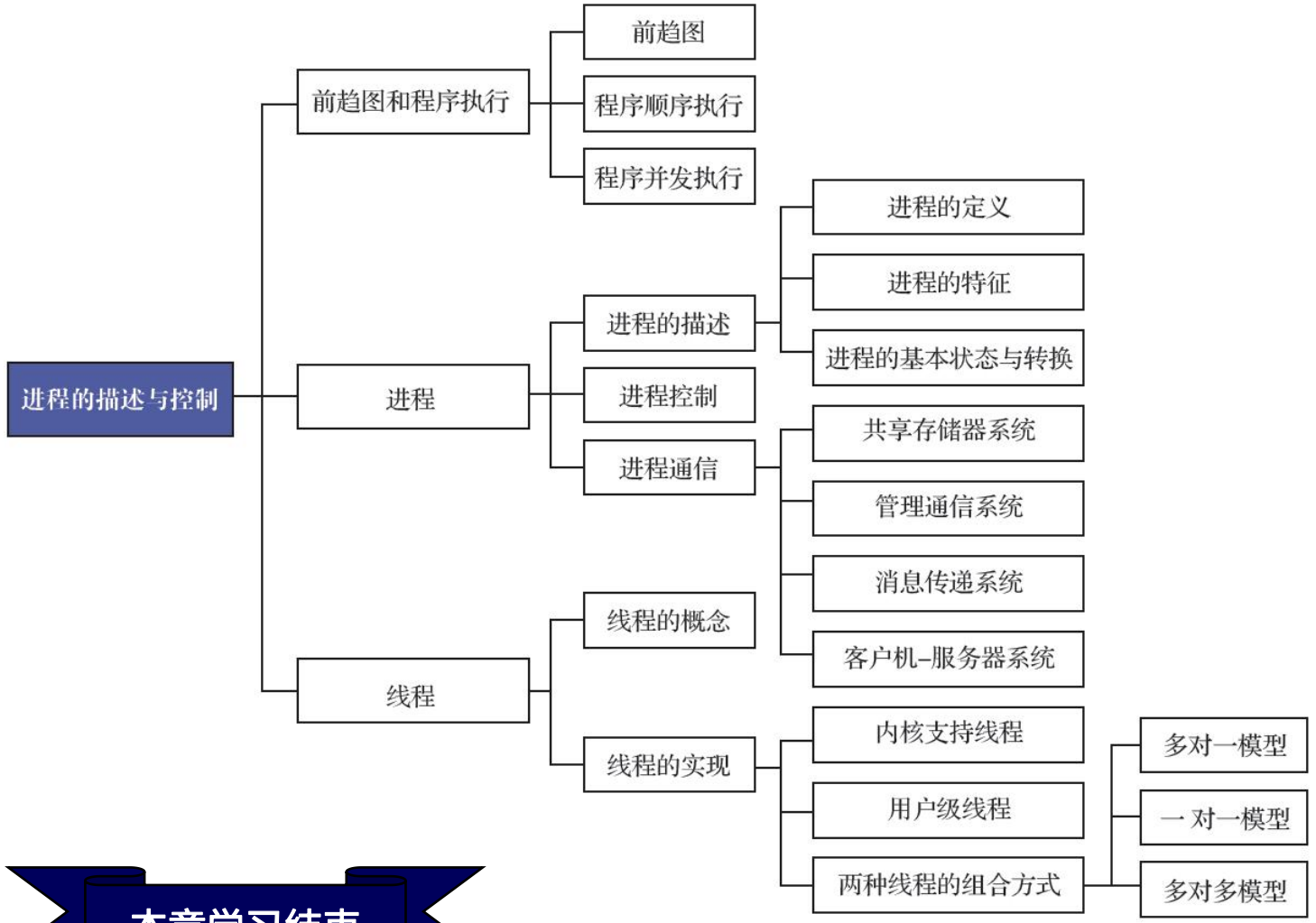
例子：

- Solaris 9 以前的版本；
- 带有ThreadFiber开发包的Windows NT/2000 。



学而时习之（第2章总结）

第1章	操作系统引论
第2章	**进程的描述与控制**
第3章	处理机调度与死锁
第4章	进程同步
第5章	存储器管理
第6章	虚拟存储器
第7章	输入/输出系统
第8章	文件管理
第9章	磁盘存储器管理
第10章	多处理机操作系统
第11章	虚拟化和云计算
第12章	保护和安全



本章学习结束

第三次作业

简答题

1 2 3 4 5 6 7 8

9 10 11 12 13 14 15 16

17 18 19 20

综合应用题

21 22

标黄色为本次作业

长期“冷战”之后

Linux与Windows究竟谁会更胜一筹？

20世纪90年代初期，基于MINIX和UNIX思想而研发的开源Linux系统面市，其是一款支持多用户、多任务、多线程和多内核的操作系统，不仅能够运行UNIX工具软件、应用程序和网络协议，还具有稳定的系统性能。发展至今，Linux已有上百种不同的发行版本。

在Linux系统稳步发展过程中，Windows系统亦不分昼夜地进行着功能完善、界面美化以及版本更新等工作。进入21世纪后，微软公司的Windows系统在个人计算机领域基本占领了垄断地位。由垄断所导致的潜在安全问题是各国相关部门尤为关心的核心问题，而解决该问题（即去微软公司化）的主流途径便是采用开源Linux系统。

长期“冷战”之后

Linux与Windows究竟谁会更胜一筹？

但是，要想简单地通过采用Linux系统实现去微软公司化，实属不易。

典型例子：2004年，德国慕尼黑政府宣布将政府办公计算机中所采用的Windows系统换为Linux系统，希望此举可以降低运维信息化成本；然而10年之后，这场“吃螃蟹”的试验并未获得预期的效果；近年来，该政府又开始逐步在政府办公计算机上重新安装Windows系统。

从此类案例中可以看出，单纯地通过采用Linux系统+开源软件的模式来降低运维信息化成本，效果并不理想，大都会陷入所须开发的专业软件多和后期维护工作量大等风险中。此外，由于Linux系统代码开源，其产品化始终不足，因此，采用Linux系统实现去微软公司化缺乏相应的产业基础。

长期“冷战”之后

Linux与Windows究竟谁会更胜一筹？

形势即便如此，我们也绝对不应放弃反垄断！在此情形下，我们知识分子与科技人才更应深度思考：究竟应当如何解决操作系统垄断问题，以及如何在解决该问题的过程中，通过发挥我们自身的价值，助力研发自主可控的国产操作系统？