



经典教材《计算机操作系统》**最新版**

# 第2章 进程的描述与控制

主讲教师：陆丽萍





## 第2章知识导图

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



# 再回来看Hello World

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

```
bash$ gcc hello.c -o hello
```

```
# 同时启动两个hello world程序
```

```
bash$ ./hello & ./hello
```

```
[1] 144
```

```
Hello World!
```

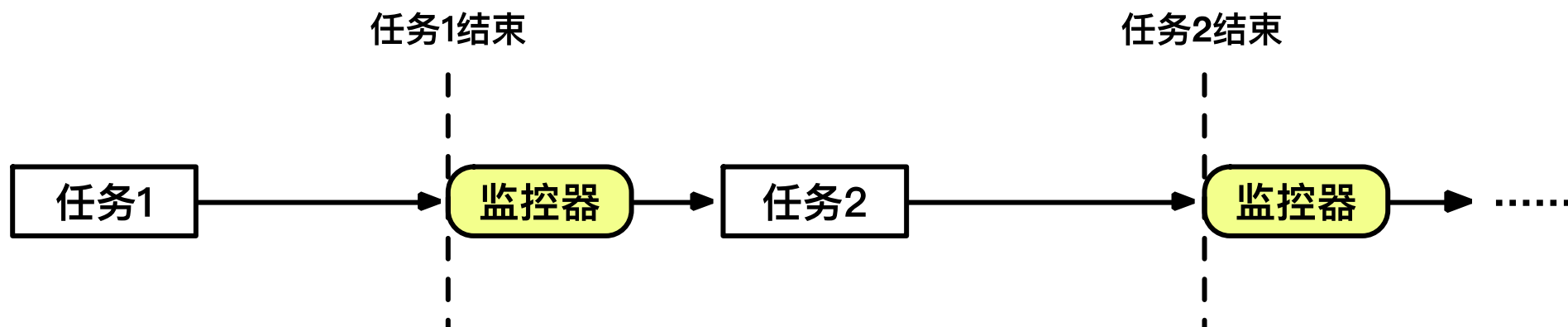
```
Hello World!
```

```
[1]+ Done      ./hello
```

## 运行多个hello时，操作系统怎么抽象与管理？

# 进程的诞生：从单任务到多任务

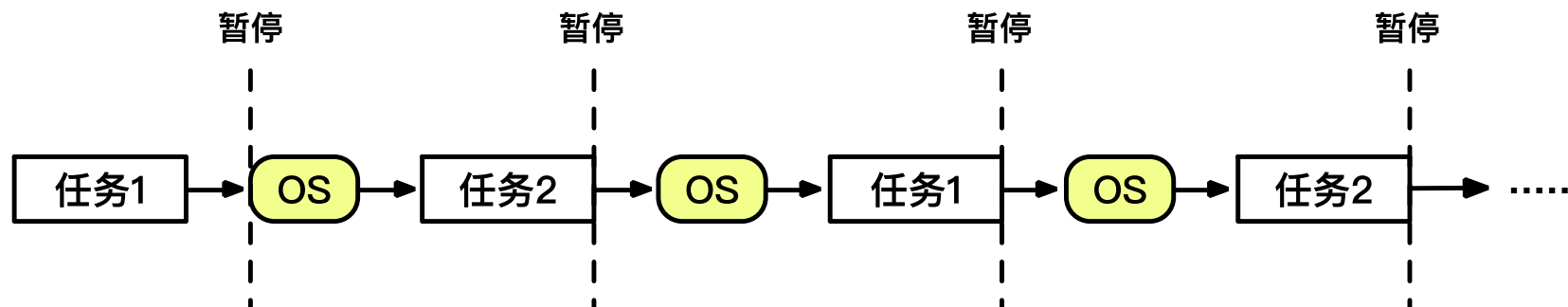
- 早期的计算机一次只能执行一个任务



- 计算机的发展趋势
  - 计算机程序种类越来越多（文本编辑、科学计算、web服务.....）
  - 外部设备种类越来越多（硬盘、显示器、网络），造成程序等待

# 进程的诞生：从单任务到多任务

- 思路：提出分时（time-sharing）操作系统
  - 多任务并行：当一个任务需要等待时，切换到其他任务



- 任务的抽象——**进程**
  - 进程的**执行状态**不断更新
  - 不断切换处理器上运行的进程（**上下文切换**）
  - 操作系统需要对进程进行**调度**（**且听下回分解**）

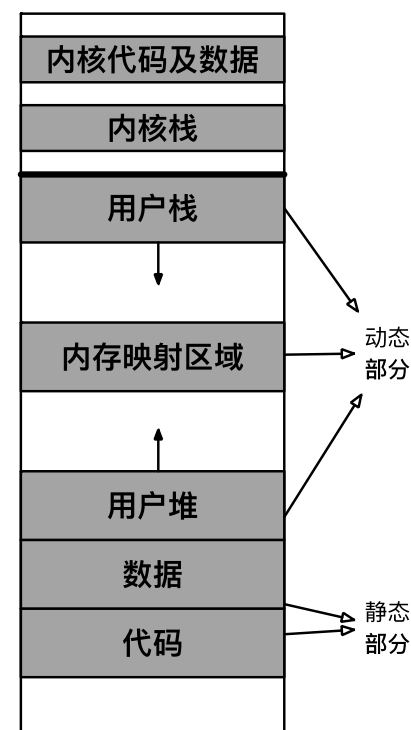
# 进程：运行中的程序

- 进程是计算机程序运行时的抽象

- 静态部分：程序运行需要的代码和数据
- 动态部分：程序运行期间的状态（程序计数器、堆、栈.....）

- 进程具有独立的虚拟地址空间

- 每个进程都具有“独占全部内存”的假象
- 内核中同样包含内核栈和内核代码、数据





## 内容导航:



2.1 前趋图和程序执行



2.2 进程的描述



2.3 进程控制



2.4 进程通信



2.5 线程的基本概念



2.6 线程的实现

## 第2章 进程的描述与控制

---



## 程序顺序执行

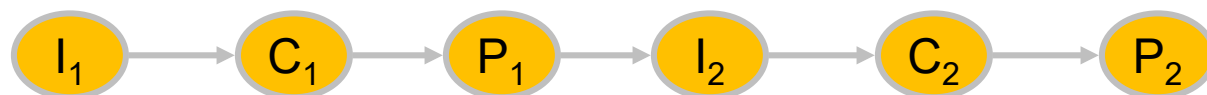
- 一个较大的程序通常都由若干个程序段组成

程序在执行时，必须按照某种先后次序逐个执行，仅当前一操作执行完后，才能执行后继操作。



## 前趋图

- 有向无循环图，用于描述进程之间执行的先后顺序
- 结点表示进程或程序段，有向边表示前趋关系



程序顺序执行时的前趋图

前趋关系： $I_i \rightarrow C_i \rightarrow P_i$

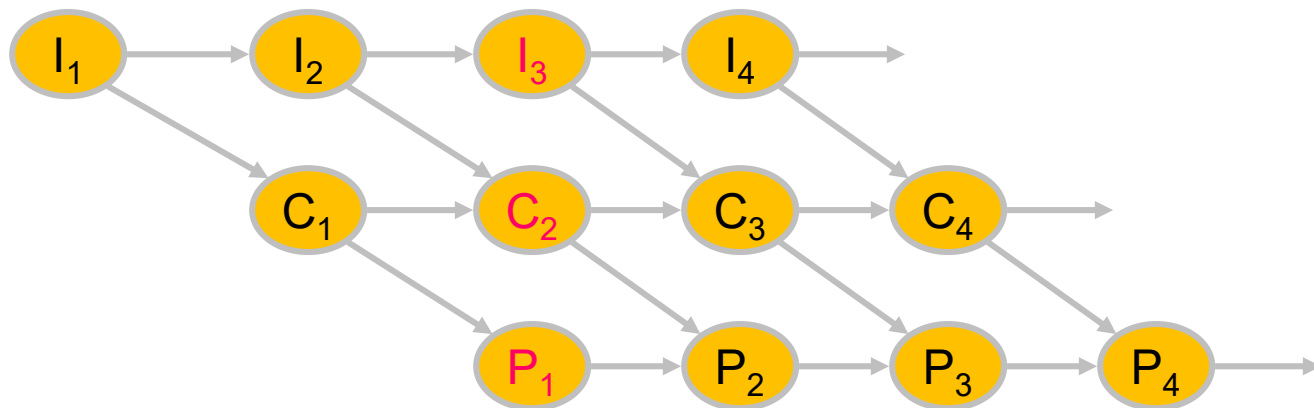
其中I代表输入操作，C代表计算操作，P代表打印操作





## 程序并发执行

- 采用多道程序技术，  
将多个程序同时装入  
内存，使之并发运行。



程序并发执行时的前趋图

前趋关系： $I_i \rightarrow C_i$ ,  $I_i \rightarrow I_{i+1}$ ,  $C_i \rightarrow P_i$ ,  $C_i \rightarrow C_{i+1}$ ,  $P_i \rightarrow P_{i+1}$

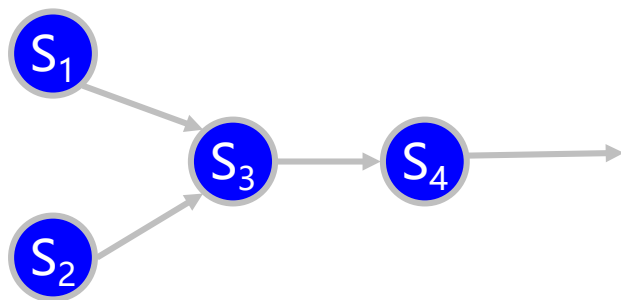
例：程序段如下：

$S_1$ :  $a := x + 2$

$S_2$ :  $b := y + 4$

$S_3$ :  $c := a + b$

$S_4$ :  $d := c + b$





## 间断性

- 并发程序之间相互制约。
- 执行——暂停执行——执行。



## 失去封闭性

- 多个程序共享全机资源。
- 执行状态受外界因素影响。



## 不可再现性

- 程序经过多次执行后，虽然其执行时的环境和初始条件都相同，但得到的结果却各不相同。例：两个循环程序共享一个变量。

**程序A**

...

 **$N := N + 1;$** 

...

**程序B**

...

 **$\text{print}(N);$**  **$N := 0;$** 

循环程序A和B共享一个变量N，他们以不同的速度运行。（假定某时刻变量N的值为n）。

- $N := N + 1$  在 $\text{print}(N)$ 和 $N := 0$ 之前，此时得到的N值分别为 $n+1, n+1, 0$
- $N := N + 1$  在 $\text{print}(N)$ 和 $N := 0$ 之后，此时得到的N值分别为 $n, 0, 1$
- $N := N + 1$  在 $\text{print}(N)$ 和 $N := 0$ 之间，此时得到的N值分别为 $n, n+1, 0$

**结论：失去了可再现性。**



## 内容导航:



2.1 前趋图和程序执行



2.2 进程的描述



2.3 进程控制



2.4 进程通信



2.5 线程的基本概念



2.6 线程的实现

## 第2章 进程的描述与控制

---



## 几种典型定义

- 进程是程序的一次执行。
- 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。
- 进程是程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。



## 进程定义：

- 进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。



## 进程控制块(process control block, PCB)

- 专门的数据结构，与进程一一对应。



# 进程例子: Suse Linux

```
Telnet 202.195.128.17
868 ?      02:23:36 oracle
870 ?      00:00:35 oracle
872 ?      00:00:01 oracle
874 ?      00:18:38 oracle
876 ?      00:17:29 oracle
878 ?      00:17:28 oracle
880 ?      00:17:22 oracle
882 ?      00:16:25 oracle
884 ?      00:17:05 oracle
886 ?      00:17:52 oracle
888 ?      00:17:19 oracle
890 ?      00:16:41 oracle
892 ?      00:17:54 oracle
913 ?      02:49:25 tnslnsr
985 ?      00:00:37 master
999 ?      00:00:00 atd
1014 ?     00:00:07 cron
1030 ?     00:01:57 nscd
1031 ?     00:00:42 nscd
1032 ?     00:01:55 nscd
1033 ?     00:01:48 nscd
1034 ?     00:01:45 nscd
1035 ?     00:01:48 nscd
1036 ?     00:01:48 nscd
--More--
```

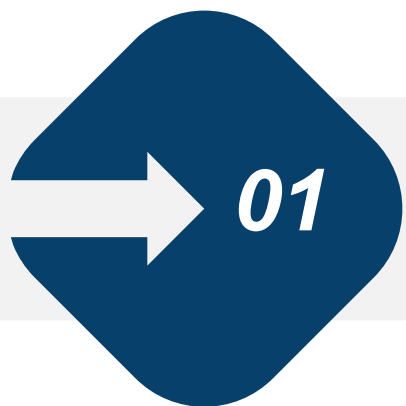


# 进程例子： Windows 10

任务管理器					
文件(F) 选项(O) 查看(V)					
进程 性能 应用历史记录 启动 用户 详细信息 服务					
名称	状态	13% CPU	50% 内存	0% 磁盘	0% 网络
> Google Chrome (17)		1.4%	206.7 MB	0.1 MB/秒	0 Mbps
> Microsoft PowerPoint		0%	174.9 MB	0 MB/秒	0 Mbps
> 腾讯QQ (32 位) (3)		1.1%	129.7 MB	0.1 MB/秒	0.1 Mbps
> Microsoft Word (3)		0.3%	76.0 MB	0 MB/秒	0 Mbps
> Clash for Windows (2)		0.6%	56.3 MB	0 MB/秒	0 Mbps
Clash for Windows		0.5%	51.3 MB	0 MB/秒	0 Mbps
360安全卫士 安全防护中心模...		0%	50.2 MB	0.1 MB/秒	0 Mbps
> Microsoft Excel		0%	49.9 MB	0 MB/秒	0 Mbps
桌面窗口管理器		0.5%	31.4 MB	0 MB/秒	0 Mbps
> Windows 资源管理器		0.3%	28.2 MB	0 MB/秒	0 Mbps
> 任务管理器		2.2%	22.3 MB	0.1 MB/秒	0 Mbps
360杀毒 实时监控		0.3%	19.5 MB	0 MB/秒	0 Mbps
> LocalServiceNoNetworkFire...		0%	14.2 MB	0 MB/秒	0 Mbps
> 服务主机: Diagnostic Policy S...		0%	13.7 MB	0 MB/秒	0 Mbps

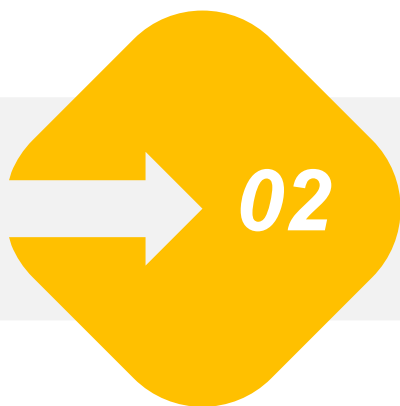
多个进程使用一个程序

## 动态性 (最基本的特征) (Concurrency)



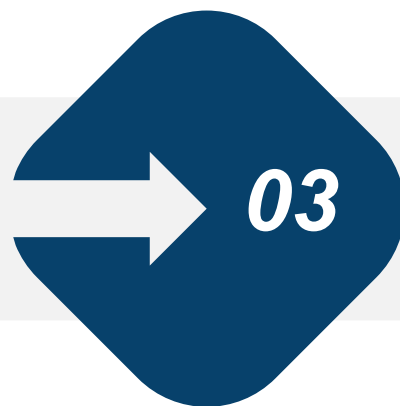
➤ 生命期  
进程由创建而产生，  
由调度而执行，由  
撤销而消亡

## 并发性



➤ 一段时间内同  
时运行

## 独立性



➤ 进程实体是一个能独立  
运行的基本单位  
➤ 是系统中独立获得资源  
和独立调度的基本单位

## 异步性



➤ 按各自独立的、  
不可预知的速度  
向前推进





# 进程和程序的区别

进程是程序的一个实例，  
是程序的一次执行。

程序是进程的代码部分。



进程是活动的，  
程序是静态的。

进程在内存中，  
程序在外存中。



## 就绪状态

- 一个较大的程序通常都由若干个程序段组成
- 程序在执行时，必须按照某种先后次序逐个执行，仅当前一操作执行完后，才能执行后继操作。



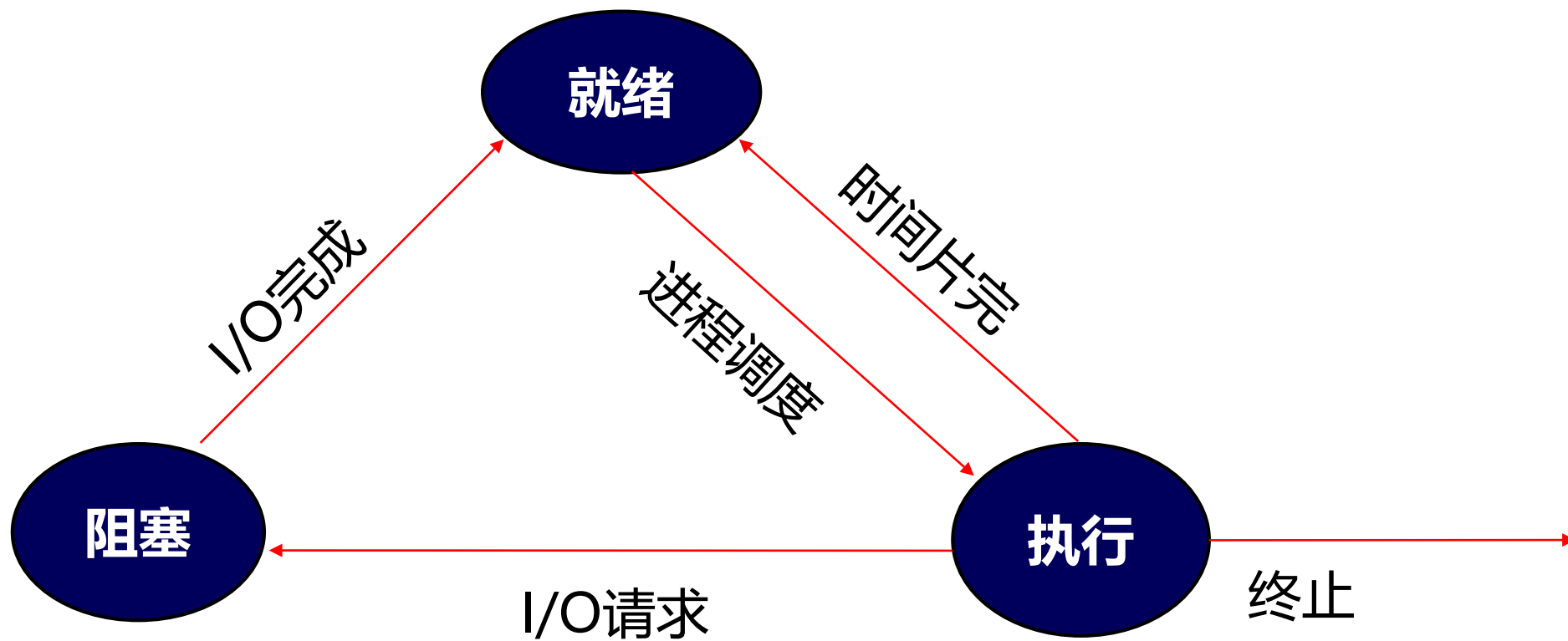
## 执行状态：已获得CPU，正在执行的状态

- 单处理机：一个进程处于执行状态
- 多处理机：多个进程处于执行状态



## 阻塞状态

- 正在执行的进程由于发生某事件而暂时无法继续执行的状态
- 典型事件：请求I/O、申请缓冲空间
- 根据阻塞原因，设置多个阻塞队列



## 01

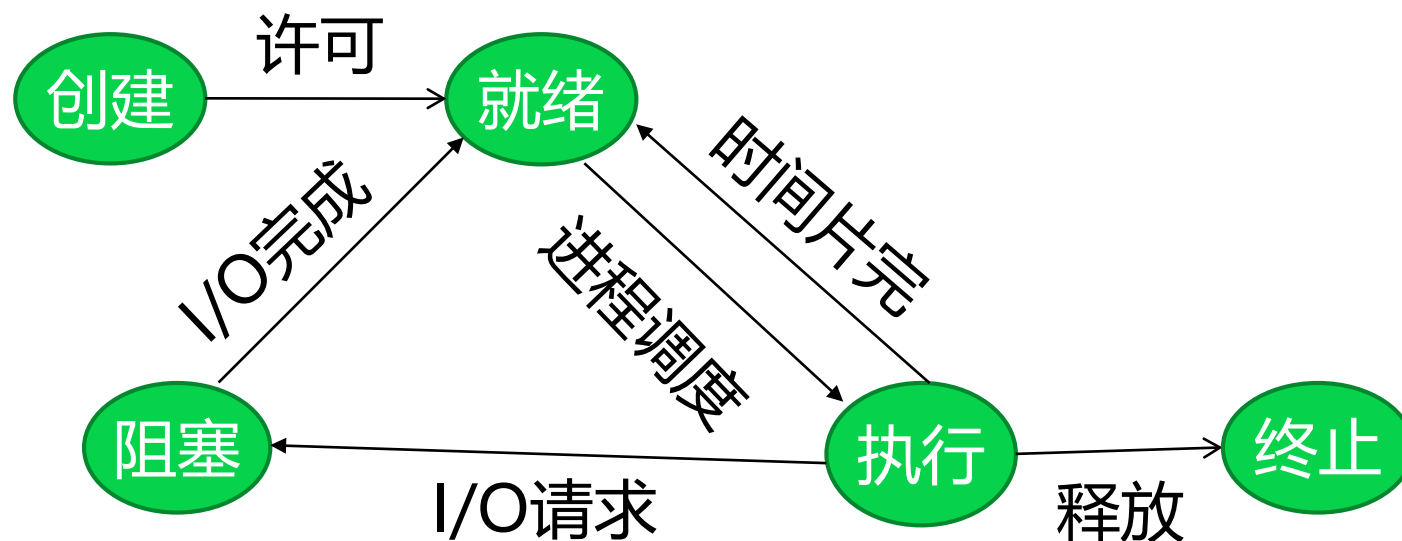
## 创建状态

- 申请一个空白PCB；填写PCB；分配资源；设置就绪状态插入就绪队列

## 02

## 终止状态

- 等待OS善后；
- 收回PCB





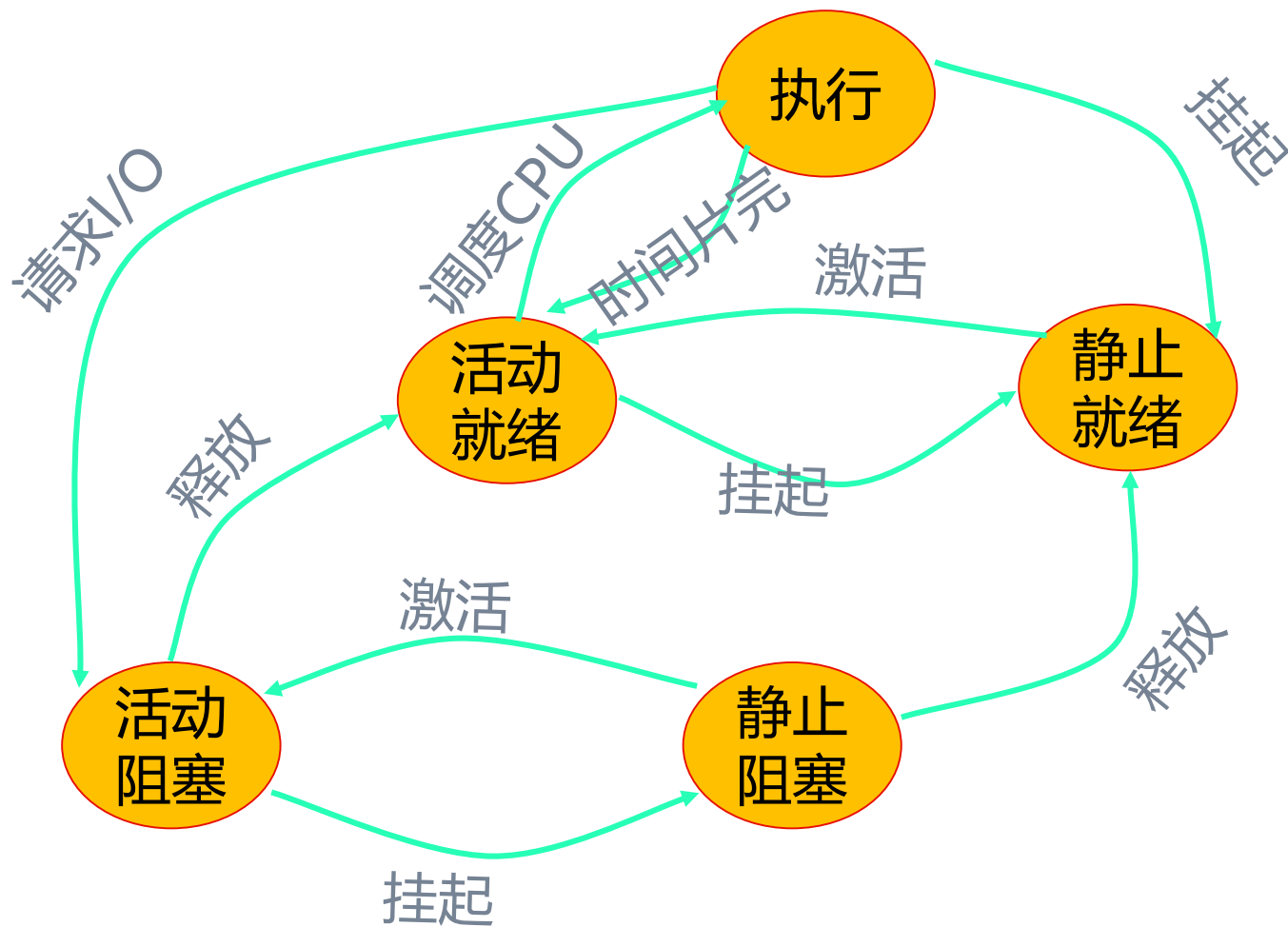
## 1. 挂起操作的引入

引入挂起操作的原因，是基于系统和用户的如下需要：

- (1) 终端用户的需要。
- (2) 父进程请求。
- (3) 负荷调节的需要。
- (4) 操作系统的需要。

在引入挂起原语Suspend和激活原语Active后，在它们的作用下，进程将可能发生以下几种状态的转换：

- (1) 活动就绪→静止就绪。
- (2) 活动阻塞→静止阻塞。
- (3) 静止就绪→活动就绪。
- (4) 静止阻塞→活动阻塞。

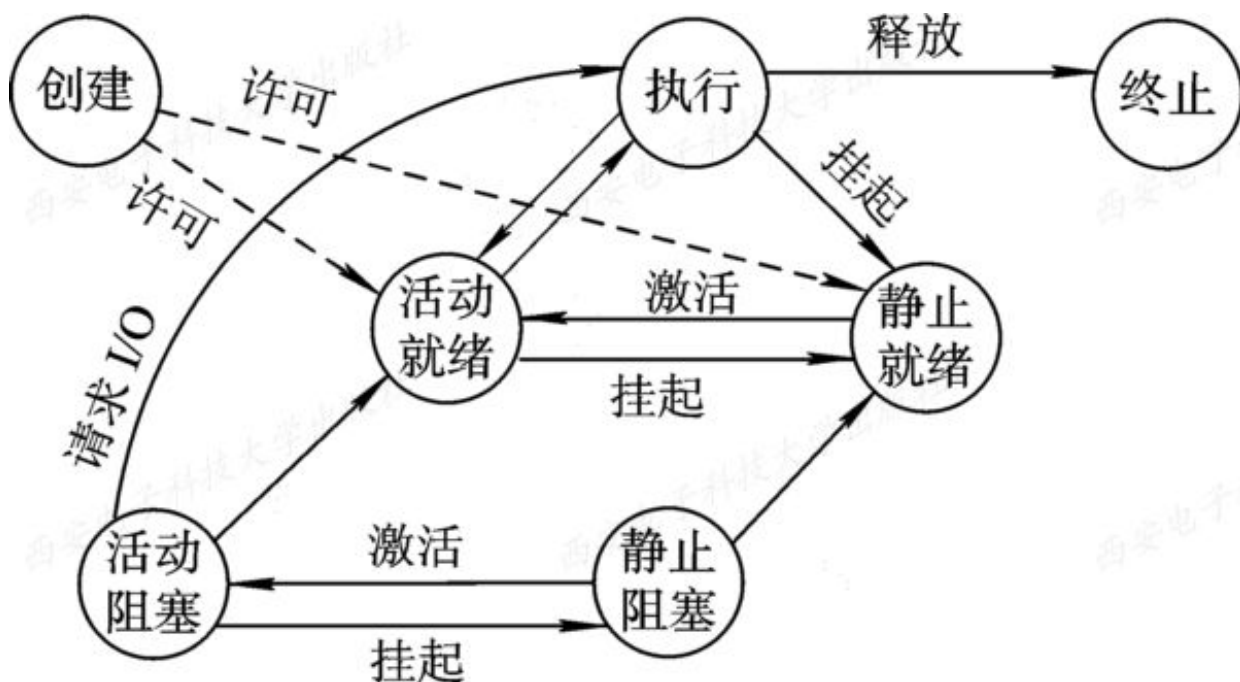


### 3. 引入挂起操作后五个进程状态的转换

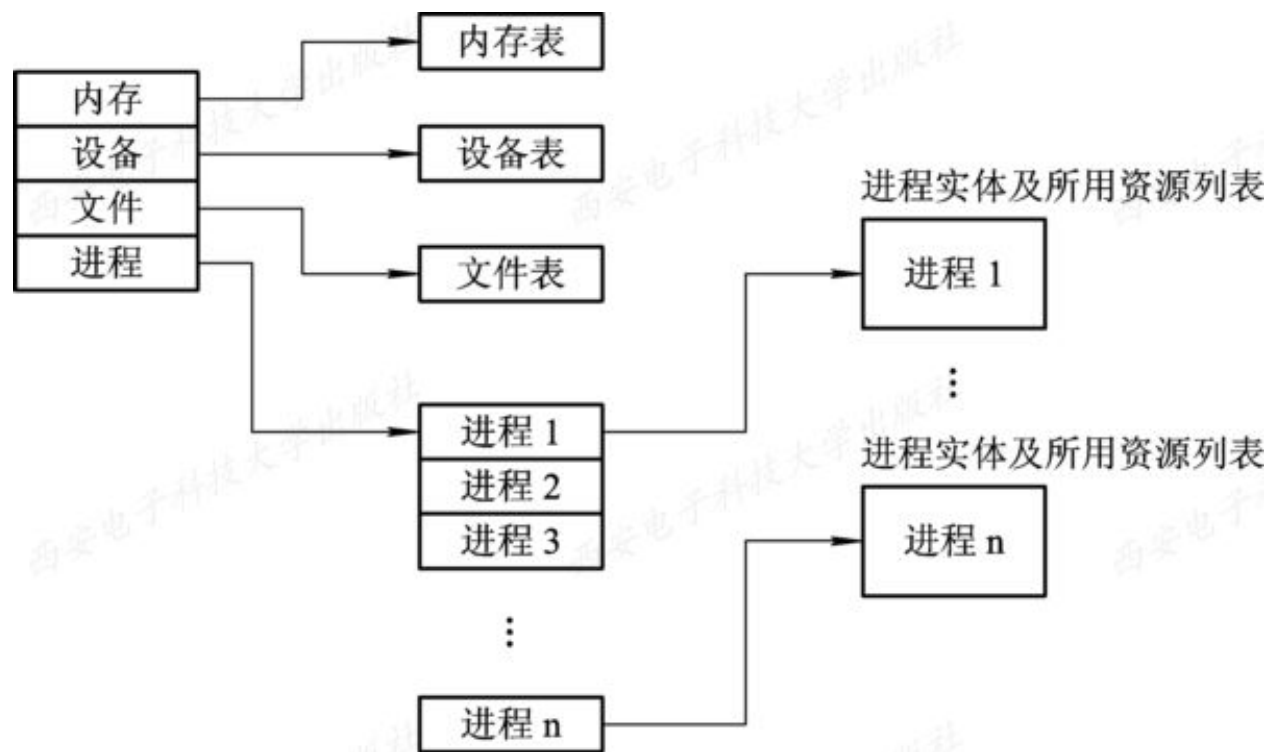
如图2-9示出了增加了创建状态和终止状态后具有挂起状态的进程状态及转换图。

增加考虑下面的几种情况：

- (1) NULL→创建：
- (2) 创建→活动就绪：
- (3) 创建→静止就绪：
- (4) 执行→终止：



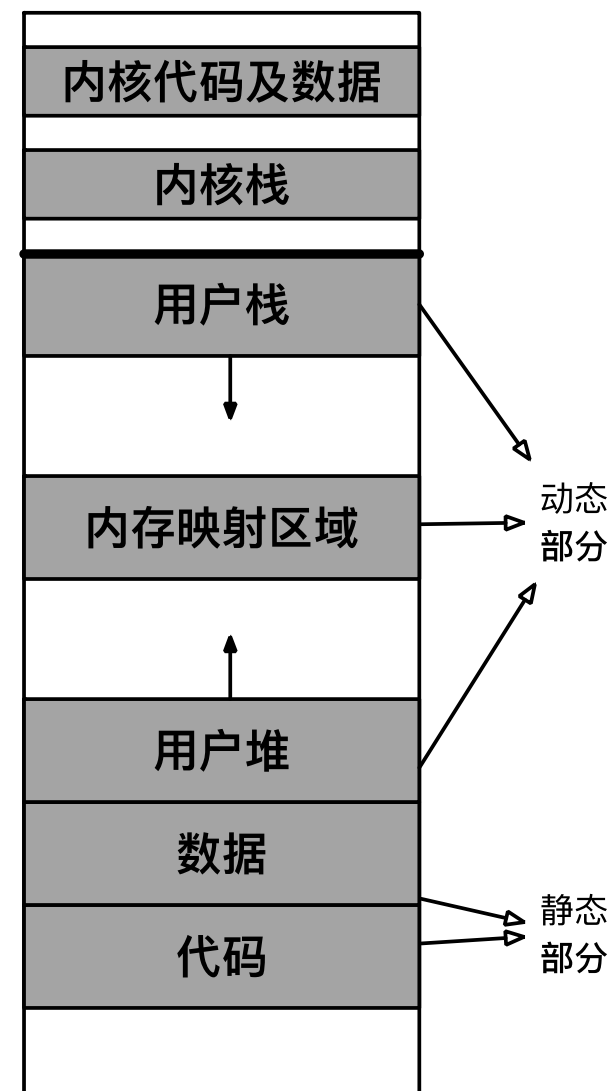
在计算机系统中，对于每个资源和每个进程都设置了一个数据结构，用于表征其实体，我们称之为资源信息表或进程信息表，其中包含了资源或进程的标识、描述、状态等信息以及一批指针。通过这些指针，可以将同类资源或进程的信息表，或者同一进程所占用的资源信息表分类链接成不同的队列，便于操作系统进行查找。





# 进程：运行中的程序

- 进程是计算机程序运行时的抽象
  - 静态部分：程序运行需要的代码和数据
  - 动态部分：程序运行期间的状态（程序计数器、堆、栈.....）
- 进程具有独立的虚拟地址空间
  - 每个进程都具有“独占全部内存”的假象
  - 内核中同样包含内核栈和内核代码、数据





PCB是进程的一部分，  
是操作系统中最重要的  
**记录型数据结构**，是进  
程存在的唯一标志，常  
驻内存。

- 存放进程相关的各种信息
  - 进程的标识符、内存、打开的文件.....
  - 进程在切换时的状态（上下文context）

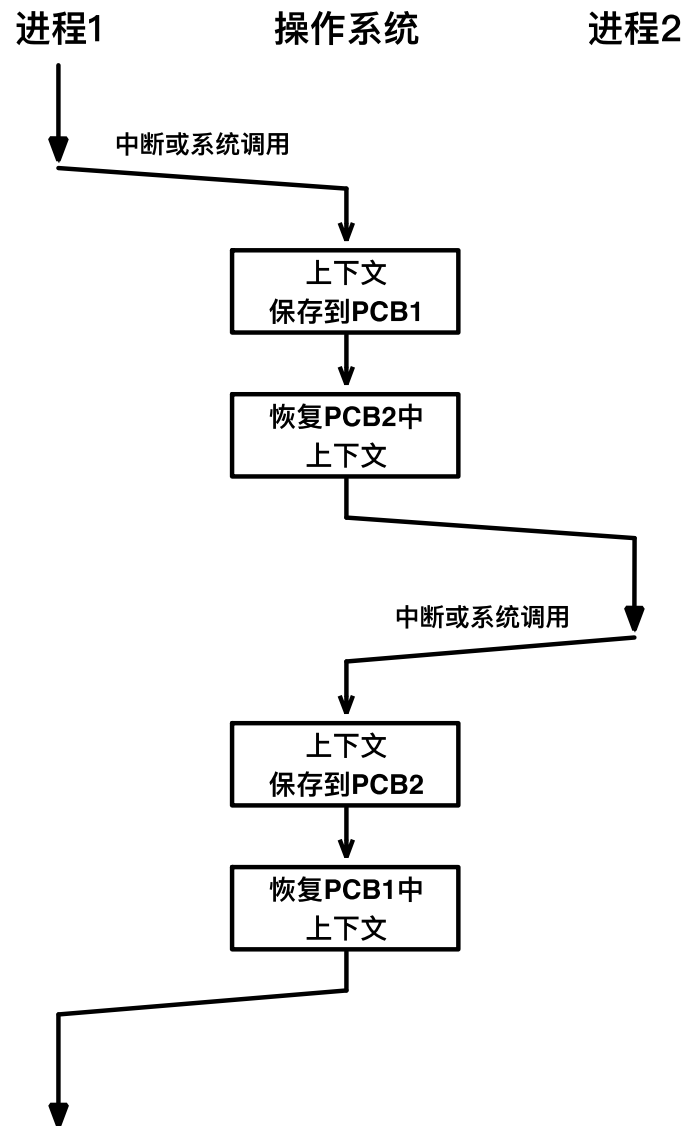
```
label_t u_qsav;      /* label variable for quits and interrupts */
label_t u_ssav;      /* label variable for swapping */
int u_signal[NSIG];  /* disposition of signals */
time_t u_ftime;      /* this process user time */
time_t u_stime;      /* this process system time */
time_t u_cutime;     /* sum of chlds' utimes */
time_t u_cstime;     /* sum of chlds' stimes */
int *u_ar0;          /* address of users saved R0 */
```

UNIX v7的部分PCB（u\_ar0为上下文信息）



## PCB的作用:

- 作为独立运行基本单位的标志;
- 能实现间断性运行方式;
- 提供进程管理所需要的信息;
- 提供进程调度所需要的信息;
- 实现与其他进程的同步与通信。



## PCB的信息



- 进程标识符
- 处理机状态
- 进程调度信息
- 进程控制信息

### 3. 进程控制块中的信息

在进程控制块中，主要包括下述四个方面的信息。

#### 1) 进程标识符

进程标识符用于唯一地标识一个进程。一个进程通常有两种标识符：

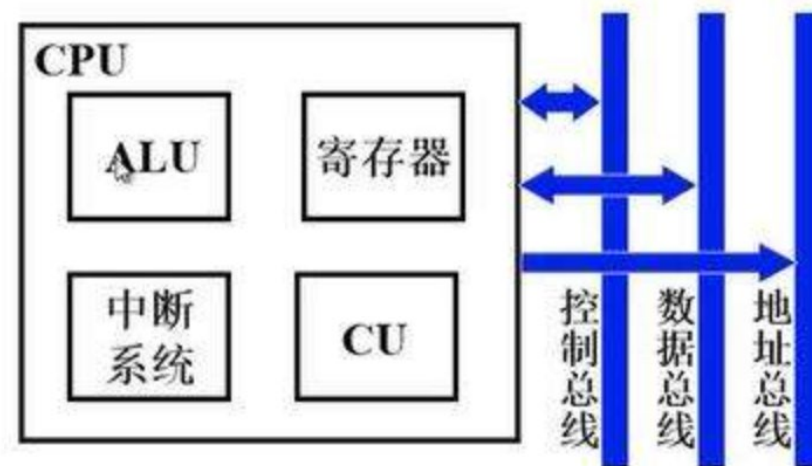
(1) 外部标识符。

(2) 内部标识符。

## 2) 处理机状态

处理机状态信息也称为处理机的上下文，  
主要是由处理机的各种寄存器中的内容组成的。

指令控制	PC	IR
操作控制	}	CU 时序电路
时间控制		
数据加工		ALU 寄存器
处理中断		中断系统



### 3) 进程调度信息

在OS进行调度时，必须了解进程的状态及有关进程调度的信息，这些信息包括：① 进程状态，指明进程的当前状态，它是作为进程调度和对换时的依据；② 进程优先级，是用于描述进程使用处理机的优先级别的一个整数，优先级高的进程应优先获得处理机；③ 进程调度所需的其它信息，它们与所采用的进程调度算法有关，比如，进程已等待CPU的时间总和、进程已执行的时间总和等；④ 事件，是指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。

#### 4) 进程控制信息

是指用于进程控制所必须的信息，它包括：① 程序和数据的地址，进程实体中的程序和数据内存或外存地址(首址)，以便再调度到该进程执行时，能从PCB中找到其程序和数据；② 进程同步和通信机制，这是实现进程同步和进程通信时必需的机制，如消息队列指针、信号量等，它们可能全部或部分地放在PCB中；③ 资源清单，在该清单中列出了进程在运行期间所需的全部资源(除CPU以外)，另外还有一张已分配到该进程的资源清单；④ 链接指针，它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。





## PCB的组织方式

### ➤ 线性方式

- 即将系统中所有的PCB都组织在一张线性表中，将该表的首址存放在内存的一个专用区域中。该方式实现简单、开销小，但每次查找时都需要扫描整张表，因此适合进程数目不多的系统。

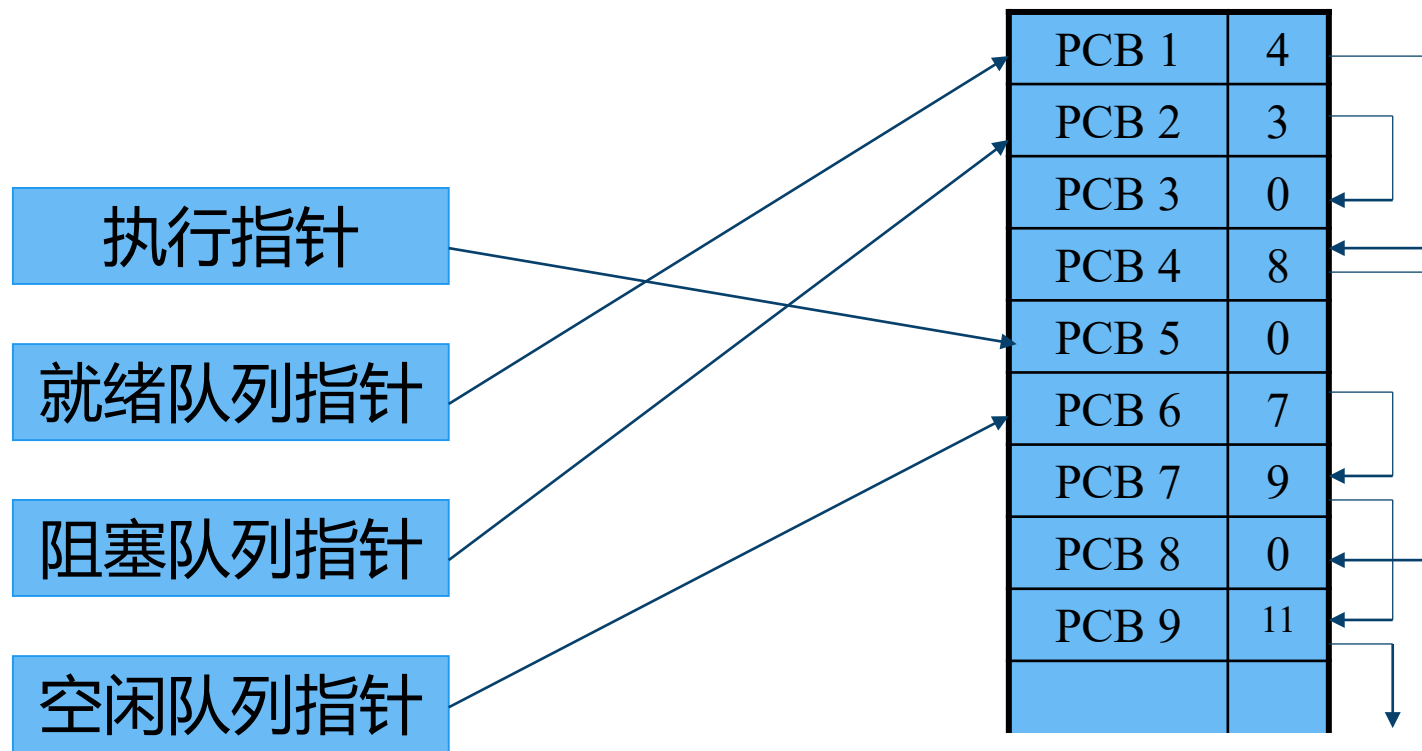
PCB 1
PCB 2
PCB 3
·
·
·
PCB $n$



## PCB的组织方式

### ➤ 链接方式

即把具有相同状态进程的PCB分别通过PCB中的链接字链接成一个队列。这样，可以形成就绪队列、若干个阻塞队列和空白队列等。

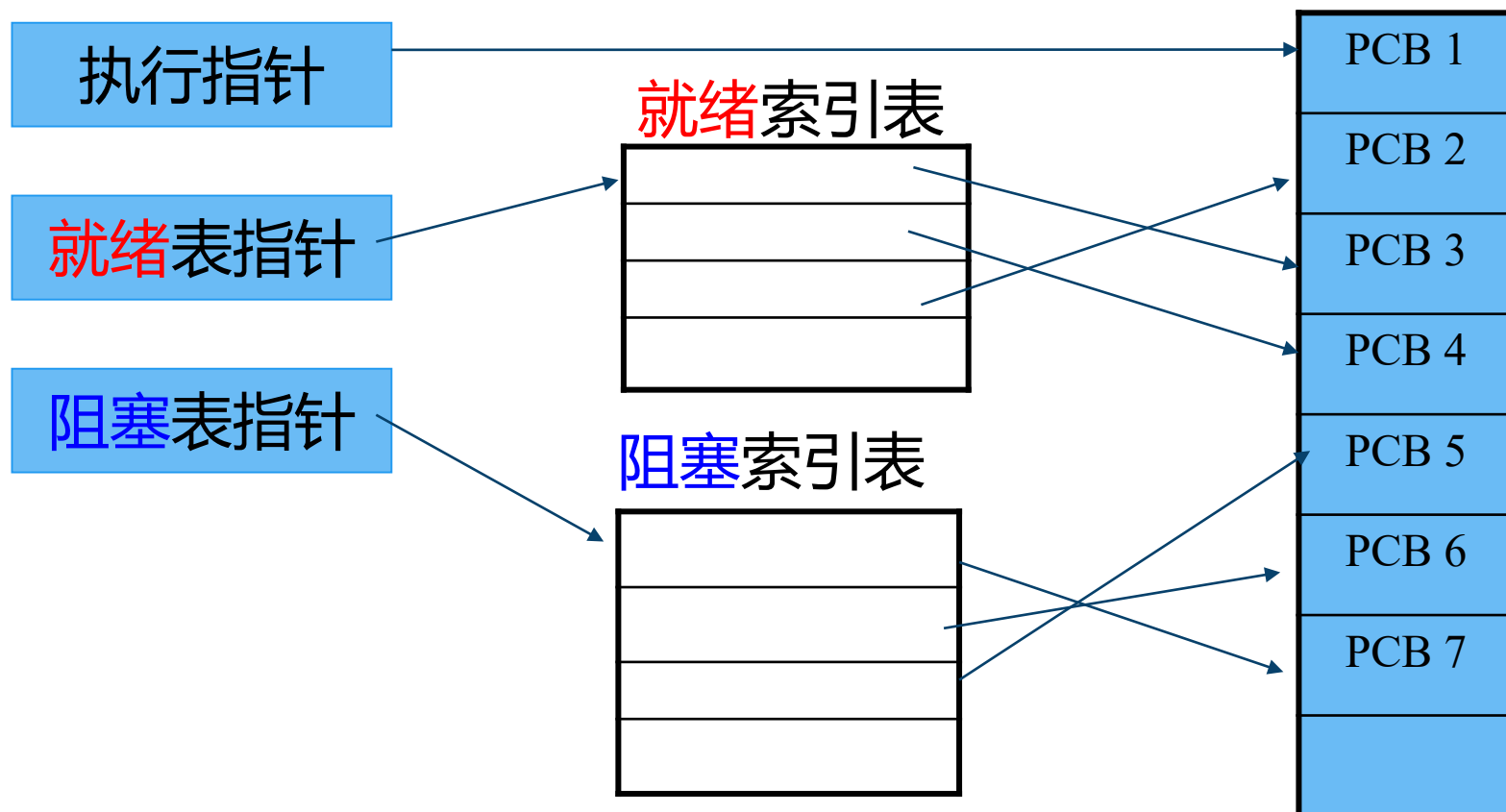




## PCB的组织方式

### ➤ 索引方式

即系统根据所有进程状态的不同，建立几张索引表，例如，就绪索引表、阻塞索引表等，并把各索引表在内存的首地址记录在内存的一些专用单元中。





## 内容导航:



2.1 前趋图和程序执行



2.2 进程的描述



2.3 进程控制



2.4 进程通信



2.5 线程的基本概念

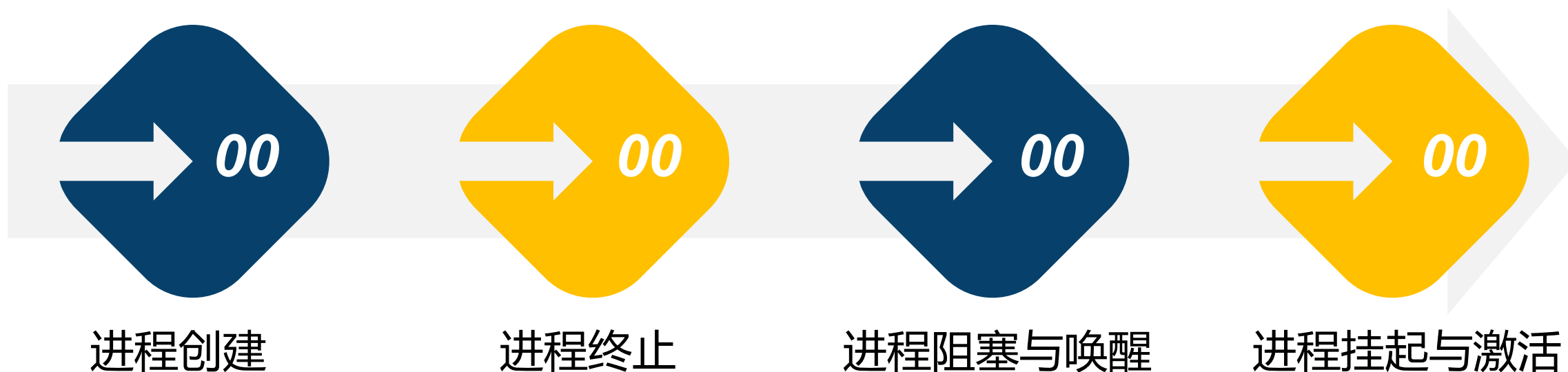


2.6 线程的实现

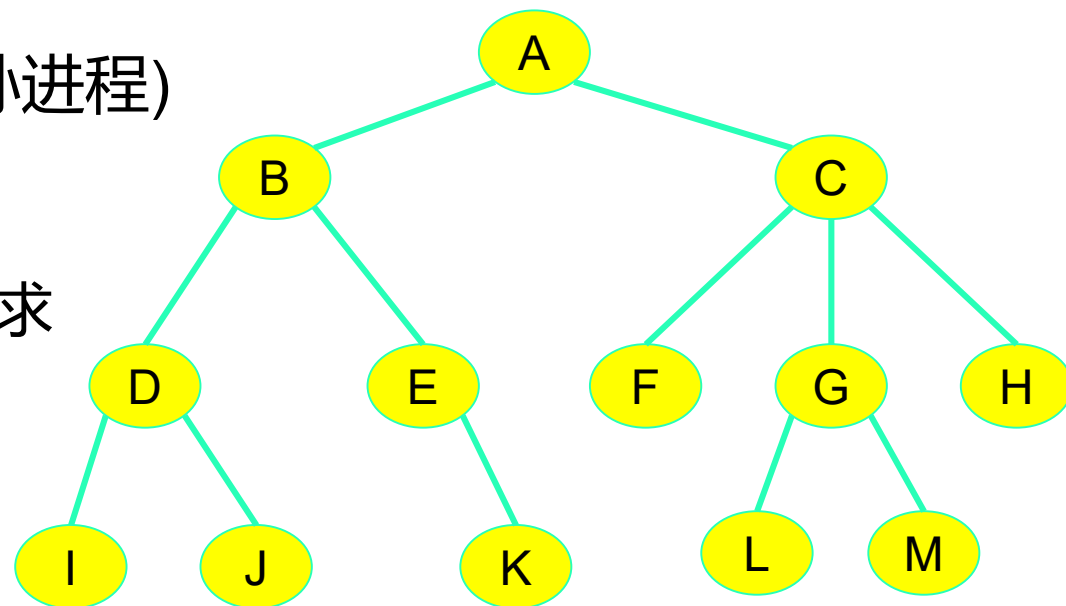
## 第2章 进程的描述与控制

---

- 进程管理最基本的功能
- 一般由OS内核中的原语实现
- 包括：



- 进程具有层次结构 (父进程、子进程、孙进程)
- 引起进程创建的事件
  - 用户登录、作业调度、提供服务、应用请求
- 进程图
  - 描述进程家族关系的有向树

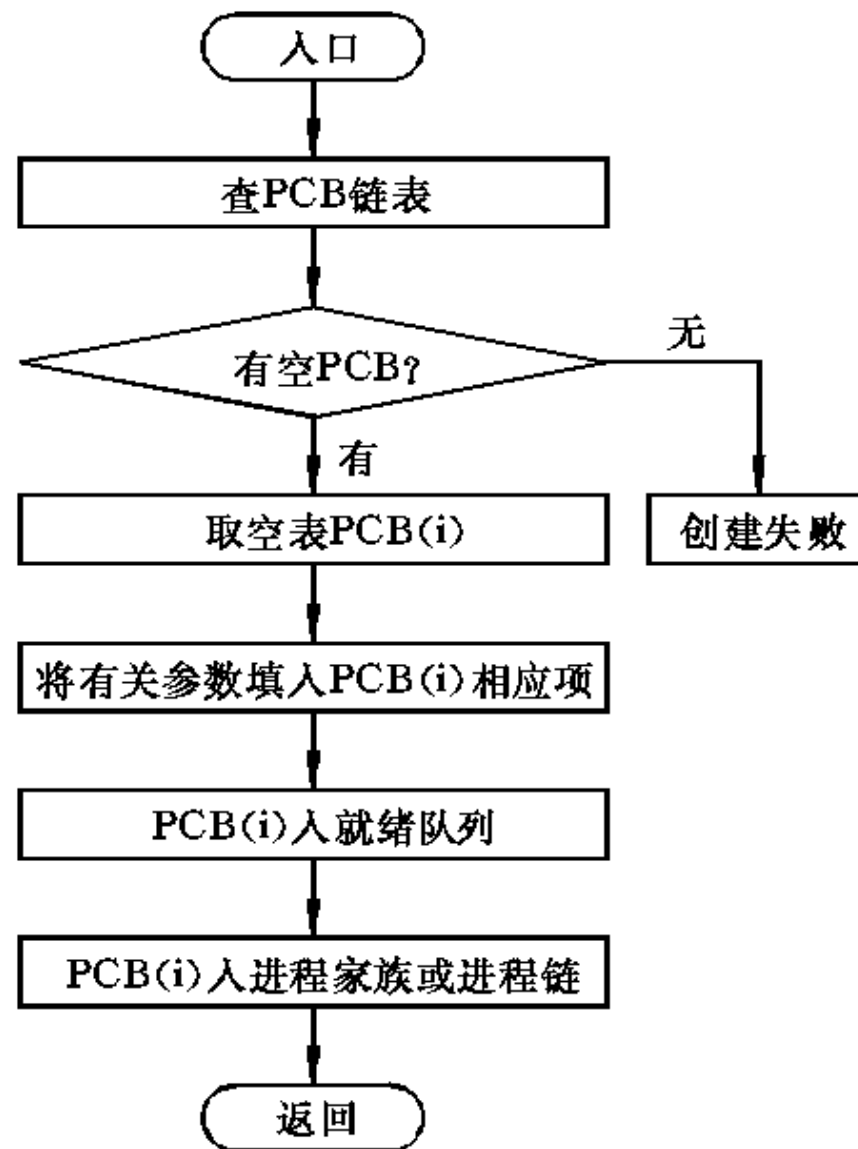


### ■ 进程创建过程:

- ① 申请空白PCB;
- ② 分配所需资源; 包括各种物理和逻辑资源, 如内存、文件、I/O设备和CPU时间等。
- ③ 初始化PCB;
- ④ 插入就绪队列。

## ■ 进程创建过程:

- ① 申请空白PCB;
- ② 分配所需资源; 包括各种物理和逻辑资源, 如内存、文件、I/O设备和CPU时间等。
- ③ 初始化PCB;
- ④ 插入就绪队列。



# 进程创建：fork()

- **语义：为调用进程创建一个一模一样的新进程**
  - 调用进程为**父进程**，新进程为**子进程**
  - 接口简单，无需任何参数
- **fork后的两个进程均为独立进程**
  - 拥有不同的进程id
  - 可以并行执行，互不干扰（除非使用特定的接口）
  - 父进程和子进程会共享部分数据结构（内存、文件等）



# fork的示例

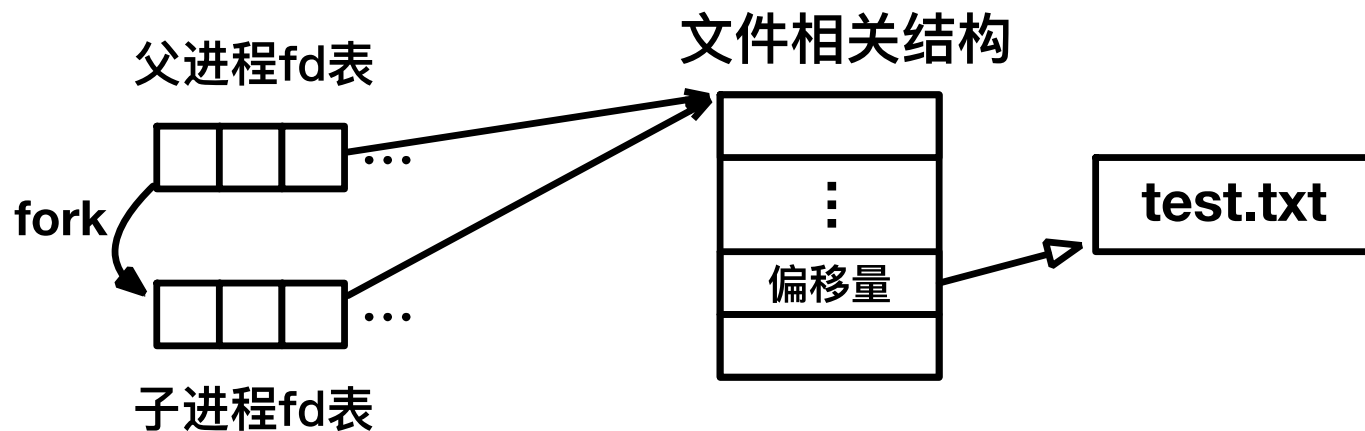
- 考虑以下代码

- 假设先执行父进程，然后再执行子进程
- test.txt中的内容: "abcdefghijklmnopqrst..."

```
char str[10];
int fd = open("test.txt", O_RDWR);
if (fork() == 0) {
    ssize_t cnt = read(fd, str, 10);
    printf("Child process: %s\n", (char *)str);
} else {
    ssize_t cnt = read(fd, str, 10);
    printf("Parent process: %s\n", (char *)str);
}
```

# fork的示例

- 执行结果(如果parent先执行):
  - Parent process: abcdefghij
  - Child process: klmnopqrst
  - 原因: 两个进程共享了同一个指向文件的结构体



# 进程的执行: exec

- 为进程指定可执行文件和参数

可执行文件位置

运行参数

```
int execve(const char *pathname, char *const argv[],  
           char *const envp[]);
```

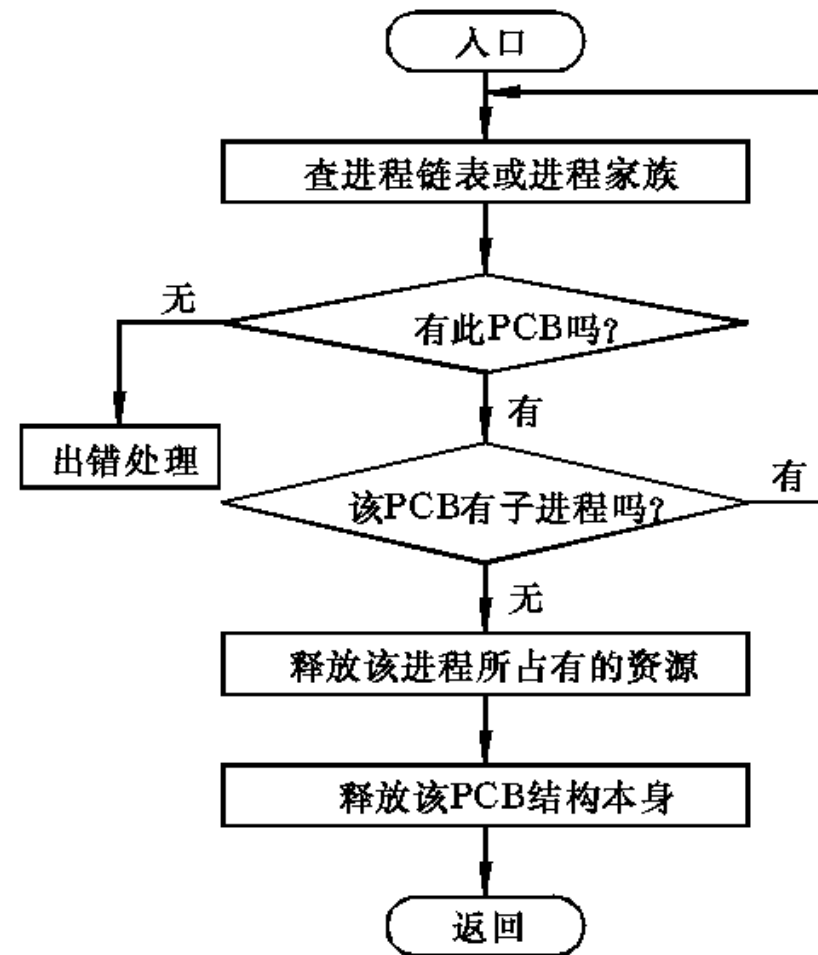
环境变量

- 在fork之后调用
  - exec在载入可执行文件后会重置地址空间

引起进程终止的事件：正常结束、异常结束、外界干预

进程的终止过程：

- 1 根据被终止进程的标识符，从PCB集合中检索出该进程的PCB，从中读出该进程的状态；
- 2 若被终止进程正处于执行状态，应立即终止该进程的执行，并设置调度标志为真，用于指示该进程被终止后应重新进行调度；
- 3 若该进程还有子孙进程，还应将其所有子进程予以终止；
- 4 将该进程所拥有的全部资源，或者归还给其父进程或系统；
- 5 将被终止进程（PCB）从所在队列中移去。





## 引起进程阻塞和唤醒的事件

- 向系统请求共享资源失败；等待某种操作的完成；新数据尚未到达；等待新任务的到达。



## 进程阻塞过程

- 阻塞原语Block()。
- 进程的阻塞是进程自身的一种主动行为。
- 具体过程：停止执行；状态由执行改为阻塞；将PCB插入阻塞队列。

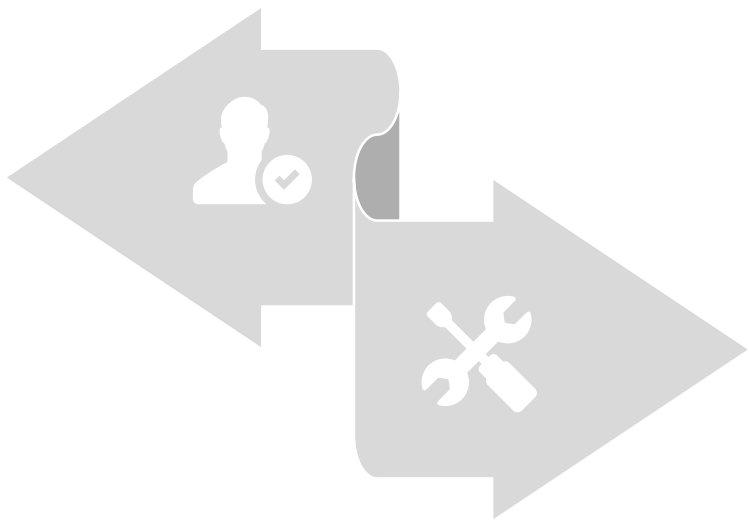


## 进程唤醒过程

- 唤醒原语Wakeup()。
- 具体过程：从阻塞队列中移出；状态由阻塞改为就绪；将PCB插入就绪队列。
- 必须成对使用Block和Wakeup原语。

## 进程的挂起

- Suspend()原语
- 执行过程



## 进程的激活过程

- Active()原语
- 执行过程

在一个单CPU的多道程序设计系统中，若在某时刻有 $N$ 个进程同时存在，那么处于运行态、阻塞态和就绪态进程个数的最小值和最大值分别可能是多少？

各状态最值如表所示。

只要有一个就绪态，CPU就不会空闲，就会有运行态进程，因此，就绪态最大值为 $N-1$ 。可能会存在所有进程均阻塞的情况。

	最小值	最大值
运行态	0	1
阻塞态	0	$N$
就绪态	0	$N-1$

代码片段 5-1 接收输入的问候程序

```
1 #include <stdio.h>
2 #define LEN 10
3
4 int main(int argc, char *argv[])
5 {
6     char name[LEN] = {0};
7     fgets(name, LEN, stdin);
8     printf("Hello %s\n", name);
9     return 0;
10 }
```



假设小明将该程序编译成了可执行文件`hello-name`，然后在Shell中使用`./hello-name`命令运行该程序。在程序运行的过程中，进程的状态会发生几次变化。

- 1.当小明按下回车后，Shell会接收到该命令，并请求内核创建相应的进程以处理命令。当内核创建出新进程时，该进程尚未初始化完成，处于**创建**状态。
- 2.内核会对进程需要的数据结构进行初始化，并将其交给调度器，加入运行队列，使其变为**就绪**状态。
- 3.调度器选择该进程执行，此时进程变为**执行**状态，可以开始执行main函数。
- 4.进程执行到fgets(第7行)，需要接收用户输入，此时进程变为**阻塞**状态，等待小明输入(此时进程并不在运行队列中)。
- 5.小明在键盘上输入“Xiaoming”并回车，此时进程会重新回到**执行**状态，并输出“Hello Xiaoming”。
6. 进程执行完main函数，回到内核中，变为**终止**状态，内核会回收该进程的资源。



## 内容导航:



2.1 前趋图和程序执行



2.2 进程的描述



2.3 进程控制



2.4 进程通信



2.5 线程的基本概念



2.6 线程的实现

# 第2章 进程的描述与控制

---

主讲: 王红玲 主审: 汤小丹

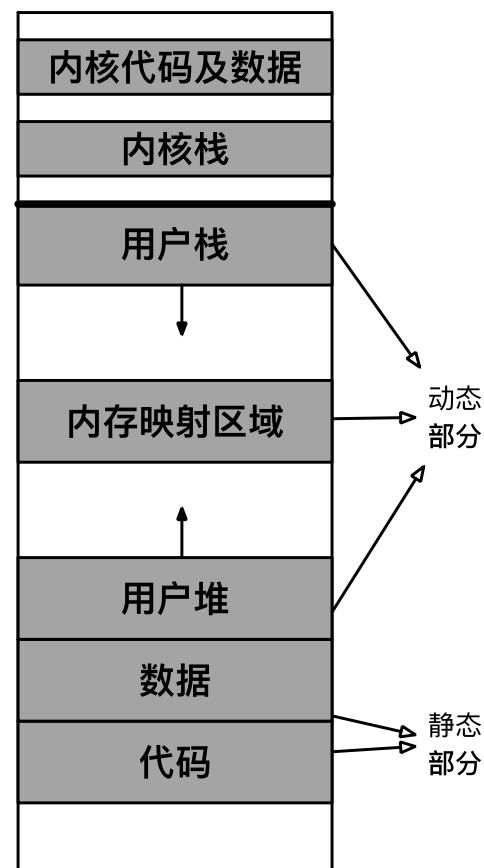
# 回顾: 进程

- **进程是计算机程序运行时的抽象**

- 静态部分: 程序运行需要的代码和数据
- 动态部分: 程序运行期间的状态  
(程序计数器、堆、栈.....)

- **进程具有独立的虚拟地址空间**

- 每个进程都具有"独占全部内存"的假象
- 内核中同样包含内核栈和内核代码、数据



# 应用程序的功能非常复杂

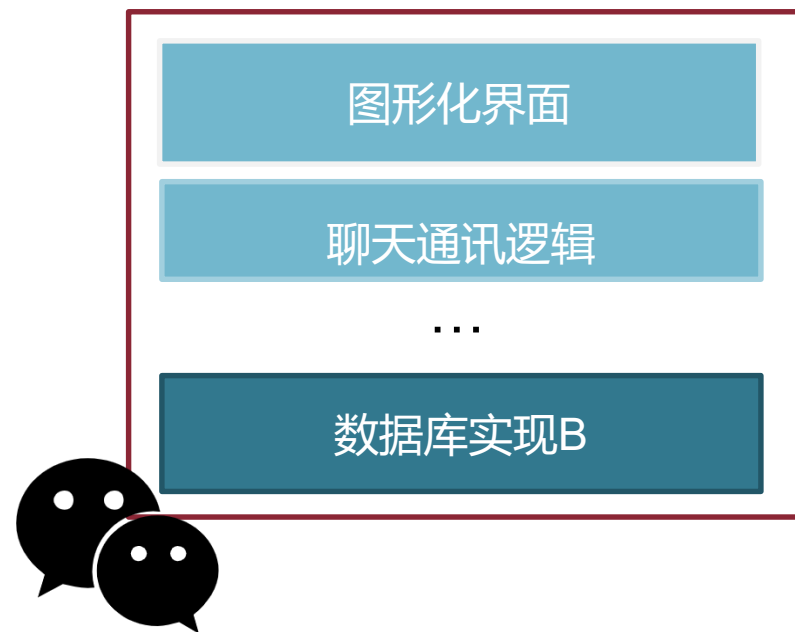
- 独立进程: 一个进程就是一个应用
  - 不会去影响其他进程的执行, 也不会被其他进程影响

单个应用的功能非常复杂



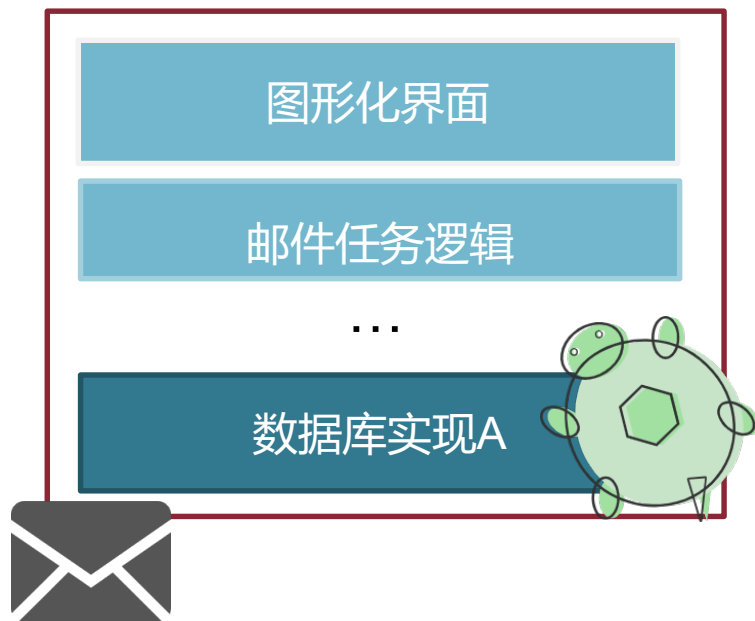
# 独立进程的问题1: 大量重复实现

- 聊天软件和邮件软件都依赖数据库
- 各自实现一份在自己的进程中



## 独立进程的问题2: 低效实现

- 聊天软件的数据库实现经过精心的优化
- 邮件软件团队的开发重心在其他组件上
  - 借用低效的某数据库开源实现

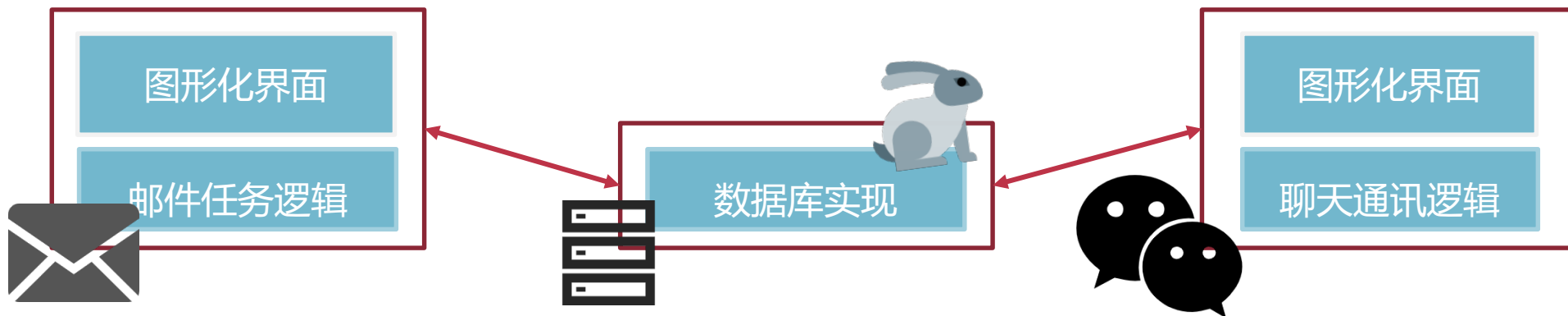


## 独立进程的问题3: 没有信息共享

- 邮件和聊天软件都需要监控系统资源信息
- 没有信息共享
  - 即使邮件软件已经完成了计算，聊天软件也要重新计算一遍

# 如果进程可以协作

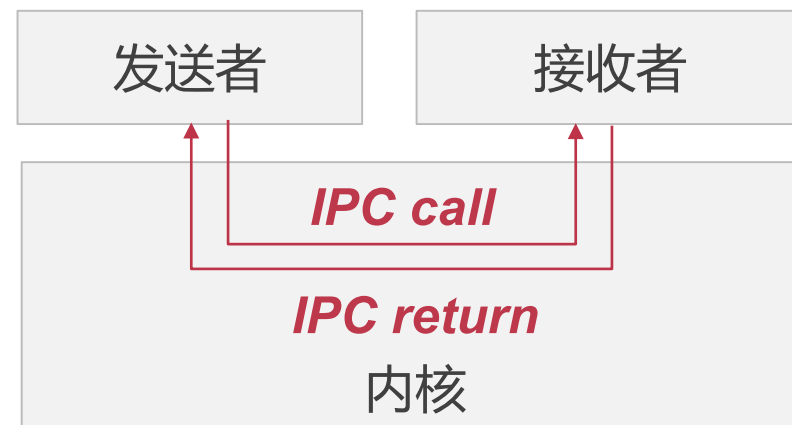
- 协作进程
  - 和独立进程相反，可以影响其他进程执行或者被影响
- 好处
  - 模块化: 数据库单独在一个进程中，可以被复用
  - 加速计算: 不同进程专注于特定的计算任务，性能更好
  - 信息共享: 直接共享已经计算好的数据，避免重复计算





# 进程间通信 (Inter-process Communication, IPC)

- 进程协作的达成依赖于进程间通信
- 进程间通信: 两个(或多个)不同的进程, 通过内核或其他共享资源进行通信, 来传递控制信息或数据
  - 交互的双方: 发送者/接收者、客户端/服务端、调用者/被调用者
  - 通信的内容一般叫做“消息”





# 新冠疫情下快递员送货问题

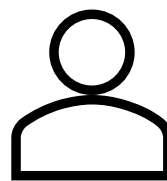
- 快递员不能送快递到家门口
- 小明的手机被没收了
- 快递员和小明怎么联络呢?
- 快递员问题→进程间通信问题



快递员



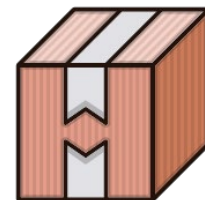
发送者



小明



接收者



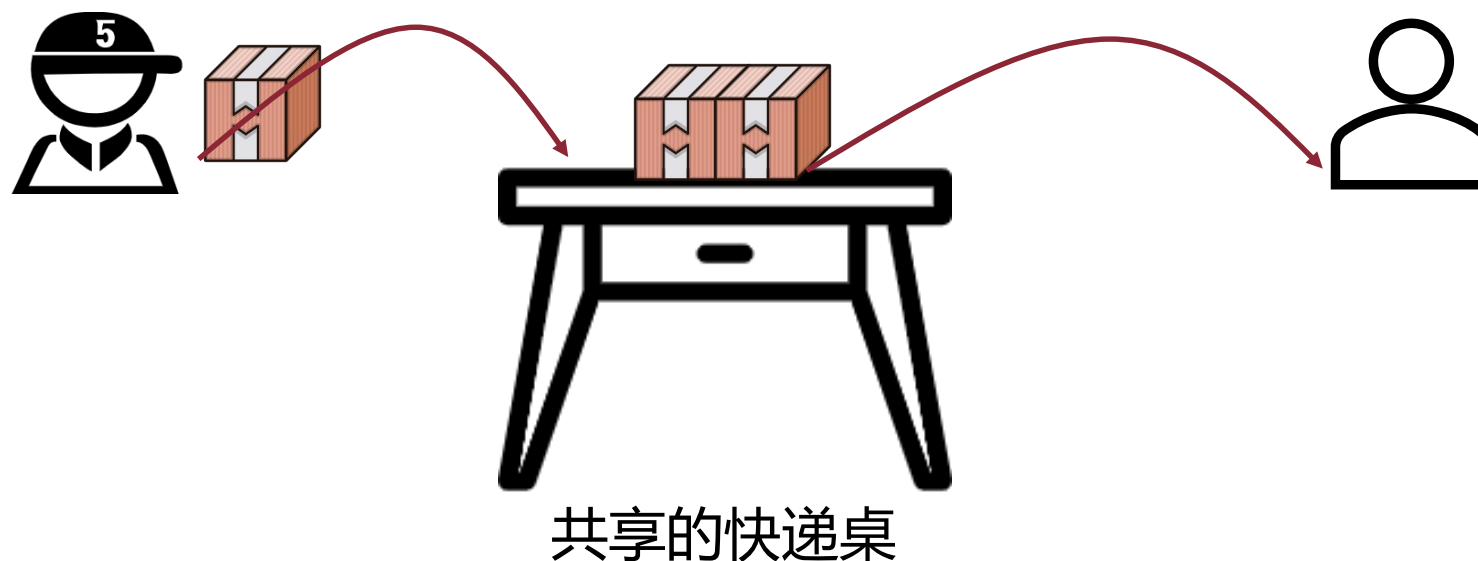
快递



消息

# 共享内存：共享一块区域

- 小区门口的桌子上允许临时放置快递
- 快递员在桌上放置快递，小明从桌上取快递



# 共享内存

- **系统内核为两个进程映射共同的内存区域**
  - 快递员和小明的快递桌
- **挑战: 做好同步**
  - 发送者不能覆盖掉未读取的数据 (新快递把旧的快递挤下桌)
  - 接收者不能读取别的数据 (小明拿错了快递)

# 共享内存

- 基础实现: 共享区域(快递桌)

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
...
```

```
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

共享数据区域，容量为10  
(快递桌可以放10个快递)

共享状态，当前有快递的位置

# 共享内存

- 基础实现: 发送者(快递员)

```
while (new_package) {  
    /* Produce an item */
```

当没有空间时，发送者盲目等待  
(快递员等待桌子有空闲空间)

```
    while (((in + 1) % BUFFER_SIZE)  
           == out)  
        ; /* do nothing -- no free buffers */
```

```
    buffer[in] = item;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
}
```

发送者放置消息  
(快递员将快递放在桌上空闲空间)

# 共享内存

- 基础实现: 接收者(小明)

```
while (wait_package) {
```

```
    while (in == out)
```

```
        ; // do nothing -- nothing to consume
```

```
    // remove an item from the buffer
```

```
    item = buffer[out];
```

```
    out = (out + 1) % BUFFER_SIZE;
```

```
    return item;
```

```
}
```

当没有新消息时, 接收者盲目等待  
(小明盲目查看桌上状态和等待)

接收者获取消息  
(小明拿到最先到达的一个快递)

# 共享内存的问题

- **轮询导致资源浪费**

- 小明时不时就得下楼检查一下快递桌子
- 快递员需要等待桌子有空闲空间
- 小明一天大部分时间都花在了上下楼和检查快递上了

- **固定一个检查时间，时延长**

- 小明每天晚上检查一下有没有新的快递过来
- 早上到达的快递要晚上才能拿到



# 消息传递：拿到了手机的小明

- 小明终于承诺不玩游戏，从而说服了妈妈拿到了手机
- 消息系统 (手机)
  - 通过中间层(如内核)保证通信时延，仍可以利用共享内存传递数据



好处： 1) 低时延 (消息立即转发)  
2) 不浪费计算资源

# 消息传递

- **基本操作:**
  - 发送消息 `Send(message)`
  - 接收消息 `Recv(message)`
- **如果两个进程  $P$  和  $Q$  希望通过消息传递进行通信, 需要:**
  - 建立一个通信连接
  - 通过 `Send/Recv` 接口进行消息传递

# 直接通信: 快递员直接拨打小明手机

- 快递员和小明通过快递网站交换手机号来建立连接
  - 手机号唯一地标识了快递员和小明
  - Send(**P**, message): 给P进程发送一个消息
  - Recv(**Q**, message): 从Q进程接收一个消息
- 直接通信下的连接
  - 连接的建立是自动的 (通过标识, 即手机号)
  - 一个连接唯一地对应一对进程
  - 一对进程之间也只会存在一个连接
  - 连接可以是单向的, 但是在大部分情况下是双向的

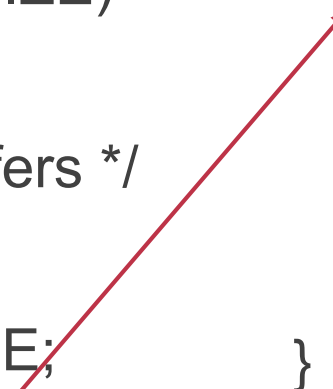
# 直接通信: 快递员直接拨打小明手机

- 发送者(快递员)

```
while (new_package) {  
    /* Produce an item */  
    while (( (in + 1) % BUFFER_SIZE)  
           == out)  
        ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    Send(XiaoMing, "Package");  
}
```

- 接收者(小明)

```
while (wait_package) {  
    Recv(Expressman, Msg);  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    return item;  
}
```



小明的Recv会阻塞，直到快递员的Send发送消息过来

# 快递员有好多苦恼 (1)

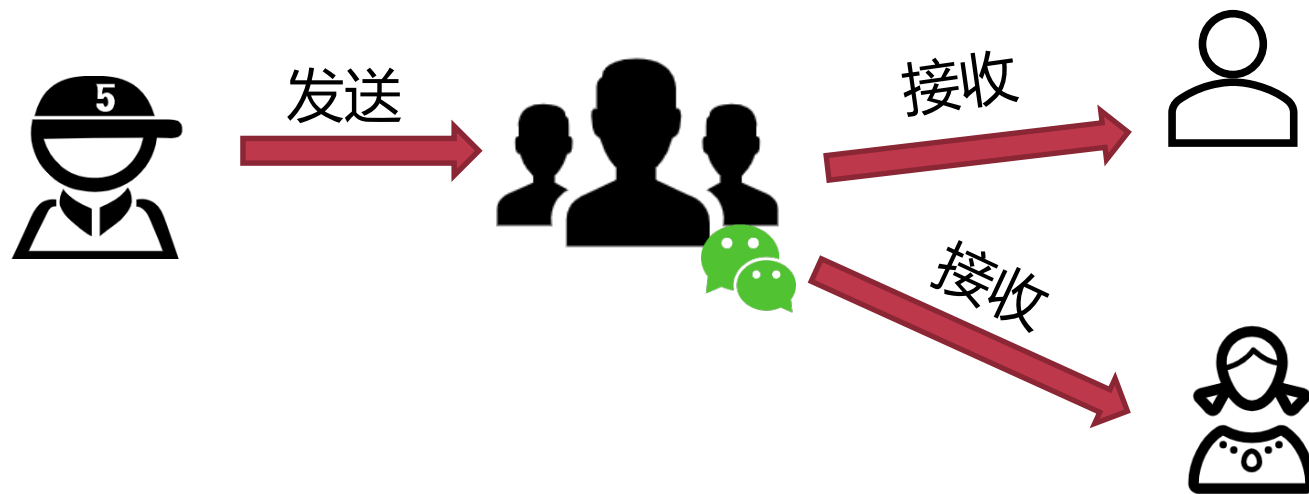


小明沉迷学习经常不接听电话，这可怎么办？

- 快递员执行Send的时候，小明还没有Recv
- 快递员知道小明妈妈经常在家，希望建立一个聊天群，在群里发布快递信息
- 小明不接听时可以拜托妈妈下来拿快递

# 间接通信：用聊天群发布快递信息

- 消息的发送和接收需要经过一个“信箱”
  - 聊天群 (所有在群内的人都可以接收消息)
  - 每个“信箱”有自己唯一的标识符 (这里的群号)
  - 发送者往“信箱”发送消息，接收者从“信箱”读取消息



# 间接通信

- **间接通信下的连接**
  - 进程间连接的建立发生在共享一个信箱时
  - 每对进程可以有多个连接 (共享多个信箱)
  - 连接同样可以是单向或双向的
- **间接进程间通信的操作**
  - 创建一个新的信箱
  - 通过信箱发送和接收消息
  - 销毁一个信箱
- **原语**
  - Send(**M**, message): 给信箱M发送一个消息
  - Recv(**M**, message): 从信箱M接收一个消息

# 间接通信: 用聊天群发布快递信息

- 发送者(快递员)

```
while (new_package) {  
    /* Produce an item */  
    while (( (in + 1) % BUFFER_SIZE)  
           == out)  
        ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    Send(Mailbox, "Package");  
}
```

- 接收者(小明)

**Recv(Mailbox, Msg);**

- 接收者(小明妈妈)

**Recv(Mailbox, Msg);**



小明或者妈妈任何一个人只要上聊天群，就能看到快递信息



## 快递员有好多苦恼 (2)



快递信息发布到群里，经常是小明和妈妈一起下来了。我都被投诉好几次了，这可怎么办好？

- 如何解决“信箱”共享带来的多接收者的问题呢？

# 信箱共享的挑战

- 信箱的共享

- 进程 $P_1$ 、 $P_2$ 和 $P_3$ 共享一个信箱M
- $P_1$ 负责发送消息， $P_2$ 、 $P_3$ 负责接收消息
- 当一个消息发出的时候，谁会接收到最新的消息呢？

- 可能的解决方案

- 让一个连接(信箱)只能被最多两个进程共享，避免该问题
- 同一时间，只允许最多一个进程在执行接收信息的操作
- 让消息系统任意选择一个接收者 (需要通知发送者谁是最终接收者)

## 快递员有好多苦恼 (3)

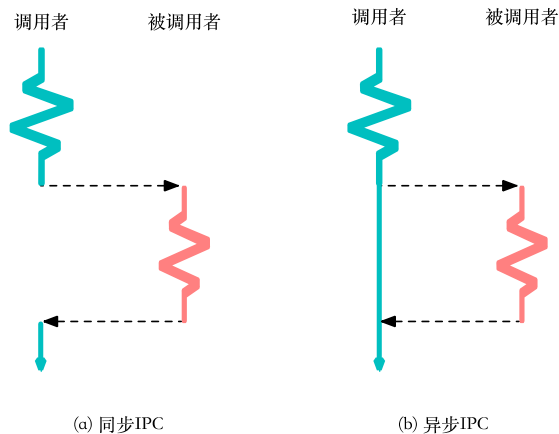


小明和妈妈回复消息很慢，如果我发完消息等待回复可能要等很久；如果我放下快递直接走的话事后又可能被投诉，怎么办呢？

- 快递员想要弄清楚自己是等待消息被确认呢(阻塞)，还是发送完消息就赶去送下一个快递呢(非阻塞)？

# 消息传递的同步与异步

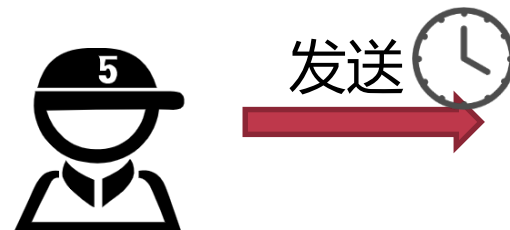
- 消息的传递可以是阻塞的，也可以是非阻塞的
- 阻塞通常被认为是同步通信
  - 阻塞的发送/接收: 发送者/接收者一直处于阻塞状态，直到消息发出/到来
  - 同步通信通常有着更低时延和易用的编程模型 (不会被投诉)
- 非阻塞通常被认为是异步通信
  - 发送者/接收者不等待操作结果，直接返回
  - 异步通信的带宽一般更高 (快递员可以送更多的快递)



# 超时机制

- 为了好评，快递员选择：

- 尽可能等待 (同步的通信)
- 但是一旦超过一个值 (如15分钟)，就先带走快递，等下再配送



- 超时机制的引入

- Send(A, message, **Time-out**)
  - 超过Time-out限定的时间就返回错误信息
- 两个特殊的超时选项: ① 一直等待 (阻塞) ; ②不等待 (非阻塞)
- 避免由通信造成的拒接服务攻击等

# 同步通信和超时机制

- 发送者(快递员)

```
while (new_package) {  
    /* Produce an item */  
    while (((in = (in + 1) % BUFFER_SIZE)  
           == out)  
           ; /* nothing, no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    if (Send(Mailbox, "Package",  
        "15min") == error)  
        goto retry;  
}
```

- 接收者(小明)

**Recv(Mailbox, Msg, Time-out);**

- 接收者(小明妈妈)

**Recv(Mailbox, Msg, Time-out);**



**快递员决定等待最多15分钟，一旦超时就放弃这一次派送**

## 快递员有好多苦恼 (4)



新冠疫情缓解后，为了避免快递丢失，小区保安不允许快递员将快递放在快递桌上。快递员不能放下快递就走了。

- 有手机后，快递桌的作用有什么变化吗？

# 通信连接的缓冲：快递桌的作用

- **缓冲：**通信连接可以选择保留住还没有处理的消息
- **常见的三种设计**
  - **零容量：**通信连接本身不缓冲消息，发送者只能阻塞等待接收者接收消息 (保安不提供快递桌)
  - **有限容量：**连接可以缓冲最多N个消息，当缓冲区满之后发送者只能阻塞等待 (快递只能放在桌上，但是空间有限)
  - **无限容量：**连接可以缓冲系统资源允许下的任意数量的消息，发送者几乎不需要等待 (快递可以放在门口任何位置)





进程通信是指进程之间的信息交换



**低级进程通信**：进程的同步和互斥

- 效率低
- 通信对用户不透明

我们以信号量机制为例来说明，它们之所以低级的原因在于：① 效率低，生产者每次只能向缓冲池投放一个产品(消息)，消费者每次只能从缓冲区中取得一个消息；② 通信对用户不透明，OS只为进程之间的通信提供了共享存储器。



## 高级进程通信

- 使用方便
- 高效地传送大量数据

**在进程之间要传送大量数据时，应当利用OS提供的高级通信工具，该工具最主要的特点是：**

**(1) 使用方便。** OS隐藏了实现进程通信的具体细节，向用户提供了一组用于实现高级通信的命令(原语)，用户可方便地直接利用它实现进程之间的通信。或者说，通信过程对用户是透明的。这样就大大减少了通信程序编制上的复杂性。

**(2) 高效地传送大量数据。** 用户可直接利用高级通信命令(原语)高效地传送大量的数据。



## 共享存储器系统

### ➤ 基于共享数据结构的通信方式（效率低）

OS仅提供共享存储器，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。这种通信方式仅适用于传送相对较少量的数据，通信效率低下，属于低级进程通信。

### ➤ 基于共享存储区的通信方式（高级）

为了传送大量数据，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，而非OS。需要通信的进程在通信前，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，读 / 写完成或不再需要时，将分区归还给共享存储区即可。



## 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，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。
  - ✓ 同步，指当写(输入)进程把一定数量(如4 KB)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它唤醒。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。
  - ✓ 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。

# Unix 管道

- 管道是Unix等宏内核系统中非常重要的进程间通信机制
- 管道(Pipe): 两个进程间的一根通信通道
  - 一端向里投递, 另一端接收
  - 管道是间接消息传递方式, 通过共享一个管道来建立连接
- 例子: 我们常见的命令 `ls | grep`

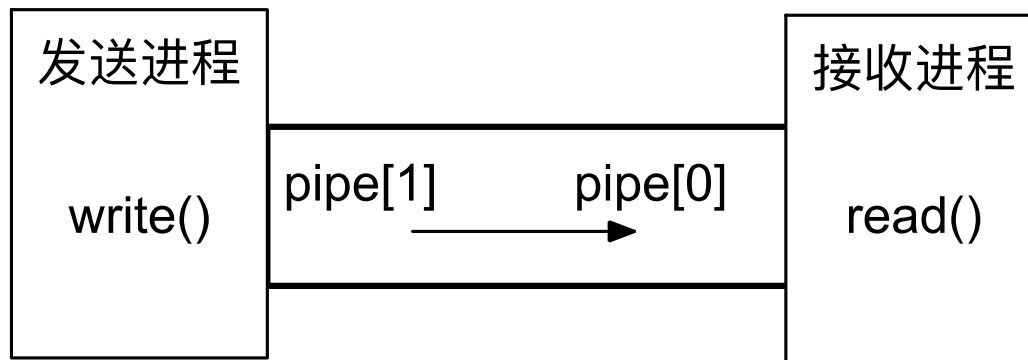
```
→ os-textbook git:(master) ls | grep ipc  
ipc.tex
```

# Unix 管道

```
int fd[2];  
pipe(fd);  
fd[0]; // read side  
fd[1]; // write side
```

- 管道的特点:

- 单向通信，当缓冲区满时阻塞
- 一个管道有且只能有两个端口: 一个负责输入 (发送数据)，一个负责输出 (接收数据)
- 数据不带类型，即字节流
- 基于Unix的文件描述符使用





## 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(计算机间的进程通信)

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



## 直接通信方式

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

## ➤ 进程的同步方式

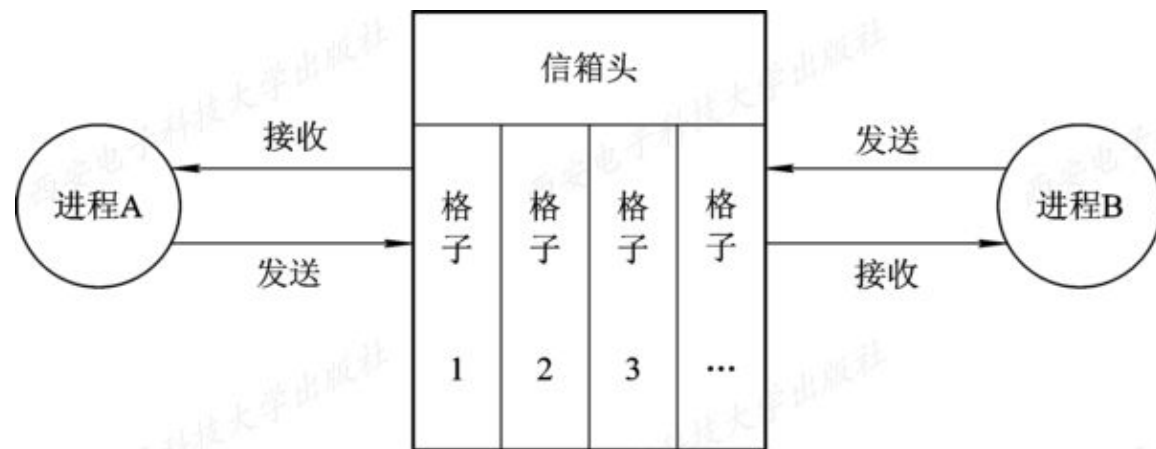
在进程之间进行通信时，同样需要有进程同步机制，以使诸进程间能协调通信。不论是发送进程还是接收进程，在完成消息的发送或接收后，都存在两种可能性，即进程或者继续发送(或接收)或者阻塞。  
**最常用的是：发送进程不阻塞，接收进程阻塞。**





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



双向信箱示意图



```
#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); } }
```



```
#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);
}
```

---

---

---

---

---

---



2.1 前趋图和程序执行



2.2 进程的描述



2.3 进程控制



2.4 进程通信



2.5 线程的基本概念



2.6 线程的实现

## 第2章 进程的描述与控制

---

主讲: 王红玲 主审: 汤小丹



## 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。

# 为什么需要线程？

- **创建进程的开销较大**
  - 包括了数据、代码、堆、栈等
- **进程的隔离性过强**
  - 进程间交互：可以通过进程间通信（IPC），但开销较大
- **进程内部无法支持并行**



## 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：更加轻量级的运行时抽象

- 线程只包含运行时的状态
  - 静态部分由**进程**提供
  - 包括了执行所需的**最小状态**（主要是寄存器和栈）
- 一个进程可以包含多个线程
  - 每个线程共享同一地址空间（方便数据共享和交互）
  - 允许进程内并行

## 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。

## 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。

## 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



-  2.1 前趋图和程序执行
-  2.2 进程的描述
-  2.3 进程控制
-  2.4 进程通信
-  2.5 线程的基本概念
-  2.6 线程的实现

## 第2章 进程的描述与控制

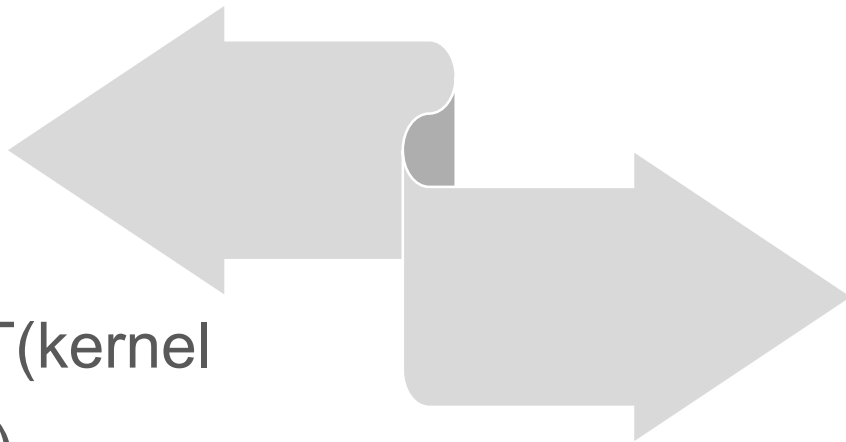
---

主讲：王红玲 主审：汤小丹



实现方式:

- 内核支持线程KST(kernel supported thread)
- 用户级线程ULT (user level thread)
- 组合方式



具体实现:

- 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)



## 在内核空间实现



### 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。





在用户空间实现



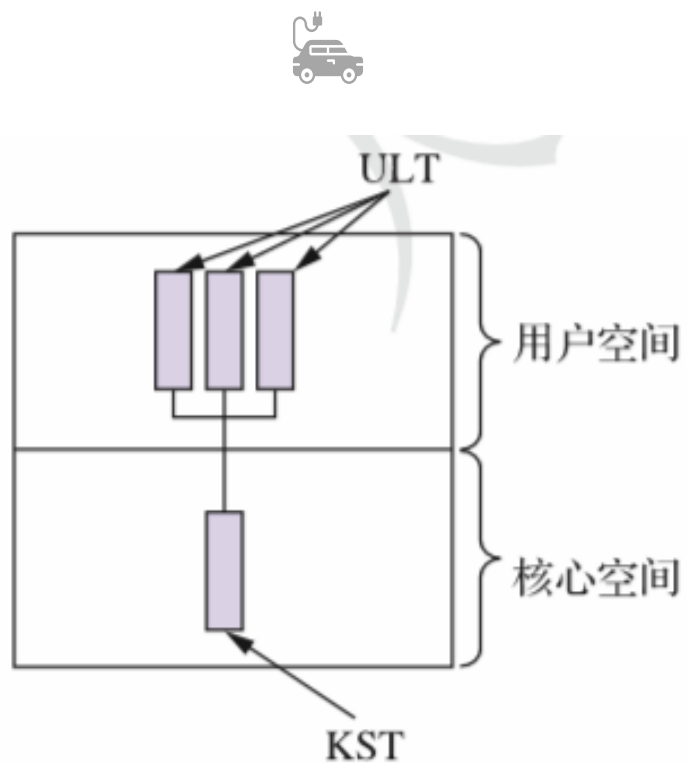
优点:

- 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。

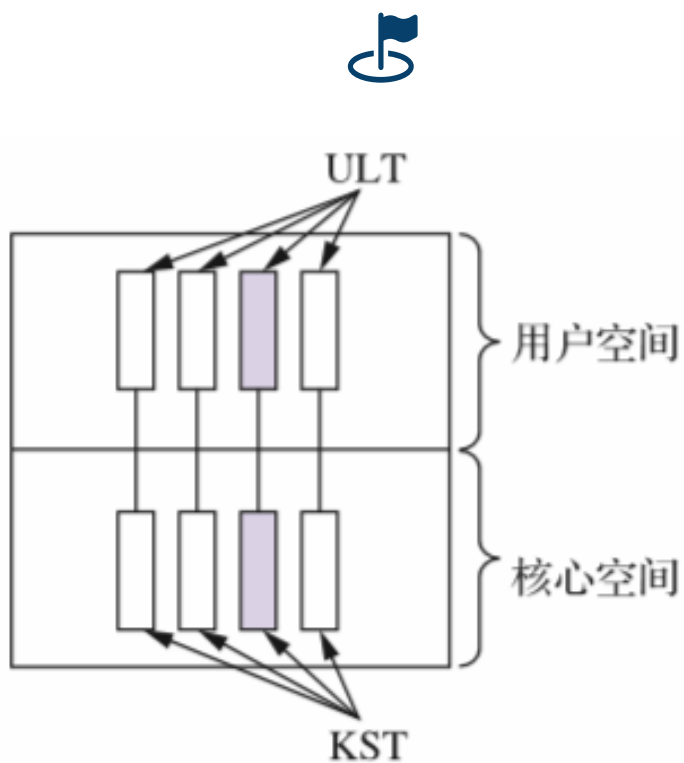


缺点:

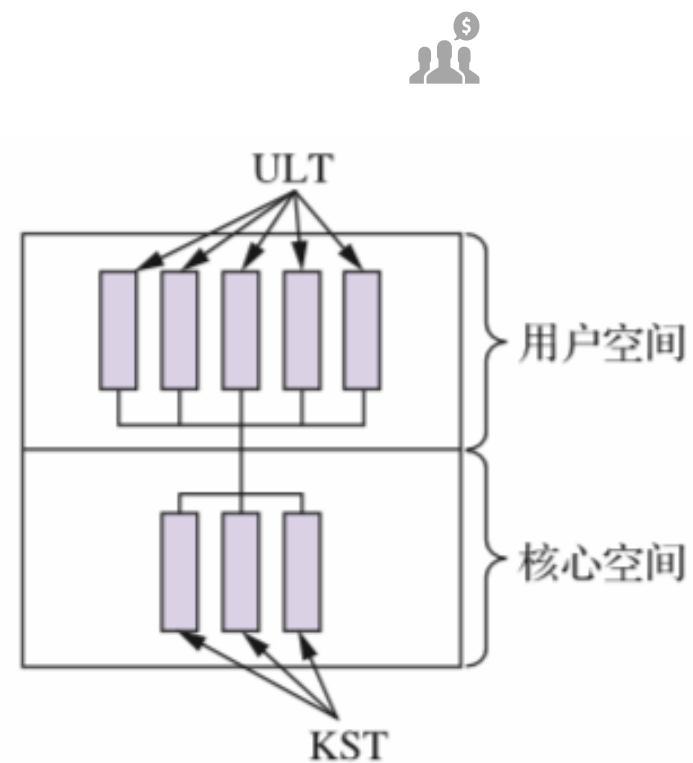
- 系统调用的阻塞问题。
- 多线程应用不能利用多处理机进行多重处理的优点。



- 多对一模型



- 一对一模型



- 多对多模型

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。

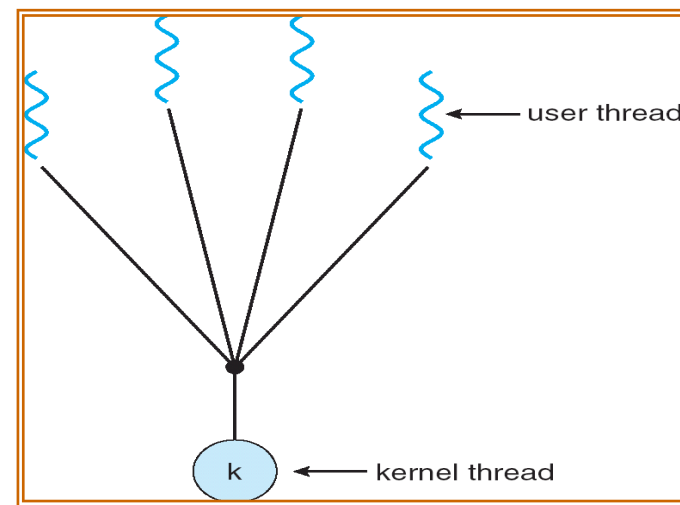
04

用于不支持内核线程的系统中。

05

例子：

- Solaris Green Threads
- GNU Portable Threads



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。

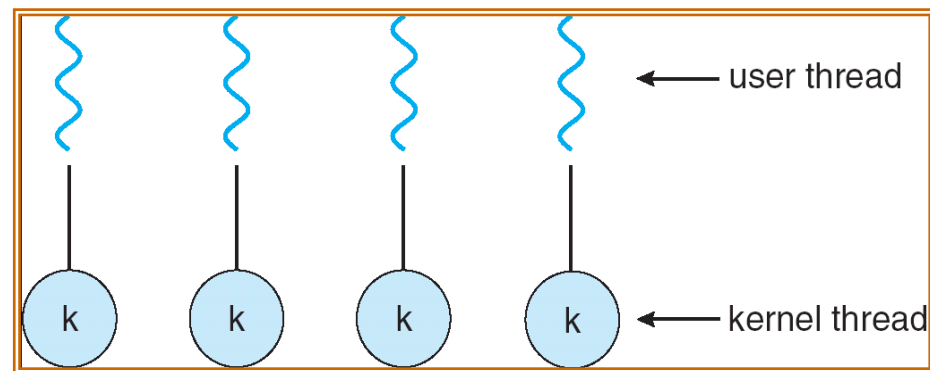
05

例子： ➤ Windows 95/98/NT/XP/2000

➤ Solaris 9 and later

➤ Linux

➤ OS/2





多个用户级线程映射为相等或小于数目的内核线程。

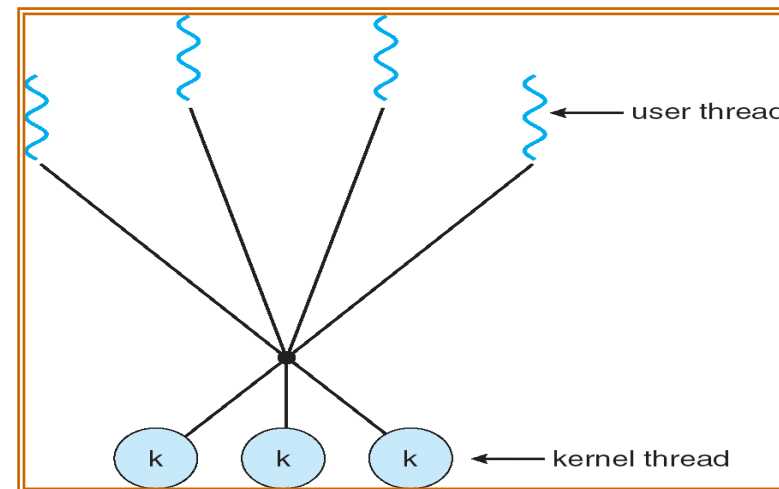


允许操作系统创建足够多的KLT。



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



本章学习结束