第3章 进程

内容

1. 进程概念
2. 进程操作
3. 进程调度
4. 进程间通信

1、进程概念

进程(Process)

- 进程是操作系统执行CPU调度和资源分配的单位
- 操作系统执行各种程序
  - 批处理系统 - 作业(Job)
  - 分时系统 - 用户程序或任务(Task)
- 本书使用的名词作业和进程，基本可互换
  - 作业：被组装成一个整体运行的一组计算步骤
  - 任务：进程或线程
- 进程 - 执行中的程序；进程的执行必须以顺序方式进行（非正式）
- 另一个说法：一个程序在一个数据集上的一次执行

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 tnslsnr
  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--
```

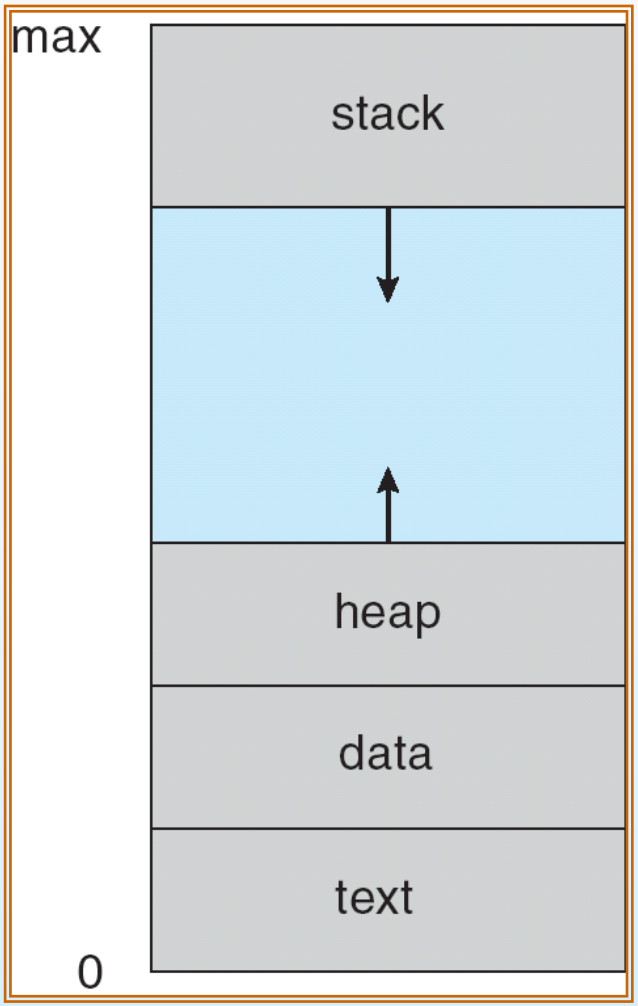
进程例子： Windows XP

进程和程序

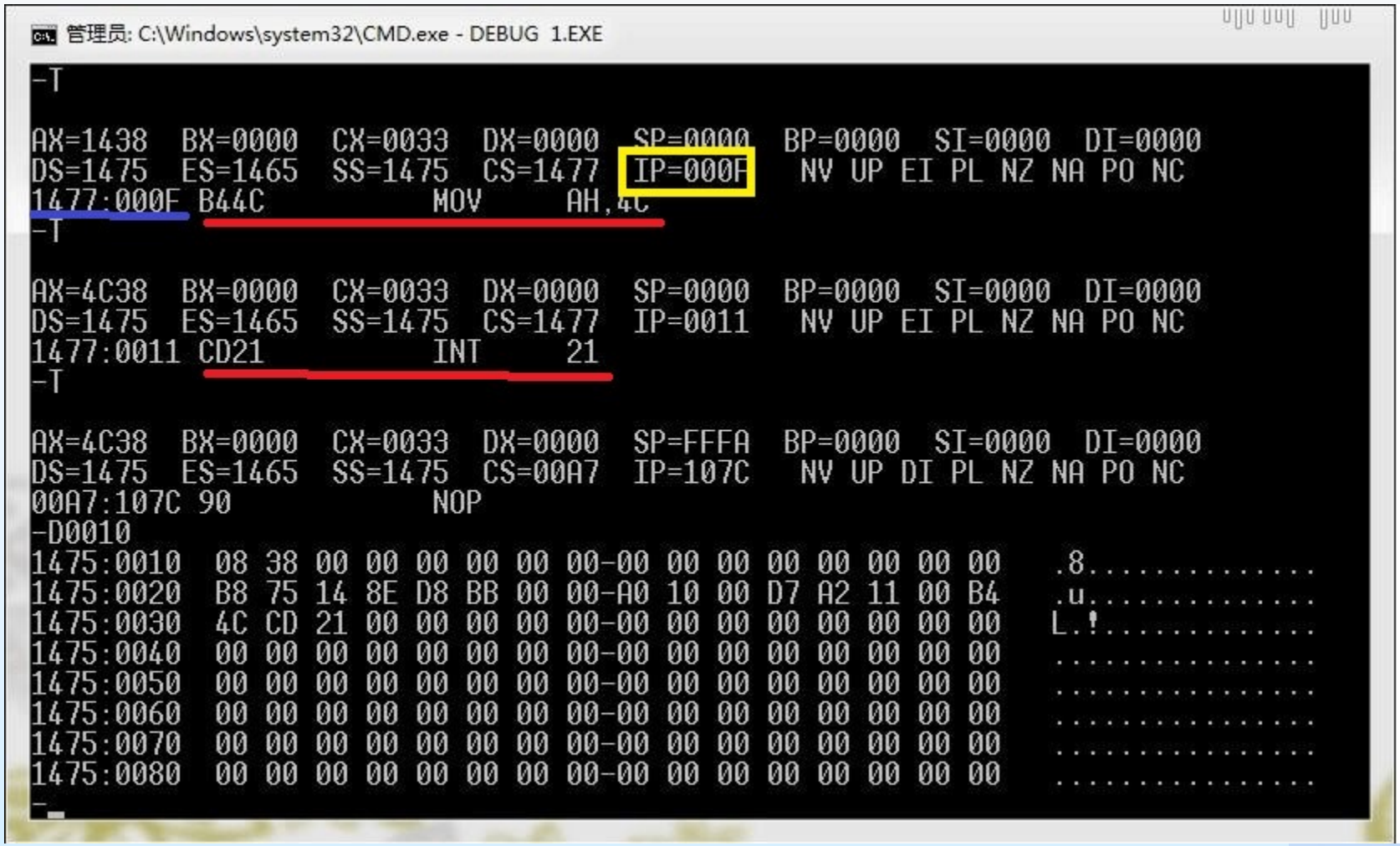
- 进程是程序的一个实例，是程序的一次执行
- 一个程序可对应一个或多个进程，同样一个进程可对应一个或多个程序
- 程序是进程的代码部分
- 进程是活动实体，程序静止（被动）实体
- 进程在内存，程序在外存

内存中的进程

- 进程包括
  - 代码（**Text**）
  - 当前活动
    - **程序计数器（PC）**-指向当前要执行的指令（地址）
    - **堆栈（Stack）**：存放函数参数、临时变量等临时数据
    - **数据（Data）**：全局变量，处理的文件
    - **堆（Heap）**：动态内存分配



Debug例子-程序计数器PC

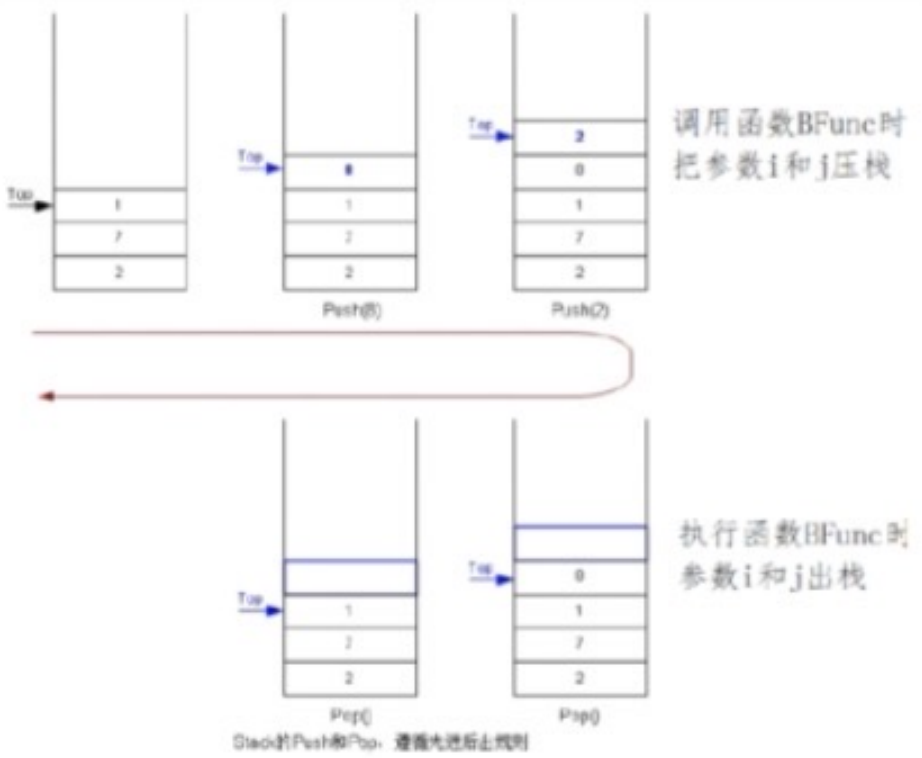


堆栈

- 内存管理角度
  - 堆栈是就是一块连续的内存空间
  - 对它的操作采用先入后出的规则
- 进程角度
  - 每一个进程有自己的堆栈
  - 主要用来给线程提供一个暂时存放数据的区域
  - 使用pop/push机器指令来对堆栈进行操作

例子

```
int BFunc(int i,int j) {
        int m = 1;
        m += i+ j;
        return m;
}
int AFunc() {
int m=2, n=8;
int k=BFunc(m,n);
return k;
}
```



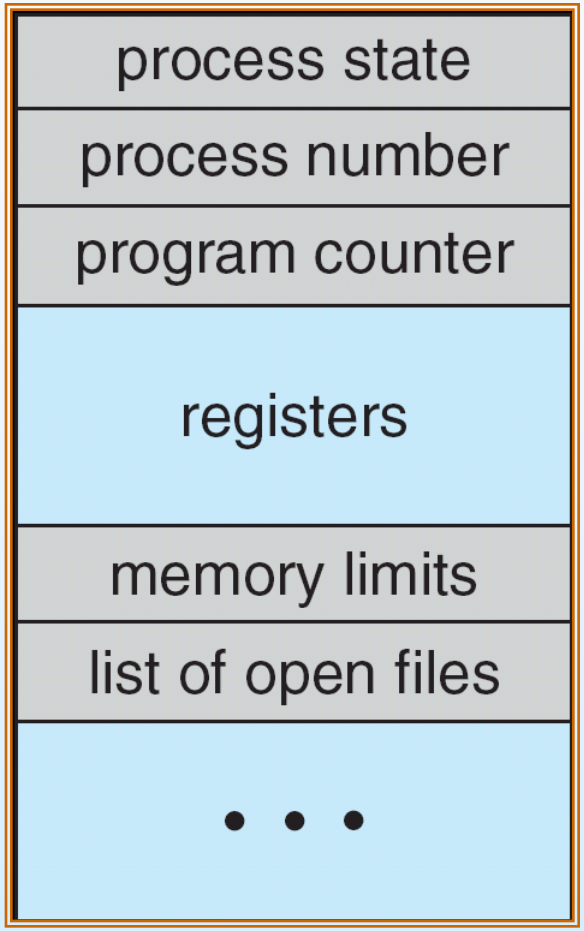
堆

- 允许程序在运行时动态申请内存空间
- 获得堆内存
  - Extern void *malloc(unsigned int num_bytes);
  - 分配长度为num_bytes字节的内存快
  - 例子：
    - Int *p;
    - P = (int*)malloc(sizeof(int));

进程控制块(PCB)

PCB包含同进程有关的信息，包括：

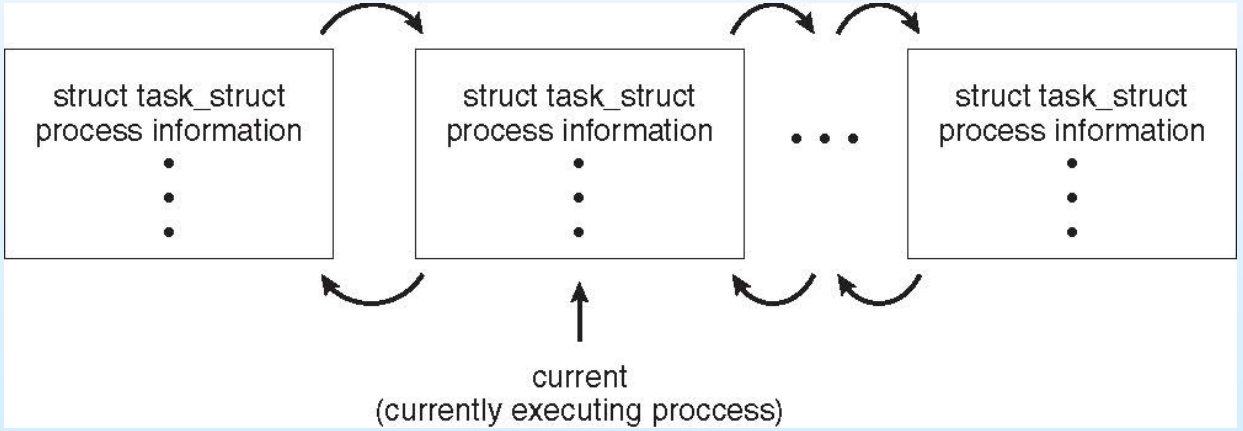
- 进程状态
- 进程号
- 程序计数器
- CPU寄存器
- CPU调度信息
- 内存管理信息
- 计账信息
- I/O状态信息



Linux PCB

C结构：task_struct

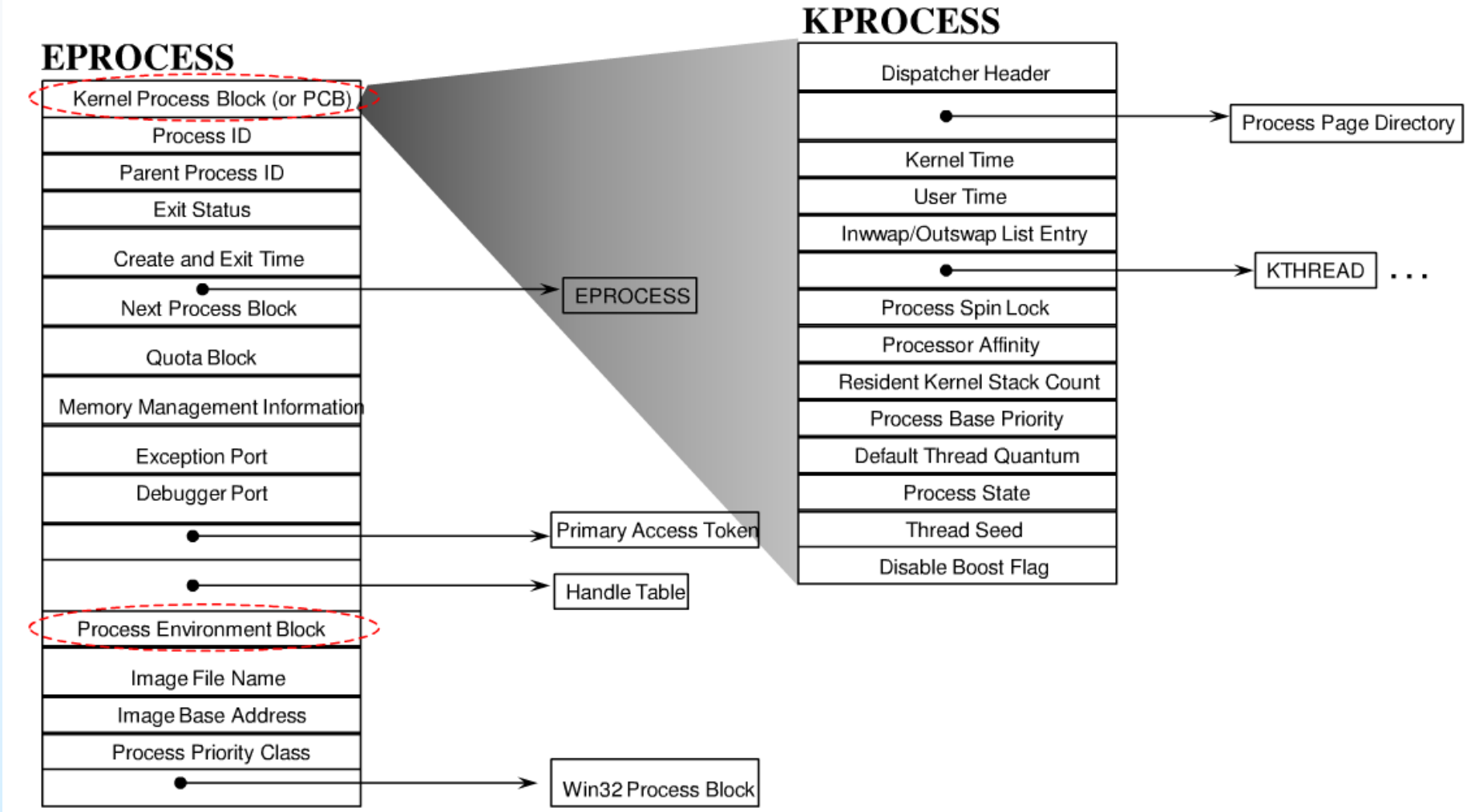
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process
*/
```



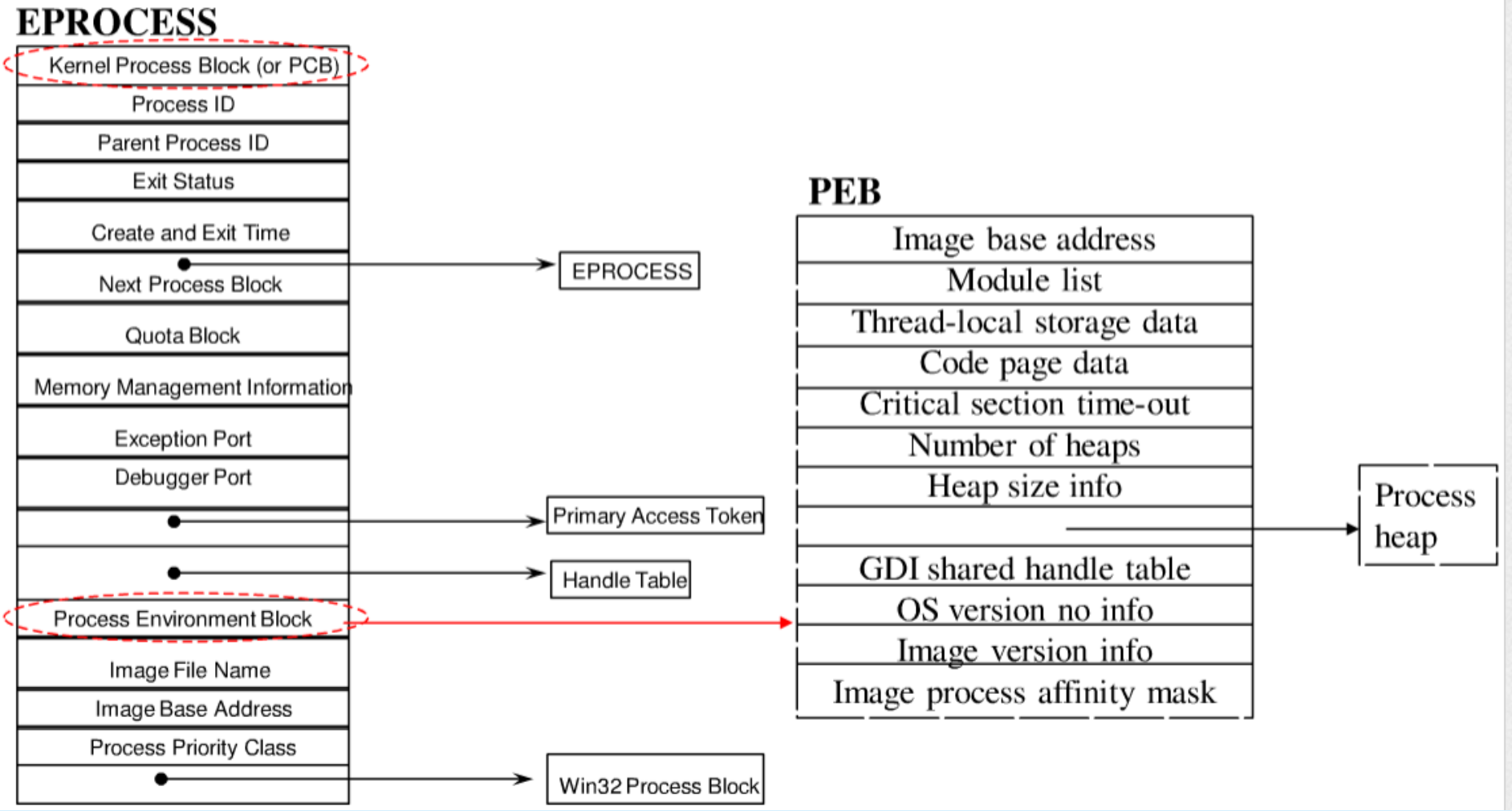
Windows PCB

- 每个Win32进程都由一个执行体进程块（executive process block）EPROCESS：PID, PCB, Access Token, Base Priority, 句柄表，指向进程环境块PEB指针，默认和处理器集合等
- Windows的PCB称为内核进程对象KPROCESS：
- 执行体进程对象EPOCESS和KPROCESS位于内核空间
- 进程环境块PEB （ Process Environment Block ），PEB位于用户空间

EPROCESS/KPROCESS

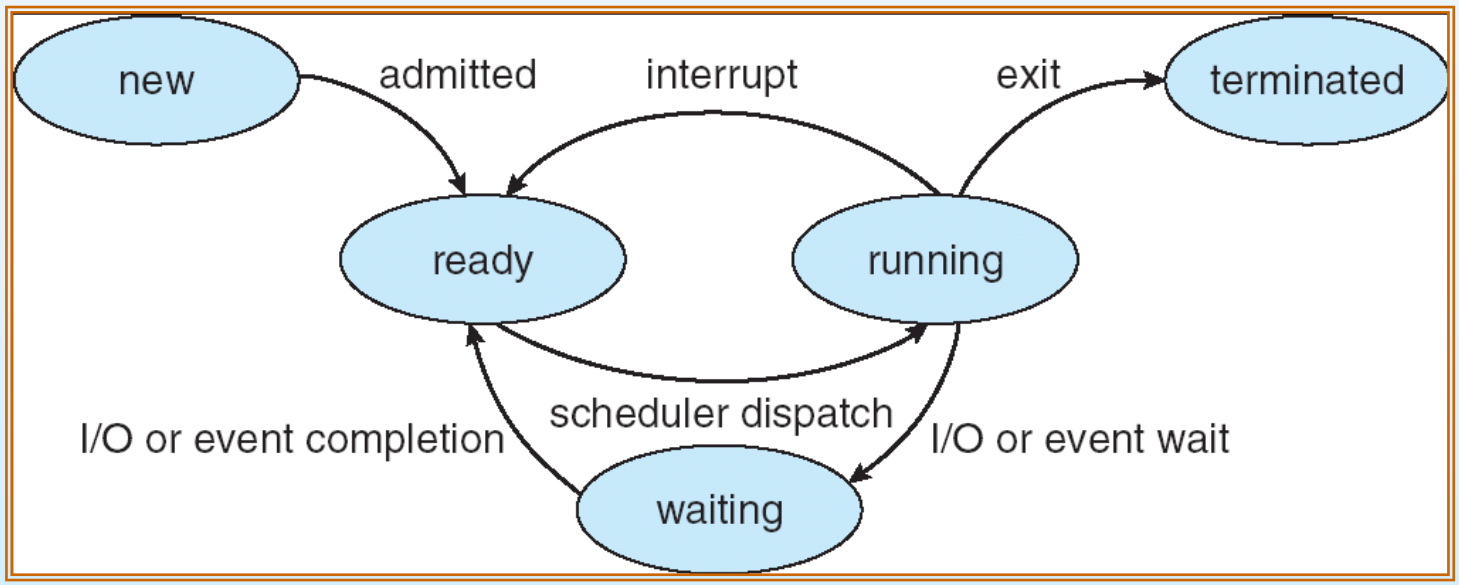


EPROCESS/PEB



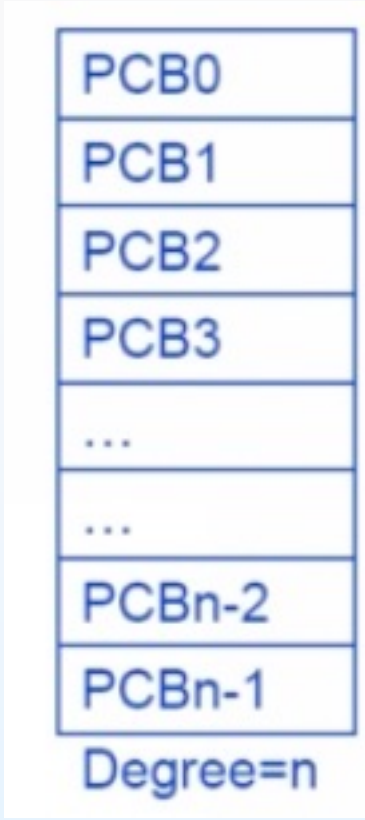
进程状态

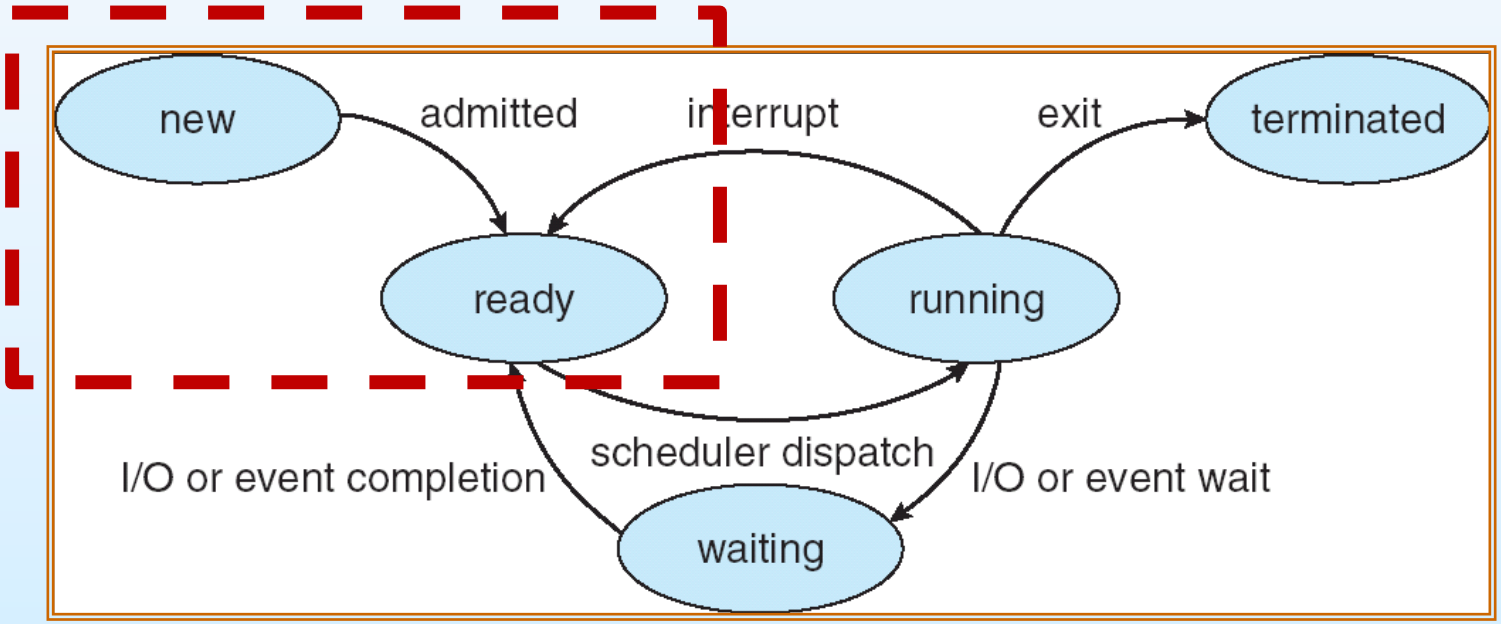
- 进程执行时，改变状态
  - 新建（new）：在创建进程
  - 就绪（ready）：进程等待分配处理器
  - 运行（running）：指令在执行
  - 等待、阻塞（waiting）：进程等待某些事件发生
  - 终止（terminated）：进程执行完毕



创建->就绪

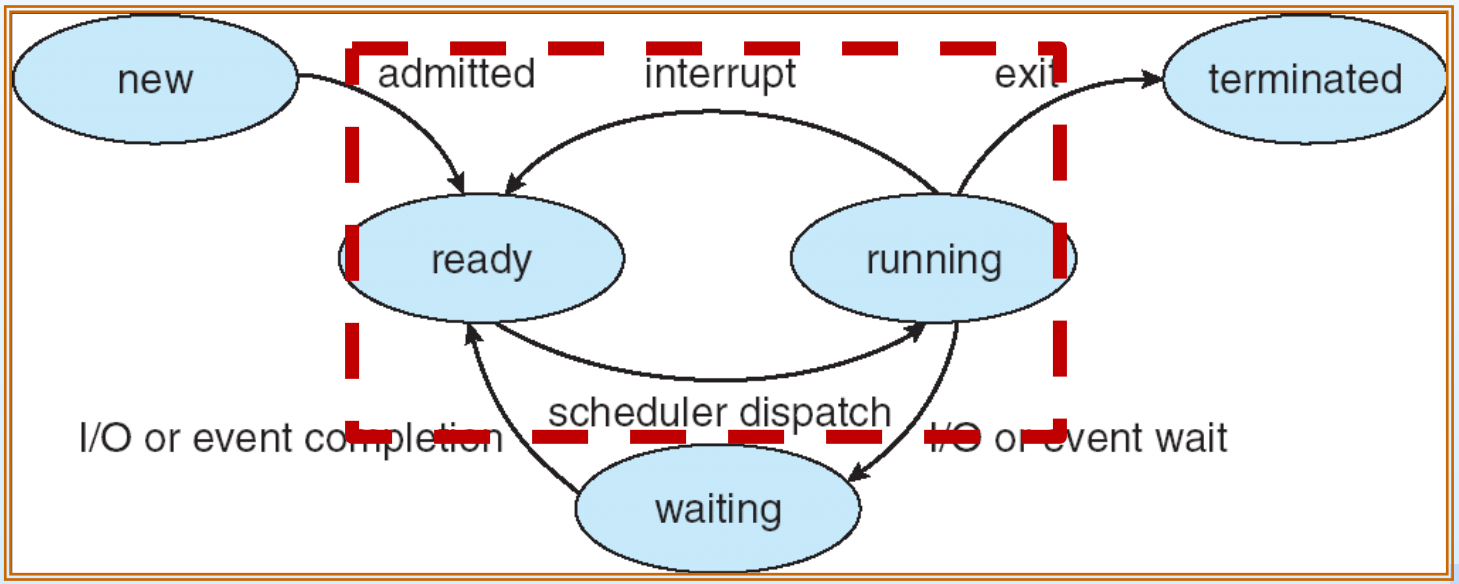
- 道（Degree）
  - 内存中可以容纳进程的数量
- 作业调度





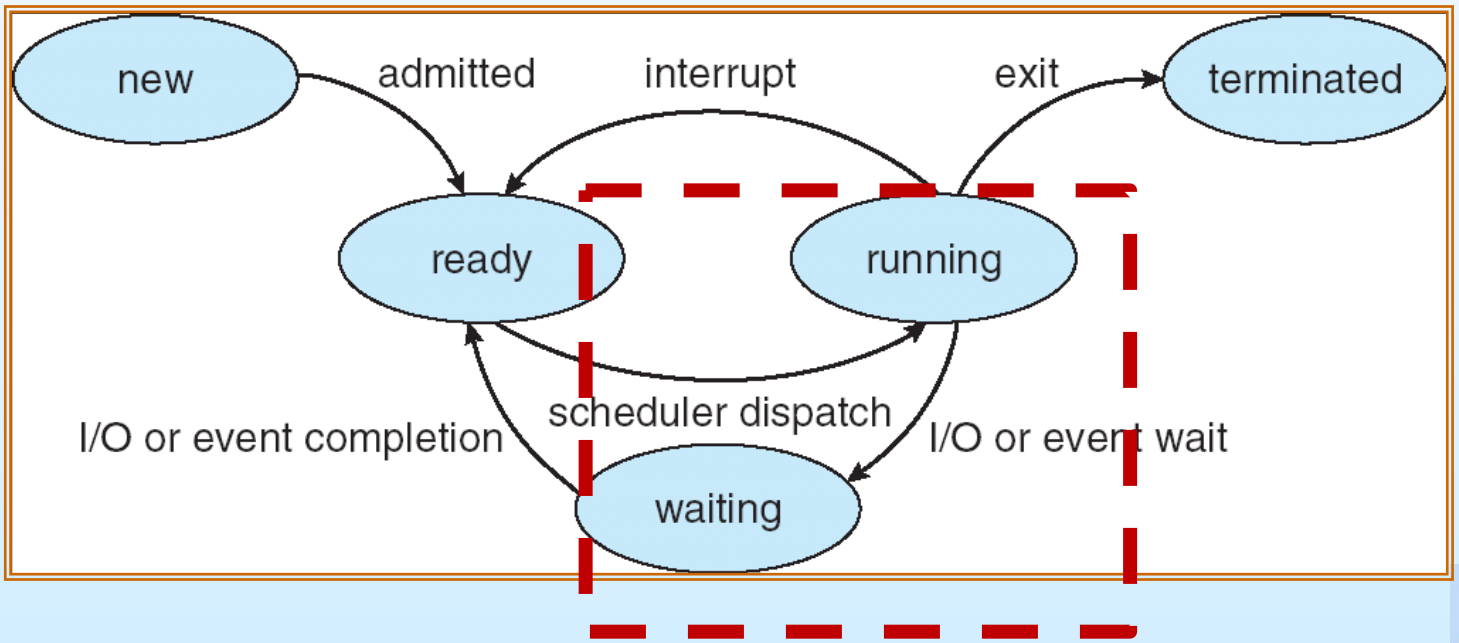
就绪->执行

- 进程调度
  - 时机：CPU空闲
  - 根据调度策略，从就绪队列中挑选一个进程去运行



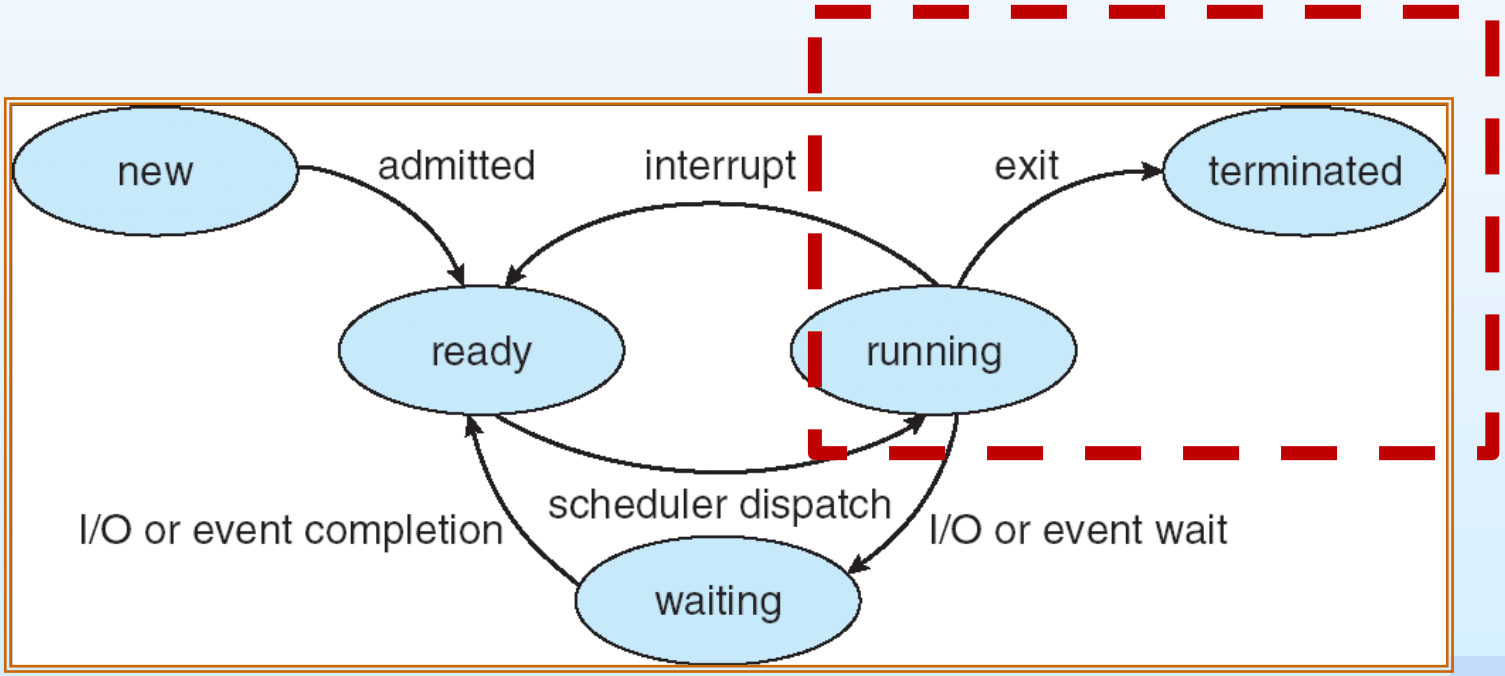
执行->阻塞

- 处于运行状态的进程无法继续运行
  - 等待某个事件发生（I/O，消息，事件等）
  - 进程主动放弃CPU



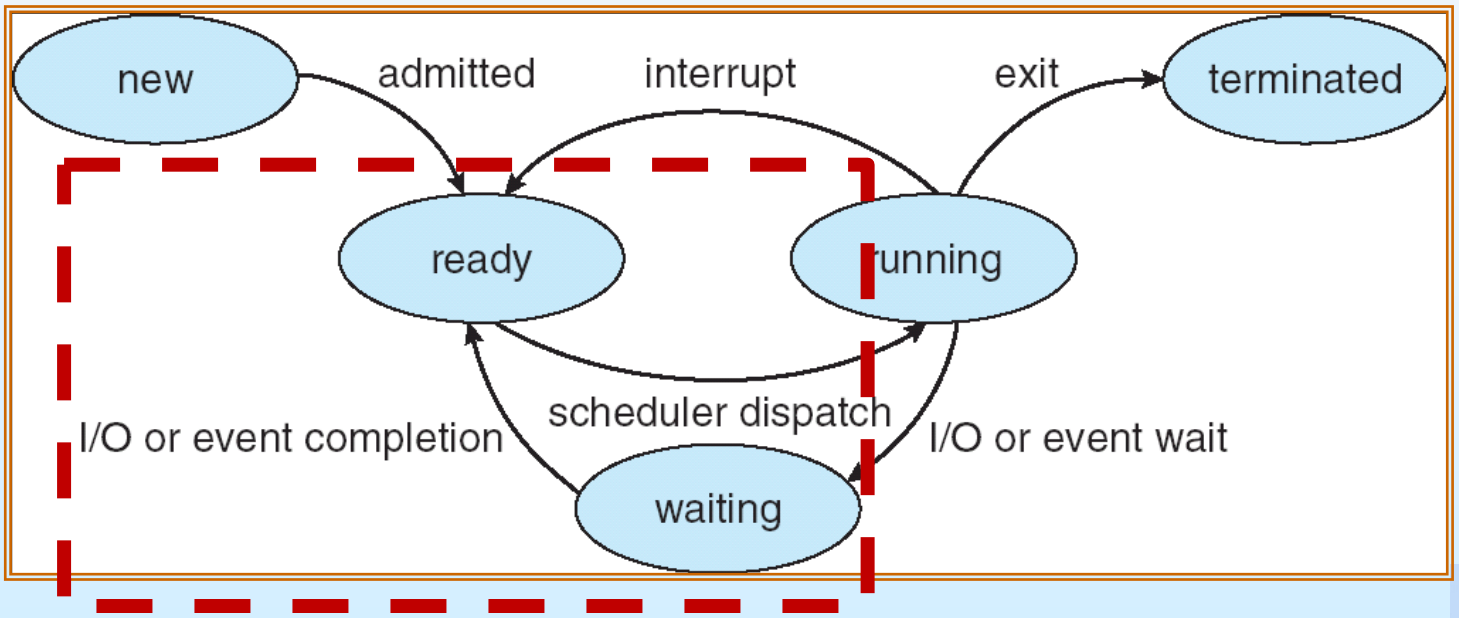
执行->终止

- 正常终止：进程运行完
- 异常终止：进程运行出现问题



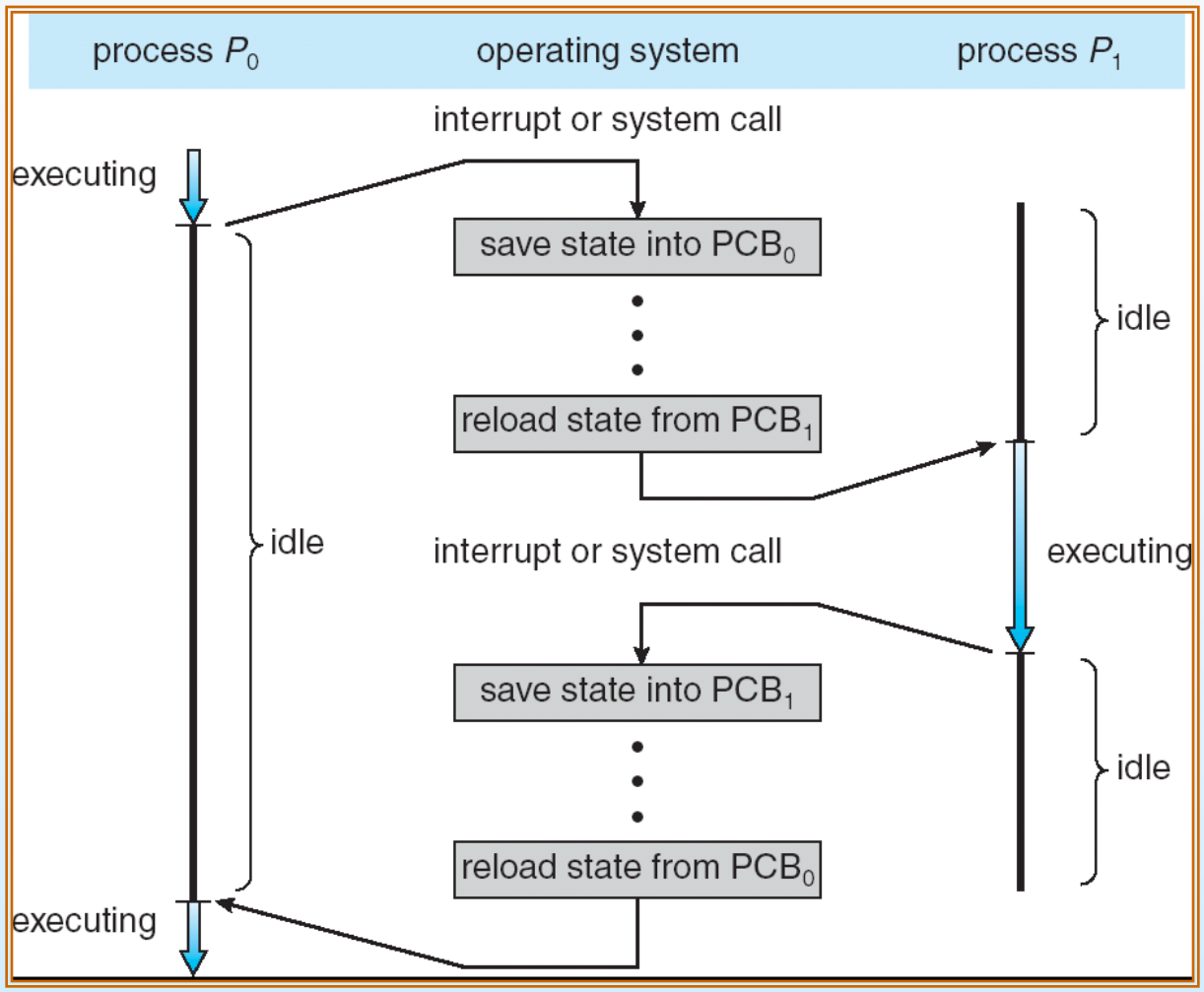
阻塞->就绪

- 等待的时间发生了
- 就绪和阻塞的区别
  - 就绪态进程只缺CPU，阻塞态进程分配给它CPU也无法运行



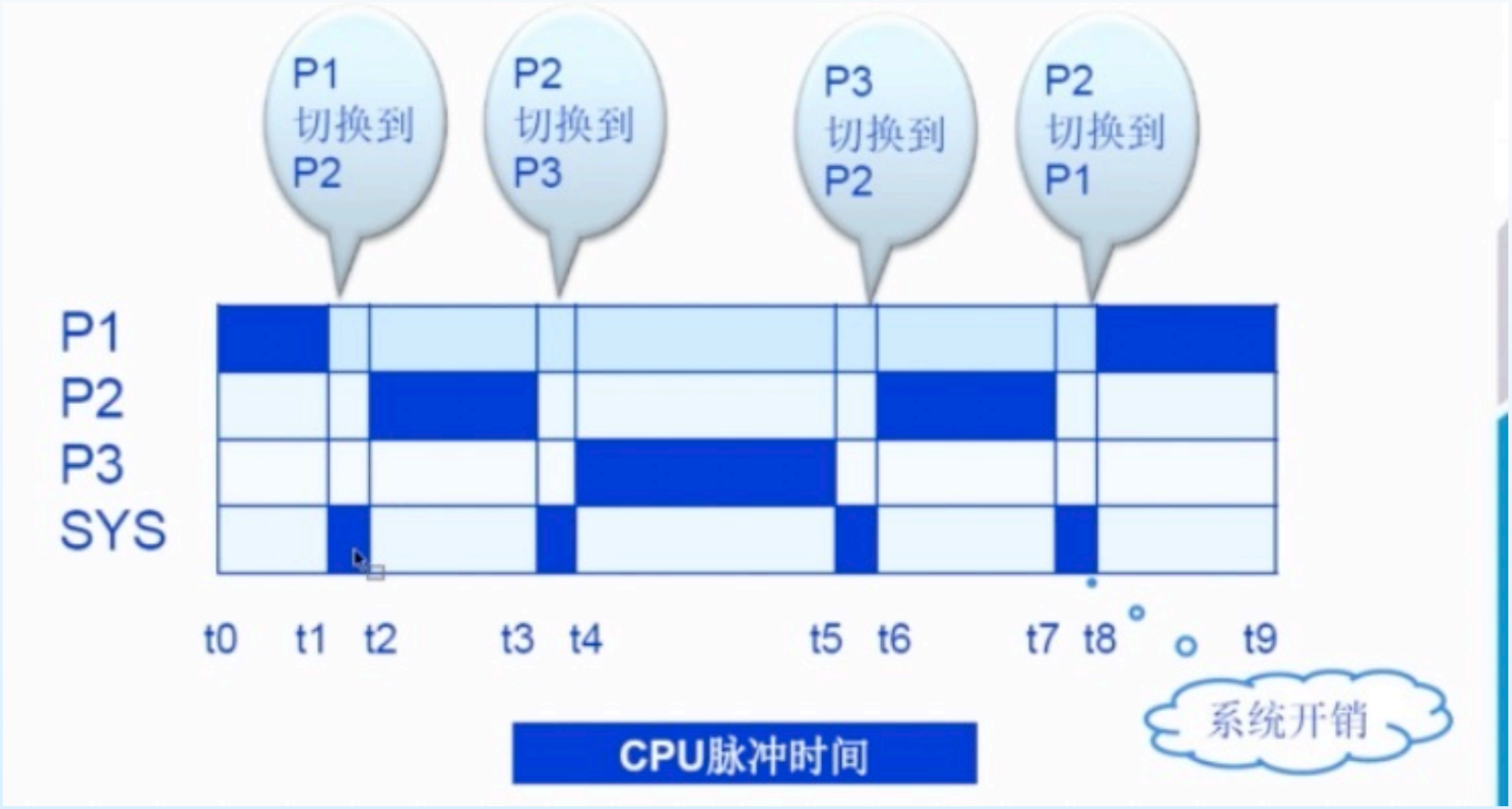
CPU 在进程间切换

进程的并发执行需要PCB保存和恢复现场



上下文切换(contact switch)

上下文切换的例子


P1
切换到
P2
P2
切换到
P3
P3
切换到
P2
P2
切换到
P1
P1
P2
P3
SYS
t0
t1
t2
t3
t4
t5
t6
t7
t8
t9
系统开销
CPU脉冲时间

为什么保护现场

- 现场：进程计算环境，包括寄存器、标志位等

P1

```
sum DWORD 0          (11)
.code
main PROC
    mov rax,5        (1)
    add rax,6        (2)
    mov sum,rax      (3)
```

P2

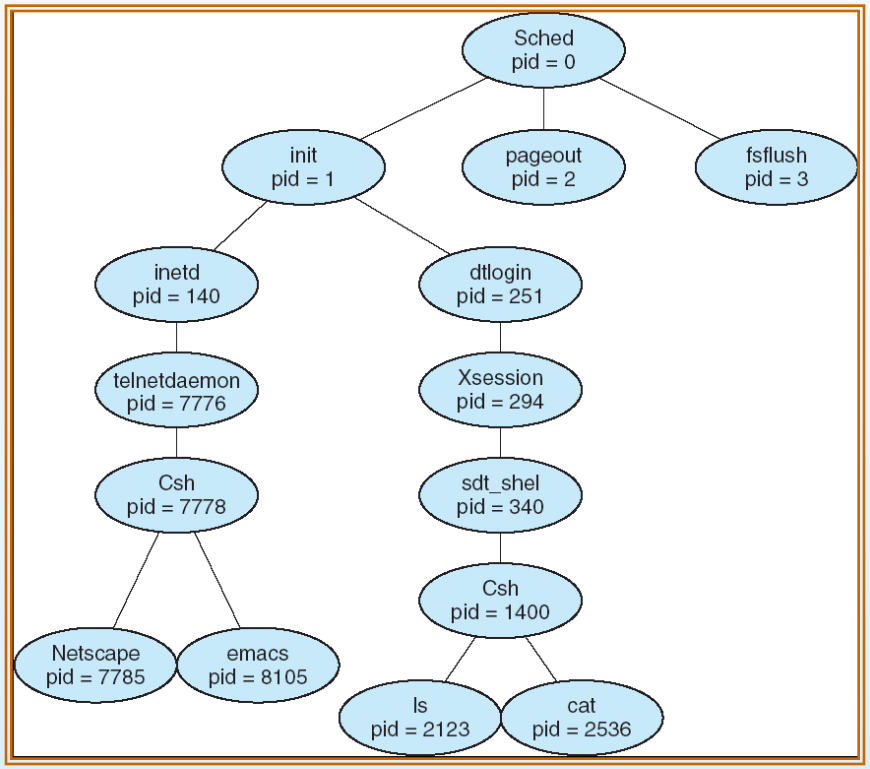
```
i DWORD 0            (21)
.code
main PROC
    mov rax,20       (4)
    inc  rax         (5)
    mov i,rax        (6)
```

步骤	RX
1	5
4	20
2	26
5	27
3	SUM=27
6	i=27

2、进程操作

进程创建

- **父进程**创建**子进程**，如此轮流创建进程下去，构成一棵进程树
- 资源共享
  - 父进程子进程共享所有的资源
  - 子进程共享父进程资源的子集
  - 父进程和子进程无资源共享
- 执行
  - 父进程和子进程并发执行
  - 父进程等待，直到子进程终止



进程创建

- 地址空间
  - 子女复制双亲
  - 子女有一个程序被调入
- UNIX例子
  - **fork** 系统调用创建新进程，创建时子进程完全复制父进程的空间，这种机制允许子进程和父进程方便的进行通讯
  - 在**fork**之后采用**exec**系统调用，以一个新程序替代进程的内存空间
  - 这种创建方式两个进程既能相互通讯，又可能按照自己的方式执行
- 原子操作
  - 该操作不能被打断，要么创建成功，要么创建失败

进程终止

- 进程执行最后一项并退出（exit）
  - 从子进程向父进程输出数据（通过wait）
  - 操作系统收回进程的资源
- 父进程可中止子进程的执行（abort）
  - 子进程超量分配资源
  - 赋予子进程的任务不再需要
  - 如果父进程结束
    - 若父进程终止，一些系统不允许子进程继续存在
      - 所有子进程终止-- **级联终止**
- 父进程可以等子进程结束
  - 如调用wait()系统调用

原子操作

- 原子操作（Atomic Operation）
  - 不会被打断的操作
  - 一旦开始就一直运行到结束，中间不过有任何上下文切换
  - 进程创建是原子操作
- 原子性必须需要硬件的支持
  - X86: CPU在指令执行期间对总线加锁的手段
  - CPU引线#HLOCK pin

操作原语

- 原子操作
  - 也称为操作原语
  - 操作系统内核中
  - 由若干条指令构成，用于完成一个特定的功能的一个过程
  - 该操作不能被打断，要么成功，要么失败

  - 创建进程原语 create(n)
  - 撤销进程原语 destroy(n)
  - 阻塞进程原语 block()
  - 唤醒进程原语 wakeup(n)

进程原语

- 进程创建原语的主要功能：
  - 为新建进程申请一个PCB
  - 将创建者（即父进程）提供的新建进程的信息填入PCB
  - 将新建进程设置为就绪状态，并按照所采用的调动算法，把PCB排入就绪队列。

进程原语

- 进程撤销
  - 该进程已经完成所要求的功能而正常终止
  - 由于某种错误导致非正常终止
  - 祖先进程要求撤销某个子进程
- 进程阻塞
  - 进程的执行状态变化到等待状态（一个进程期待某事件的发生，但发生尚不具备条件时），由该进程自己调用阻塞原语来阻塞自己。
  - 进程状态 运行->等待
- 进程唤醒
  - 唤醒一个进程的两种方式：（1）有系统进程唤醒；（2）由事件发生进程唤醒
  - 进程状态 等待->就绪

Windows进程操作

- CreateProcess：进程创建
  - 新进程可以继承父进程的一些资源：打开文件的句柄、各种对象（如进程、线程、信号量、管道等）的句柄、环境变量、当前目录、原进程的控制终端、原进程的进程组（用于发送Ctrl+C或Ctrl+Break信号给多个进程）——每个句柄在创建或打开时能指定是否可继承
  - 新进程不能继承：优先权类、内存句柄、DLL模块句柄
- ExitProcess和TerminateProcess：进程退出
- WaitForSingleObject:等待子进程结束

创建子进程API函数

- CreateProcess函数：

```
BOOL CreateProcess(
    LPCTSTR  lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTURINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

结束进程

- 如果某个process想自己停止执行， 可调用ExitProcess()
  - C程序库中的exit(), exit()在自动执行一些清除垃圾工作后， 再调用ExitProcess()终止进程

    VOID ExitProcess(UNT uExitCode)
- 如果process A 想要process B 停止执行， 可在取得process B 的handle 后， 调用TerminateProcess():

  BOOL TerminateProcess(HANDLE hProcess, UNIT uExitCode)

ExitProcess——主动终止

TerminateProcess——强制终止

例子-子进程

- Child Process:

```
#include "stdafx.h"
#include <conio.h>
int _tmain(int argc, _TCHAR* argv[])
{
        wprintf(L"ParamTest Output:\n");
        wprintf(L" Number of parameters: %d\n", argc);
        wprintf(L" Parameter Info:\n");
        for (int c=0; c < argc; c++){
                wprintf(L" Param #%d: %s\n", c, argv[c]);
        }
        return 0;
}
```

例子-父进程

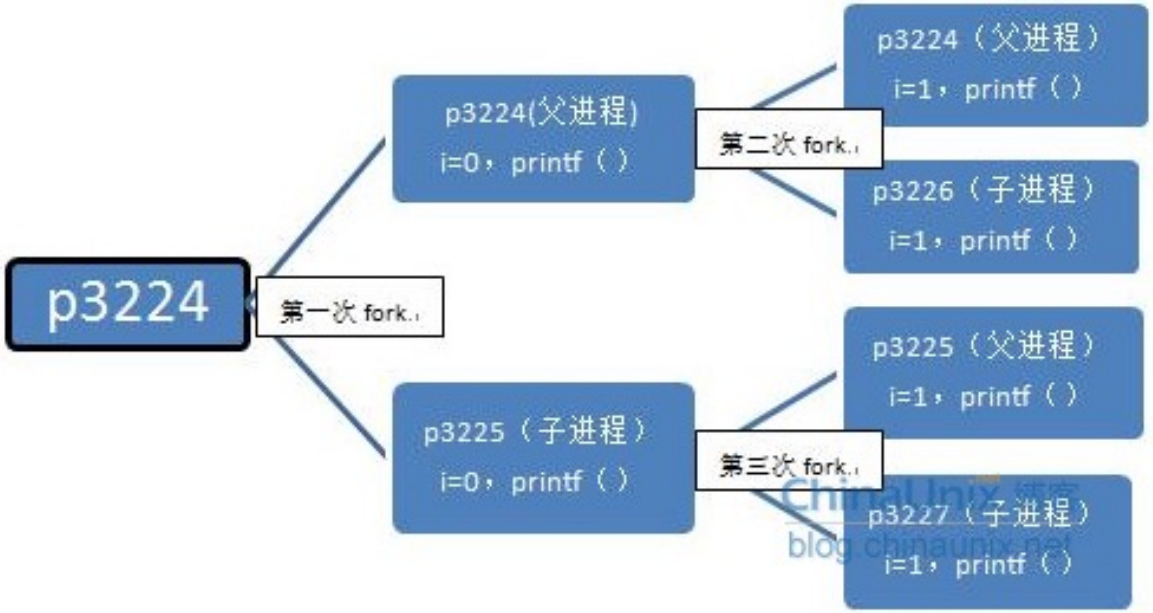
```
#include "stdafx.h"
#include <windows.h>
#include <strsafe.h>
#include <direct.h>
#include <string.h>
int _tmain(int argc, _TCHAR* argv[]){
     PROCESS_INFORMATION processInformation;
      STARTUPINFO startupInfo;
      memset(&processInformation, 0, izeof(processInformation));
      memset(&startupInfo, 0, sizeof(startupInfo));
      startupInfo.cb = sizeof(startupInfo);

      BOOL result;
      result = ::CreateProcess(AppName, CmdLine, NULL, NULL, FALSE,
      NORMAL_PRIORITY_CLASS, NULL, NULL, &startupInfo,
      &processInformation);
```

例子-父进程

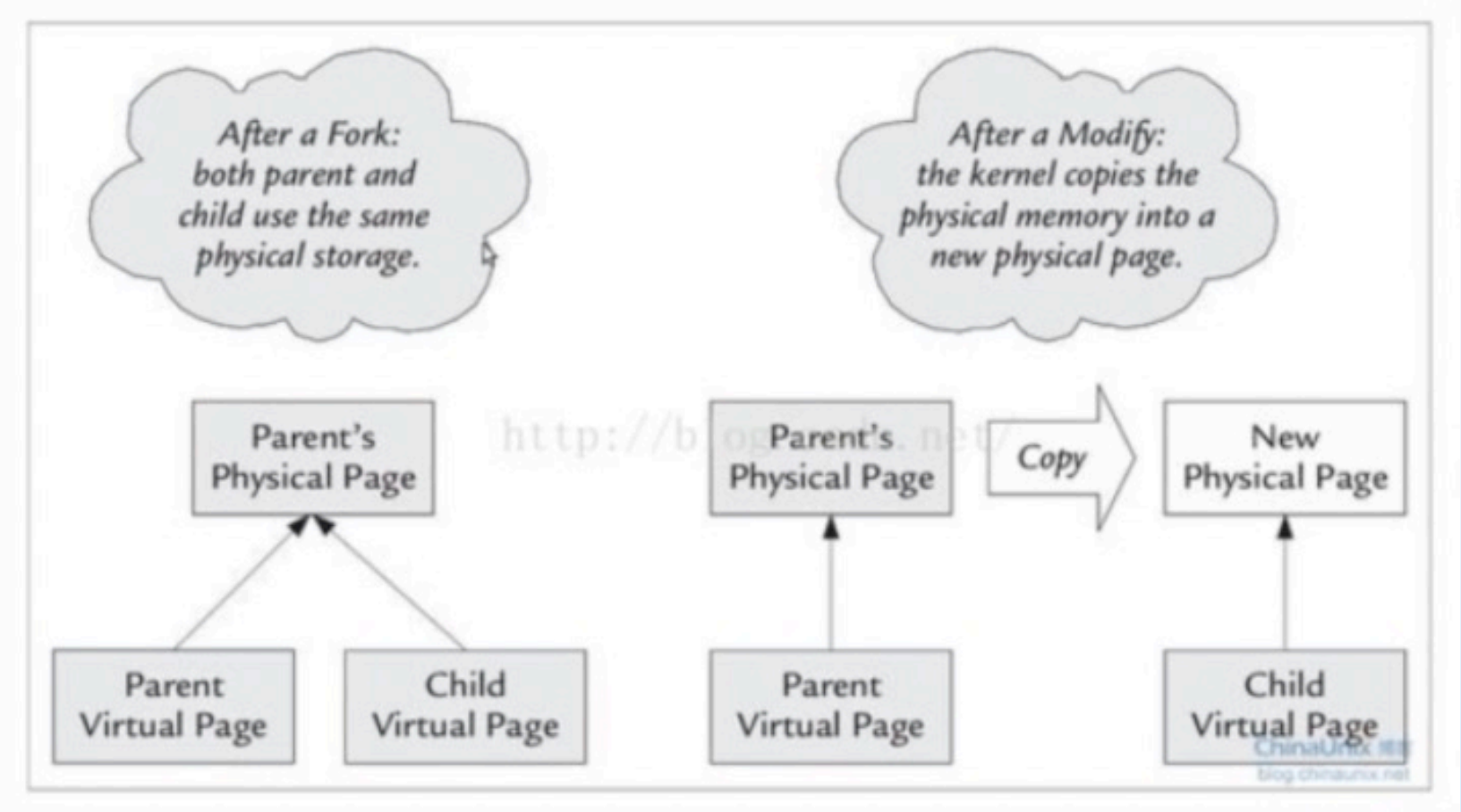
```
if (result == 0)    {
        wprintf(L"ERROR: CreateProcess failed!");
    }    else    {
       WaitForSingleObject( processInformation.hProcess, INFINITE );
        CloseHandle( processInformation.hProcess );
        CloseHandle( processInformation.hThread );
    }
return 0;
}
```

Linux进程创建

- fork 函数：
  - #include <unistd.h>
  - pid_t fork();
- 当一个进程调用fork 后会创建一个子进程
- 这个子进程和父进程不同：进程ID



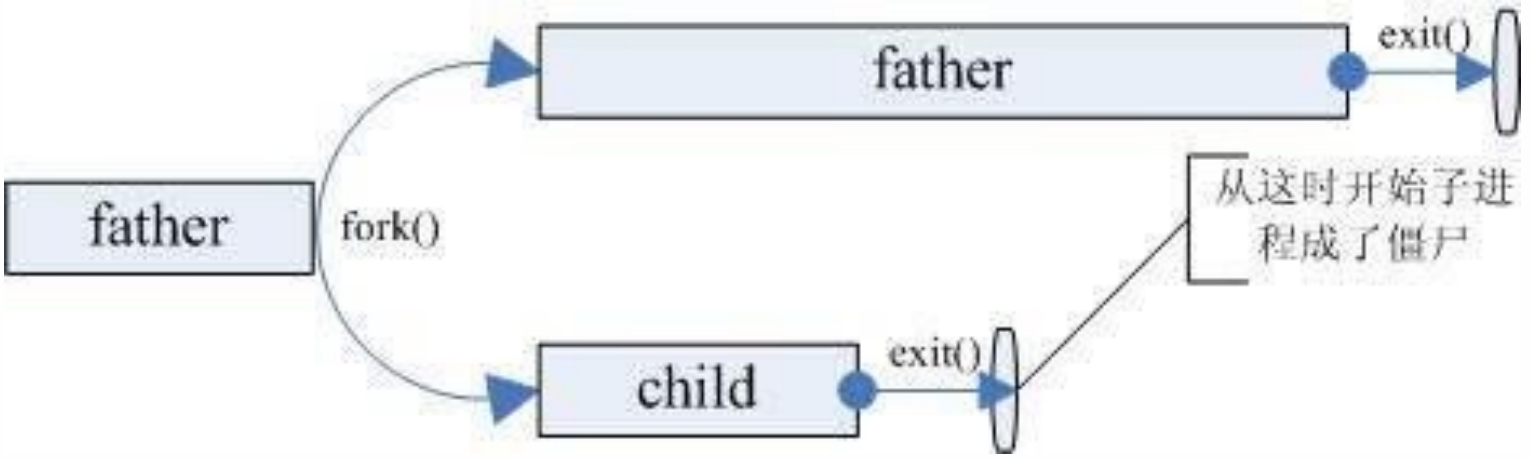
写时复制技术


After a Fork:
both parent and
child use the same
physical storage.
After a Modify:
the kernel copies the
physical memory into a
new physical page.
Parent's
Physical Page
Parent
Virtual Page
Child
Virtual Page
Parent's
Physical Page
Copy
New
Physical Page
Parent
Virtual Page
Child
Virtual Page

父进程和子进程

- 区分父进程和子进程:
  - 跟踪fork返回值
    - 失败:-1
    - 否则
      - 父进程fork 返回子进程的ID
      - fork 子进程返回0
- 可根据这个返回值来区分父子进程

fork

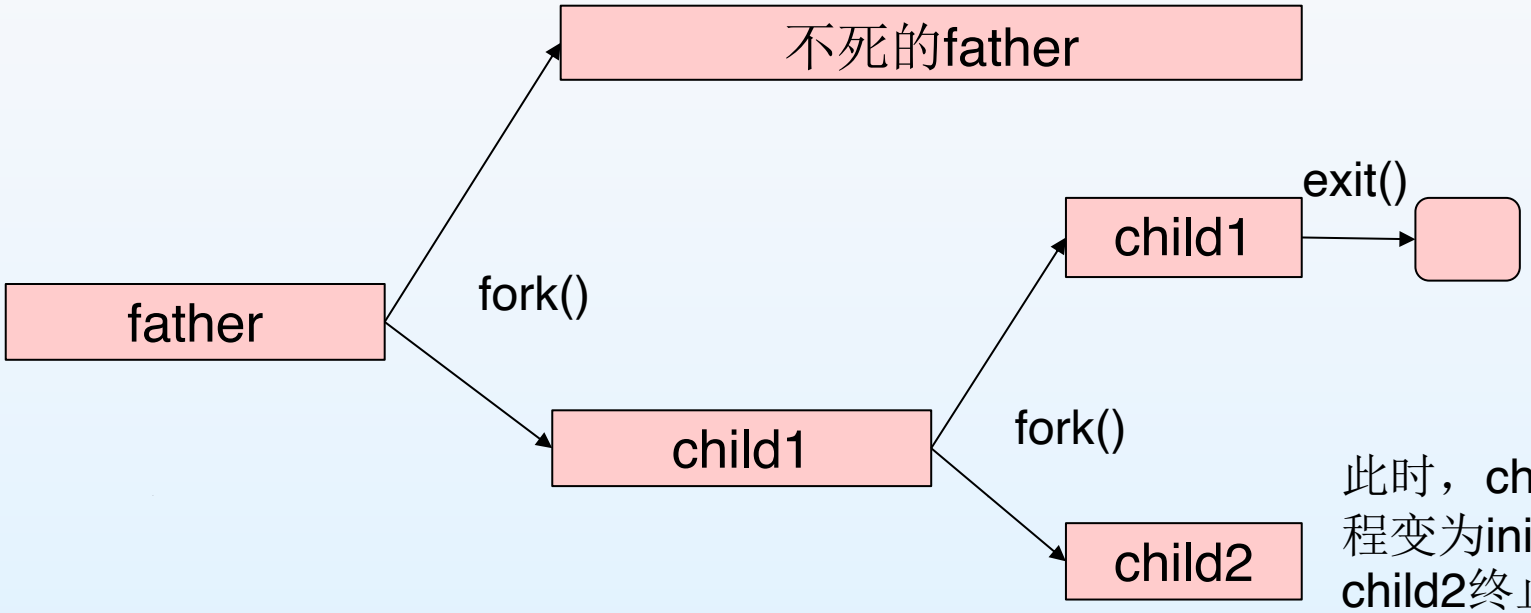


执行其它程序

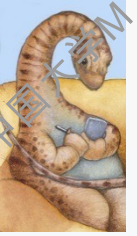
- exec 族系统调用有6个函数：
  - #include <unistd.h>
  - int execl(const char *path,const char *arg,...);
  - int execlp(const char *file,const char *arg,...);
  - int execle(const char *path,const char *arg,...char *const envp[]);
  - int execv(const char *path,char *const argv[]);
  - int execvp(const char *file,char *const argv[]);
  - int execve(const char *path,char *const argv[],char *const envp[]);
- exec 族作用：根据指定的文件名找到可执行文件，并用它来取代调用进程的内容（在调用进程内部执行一个可执行文件）
- exec族的函数执行成功后不会返回

等待

- 父进程阻塞直到子进程完成任务
  - 进程一旦调用了wait函数，将立即阻塞自己
  - 由wait自动分析是否当前进程的某个子进程已经退出
    - 如果让它找到了这样一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回；
    - 如果没有找到这样一个子进程，wait就会一直阻塞在这里，直到有一个出现为止
- 调用wait 或者waitpid 系统调用
  - #include <sys/types.h>
  - #include <sys/wait.h>
  - pid_t wait(int *stat_loc);
  - pid_t waitpid(pid_t pid,int *stat_loc,int options);

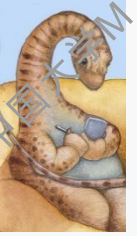


此时，child2的父进程变为init，init负责child2终止时的资源回收



例子1

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>
#include <math.h>

void main(void)
{
pid_t child;
int status;
printf("This will demostrate how to get child status\n");
if((child=fork())==-1)
{
  printf("Fork Error  : %s\n",strerror(errno));
  exit(1);
```



例子1（续）

```
} else if(child==0) {
  int i;
  printf("I am the child : %ld\n",getpid());
  for(i=0;i<1000000;i++) i++;
  i=5;
  printf("I exit with %d\n",i);
  exit(i);
}
while(((child=wait(&status))==-1)&(errno==EINTR));
}
```

fork ( )

child=fork()

child=0

child=1324

```
int i;
printf("I am the child : %ld\n",getpid());
for(i=0;i<1000000;i++) i++;
i=5;
printf("I exit with %d\n",i);
exit(i);
```

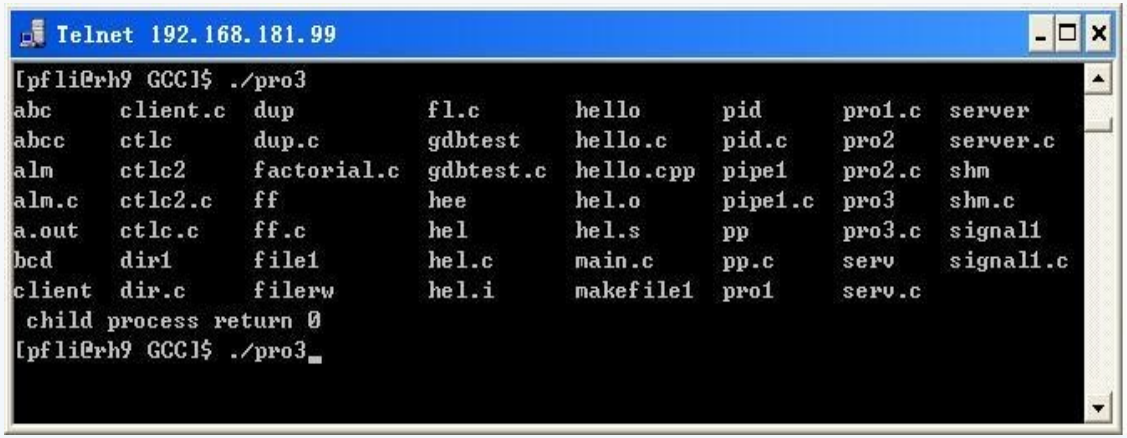
```
while(((child=wait(&status))==-1)
    &(errno==EINTR));
```

子进程

父进程

例子2

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>

int main()
{
int rtn; /*子进程的返回数值*/
if ( fork() == 0 ) {
/* 子进程执行此命令*/
execlp("/bin/ls","ls -al ",(char *)0);
/* 如果exec函数返回，表明没有正常执行命令，打印错误信息*/
exit( 1 );
}
else {
/* 父进程， 等待子进程结束，并打印子进程的返回值 */
wait ( &rtn );
printf( " child process return %d\n",. rtn );
}
}
```

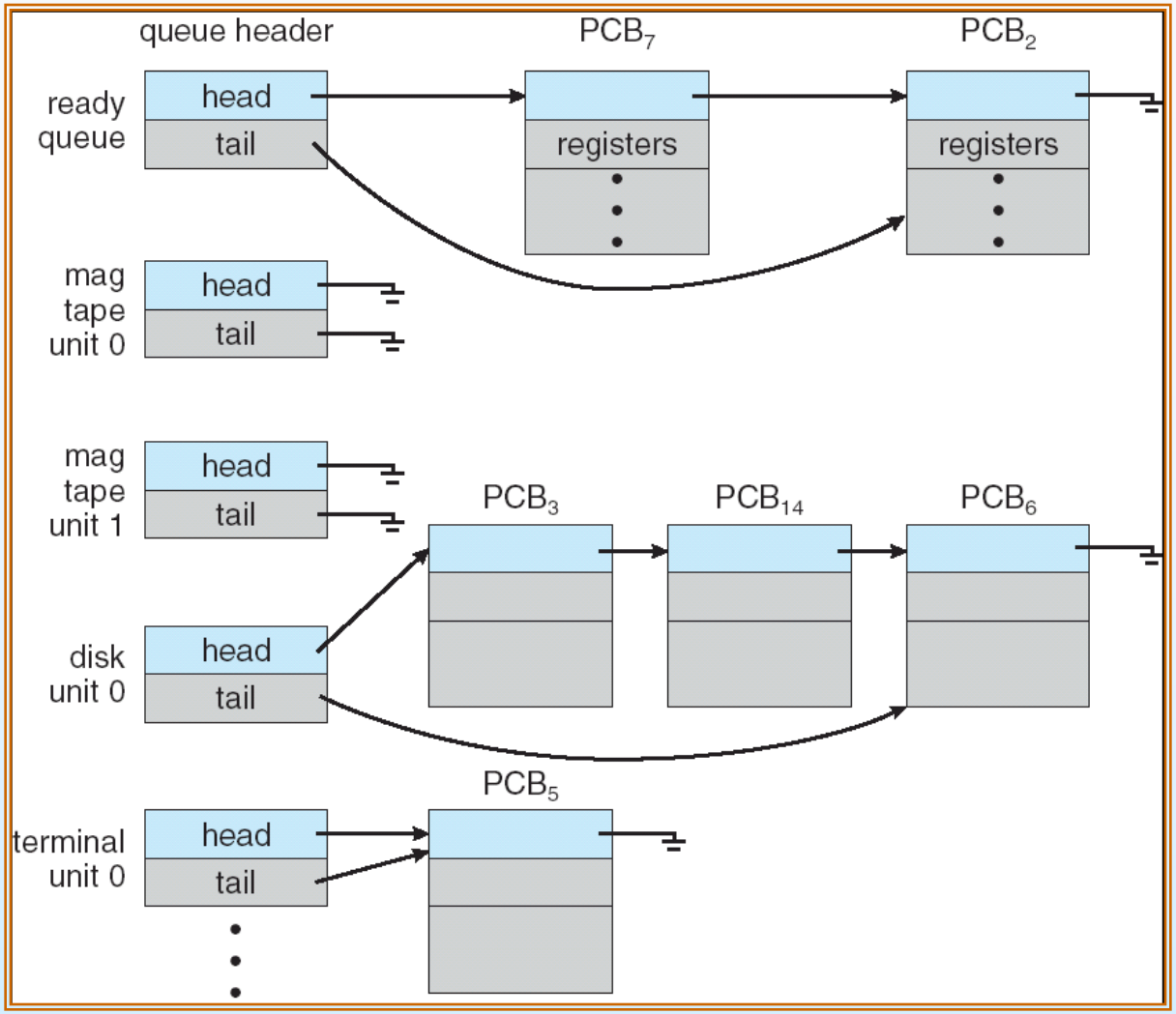


3、进程调度

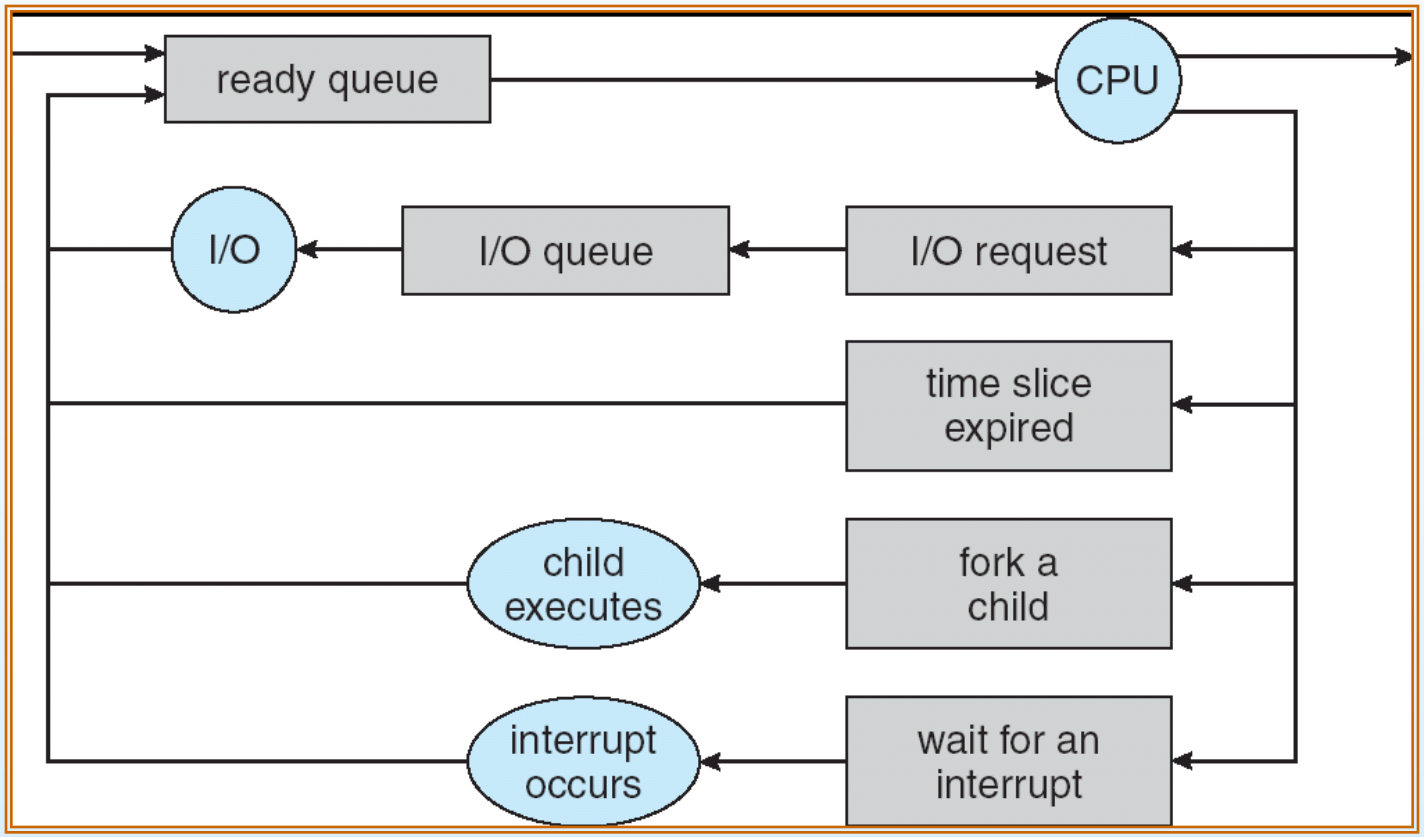
进程调度队列

- 作业队列（job queue） - 在系统中的所有进程的集合
- 就绪队列（ready queue） - 在主内存中的，就绪并等待执行的所有进程的集合
- 设备队列（device queue） - 等待某一I/O设备的进程队列
- 在各种队列之间进程的迁移

就绪队列和各种 I/O 设备队列

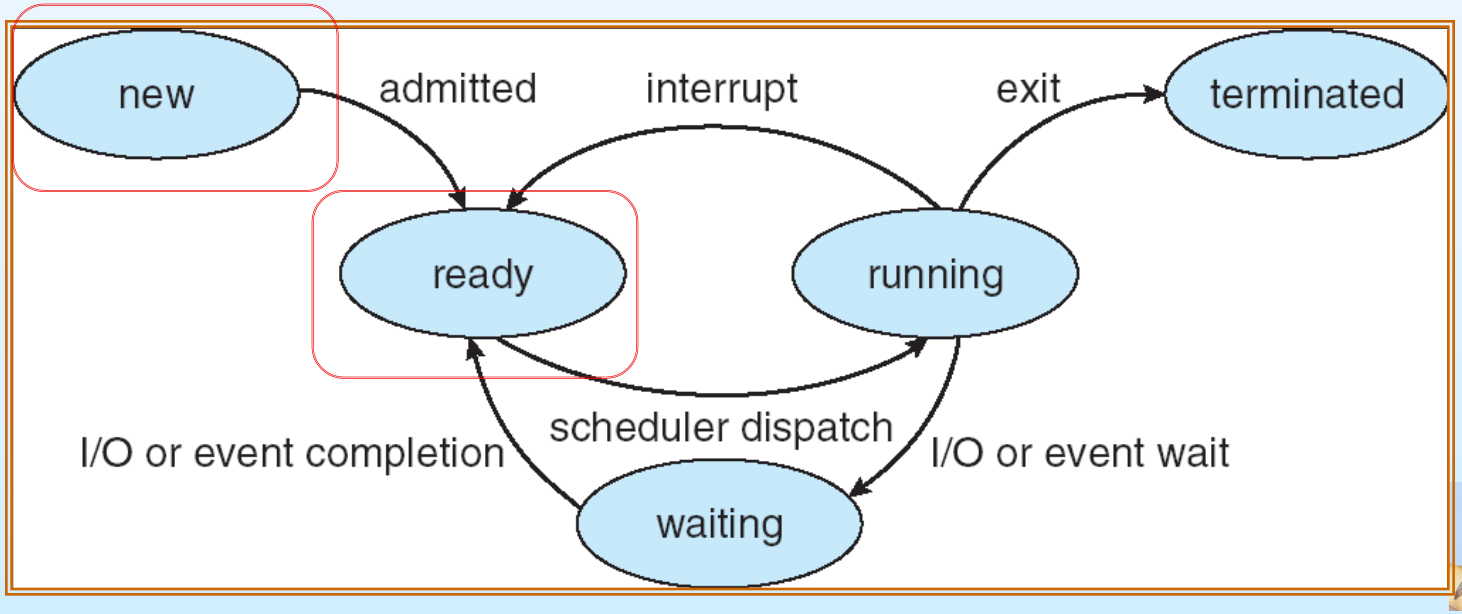


进程调度的队列表示图


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

调度程序

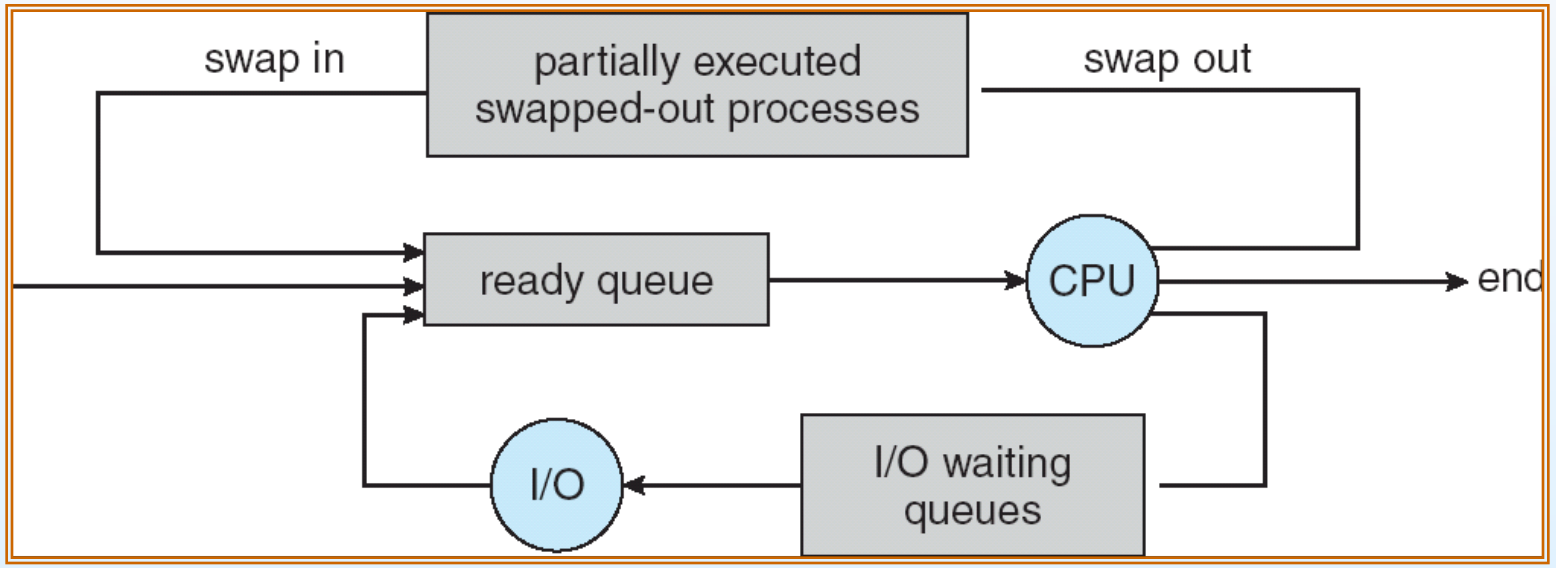
- 长程调度（或作业调度）- 选择可以进入就绪队列的进程
- 短程调度（或CPU调度）- 选择可被下一个执行并分配CPU的进程



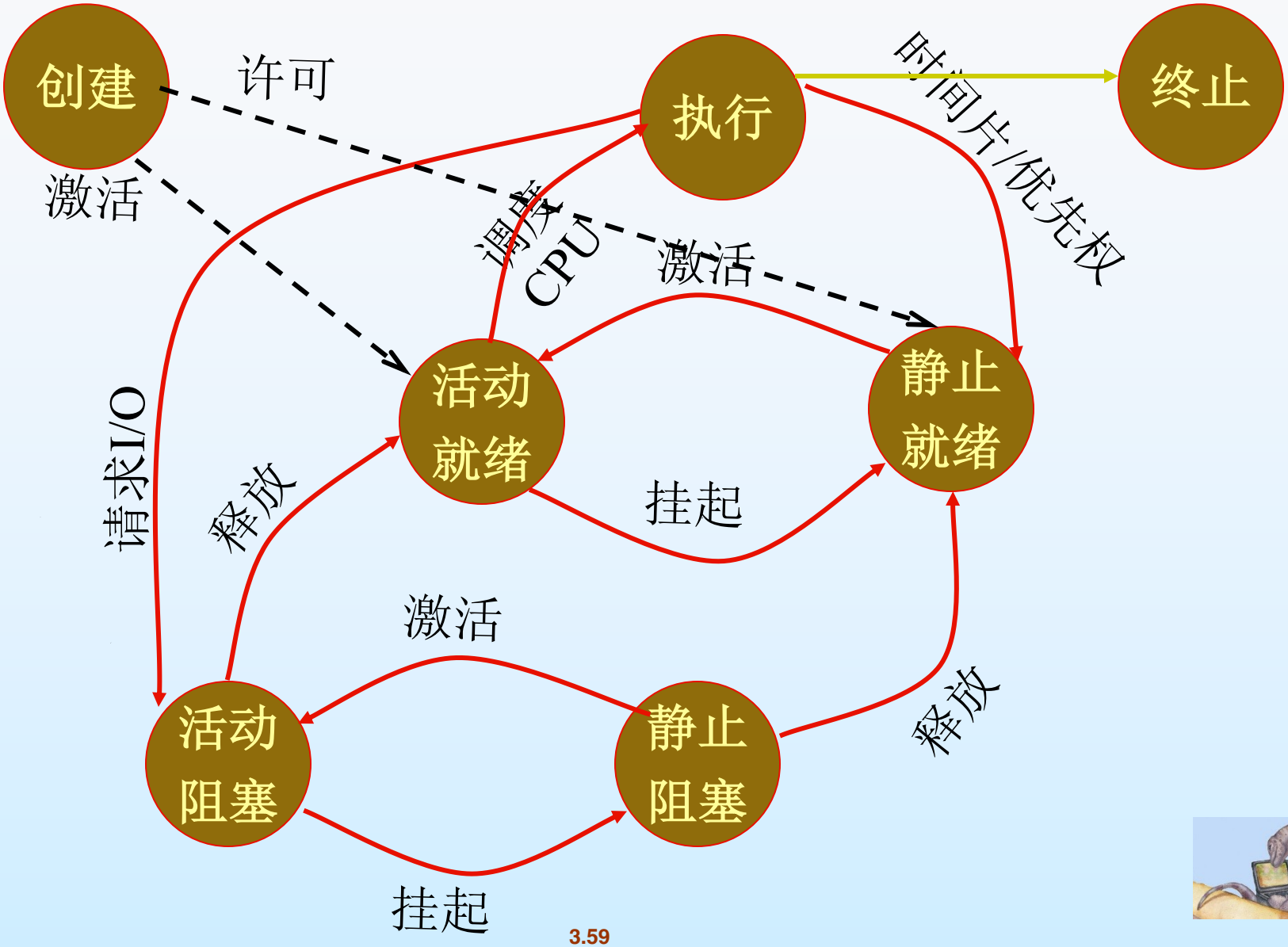
调度程序(Cont.)

- (milliseconds) ⇒ (切换必须快). 短程调度切换频率高
- (seconds, minutes) ⇒ (切换可以慢). 长程调度不快
- 长程调度控制了多道程序的“道”
- 进程可以用下列方式描述：
  - I/O型进程 - 花费I/O 时间多于计算，许多短CPU处理
  - CPU 型进程 - 花费更多时间于计算，许多长CPU处理

中程调度-交换

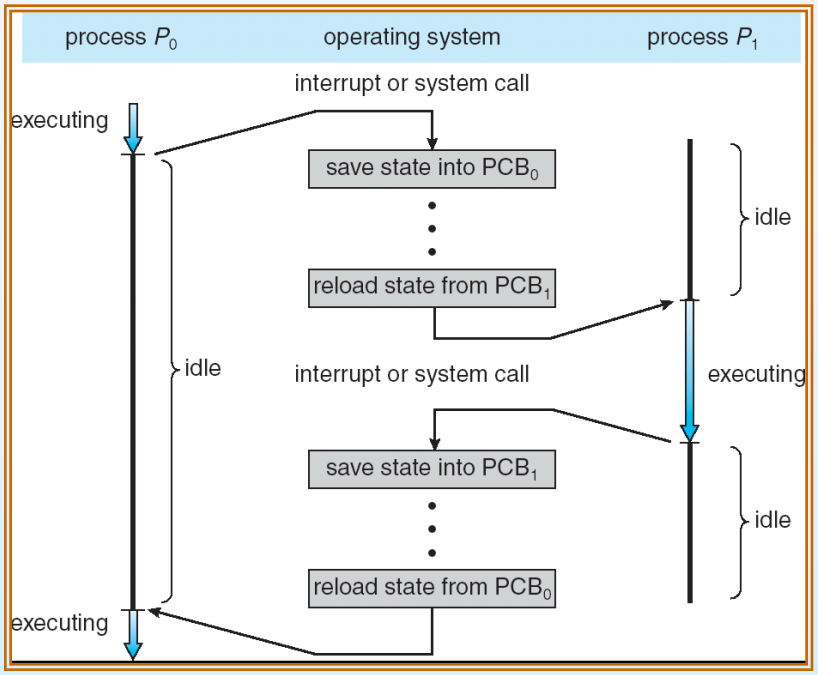

swap in
partially executed swapped-out processes
swap out
ready queue
CPU
end
I/O
I/O waiting queues

带有中程调度的进程状态图



上下文切换

- 当CPU切换至另一个进程时，系统必须保存旧进程状态并为新进程调入所保留的状态
- 上下文切换的时间开销较重；在切换时，系统没有做有用的工作
- 时间取决于硬件的支持



Android多任务

- 源于Linux，真正多任务
- 进程分前台（foreground）和后台（background）
  - 后台进程利用服务执行任务
  - 服务会保持运行即使后台进程被暂停
  - 服务没有运行界面，需要小内存

iOS多任务

- iOS 4.0前不支持多任务
  - 没落的Palm

- “伪”多任务
  - **1、保持退出时状态**
    - 一种看起来像多任务的单任务，当按下Home键时，程序会保持状态。待到下次呼出时便恢复保存时的状态
  - **2、有限多任务**
    - 一些系统允许后退运行的进程，如音乐播放和下载等。按下Home键，程序会保存到内存中
  - **3、真正多任务**
    - Safari和Mail

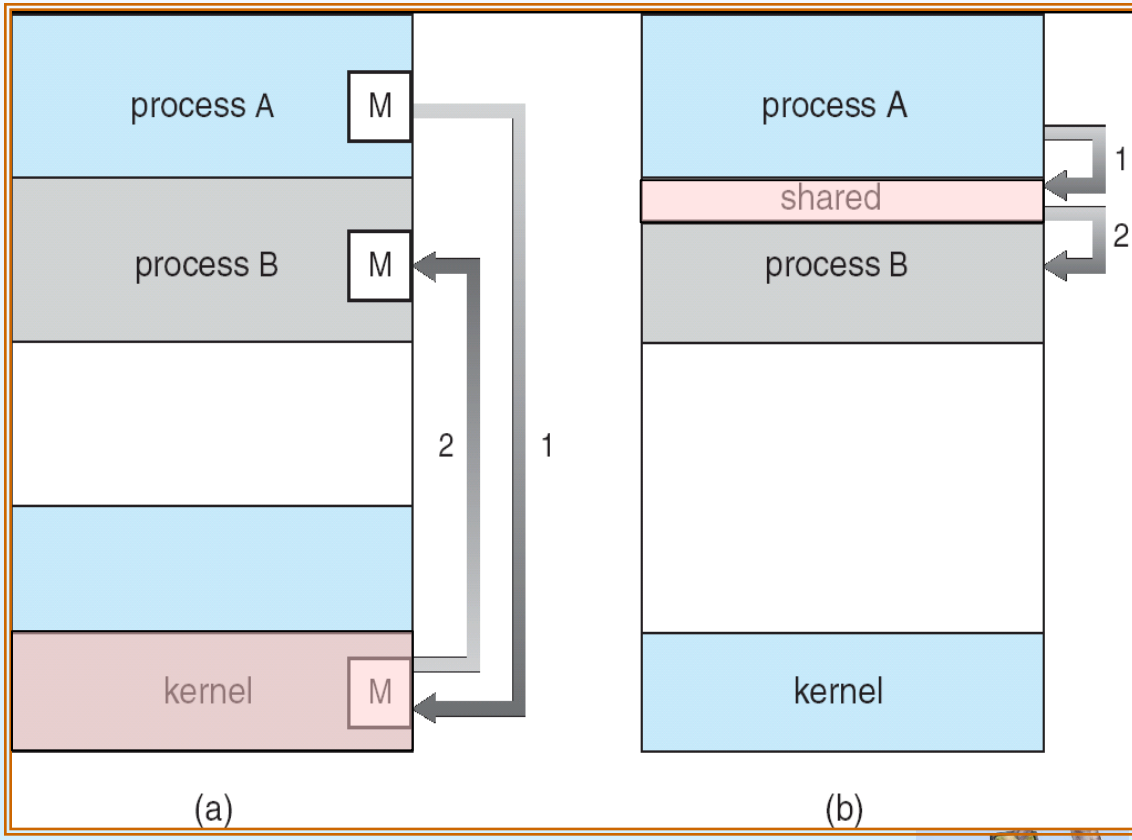
4、进程间通信

协同进程

- 独立进程：不会影响另一个进程的执行或被另一个进程执行影响
- 协同进程：可能影响另一个进程的执行或被另一个进程执行影响
- 进程协同的优点
  - 信息共享
  - 加速运算
  - 模块化
  - 方便

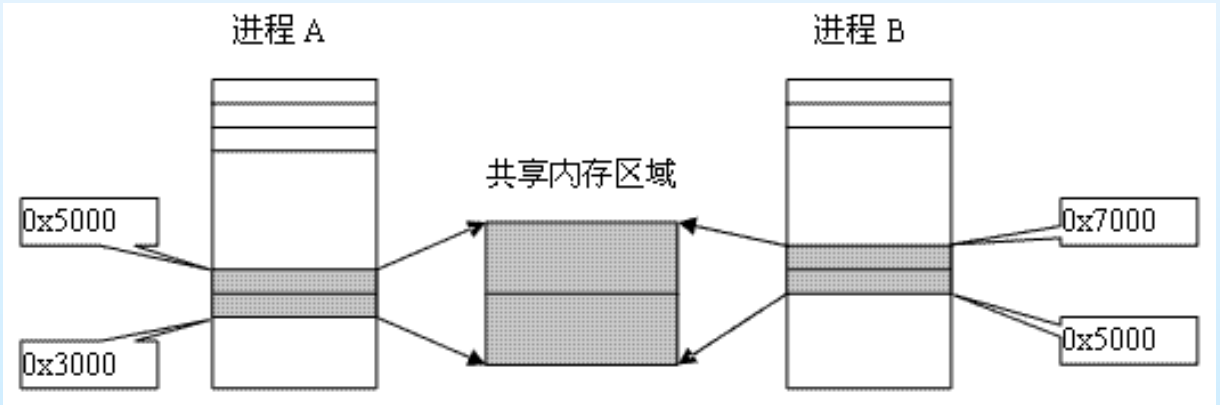
进程间通信(IPC)

- 用于进程通信的机制，同步其间的活动
- 两种基本模式：
  - 共享内存
    - 以最快的速度进行方便的通信
  - 消息传递
    - 交换较少数量的数据



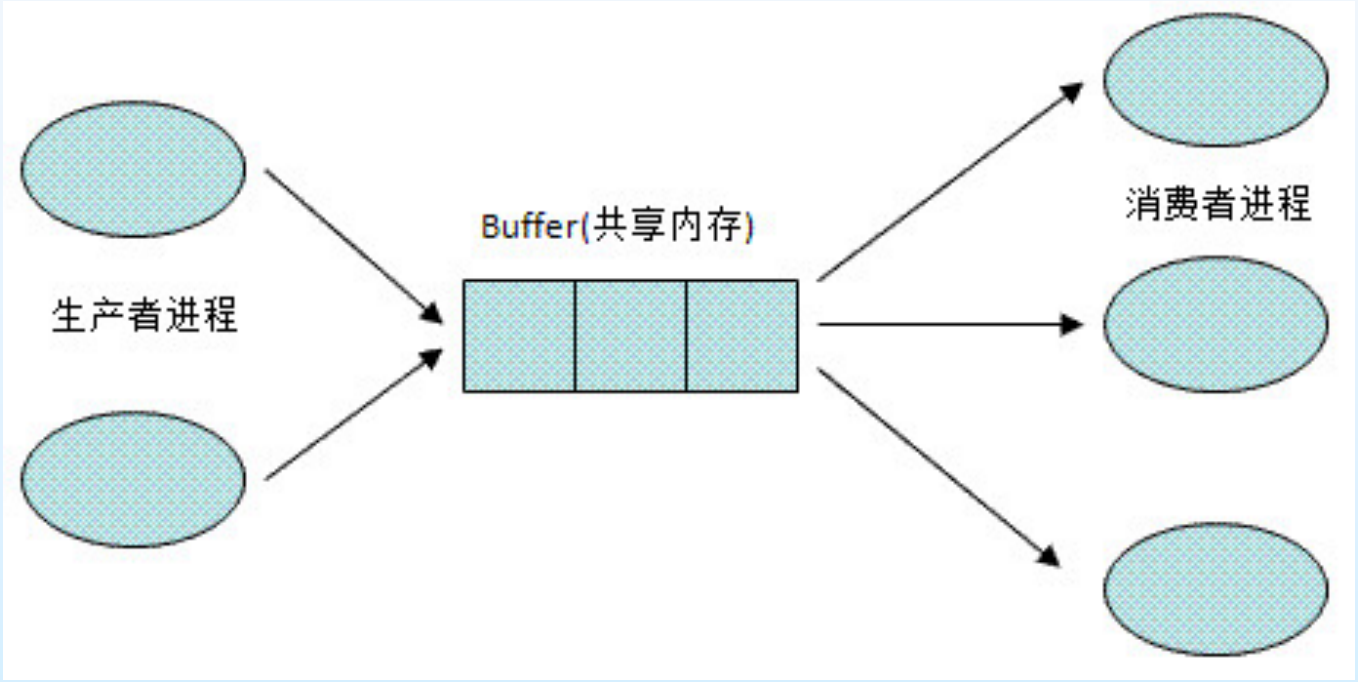
共享内存

- 一块内存在多个进程间共享
- 通信由应用程序自己控制
- 一般用于大数据通信
- 实现手段：
  - 文件映射
  - 管道
  - 剪贴板



例子：生产者-消费者

- 生产者进程生产，供消费者进程消费的信息
  - 无界缓冲(Unbounded-buffer)没有对缓冲区大小的限制
  - 有界缓冲(Bounded-buffer)对缓冲区大小作了限定



有界缓冲

- Shared data

```
#define BUFFER_SIZE 10
Typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- in指向缓冲区中下一个空位；out指向缓冲区中第一个非空位
- 但最多只能填满缓冲区的BUFFER_SIZE-1个项

生产者进程

```
item next_produced;
while (true) {
   /* Produce an item in next_produced*/
      while (((in = (in + 1) % BUFFER SIZE count)  == out)
                  ; /* do nothing - no free buffers */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER SIZE;
}
```

消费者进程

```
item next_consumed;
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume

    // remove an item from the buffer
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    /*consume the item in nextConsumed*/;
}
```

消息传递

- 消息传递在微内核中的应用
- 远程通信无法采用共享内存
- 两个原子操作：
  - 发送**send**(*message*) - 固定或可变大小消息
  - 接收**receive**(*message*)
- 若*P*与*Q*要通信，需要:
  - 建立通信连接
  - 通过send/receive交换消息
- 通信连接的实现
  - 物理的（如，共享存储，硬件总线）
  - 逻辑的（如，逻辑特性）

消息传递实现问题

- 连接如何建立？
- 连接可同多于两个的进程相关吗？
- 每对在通信进程有多少连接？
- 一个连接的容量是多少？
- 连接可使用的固定或可变消息的大小？
- 连接是单向的还是双向的？

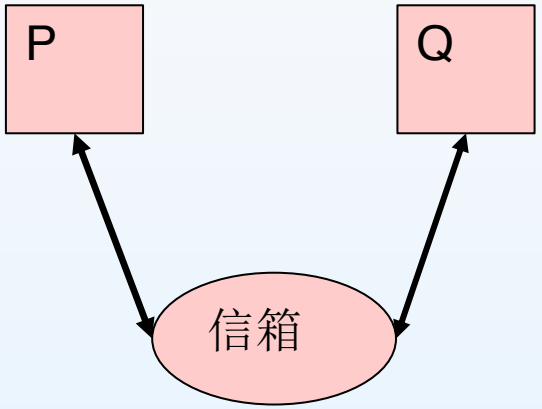
直接通信

- 进程必须显式的命名接受者和发送者
  - **send** (*P, message*) –向进程P发消息
  - **receive**(*Q, message*) –从进程Q收消息
- 通信连接的特性
  - 连接自动建立
  - 连接精确地与一对通信进程相关
  - 在每一对通信进程间存在一个连接
  - 连接可单向，但通常双向

直接通信应用例子

间接通信

- 消息导向至信箱并从信箱接收
  - 每一个信箱有一个唯一的id
  - 仅当共享一个信箱时进程才能通信
- 通信连接的特性
  - 仅当进程共有一个信箱时连接才能建立
  - 连接可同多个进程相关
  - 每一对进程可共享多个通信连接
  - 连接可是单向或双向的



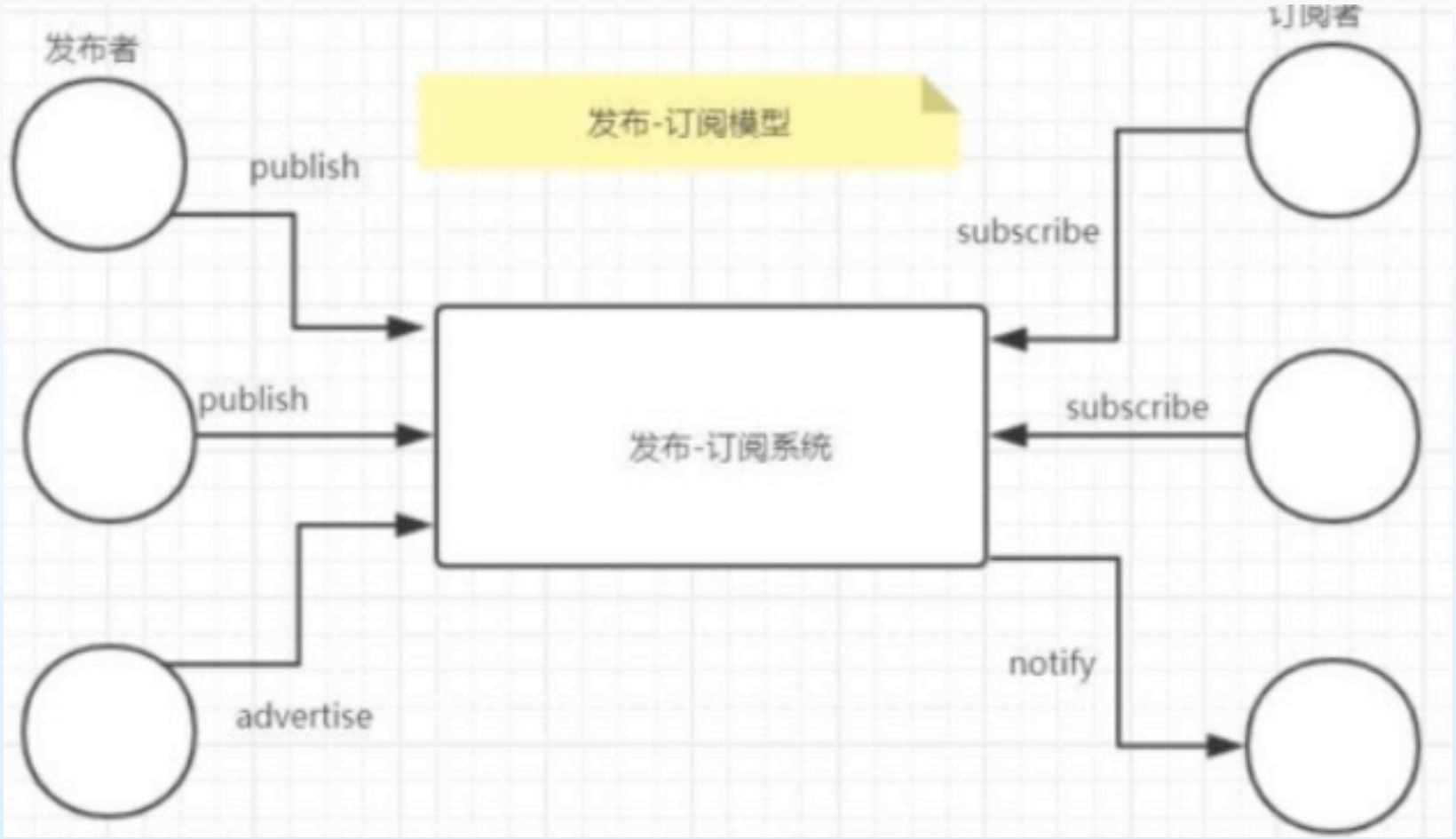
间接通信

- 操作
  - 创建新的信箱
  - 通过信箱发送和接收消息
  - 销毁信箱
- 两个原语被定义
  - **send**(*A, message*) – 发送消息到信箱 A
  - **receive**(*A, message*) – 从信箱 A接收消息

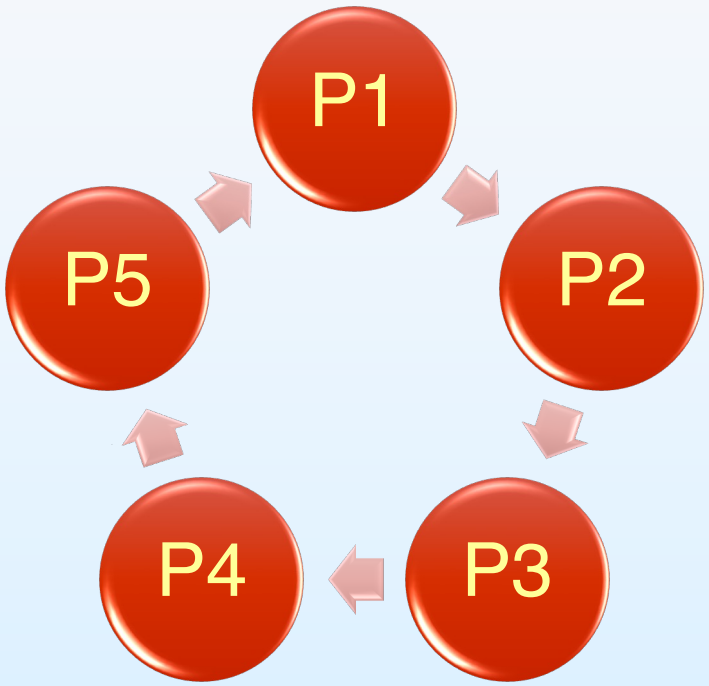
间接通信

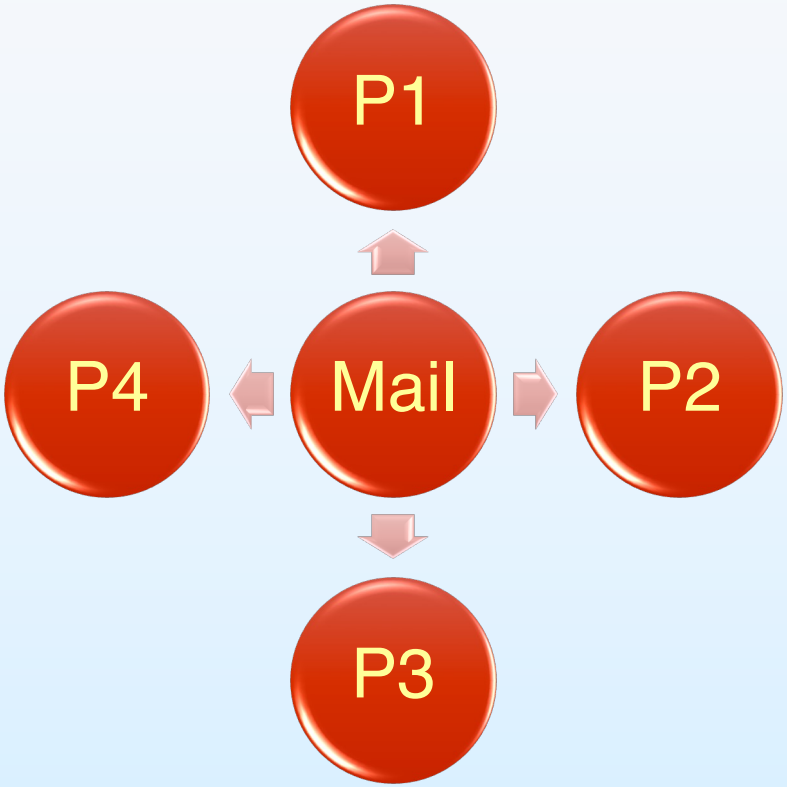
- 信箱共享
  - P_1, P_2与P_3共享信箱A
  - P_1发送； P_2与P_3接受
  - 谁得到消息？
- 解决方案
  - 允许一个连接最多同两个进程相关
  - 只允许一个时刻有一个进程执行接收操作
  - 允许系统任意选择接收者。发送者被通知谁是接收者。

间接通信应用例子


发布者
订阅者
发布-订阅模型
publish
subscribe
publish
subscribe
发布-订阅系统
notify
advertise

直接与间接的比较





同步

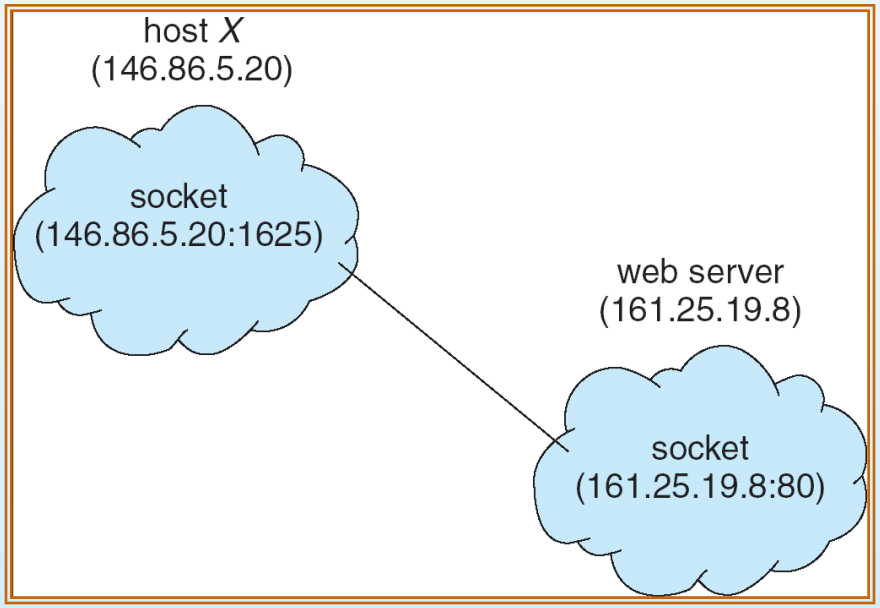
- 消息传递可阻塞（blocking）或非阻塞（non-blocking），也称为同步或异步
- 阻塞-同步
  - 阻塞send：发送进程阻塞，直到消息被接收
  - 阻塞receive：接受者进程阻塞，直到有消息可用
- 非阻塞-异步
  - 非阻塞send：发送进程发送消息并继续操作
  - 非阻塞receive: 接收者收到一个有效消息或无效消息

远程通信-客户机服务器通信

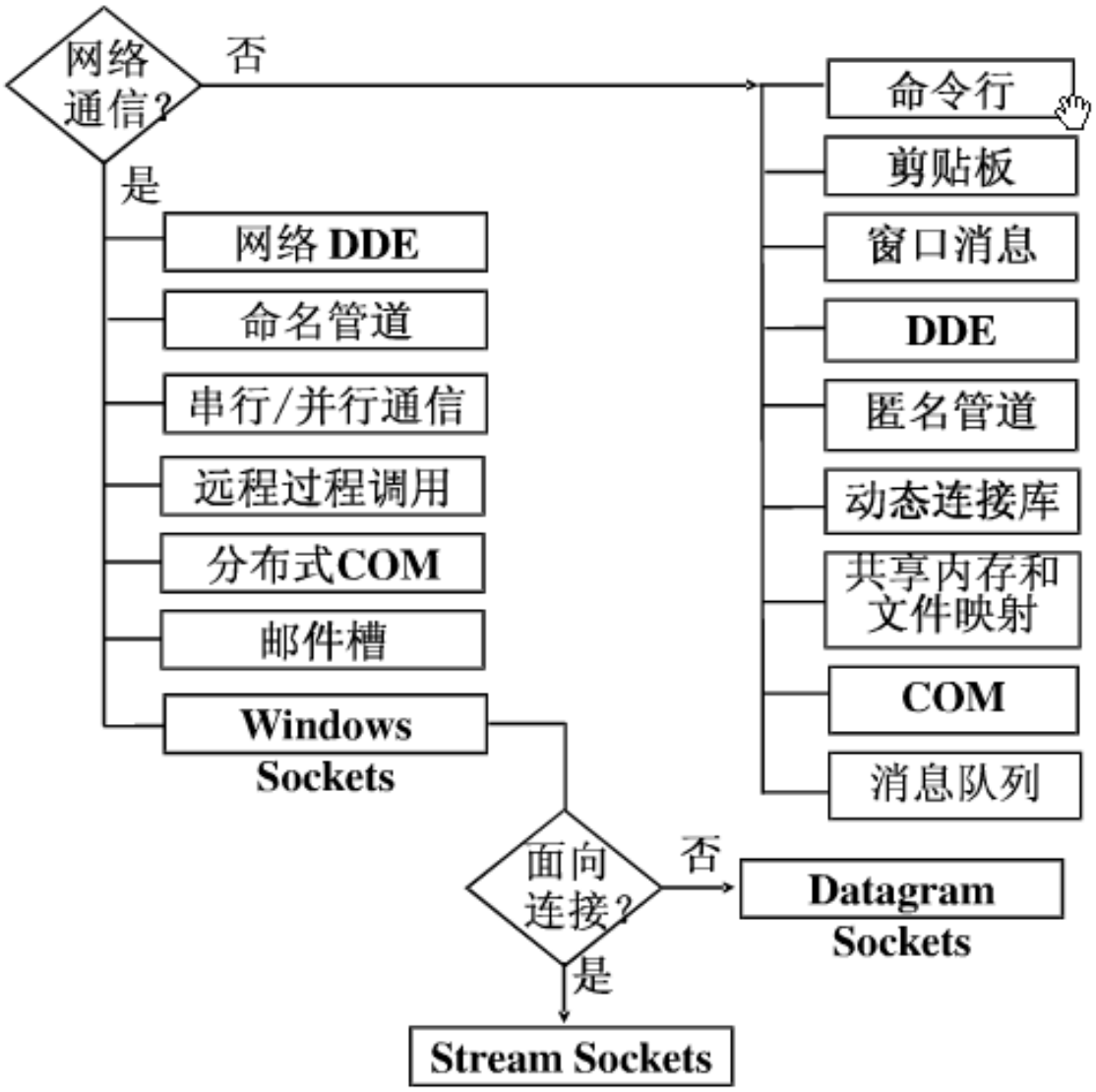
- 套接字(Socket)
- 远程过程调用(RPC)
- 远程方法调用(RMI) (Java)

Sockets

- 套接字被定义为通信的端点
- 套接字由IP地址和端口号连接组成
- **套接字161.25.19.8:80** 指的是主机**161.25.19.8**上的**80**端口
- 连接由一对套接字组成



Windows进程间通信



基于文件映射的共享存储区

- 将整个文件映射为进程虚拟地址空间的一部分来访问
  - CreateFileMapping为指定文件创建一个文件映射对象，返回对象指针
  - OpenFileMapping打开一个命名的文件映射对象，返回对象指针
  - MapViewOfFile把文件映射到本进程的地址空间，返回映射地址空间的首地址
- 利用首地址进行读写
  - FlushViewOfFile可把映射地址空间的内容写到物理文件中；
  - UnmapViewOfFile拆除文件映射与本进程地址空间间映射关系；
- 随后，利用CloseHandle关闭文件映射对象

例子-主程序

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#define BUF_SIZE 256
TCHAR szName[]=TEXT("MyFileMappingObject111");
TCHAR szMsg[]=TEXT("Message from first process.");
int _tmain() {
   HANDLE hMapFile;      LPCTSTR pBuf;
   hMapFile = CreateFileMapping(
         INVALID_HANDLE_VALUE,    // use paging file
          NULL,                    // default security
          PAGE_READWRITE,       // read/write access
          0,                // maximum object size (high-order DWORD)
         BUF_SIZE, // maximum object size (low-order DWORD)
         szName);    // name of mapping object
   if(hMapFile == NULL){
               _tprintf(TEXT ”Could not create file mapping object (%d). \n,
GetLastError());
               return 1;
}
```

例子-主程序

```
pBuf = (LPTSTR) MapViewOfFile(hMapFile,   // handle to map object
                    FILE_MAP_ALL_ACCESS, // read/write permission
                    0,
                    0,
                    BUF_SIZE);
 if(pBuf == NULL){
              _tprintf(TEXT ”Could not map view of file(%d). \n, GetLastError());
              CloseHanle(hMapFile);
              return 1;

 CopyMemory((PVOID)pBuf, szMsg, (_tcslen(szMsg) * sizeof(TCHAR)));

  _getch();
   UnmapViewOfFile(pBuf);
   CloseHandle(hMapFile);
   return 0;
}
```

例子-子程序

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#pragma comment(lib, "user32.lib")

#define BUF_SIZE 256
TCHAR szName[]=TEXT("MyFileMappingObject111");

int _tmain() {
   HANDLE hMapFile;
   LPCTSTR pBuf;
```

例子-子程序

```
hMapFile = OpenFileMapping(
                FILE_MAP_ALL_ACCESS,   // read/write access
                FALSE,                 // do not inherit the name
                szName);               // name of mapping object

if  (hMapFile == NULL){
                _tprintf(TEXT ”Could not open file mapping object (%d). \n,
                 GetLastError());
                return 1;
}
 pBuf = (LPTSTR) MapViewOfFile(hMapFile, // handle to map object
                FILE_MAP_ALL_ACCESS,  // read/write permission
                0,
                0,
                BUF_SIZE);

 if(pBuf == NULL){
                _tprintf(TEXT ”Could not map view of file(%d). \n, GetLastError());
                CloseHanle(hMapFile);
                return 1;
}
```

例子-子程序

```
 MessageBox(NULL, pBuf, TEXT("Process2"), MB_OK);

   UnmapViewOfFile(pBuf);  //拆除文件映射
   CloseHandle(hMapFile);   //关闭文件映射对象
   return 0;
}
```

剪帖板(Clipboard)

- 当进程间的复杂信息交流需要约定交流信息的格式。剪帖板就是Windows 提供的一种信息交流方式，可增强进程的信息交流能力。
- Windows 提供了一组相关的API来完成应用进程与剪帖板间的格式化信息交流。
- 当执行复制操作时，应用程序将选中的数据以标准的格式或者应用程序定义的格式放到剪贴板中，然后其他的应用程序可以从剪贴板中以其可以支持的格式获取所需要的数据。

剪帖板信息格式

- 剪帖板中提供了许多标准的剪帖板信息格式。如：文本格式和位图格式。
- 允许用户进程注册新的剪帖板信息格式。

Format Type	*Description*
Text Formats	
CF_OEMTEXT	Text containing characters from the OEM character set
CF_TEXT	Text containing characters from the ANSI character set
CF_UNICODETEXT	Text containing Unicode characters
Bitmap formats	
CF_BITMAP	Device-dependent bitmap (HBITMAP)
CF_DIB	Device independent bitmap (HBITMAPINFO)
CF_TIFF	Tagged Image File Format

- 与剪帖板相关的API
  - OpenClipboard: 打开剪帖板；
  - CloseClipboard：关闭剪帖板；
  - EmptyClipboard：清空剪帖板；
  - SetClipboardData：把数据及其格式加入剪帖板；
  - GetClipboardData：从剪帖板读取数据；
  - RegisterClipboardFormat：注册剪帖板格式

Linux 进程通信

- 主要手段：
  - 管道(Pipe)
  - 信号(Signal)
  - 消息(Message)
  - 共享内存(Shared memory)
  - 信号量(Semaphore)
  - 套接口(Socket)

共享内存

- System 提供了以下几个函数以实现共享内存：
  - #include <sys/types.h>
  - #include <sys/ipc.h>
  - #include <sys/shm.h>
  - int shmget(key_t key,int size,int shmflg);
  - void *shmat(int shmid,const void *shmaddr,int shmflg);
  - int shmdt(const void *shmaddr);
  - int shmctl(int shmid,int cmd,struct shmid_ds *buf);
- size 是共享内存的大小
- shmat 是用来连接共享内存
- shmdt 是用来断开共享内存的

例子

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define PERM S_IRUSR|S_IWUSR
int main(int argc,char **argv)
{
        int shmid;
        char *p_addr,*c_addr;
        if(argc!=2)   { printf("Usage：%s\n\a",argv[0]); exit(1); }

        if((shmid=shmget(IPC_PRIVATE,1024,PERM))==-1){
                printf("Create Share Memory Error\n\a");
                exit(1);
        }
```

例子

```
        if(fork()){
                p_addr=shmat(shmid,0,0);
                memset(p_addr,'\0',1024);
                strncpy(p_addr,argv[1],1024);
                exit(0);
        }
        else{
                c_addr=shmat(shmid,0,0);
                printf("Client get %s \n",c_addr);
                exit(0);
        }
}
```

```
chuxiaomindeMacBook-Air:ch03 wendy$ ./message "hello"
Client get hello
chuxiaomindeMacBook-Air:ch03 wendy$
```

消息队列

- #include <sys/types.h>;
- #include <sys/ipc.h>;
- #include <sys/msg.h>;

- int msgget(key_t key,int msgflg);
- int msgsnd(int msgid,struct msgbuf *msgp,int msgsz,int msgflg);
- int msgrcv(int msgid,struct msgbuf *msgp,int msgsz, long msgtype,int msgflg);
- int msgctl(Int msgid,int cmd,struct msqid_ds *buf);
- struct msgbuf {
- long msgtype; /* 消息类型*/
- ..... /* 其他数据类型*/
- }

server.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <sys/msg.h>

#define MSG_FILE "server.c"
#define BUFFER 255
#define PERM S_IRUSRIS_IWUSR

struct msgtype {
long mtype;
char buffer[BUFFER+1];
};
```

server.c

```
int main(){
        struct msgtype msg;  key_t key;  int msgid;

        if((key=ftok(MSG_FILE,'a'))==-1) {
          printf("Creat Key Error\n");
          exit(1);
        }

        if((msgid=msgget(key,PERM|IPC_CREAT|IPC_EXCL))==-1){
          printf("Creat Message Error\n");
          exit(1);
        }

        while(1){
           msgrcv(msgid,&msg,sizeof(struct msgtype),1,0);
           printf("Server Receive：%s\n",msg.buffer);
           msg.mtype=2;
           msgsnd(msgid,&msg,sizeof(struct msgtype),0);
        }
        exit(0);
}
```

client.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>

#define MSG_FILE "server.c"
#define BUFFER 255
#define PERM S_IRUSRIS_IWUSR

struct msgtype {
long mtype;
char buffer[BUFFER+1];
};
```

client.c

```
int main(int argc,char **argv){
        struct msgtype msg;
        key_t key; int msgid;

        if(argc!=2){
          printf("Usage：%s string\n\a",argv[0]);
          exit(1);
        }
        if((key=ftok(MSG_FILE,'a'))==-1){
         printf("Creat Key Error：\n");
         exit(1);
        }

        if((msgid=msgget(key,PERM))==-1){
          printf("Creat Message Error：\a\n");
          exit(1);
        }
        msg.mtype=1;
        strncpy(msg.buffer,argv[1],BUFFER);
        msgsnd(msgid,&msg,sizeof(struct msgtype),0);
        memset(&msg,'\0',sizeof(struct msgtype));
        msgrcv(msgid,&msg,sizeof(struct msgtype),2,0);
        printf("Client receive：%s\n",msg.buffer);
        exit(0);
}
```

Telnet 192.168.181.99

```
[pfli@rh9 GCC]$ ./server &
[1] 3709
[pfli@rh9 GCC]$ ./client hello
Server Receive: hello
Client receive: hello
[pfli@rh9 GCC]$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 32768      pfli       600        1024       0
0x00000000 65537      pfli       600        1024       0
0x00000000 98306      pfli       600        1024       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x6102c2ad 32768      pfli       600        0            0

[pfli@rh9 GCC]$ kill 3709
[pfli@rh9 GCC]$ ipcrm -q 32768
[1]+  Terminated              ./server
[pfli@rh9 GCC]$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 32768      pfli       600        1024       0
0x00000000 65537      pfli       600        1024       0
0x00000000 98306      pfli       600        1024       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```