

# 操作系统原理

# PRINCIPLES OF OPERATING SYSTEM

北京大学计算机科学技术系 陈向群

Department of computer science and Technology

Peking University

2015 春季

# 第3讲

## 进程/线程模型

# 进程/线程模型

---

## ◎ 进程模型

- 多道程序设计
- 进程的概念、进程控制块
- 进程状态及转换、进程队列
- 进程控制  
进程创建、撤销、阻塞、唤醒、.....

## ◎ 线程模型

- 为什么引入线程？
- 线程的组成
- 线程机制的实现  
用户级线程、核心级线程、混合方式

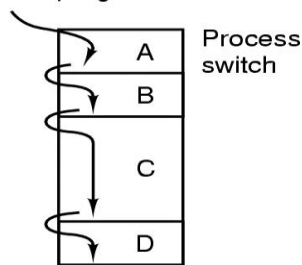
# 进程基本概念

# 多道程序设计 (MULTIPROGRAMMING)

## 多道程序设计

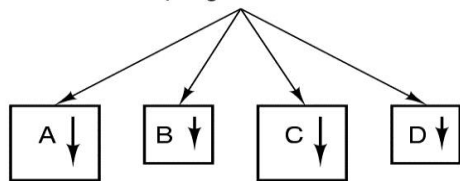
允许多个程序同时进入内存并运行，其目的是为了提高系统效率

1个程序计数器  
One program counter

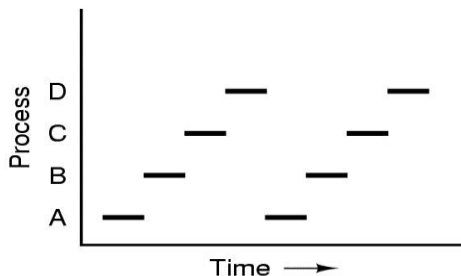


(a)

4个程序计数器  
Four program counters



(b)



轮流执行

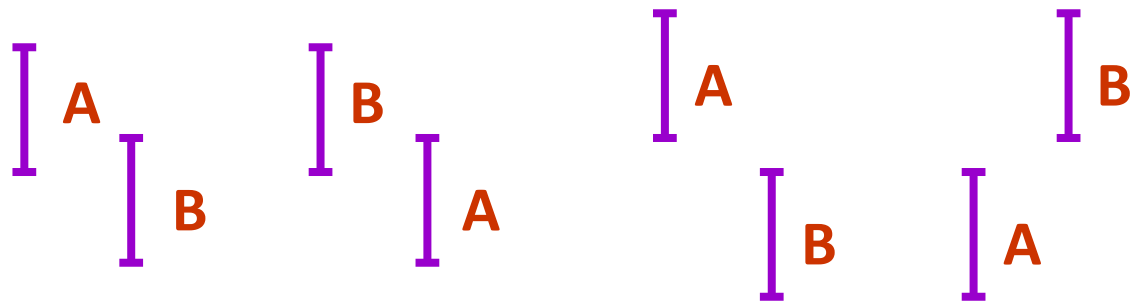
(c)

# 并发环境与并发程序

---

并发环境：

一段时间间隔内，单处理器上有两个或两个以上的程序同时处于开始运行但尚未结束的状态，并且次序不是事先确定的



并发程序：在并发环境中执行的程序

# 进程的定义

对CPU的抽象

定义: Process

进程是具有独立功能的程序关于 **某个数据集集合上的一次运行活动**，是系统进行资源分配和**调度**的独立单位

又称 任务 (Task or Job)

如何查看当前系统中有多少个进程?

- 程序的一次执行过程
- 是正在运行程序的抽象
- 将一个CPU变幻成多个虚拟的CPU
- 系统资源以进程为单位分配，如内存、文件、.....  
每个具有独立的地址空间
- 操作系统将CPU **调度**给需要的进程

# 进程控制块PCB

---

- ◎ **PCB: Process Control Block**

- 又称 **进程描述符、进程属性**
- 操作系统用于管理控制进程的一个专门数据结构
- 记录进程的各种属性，描述进程的动态变化过程

- ◎ **PCB是系统感知进程存在的唯一标志**

→ 进程与PCB是一一对应的

- ◎ **进程表: 所有进程的PCB集合**



PCB的内容应  
该包括什么  
呢？







## 进程描述信息

- ◆ 进程标识符(process ID), 唯一, 通常是一个整数
- ◆ 进程名, 通常基于可执行文件名, 不唯一
- ◆ 用户标识符(user ID)
- ◆ 进程组关系





## 进程控制信息

- ◆ 当前状态
- ◆ 优先级(priority)
- ◆ 代码执行入口地址
- ◆ 程序的磁盘地址
- ◆ 运行统计信息(执行时间、页面调度)
- ◆ 进程间同步和通信
- ◆ 进程的队列指针
- ◆ 进程的消息队列指针







## 所拥有的资源和使用情况

- ◆ 虚拟地址空间的状况
- ◆ 打开文件列表



进程不运行时，操作系统要保存哪些硬件执行状态呢？



## CPU现场信息

- ◆ 寄存器值(通用寄存器、程序计数器PC、程序状态字PSW、栈指针)
- ◆ 指向该进程页表的指针



# 换个角度看PCB的内容

<u>Process management</u>	<u>Memory management</u>	<u>File management</u>
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Linux: task\_struct  
Windows: EPROCESS、KPROCESS、PEB

# LINUX TASK\_STRUCT(1)

Linux 2.6.x

```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags; /* per process flags, defined below */
    unsigned long ptrace;

    int lock_depth; /* Lock depth */

    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;

    unsigned long sleep_avg;
    long interactive_credit;
    unsigned long long timestamp;
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;

    struct list_head tasks;
    /*
     * ptrace_list/ptrace_children forms the list of my children
     * that were stolen by a ptracer.
     */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;
} /* task state */
struct linux_binfmt *binfmt;
int exit_code, exit_signal;
int pdeath_signal; /* The signal sent when the parent dies */
/* ??? */
unsigned long personality;

int did_exec:1;
pid_t pid;
pid_t tgid;
/*
 * pointers to (original) parent process, youngest child,
 * younger sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->parent->pid)
 */
struct task_struct *real_parent; /* real parent process (when
being debugged) */
struct task_struct *parent; /* parent process */
/*
 * children/sibling forms the list of my children plus the
 * tasks I'm ptracing.
 */
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list
*/
struct task_struct *group_leader; /* threadgroup leader */

/* PID/PID hash table linkage. */
struct pid_link pids[PIDTYPE_MAX];

wait_queue_head_t wait_chldexit; /* for wait4() */
struct completion *vfork_done; /* for vfork() */
int __user *set_child_tid; /* CLONE_CHILD_SETTID */
int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */

unsigned long rt_priority;
unsigned long it_real_value, it_prof_value, it_virt_value;
unsigned long it_real_incr, it_prof_incr, it_virt_incr;
struct timer_list real_timer;
unsigned long utime, stime, cutime, cstime;
unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw; /* context
switch counts */
u64 start_time;
```



# LINUX TASK\_STRUCT(2)

Linux 2.6.x

```
/* mm fault and swap info: this can arguably be seen as either
mm-specific or thread-specific */
unsigned long min_flt, maj_flt, cmin_flt, cmaj_flt;
/* process credentials */
uid_t uid, euid, suid, fsuid;
gid_t gid, egid, sgid, fsgid;
struct group_info *group_info;
kernel_cap_t cap_effective, cap_inheritable,
cap_permitted;
int keep_capabilities:1;
struct user_struct *user;
/* limits */
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];
/* file system info */
int link_count, total_link_count;
/* ipc stuff */
struct sysv_sem sysvsem;
/* CPU-specific state of this task */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* namespace */
struct namespace *namespace;
/* signal handlers */
struct signal_struct *signal;
struct sighand_struct *sighand;

sigset_t blocked, real_blocked;
struct sigpending pending;

unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
```

```
void *notifier_data;
sigset_t *notifier_mask;

void *security;
struct audit_context *audit_context;

/* Thread group tracking */
u32 parent_exec_id;
u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty */
spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock,
write_lock_irq(&tasklist_lock); */
spinlock_t proc_lock;
/* context-switch lock */
spinlock_t switch_lock;

/* journalling filesystem info */
void *journal_info;

/* VM state */
struct reclaim_state *reclaim_state;

struct dentry *proc_dentry;
struct backing_dev_info *backing_dev_info;

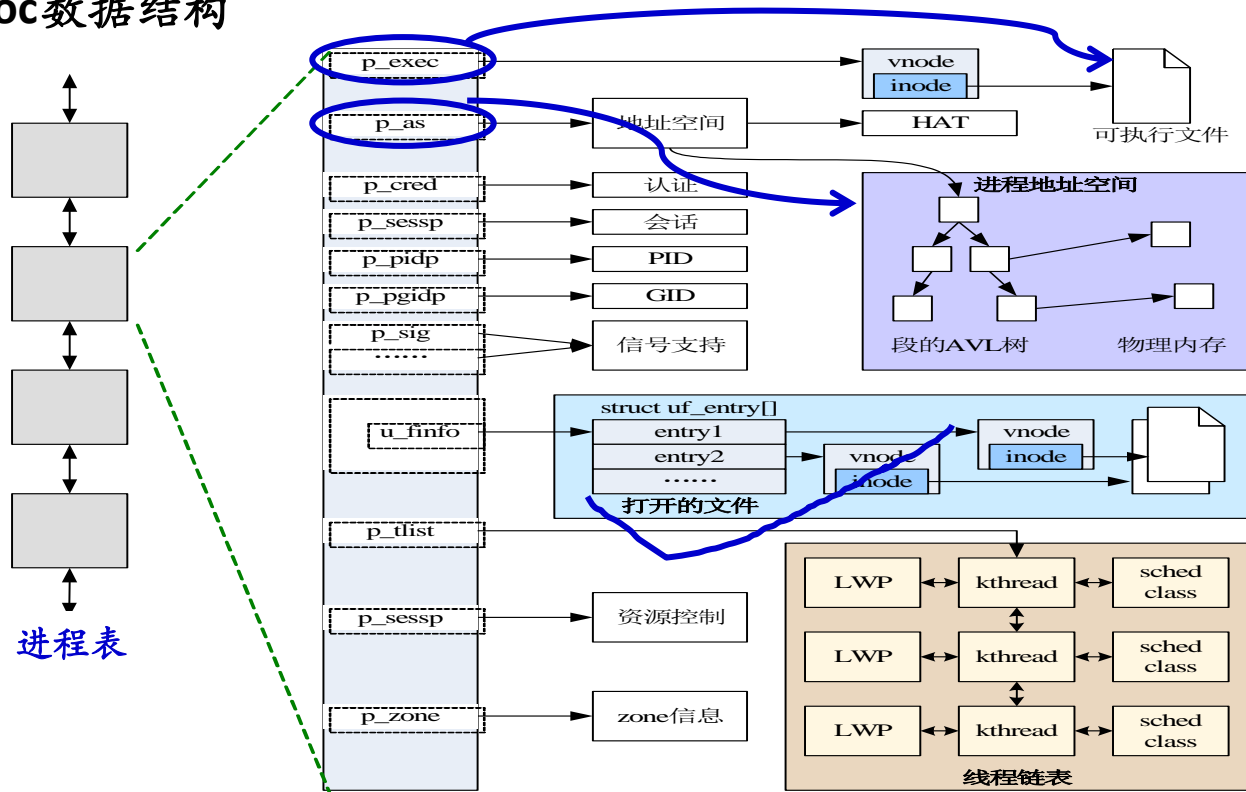
struct io_context *io_context;

unsigned long ptrace_message;
siginfo_t *last_siginfo; /* For ptrace use. */

#ifdef CONFIG_NUMA
struct mempolicy *mempolicy;
short il_next; /* could be shared with used_math */
#endif
};
```

# SOLARIS的进程控制块与进程表

## proc数据结构



三状态模型、五状态模型、七状态模型

# 进程状态及状态转换

# 进程的三种基本状态

---

## ◎ 进程的三种基本状态： 运行态、就绪态、等待态

- 运行态（Running）

占有CPU，并在CPU上运行

- 就绪态（Ready）

已经具备运行条件，但由于没有空闲CPU，而暂时不能运行

- 等待态（Waiting/Blocked）

因等待某一事件而暂时不能运行

阻塞态、封锁态、睡眠态

如：等待读  
盘结果

# 三状态模型及状态转换

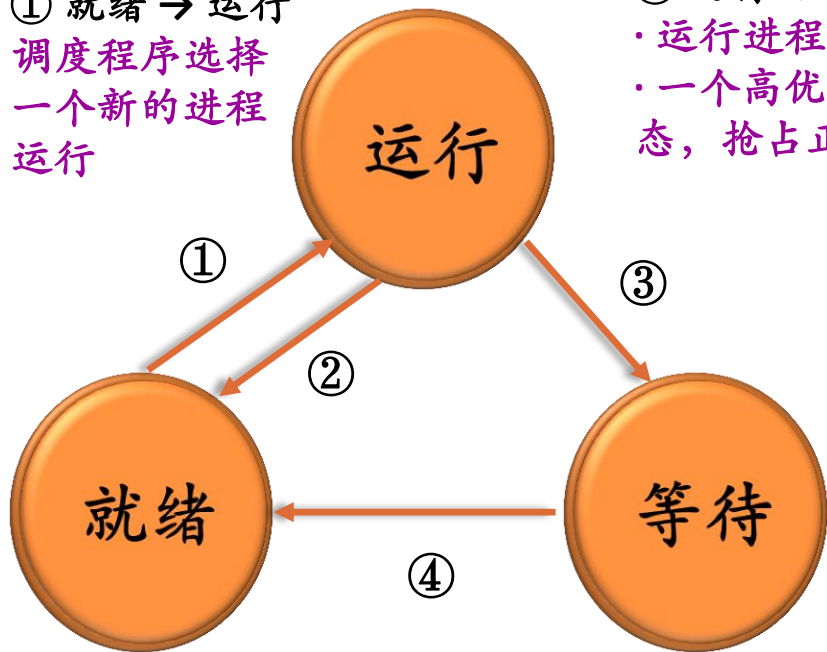
① 就绪 → 运行  
调度程序选择  
一个新的进程  
运行

② 运行 → 就绪  
· 运行进程用完了时间片  
· 一个高优先级进程进入就绪状态, 抢占正在运行的进程

③ 运行 → 等待  
当一个进程等待某个事件发生时

- 请求OS服务
- 对资源的访问尚不能进行
- 等待I/O结果
- 等待另一进程提供信息
- ... ..

④ 等待 → 就绪  
所等待的事件发生了



# 进程的其他状态

创建  
new

- 已完成创建一进程所必要的工作
  - PID、PCB
- 但尚未同意执行该进程
  - 因为资源有限

终止  
Terminated

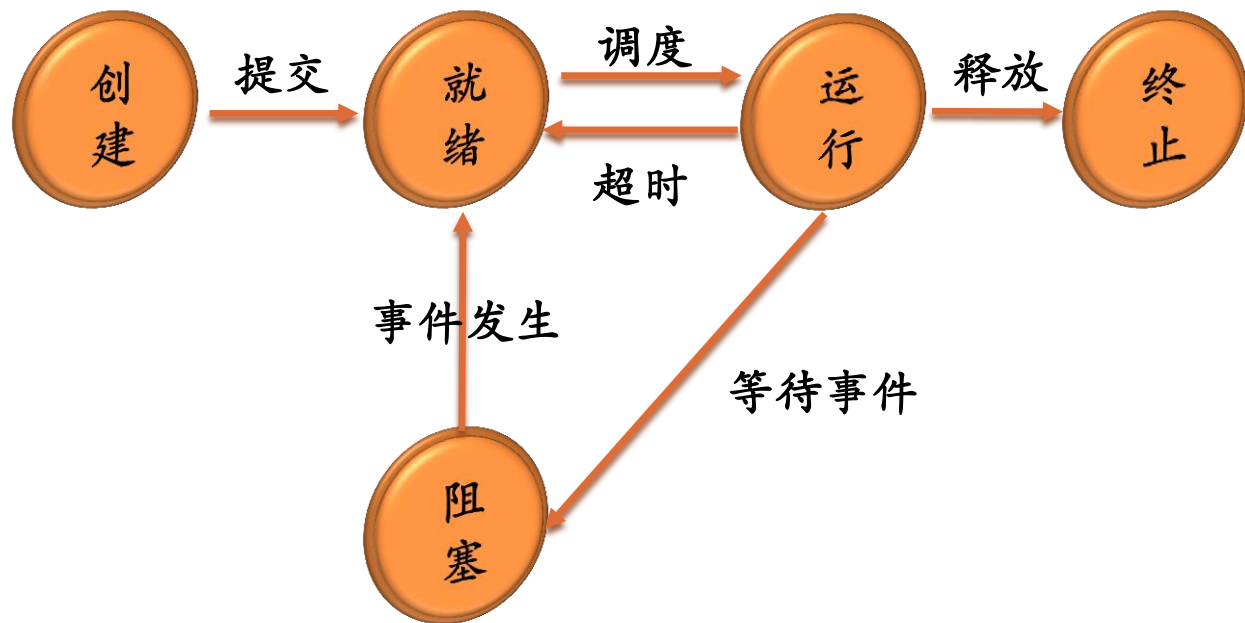
- 终止执行后，进程进入该状态
- 可完成一些数据统计工作
- 资源回收

挂起  
suspend

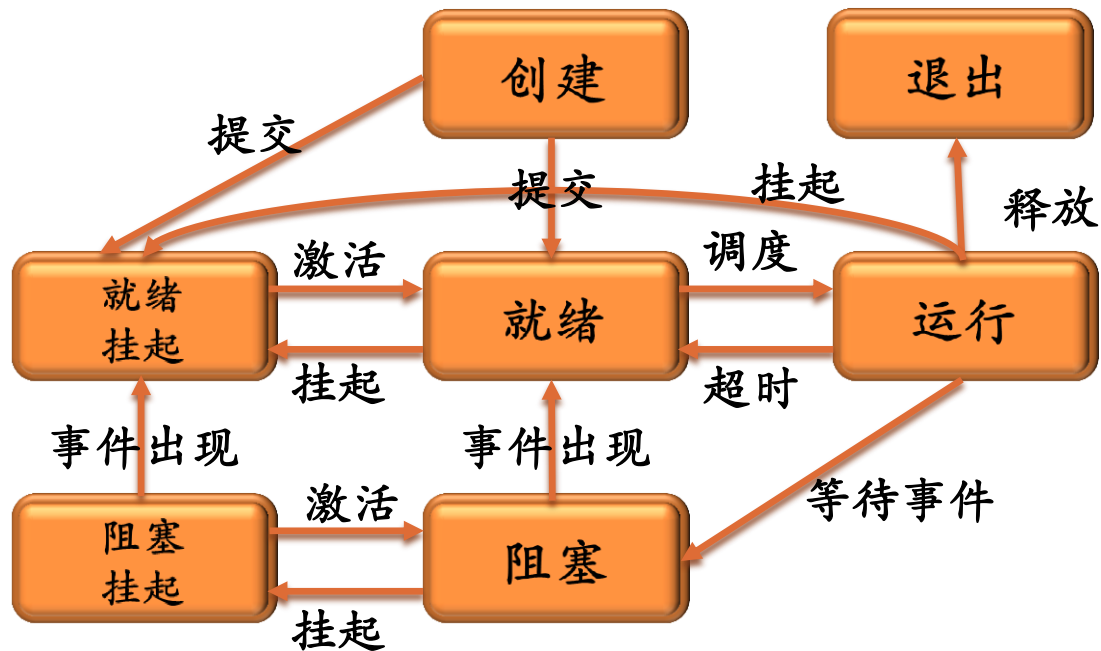
- 用于调节负载
- 进程不占用内存空间，其进程映像交换到磁盘上

# 五状态进程模型

---

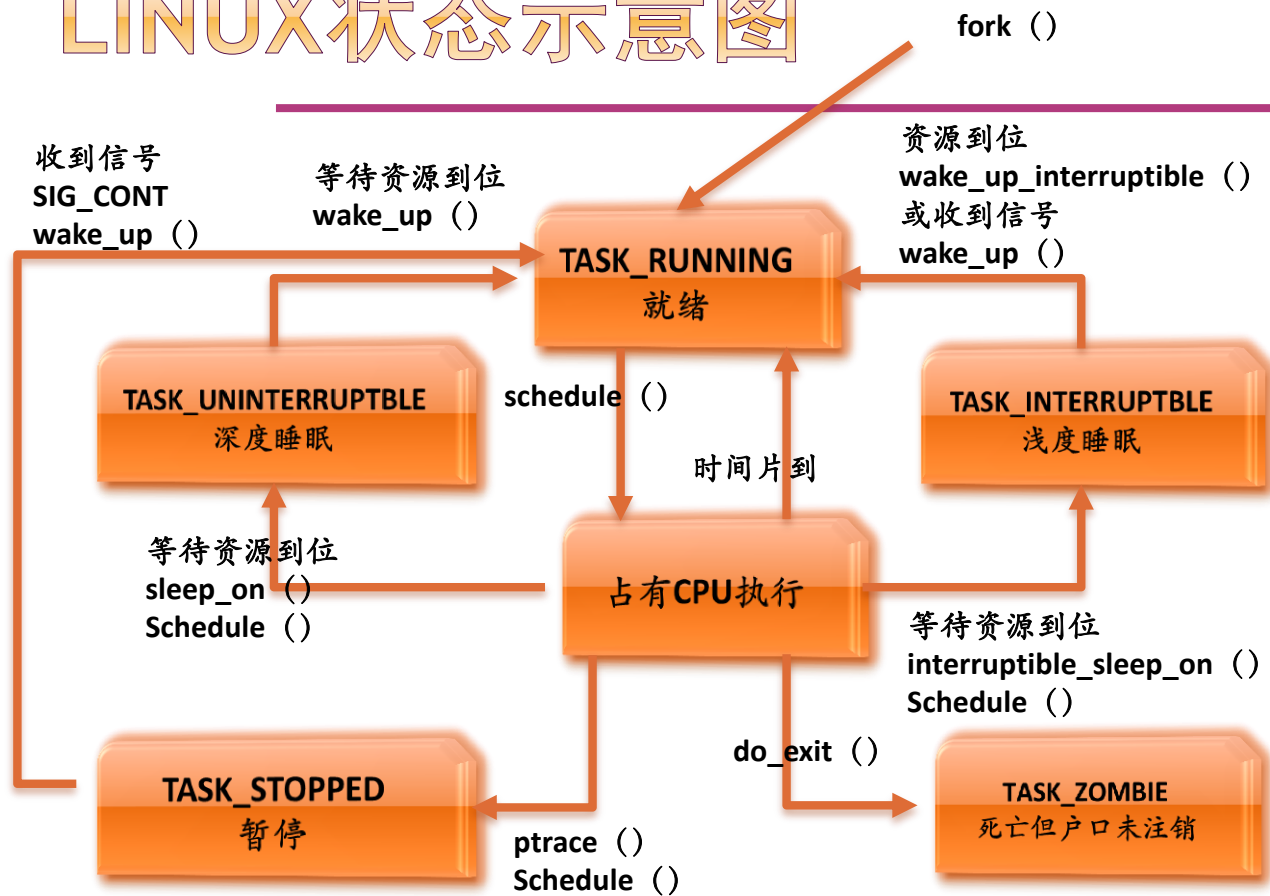


# 七状态进程模型



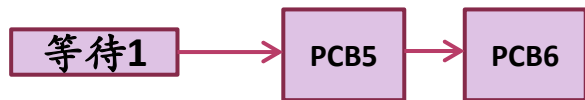
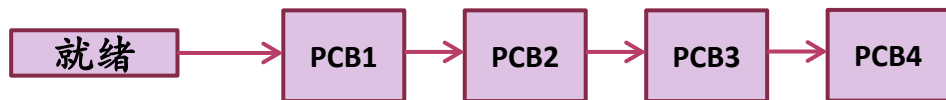


# LINUX状态示意图



# 进程队列

- 操作系统为每一类进程建立一个或多个队列
- 队列元素为**PCB**
- 伴随进程状态的改变，其**PCB**从一个队列进入另一个队列

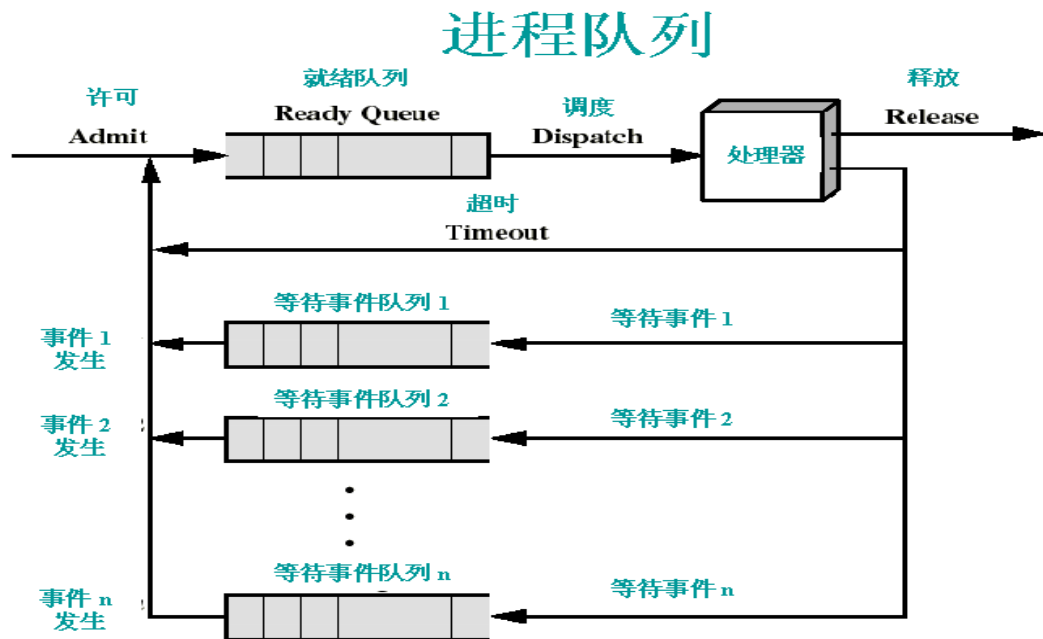


.....



- 多个等待队列等待的事件不同
- 就绪队列也可以多个
- 单CPU情况下，运行队列中只有一个进程

# 五状态进程模型的队列模型



创建、撤销、阻塞、唤醒……

# 进程控制

# 进程控制

进程控制操作完成进程各状态之间的转换，由具有特定功能的**原语**完成

- 进程创建原语
- 进程撤消原语
- 阻塞原语
- 唤醒原语
- 挂起原语
- 激活原语
- 改变进程优先级
- .....

## 原语 (primitive)

完成某种特定功能的一段程序，具有不可分割性或不可中断性

即原语的执行必须是连续的，在执行过程中不允许被中断


原子操作 (atomic)

# 1.进程的创建

---

- ◎ 给新进程分配一个唯一标识以及进程控制块
- ◎ 为进程分配地址空间
- ◎ 初始化进程控制块
  - 设置默认值 (如: 状态为 **New**, ...)
- ◎ 设置相应的队列指针

如: 把新进程加到就绪队列链表中



UNIX: fork/exec  
WINDOWS: CreateProcess

## 2.进程的撤消

---

### 结束进程

- ◎ 收回进程所占有的资源
  - 关闭打开的文件、断开网络连接、回收分配的内存、.....
- ◎ 撤消该进程的PCB



UNIX: exit  
WINDOWS:  
TerminateProcess

### 3.进程阻塞

---

处于运行状态的进程，在其运行过程中期待某一事件发生，如等待键盘输入、等待磁盘数据传输完成、等待其它进程发送消息，当被等待的事件未发生时，由**进程自己执行阻塞原语**，使自己由运行态变为阻塞态



UNIX: wait

WINDOWS: WaitForSingleObject



## 4.UNIX的几个进程控制操作

---

- ◎ **fork()** 通过复制调用进程来建立新的进程，是最基本的进程建立过程
- ◎ **exec()** 包括一系列系统调用，它们都是通过用一段新的程序代码覆盖原来的地址空间，实现进程执行代码的转换
- ◎ **wait()** 提供初级进程同步操作，能使一个进程等待另外一个进程的结束
- ◎ **exit()** 用来终止一个进程的运行



系统调用

# UNIX的FORK()实现

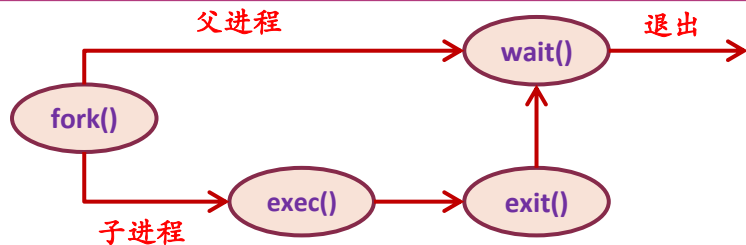
---

- ◉ 为子进程分配一个空闲的进程描述符  
proc 结构
- ◉ 分配给子进程唯一标识 pid
- ◉ 以一次一页的方式复制父进程地址空间 ?
- ◉ 从父进程处继承共享资源，如打开的文件和当前工作目录等
- ◉ 将子进程的状态设为就绪，插入到就绪队列
- ◉ 对子进程返回标识符 0
- ◉ 向父进程返回子进程的 pid

Linux采用了写时  
复制技术COW加  
快创建进程  
Copy-On-Write

# 使用FORK()的示例代码

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
void main(int argc, char *argv[])
{
    pid_t pid;
```



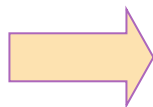
```
① → pid = fork();    /* 创建一个子进程 */
    if (pid < 0) {     /* 出错 */
        fprintf(stderr, "fork failed");
        exit(-1); }
    else if (pid == 0) { /* 子进程 */
② → execlp("/bin/ls", "ls", NULL); }
    else {             /* 父进程 */
③ → wait(NULL);        /* 父进程等待子进程结束 */
        printf("child complete");
        exit(0);
    }
}
```

**pid = xxx**

PC →

```
pid = fork();  
if (pid == 0) {  
    printf("child");  
} else {  
    printf("parent");  
}
```

父进程



**pid = 0**

PC →

```
pid = fork();  
if (pid == 0) {  
    printf("child");  
} else {  
    printf("parent");  
}
```

子进程

**pid = xxx**

PC →

```
pid = fork();  
if (pid == 0) {  
    printf("child");  
} else {  
    printf("parent");  
}
```

PC →

**pid = 0**

```
pid = fork();  
if (pid == 0) {  
    printf("child");  
} else {  
    printf("parent");  
}
```

Acknowledgement: Prof. Yuanyuan Zhou at UCSD

# 深入理解进程概念

# 关于进程的讨论

---

## ◎ 进程的分类

## ◎ 进程层次结构

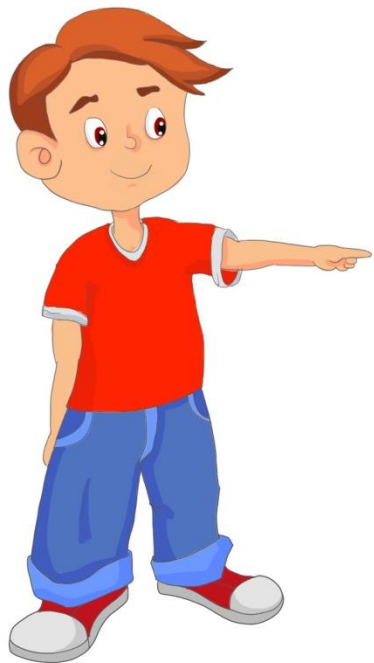
**UNIX进程家族树：init为根**  
**Windows：地位相同**

➤ 系统进程  
➤ 用户进程

➤ 前台进程  
➤ 后台进程

➤ CPU密集型进程  
➤ I/O密集型进程

# 进程与程序的区别



- 进程更能准确刻画并发，而程序不能
- 程序是静态的，进程是动态的
- 进程有生命周期的，有诞生有消亡，是短暂的；而程序是相对长久的
- 一个程序可对应多个进程
- 进程具有创建其他进程的功能

生活中类比例子

# 进程地址空间

---



操作系统给每个进程都分配了一个地址空间



# 先看一段程序

---

```
int myval;  
int main(int argc, char *argv[])  
{  
    myval = atoi(argv[1]);  
    while (1)  
        printf("myval is %d, loc 0x%lx\n",  
               myval, (long) &myval);  
}
```

同时运行两个Myval程序



输出?

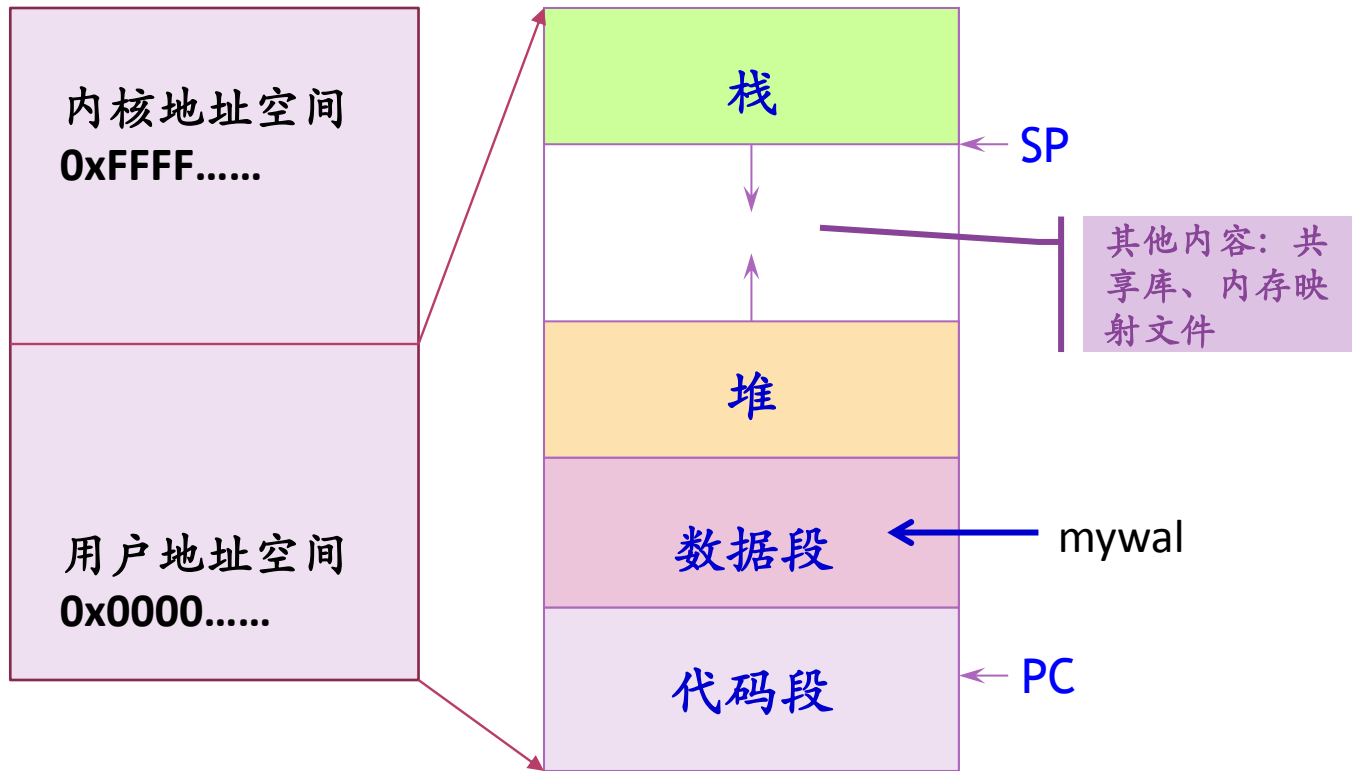
- ◉ Myval 7
- ◉ Myval 8

Acknowledgement: Prof. Yuanyuan Zhou at UCSD

变量myval  
的值不同，  
但地址相同

[illegible]

# 进程地址空间



# 进程映像 (IMAGE)

---

## 对进程执行活动全过程的静态描述

由进程地址空间内容、硬件寄存器内容及与该进程相关的内核数据结构、内核栈组成

- **用户相关：**进程地址空间（包括代码段、数据段、堆和栈、共享库.....）
- **寄存器相关：**程序计数器、指令寄存器、程序状态寄存器、栈指针、通用寄存器等值
- **内核相关：**
  - **静态部分：**PCB及各种资源数据结构
  - **动态部分：**内核栈（不同进程在进入内核后使用不同的内核栈）

# 上下文（CONTEXT）切换

---

- 将**CPU**硬件状态从一个进程换到另一个进程的过程称为**上下文切换**
- 进程运行时，其硬件状态保存在**CPU**上的寄存器中  
寄存器：程序计数器、程序状态寄存器、栈指针、通用寄存器、其他控制寄存器的值
- 进程不运行时，这些寄存器的值保存在进程控制块**PCB**中；当操作系统要运行一个新的进程时，将**PCB**中的相关值送到对应的寄存器中

# 线程的引入

# 线程的引入

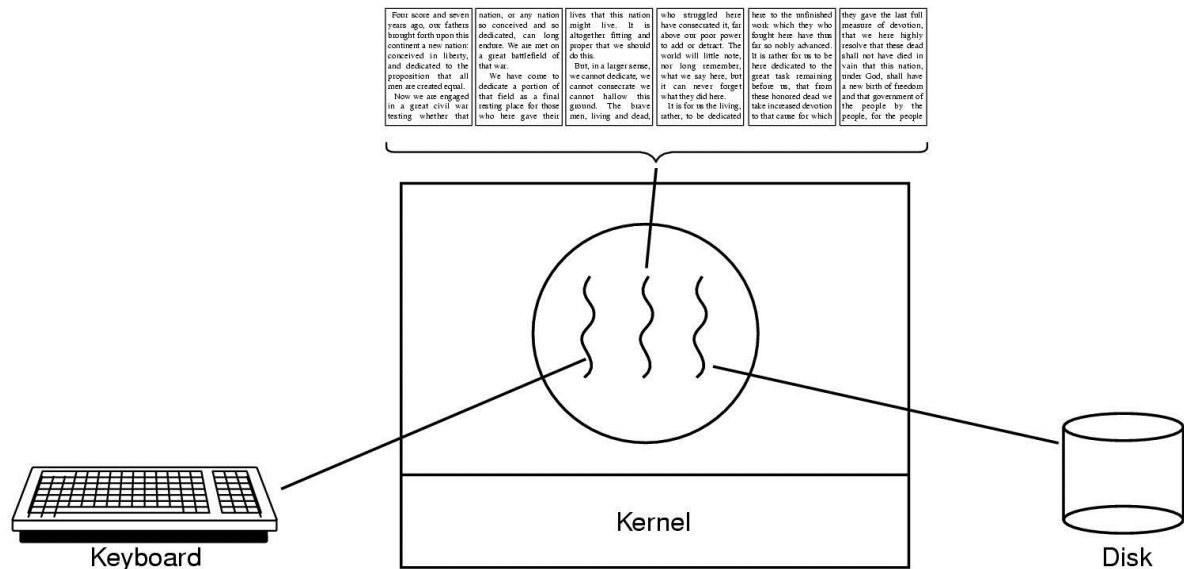
---

◎ 为什么在进程中再派生线程？

## 三个理由

- 应用的需要
- 开销的考虑
- 性能的考虑

# 应用的需要——示例1



有三个线程的字处理软件



# 应用的需要——示例2(1/5)

---

- ◎ 典型的应用

  - Web服务器**

- ◎ 工作方式

  - 从客户端接收网页请求（**http**协议）
  - 从磁盘上检索相关网页，读入内存
  - 将网页返回给对应的客户端

- ◎ 如何提高服务器工作效率？

  - 网页缓存（Web page Cache）**

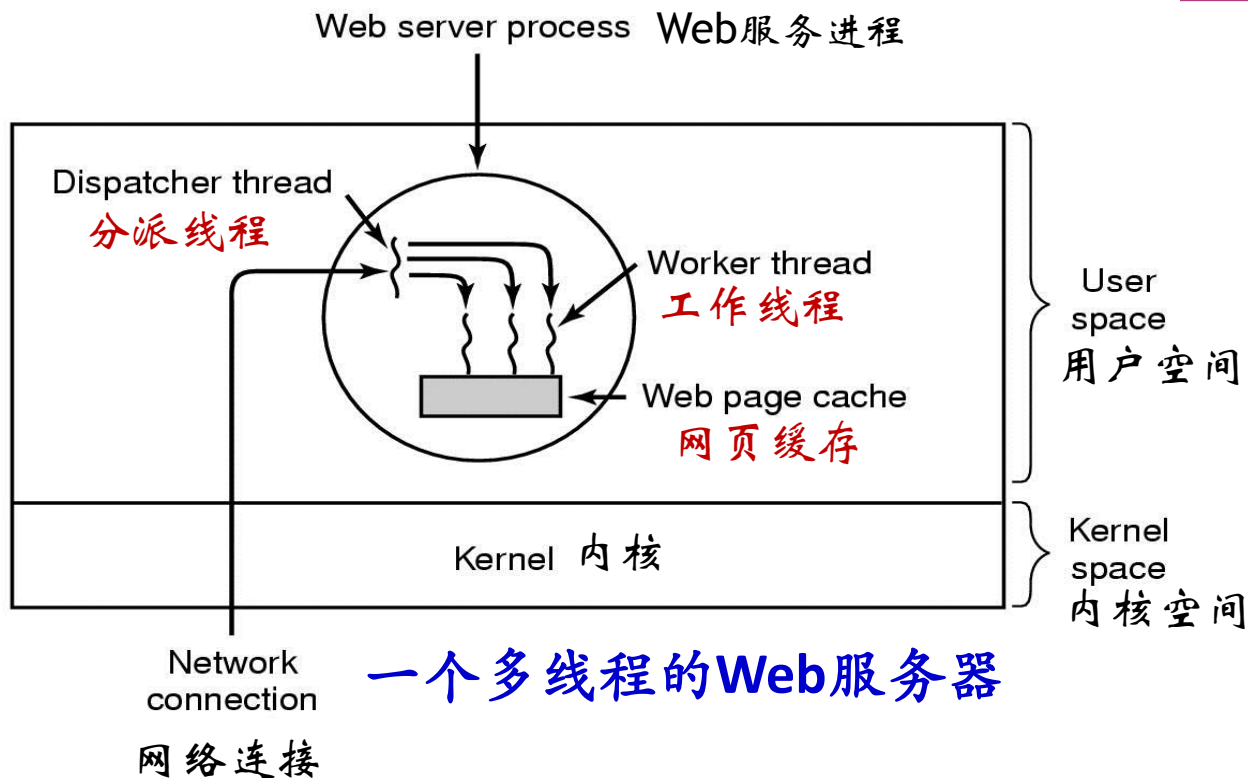
## 示例2(2/5)——如果没有线程？

---

两种解决方案：

- ◎ 一个服务进程  
顺序编程；性能下降
- ◎ 有限状态机  
编程模型复杂；采用非阻塞I/O

# 示例2(3/5)



一个多线程的Web服务器

# 示例2(4/5)

---

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a) 分派线程

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b) 工作线程

# 示例2(5/5)

---

模型	特性
多线程	有并发、阻塞系统调用
单线程进程	无并发、阻塞系统调用
有限状态机	有并发、非阻塞系统调用、中断

## 构造服务器的三种方法

## 2. 开销的考虑

---

进程相关的操作：

- ✓ 创建进程
- ✓ 撤消进程
- ✓ 进程通信
- ✓ 进程切换

→ 时间/空间开销大，  
限制了并发度的提高

线程的开销小


- ✓ 创建一个新线程花费时间少  
(撤销亦如此)
- ✓ 两个线程切换花费时间少
- ✓ 线程之间相互通信无须调用内核（同一进程内的线程共享内存和文件）

### 3.性能考虑

---

多个线程，有的计算，有的I/O

◉ 多个处理器



如何提高软件  
运行性能?

# 线程的基本概念

在同一进程  
增加了多个  
执行序列  
(线程)

进程的两个基本属性

资源的拥有者

CPU调度单位

进程还是资源的拥有者

线程继承了这一属性

线程：进程中的一个运行实体，是CPU的调度单位  
有时将线程称为轻量级进程



# 线程的属性

---

线程：

- ◉ 有标示符ID
- ◉ 有状态及状态转换 → 需要提供一些操作
- ◉ 不运行时需要保存的上下文  
有上下文环境：程序计数器等寄存器
- ◉ 有自己的栈和栈指针 ✓
- ◉ 共享所在进程的地址空间和其他资源
- ◉ 可以创建、撤消另一个线程  
程序开始是以一个单线程进程方式运行的

用户级线程、核心级线程、混合方式

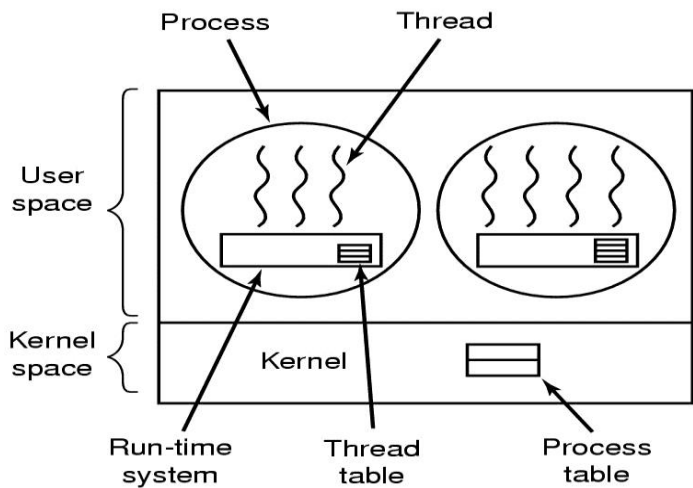
# 线程机制的实现

# 线程的实现

---

- ◎ 用户级线程
- ◎ 核心级线程
- ◎ 混合—两者结合方法

# 1. 用户级线程 (USER LEVEL THREAD)



- 在用户空间建立线程库：提供一组管理线程的过程
- 运行时系统：完成线程的管理工作（操作、线程表）
- 内核管理的还是进程，不知道线程的存在
- 线程切换不需要内核态特权
- 例子：UNIX

# 例子：POSIX线程库——PTHREAD

---

- ◎ **POSIX(Portable Operating System Interface)**
- ◎ 多线程编程接口，以线程库方式提供给用户

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

线程是自愿让出  
CPU的，why?

# 用户级线程小结

Jacketing/  
wrapper

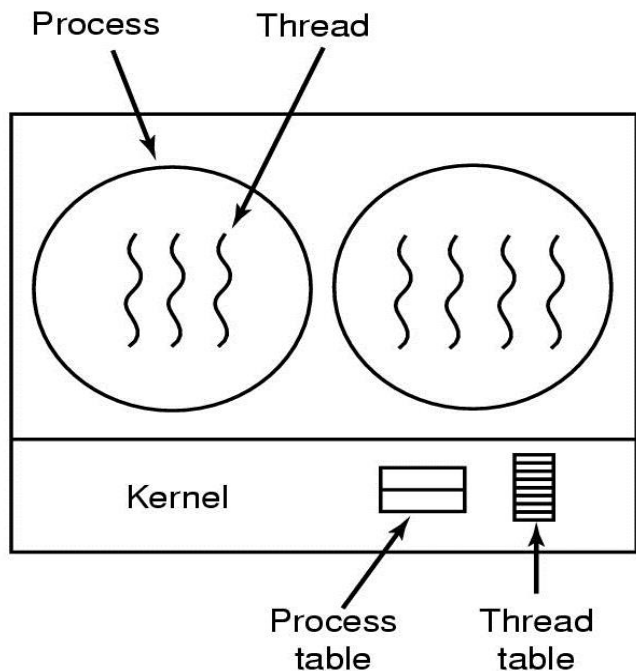
优点:

- ◎ 线程切换快
- ◎ 调度算法是应用程序特定的
- ◎ 用户级线程可运行在任何操作系统上（只需要实现线程库）

缺点:

- ◎ 内核只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上
- ◎ 大多数系统调用是阻塞的，因此，由于内核阻塞进程，故进程中所有线程也被阻塞

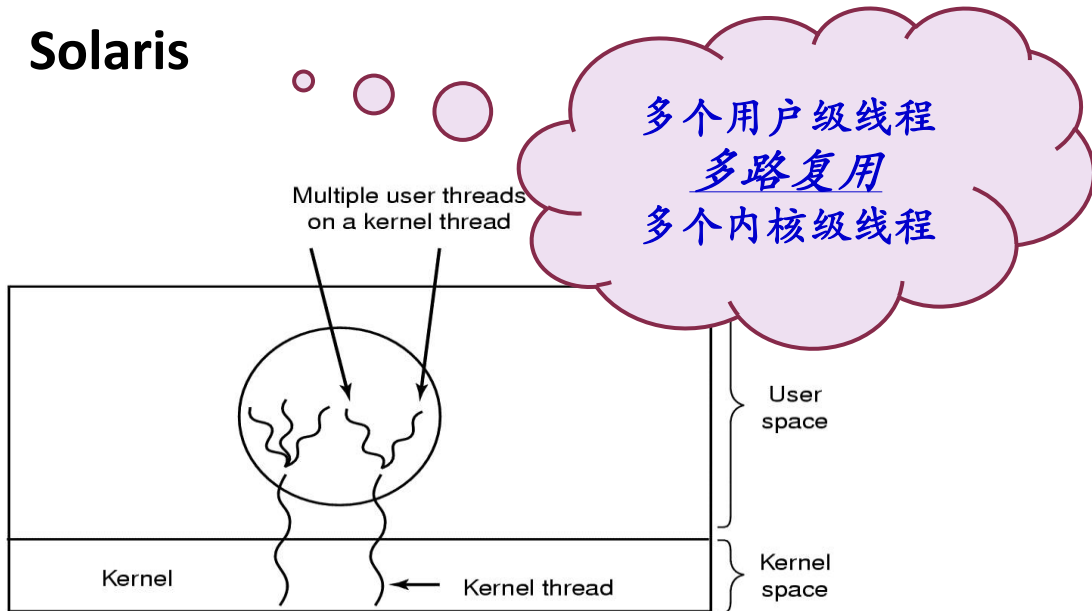
## 2.核心级线程 (KERNEL LEVEL THREAD)



- ◉ 内核管理所有线程管理，并向应用程序提供**API**接口
- ◉ 内核维护进程和线程的上下文
- ◉ 线程的切换需要内核支持
- ◉ 以线程为基础进行调度
- ◉ 例子： **Windows**

### 3.混合模型

- 线程创建在用户空间完成
- 线程调度等在核心态完成
- 例子: Solaris





# 本章重点小结(1/2)

## 进程

- **并发性** 任何进程都可以与其他进程一起向前推进
- **动态性** 进程是正在执行程序实例
  - ✓ 进程是动态产生，动态消亡的
  - ✓ 进程在其生命周期内，在三种基本状态之间转换
- **独立性** 进程是**资源分配**的一个独立单位  
例如：各进程的**地址空间相互独立**
- **交互性** 指进程在执行过程中可能与其他进程产生直接或间接的关系
- **异步性** 每个进程都以其相对独立的、不可预知的速度向前推进
- **进程映像** 程序 + 数据 + 栈(用户栈、内核栈) + PCB

# 本章重点小结(2/2)

---

## ◎ 线程

- 多线程应用场景
- 线程基本概念、属性
- 线程实现机制

可再入程序（可重入）：

可被多个进程同时调用的程序，具有下列性质：  
它是纯代码的，即在执行过程中自身不改变；  
调用它的进程应该提供数据区

# 本周要求

---

- 重点阅读教材

第2章相关内容： 2.1、 2.2(除2.2.8-2.2.10外)

- 重点概念

进程 进程状态及状态转换 进程控制  
进程控制块(PCB) 进程地址空间 进程上下文环境  
线程 线程属性 用户级线程 核心级线程  
Pthreads 可再入程序 原语 Web服务器

*THANKS*

*The End*