操作系统A

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

2020 Autumn

本章要求掌握的概念

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

思考问题

- 怎样理解“进程是对CPU的抽象”这句话?
- 进程有哪些状态？进程状态之间的转换条件以及对应的操作?
- 一个进程在生命周期内都有什么组成要素?
- 怎样描述进程?

  从静态和动态两个角度描述进程执行活动的全过程
- 什么是可再入程序？

- 哪些应用场景适合多线程?
- 线程的基本概念是什么？与进程是什么关系?
- 线程有哪些属性？为什么线程要有自己的栈?
- 线程实现机制有哪几种?

大纲

- 进程模型
  - 多道程序设计
  - 进程模型

    进程的概念、进程状态及转换、进程控制块、进程队列
  - 进程控制

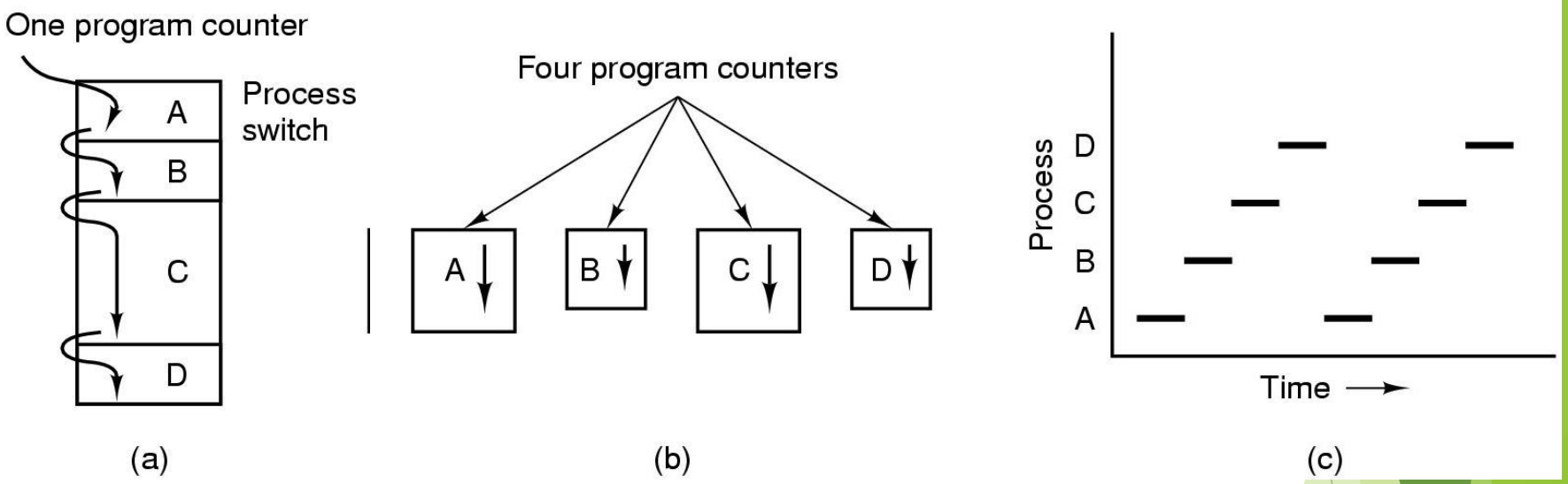
    进程创建、撤销、阻塞、唤醒、……
- 线程模型
  - 为什么引入线程?
  - 线程的组成
  - 线程机制的实现

    用户级线程、核心级线程、混合实现

进程的组成、进程控制

进程模型

一、多道程序设计（Multiprogramming）



- **多道程序设计**

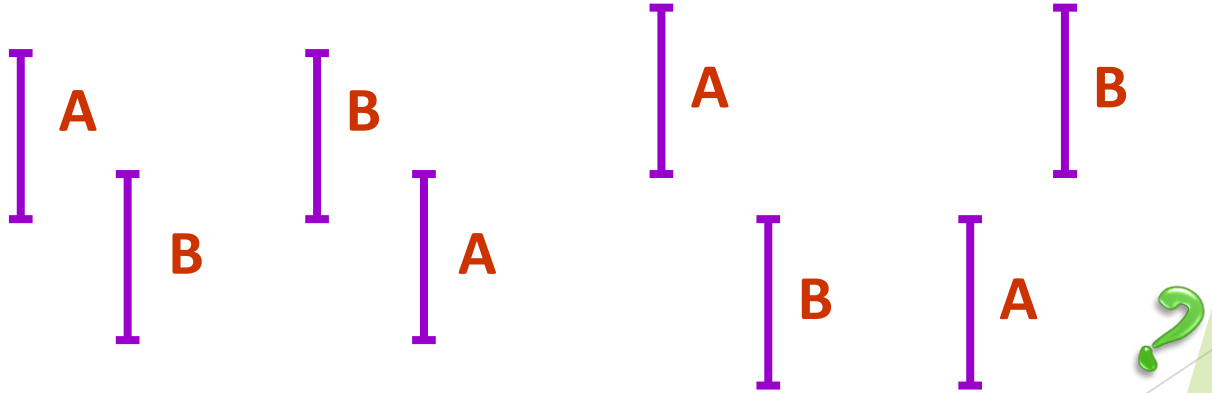
  允许多个程序 **同时** 进入内存并运行

  → 目的 为了提高系统效率

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

二、进程相关的基本概念

定义：Process（又称　任务Task or Job）

- 程序的一次执行过程
- 是正在运行程序的抽象
- 将一个CPU变换成多个虚拟的CPU
- 系统资源以进程为单位分配，如内存、文件、…

  每个进程具有独立的地址空间
- 操作系统将CPU调度给进程

进程是对CPU的抽象

如何查看当前系统中有多少个进程?

进程是具有独立功能的程序关于某个数据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

1.进程的基本状态

- 进程的三种基本状态：
  运行态、就绪态、等待态

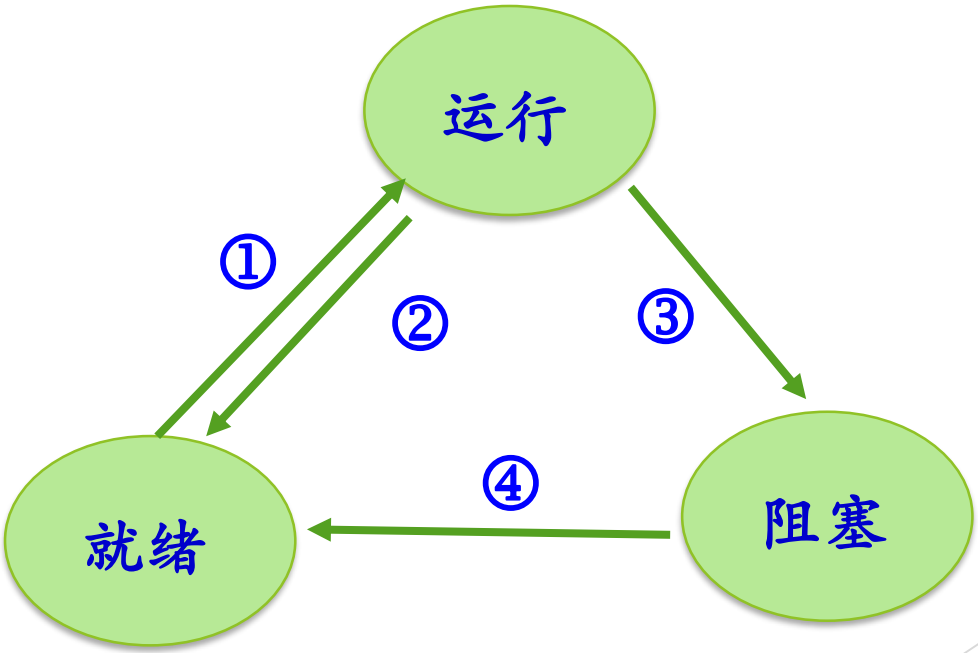
- 运行态（Running）
  占有CPU，并在CPU上运行
- 就绪态（Ready）
  已经具备运行条件，但由于没有空闲CPU，而暂时不能运行
- 等待态（Waiting/Blocked）
  因等待某一事件（如完成I/O）而暂时不能运行

阻塞态、封锁态、睡眠态

如：等待读盘结果

2.进程状态之间的转换

- 进程在消亡前处于且仅处于某一状态
- 不同系统设置的进程状态数目不同



3.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

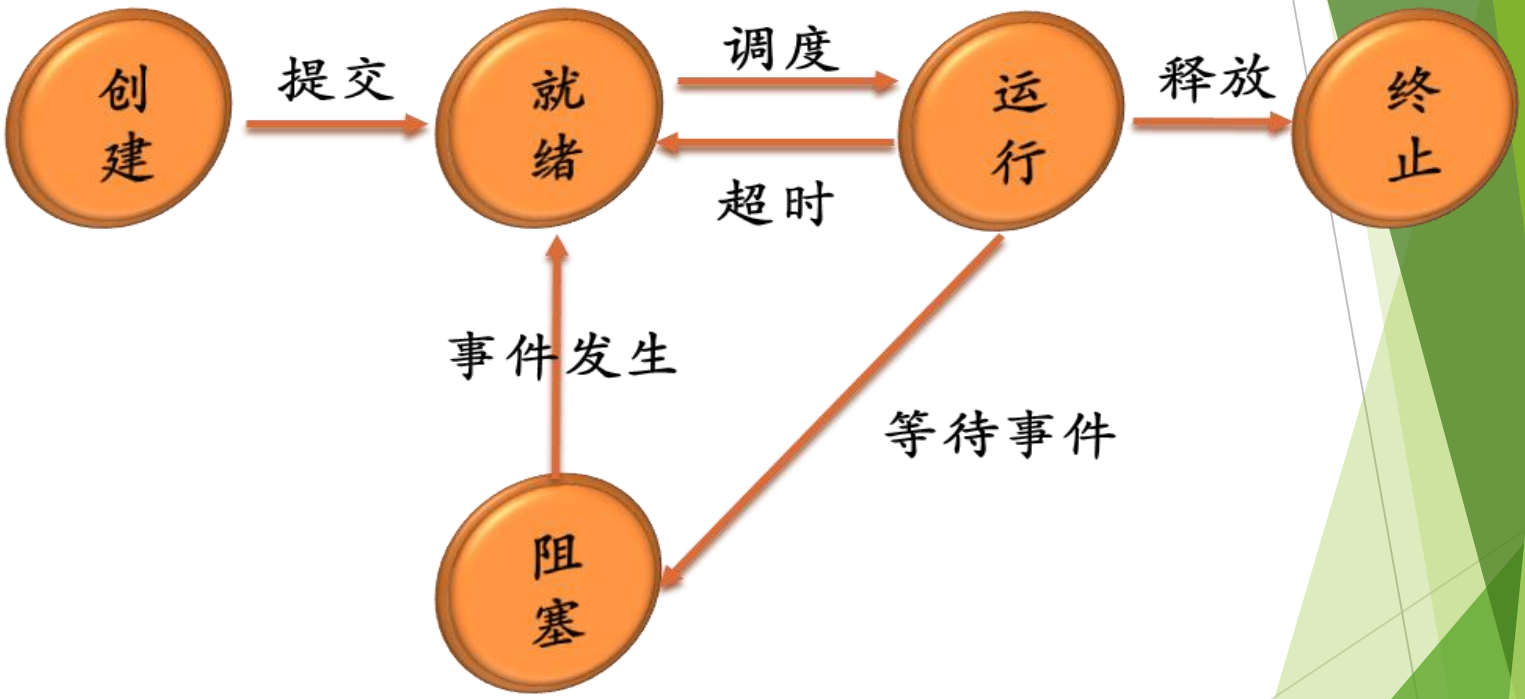
终止
Terminated

- 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

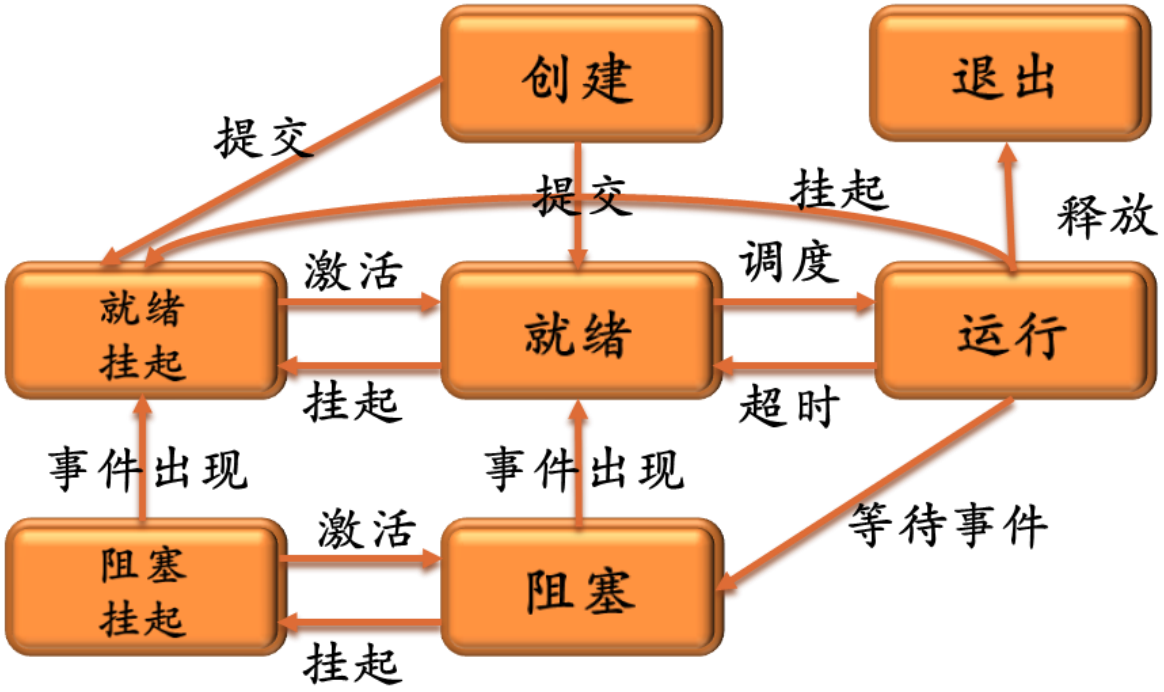
挂起
suspend

- 把一个进程从内存转到磁盘
  进程不占用内存空间，其进程映像交换到磁盘上
- 用于调节负载

五状态进程模型

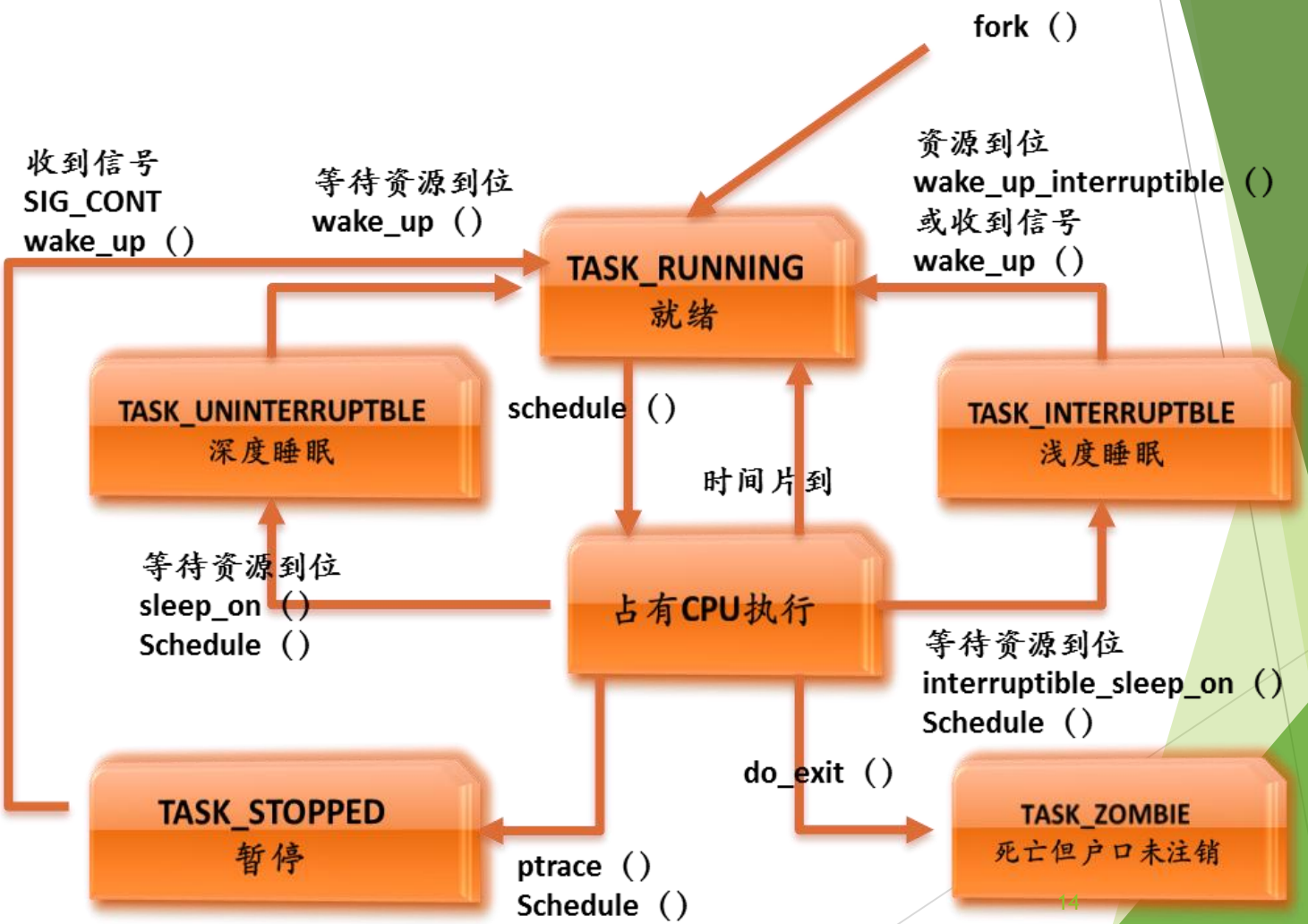


七状态进程模型


创建
退出
提交
提交
挂起
释放
激活
调度
就绪
挂起
就绪
运行
挂起
超时
事件出现
事件出现
激活
等待事件
阻塞
挂起
阻塞
挂起

挂起

Linux进程状态


fork（）
收到信号
SIG_CONT
wake_up（）
等待资源到位
wake_up（）
资源到位
wake_up_interruptible（）
或收到信号
wake_up（）
TASK_RUNNING
就绪
TASK_UNINTERRUPTBLE
深度睡眠
schedule（）
时间片到
TASK_INTERRUPTBLE
浅度睡眠
等待资源到位
sleep_on（）
Schedule（）
占有CPU执行
等待资源到位
interruptible_sleep_on（）
Schedule（）
do_exit（）
TASK_STOPPED
暂停
ptrace（）
Schedule（）
TASK_ZOMBIE
死亡但户口未注销

4. 进程控制块PCB

- PCB：Process Control Block
  - 操作系统表示进程的一个专门的数据结构
  - 记录进程的各种属性，描述进程的动态变化过程
  - 又称 **进程描述符、进程属性**

- 操作系统通过PCB来控制和管理进程

  → **PCB是系统感知进程存在的唯一标志**

进程表：所有进程的PCB的集合

Linux：task_struct
Windows：EPROCESS、KPROCESS、PEB

PCB的内容（1/2）

- 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
  - 可执行文件名（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；阻塞原因

PCB的内容（2/2）

- 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
  - 虚拟地址空间的现状
  - 打开文件列表

进程暂时不运行时，操作系统要保存该进程的硬件环境

- CPU现场信息
  - 寄存器值（通用、程序计数器PC、状态PSW，地址如栈指针）
  - 指向赋予该进程的段/页表的指针

Linux task_struct(1)

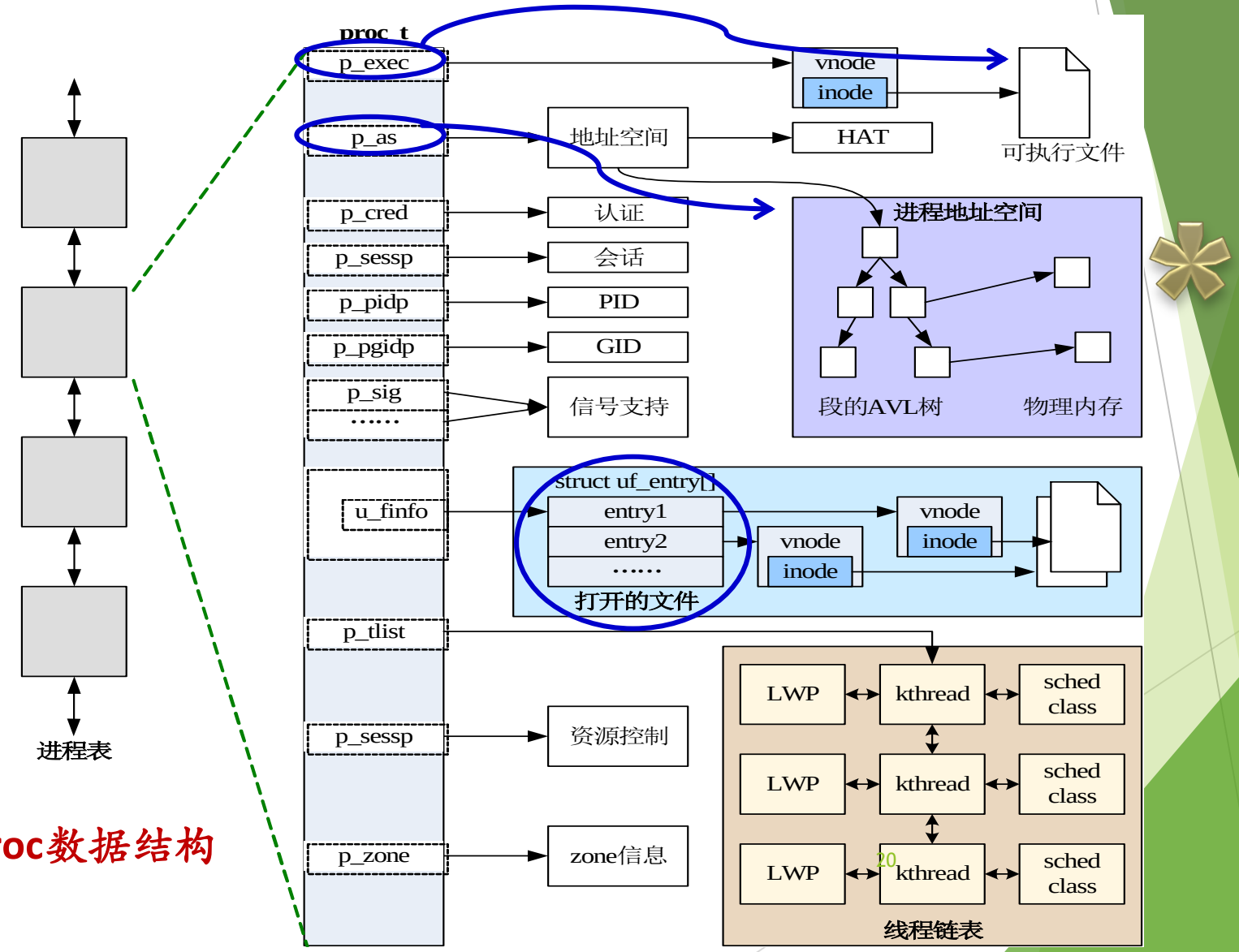
```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;    /* per process flags, defined below */
    unsigned long ptrace;

    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)

思考题：解读task_struct

```
/* 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
};
```

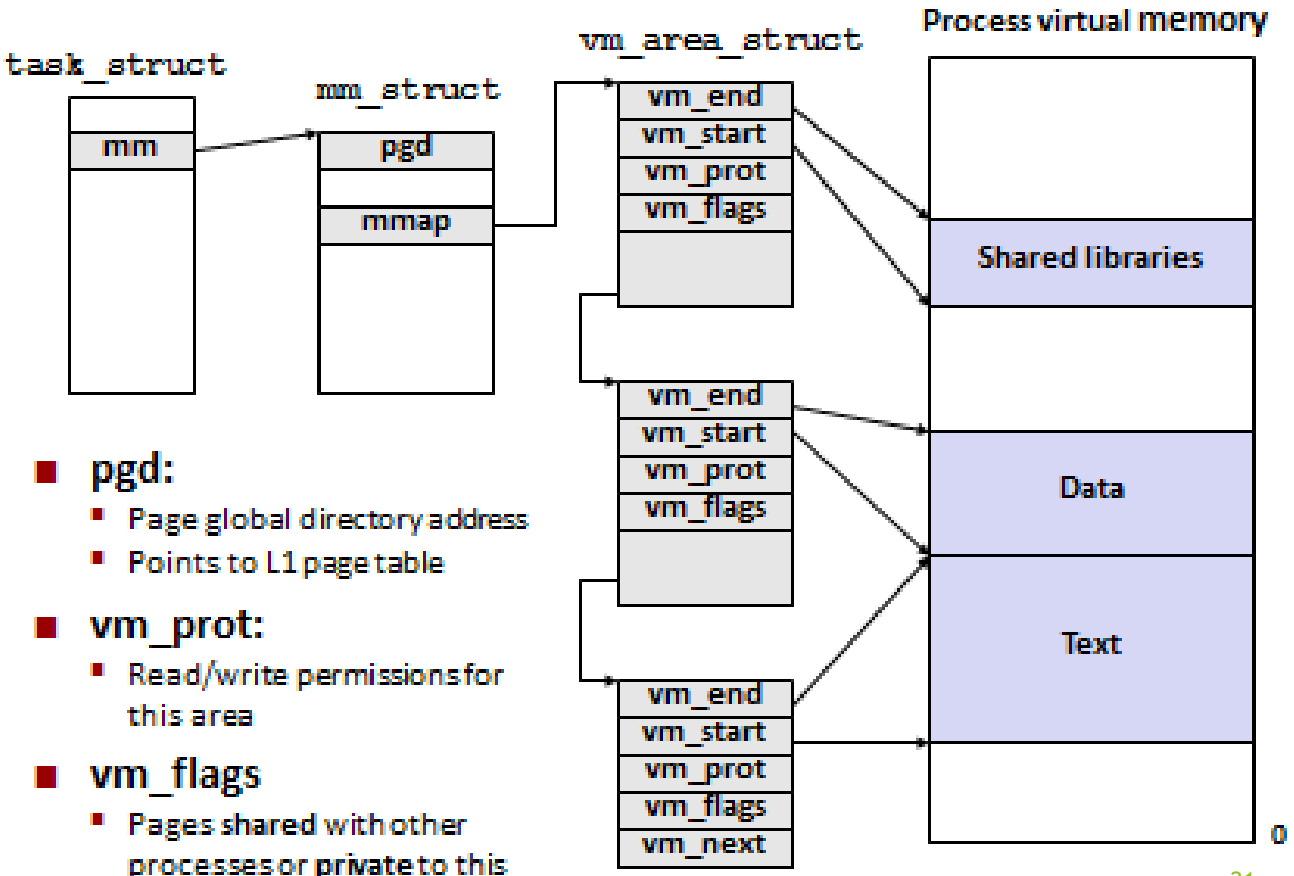
Solaris的进程控制块与进程表



proc数据结构

对照：ICS课程相关内容

Linux Organizes VM as Collection of "Areas"



- **pgd:**
  - Page global directory address
  - Points to L1 page table
- **vm_prot:**
  - Read/write permissions for this area
- **vm_flags**
  - Pages shared with other processes or **private** to this process

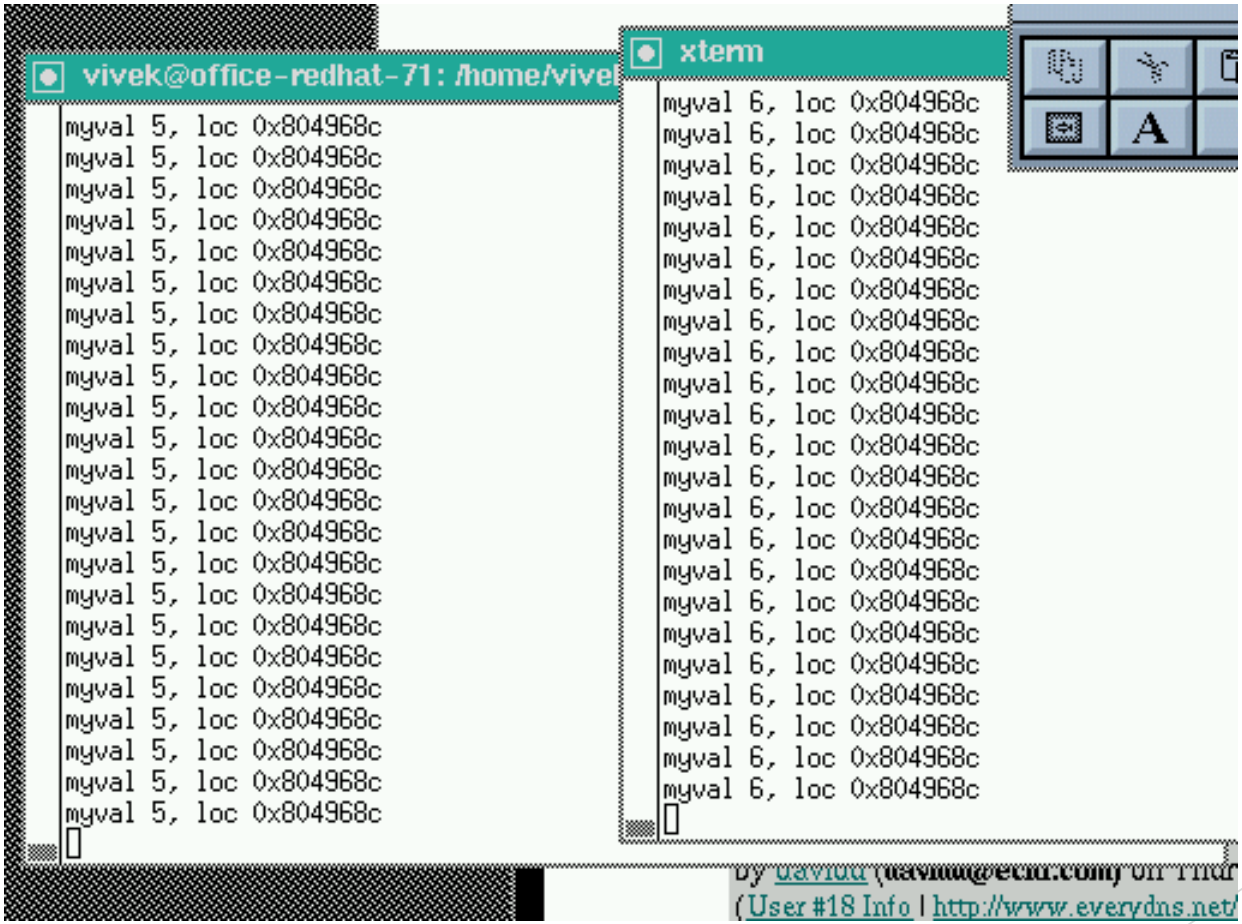
看一段程序

```
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

- Myval 5
- Myval 6
- 输出？

输出结果示意

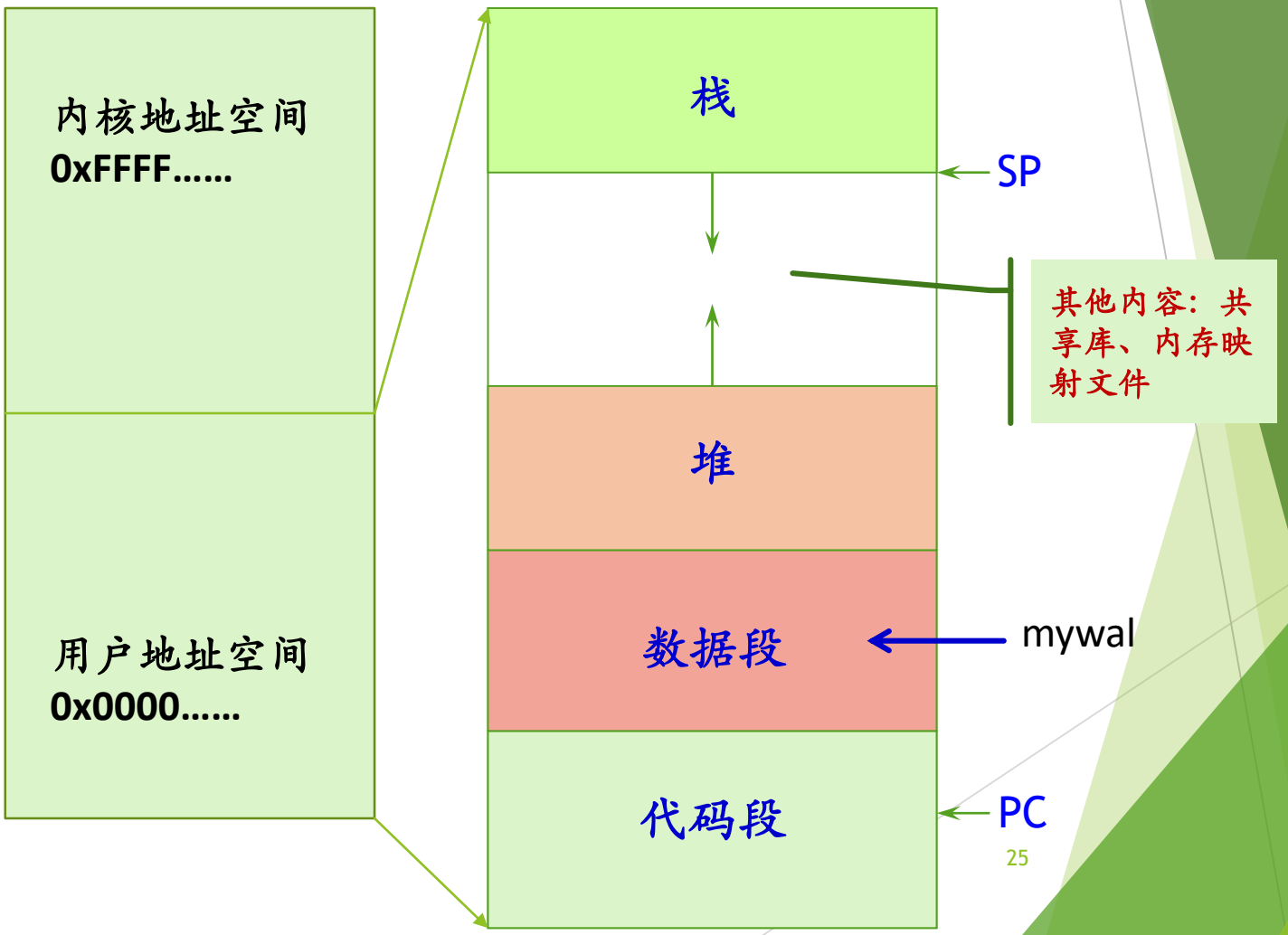


为什么变量myval的值不同，但地址相同？

5.重要概念：进程地址空间

理解：操作系统给
每个进程都分配了
一个地址空间

进程地址空间


内核地址空间
0xFFFF……
用户地址空间
0x0000……
栈
SP
其他内容：共享库、内存映射文件
堆
数据段
mywal
代码段
PC

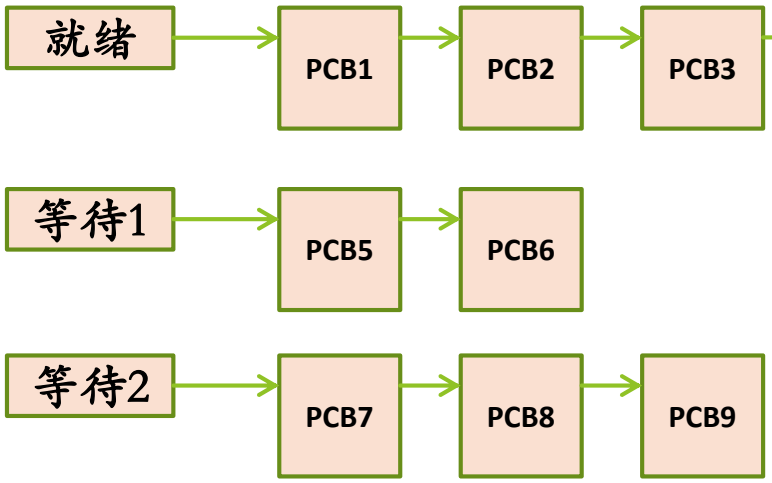
上下文（context）切换

- 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PCB中
- 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**

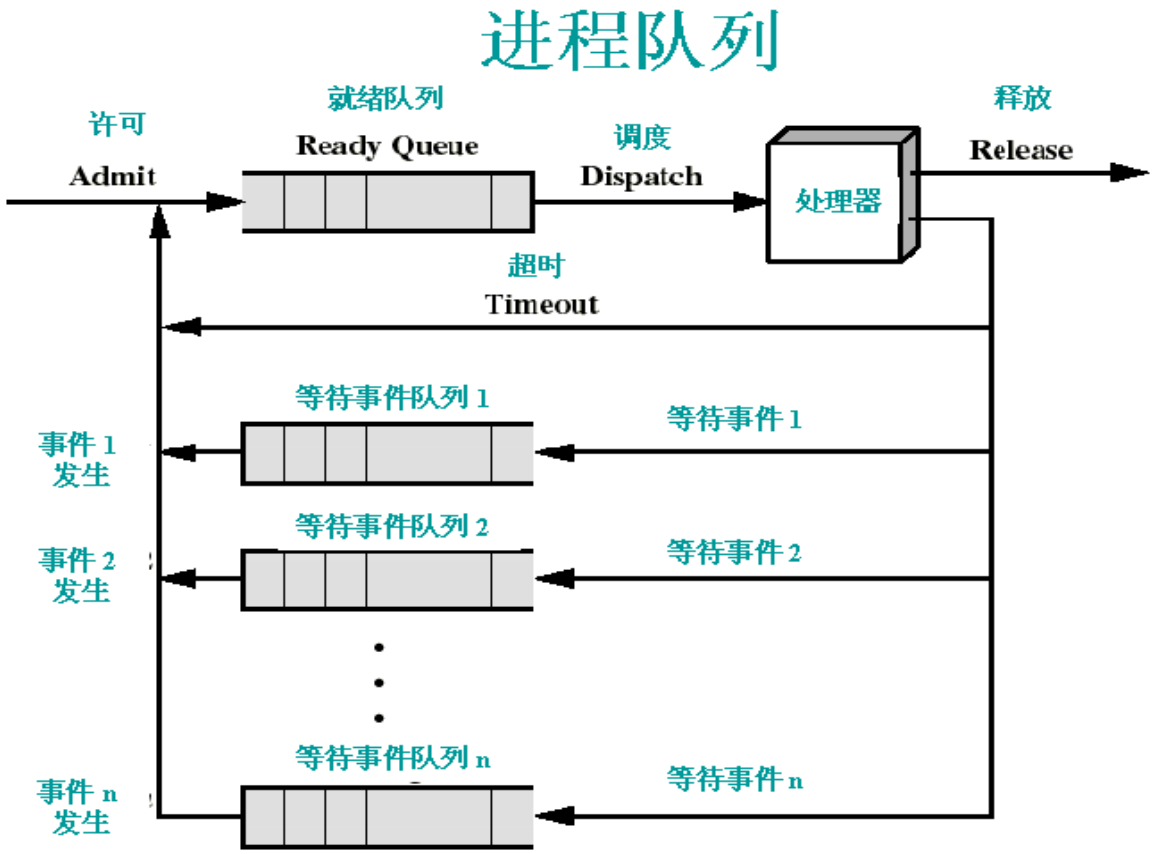
6.进程队列

- 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



- 就绪队列无优先级（例：FIFO）
- 当事件n发生，对应队列移进就绪队列

W. Stallings

三、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（解挂）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）

进程何时创建？何时终止？

- 系统初始化时
- 操作系统提供的服务
- 交互用户登录系统
- 由现有的进程派生出一个新进程
- 提交一个程序执行

- 正常退出（自愿的）
- 出错退出（自愿的）
- 严重错误（非自愿）
- 被其他进程杀死（非自愿）

1. 进程的创建

- 给新进程分配一个唯一标识(pid)以及进程控制块(PCB)
- 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的链表中
- 创建或扩充其他数据结构

UNIX：fork/exec
WINDOWS：CreateProcess

2. 进程的撤消

- 结束子进程或线程
- 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： ExitProcess

3. 进程阻塞和进程唤醒

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系统设计的进程控制操作

- 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代码覆盖原来的内存空间，实现进程执行代码的转换
- wait()　提供初级的进程同步措施，能使一个进程等待，直到另外一个进程结束为止
- exit()　用来终止一个进程的运行

系统调用

UNIX的fork()实现

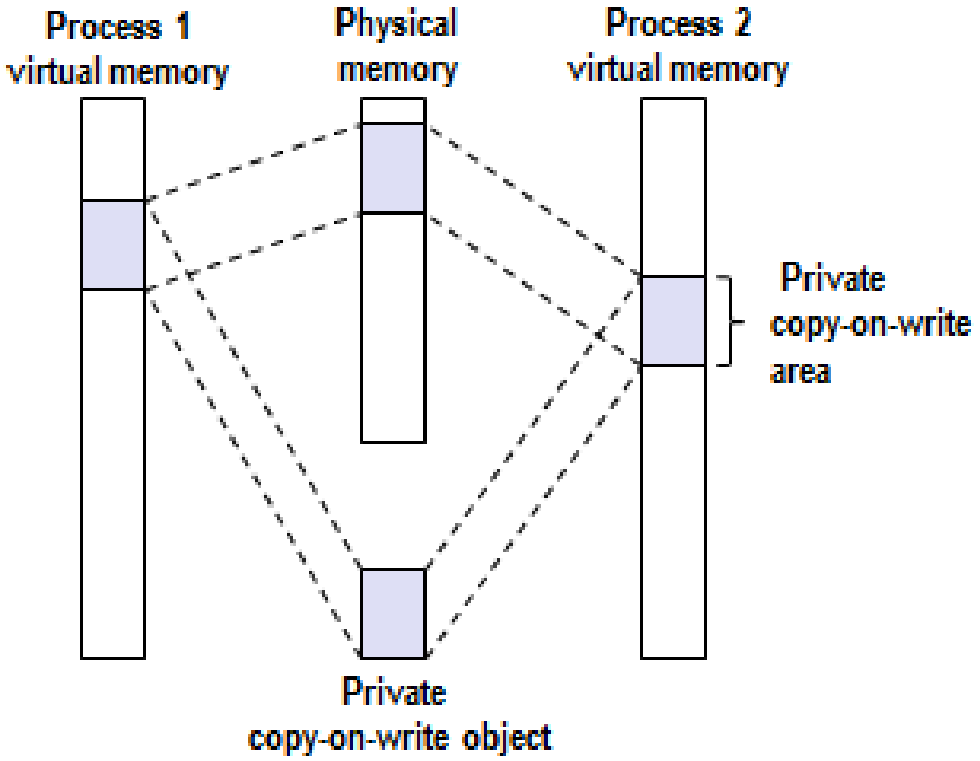
- 为子进程分配一个空闲的进程描述符
  proc 结构
- 分配给子进程唯一标识 pid
- ***以一次一页的方式复制父进程地址空间***
- 从父进程处继承共享资源，如打开的文件和当前工作目录等
- 将子进程的状态设为就绪，插入到就绪队列
- 对子进程返回标识符 **0**
- 向父进程返回子进程的 **pid**

如何优化？

Linux采用的是“写时复制技术”COW（Copy-On-Write）

对照：ICS课程相关内容1

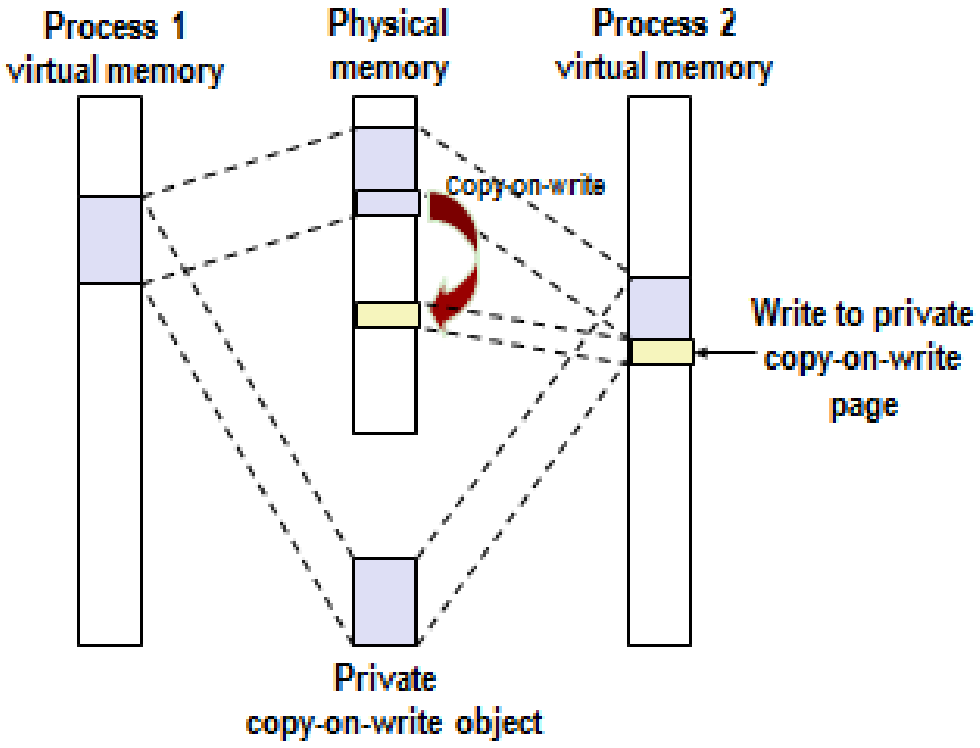
Private Copy-on-write (COW) sharing



- Two processes mapping *private copy-on-write (COW)* pages
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

对照：ICS课程相关内容2

Private Copy-on-write (COW) sharing



- Instruction writing to private page triggers protection fault
- Handler creates new R/W page
- Instruction restarts upon handler return
- Copying deferred as long as possible!

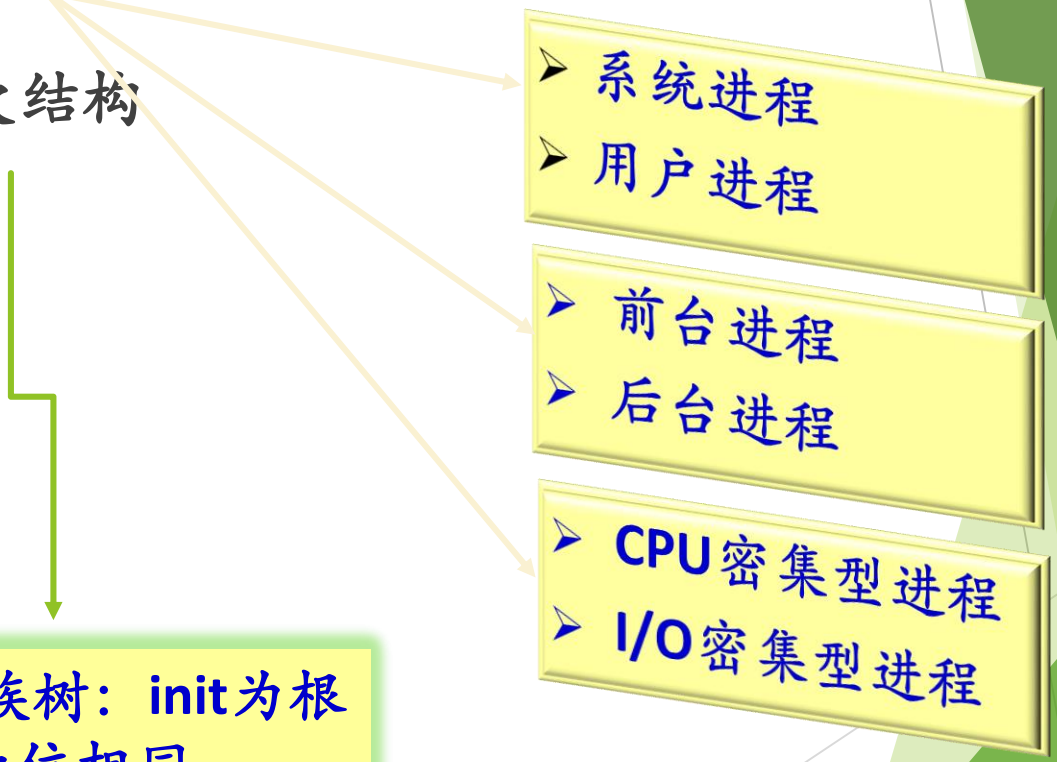
对照：ICS课程相关内容3

The `fork` Function Revisited

- `fork` provides private address space for each process
- To create virtual address for new process
  - Create exact copies of parent page tables
  - Flag each page in both processes (parent and child) as read-only
  - Flag writeable areas in both processes as private COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create new physical pages using COW mechanism
- Perfect approach for common case of fork() followed by exec()
  - Why?

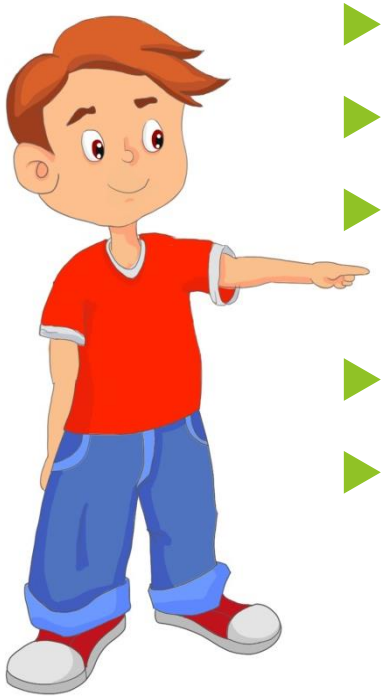
小结1——进程的讨论

- 进程举例（现实生活类比）
- 进程的分类
- 进程的层次结构



UNIX进程家族树：init为根
Windows：地位相同

小结2——进程与程序的区别

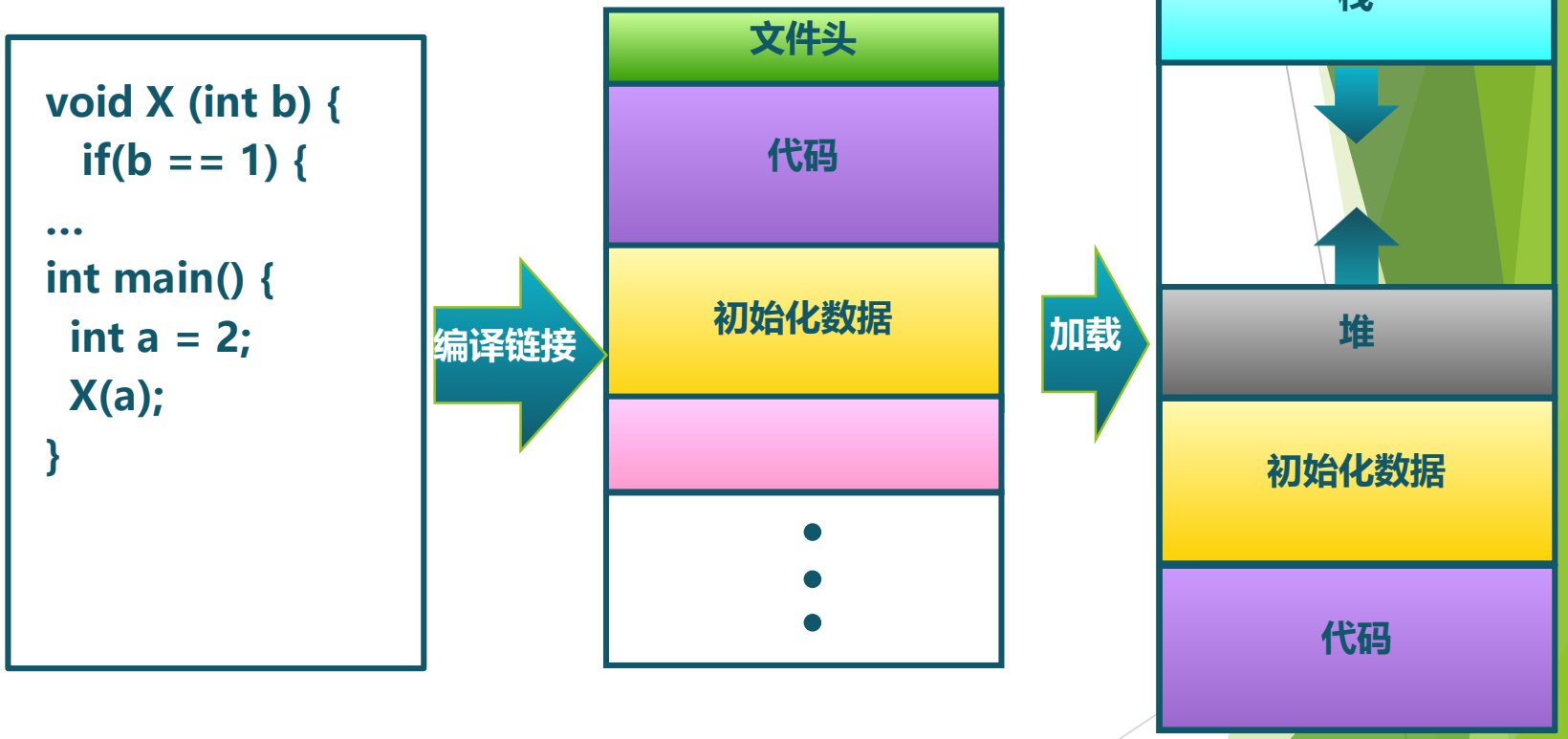


- 进程更能准确刻画并发，而程序不能
- 程序是静态的，进程是动态的
- 进程有生命周期的，有诞生有消亡，是短暂的；而程序是相对长久的
- 一个程序可对应多个进程
- 进程具有创建其他进程的功能

生活中类比例子

小结3——进程生成示意

进程是指一个具有一定独立功能的程序在一个数据集合上的一次执行过程



Acknowledgement: 清华大学 向勇、陈渝

线程的组成、线程机制的实现

线程模型

一、线程的引入

- 为什么在进程中再派生线程?

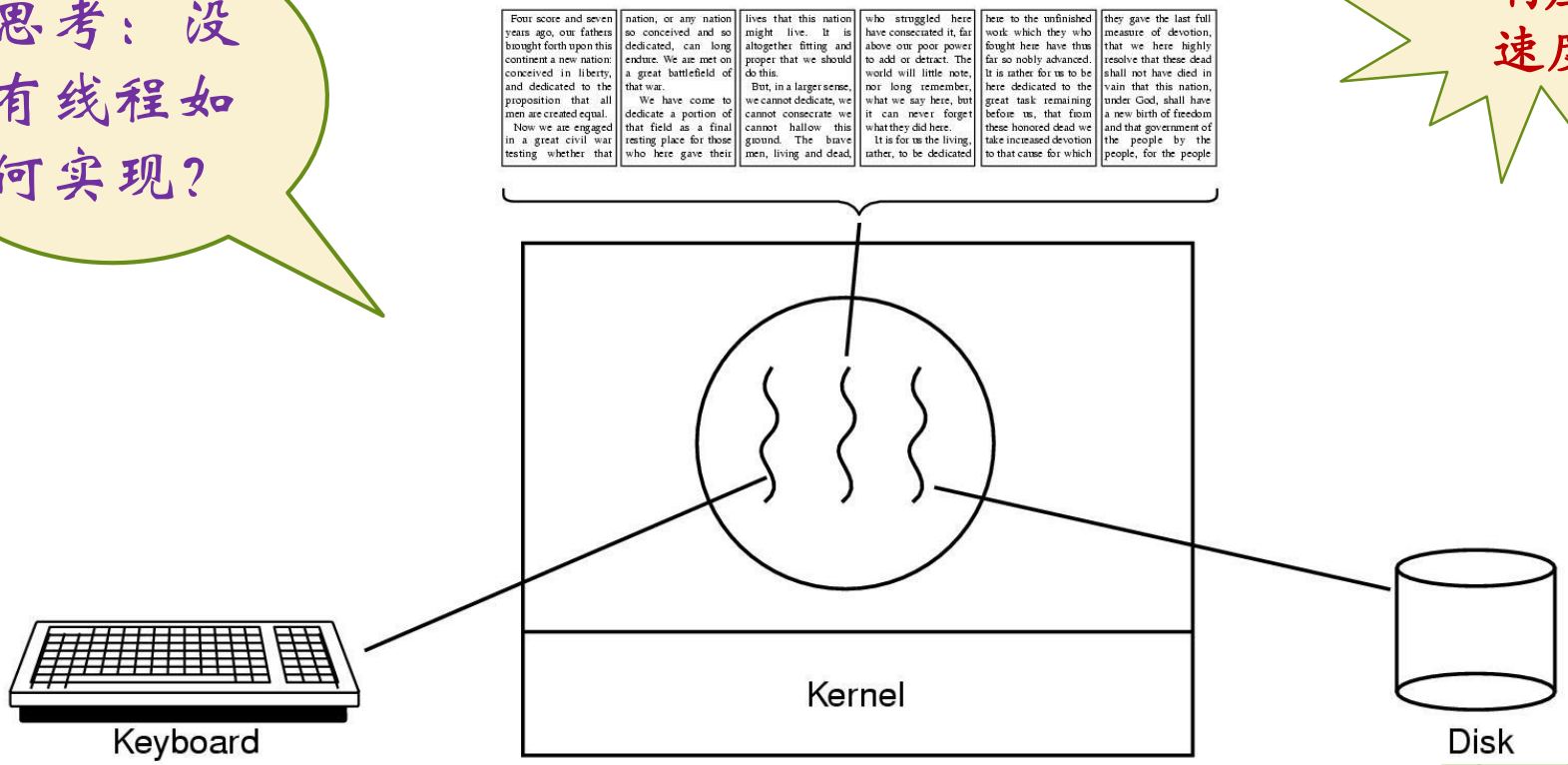
三个理由

- 应用的需要
- 开销的考虑
- 性能的考虑

1.应用的需要——示例1

思考：没有线程如何实现？

响应速度



有三个线程的字处理软件

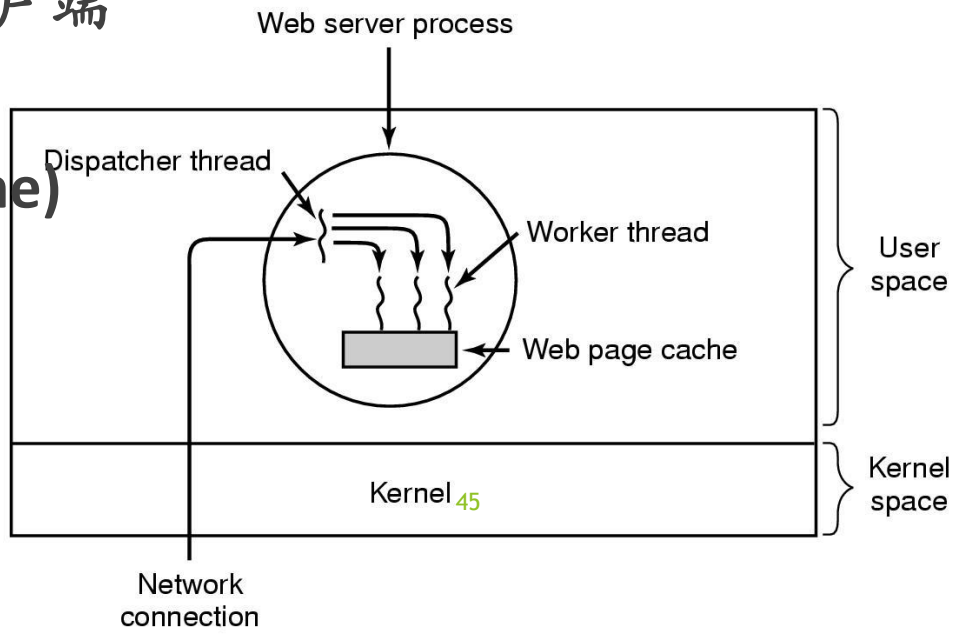
选自Tanenbaum《现代操作系统》第4版

应用的需要——示例2(1/4)

- 典型的应用

  Web服务器

- 工作原理
  - 从客户端接收网页请求（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)
  - 多线程



选自Tanenbaum《现代操作系统》第4版

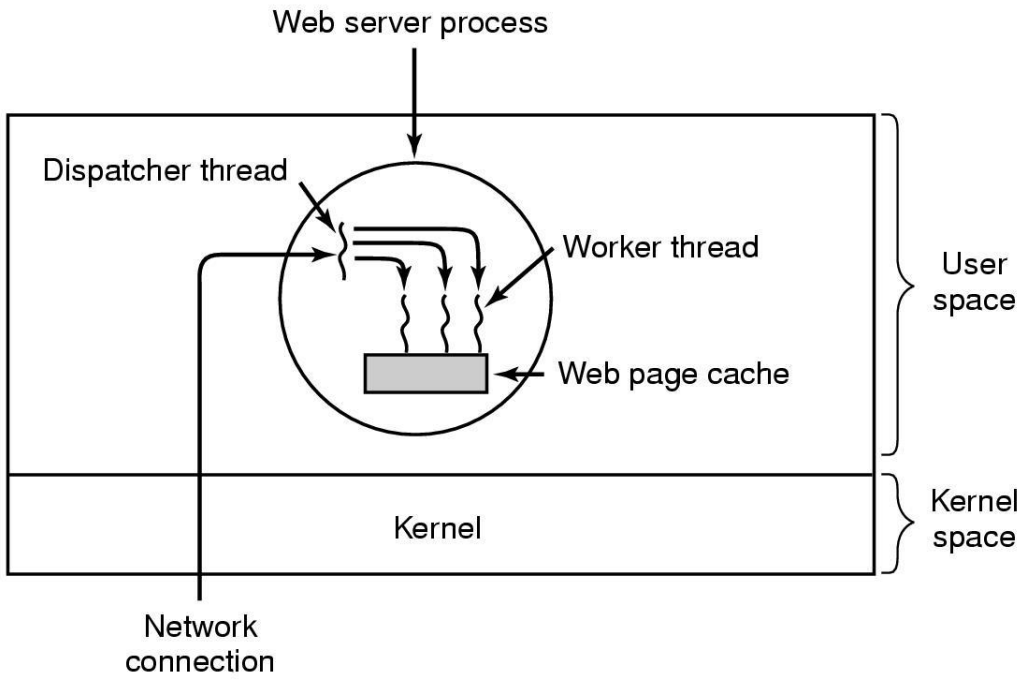
示例2(2/4)—如果没有线程?

两种解决方案:

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

示例2(3/4)

一个多线程的
Web服务器




```
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

选自Tanenbaum《现代操作系统》第4版

示例2(4/4)

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

构造服务器的三种方法

选自Tanenbaum《现代操作系统》第4版

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）
- ✓ 两个线程的切换花费时间少
- ✓ 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. 性能的考虑

- 多个线程，有的计算，有的I/O
- 多个处理器

二、线程的基本概念

- 进程的两个基本属性：
  - 资源的拥有者

    一个虚拟地址空间，一些占有的资源（文件，I/O设备），保护
  - 调度单位

    一个执行（路径）轨迹，状态、优先级

- 将原来进程的两个属性分别处理

→ 线程：进程中一个运行实体，是CPU的调度单位

→ 有时将线程称为轻量级进程

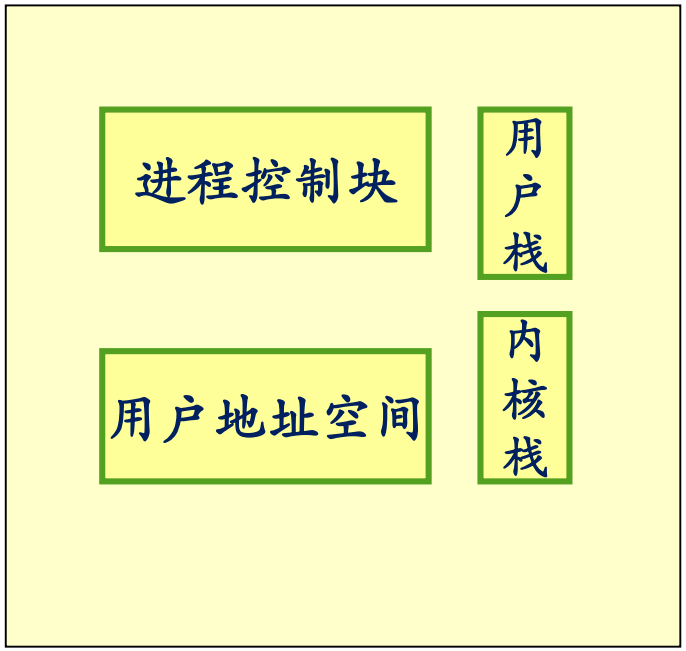
线程的属性

线程：

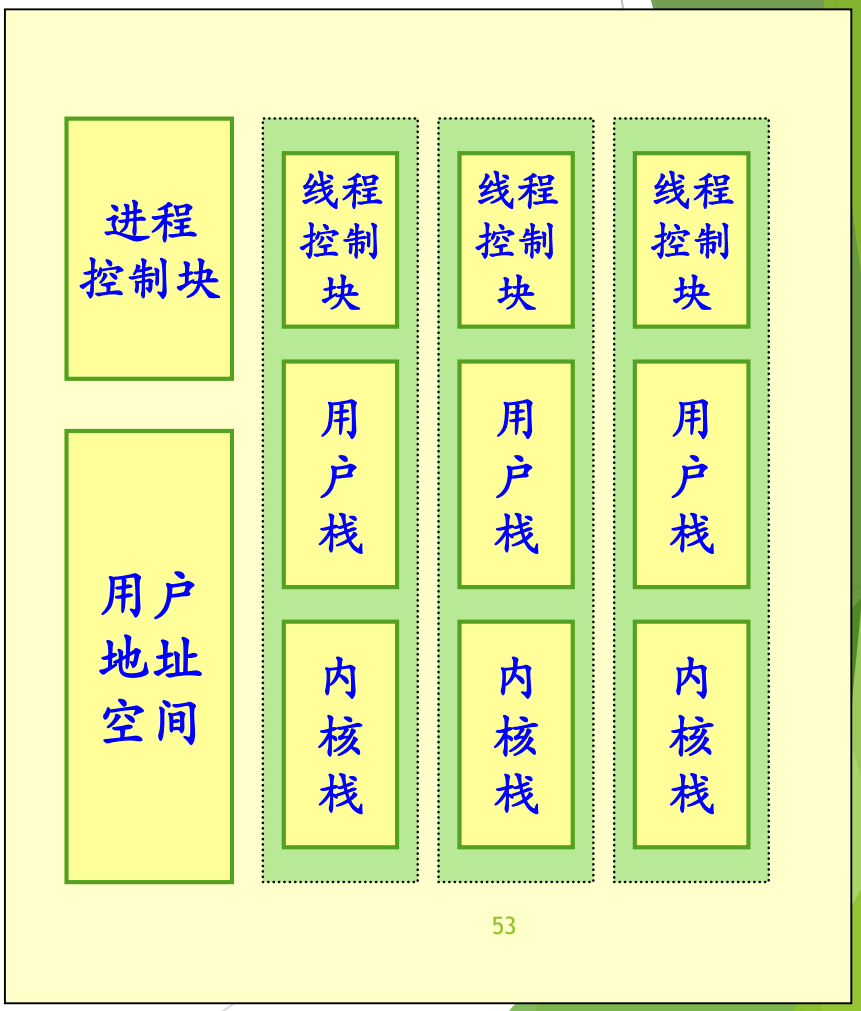
- 有状态及状态转换 → 需要提供一些操作
- 不运行时需要保存的上下文

  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）

单线程进程模型



多线程进程模型



三、线程的实现

- 用户级线程

  在用户空间实现

- 核心级线程

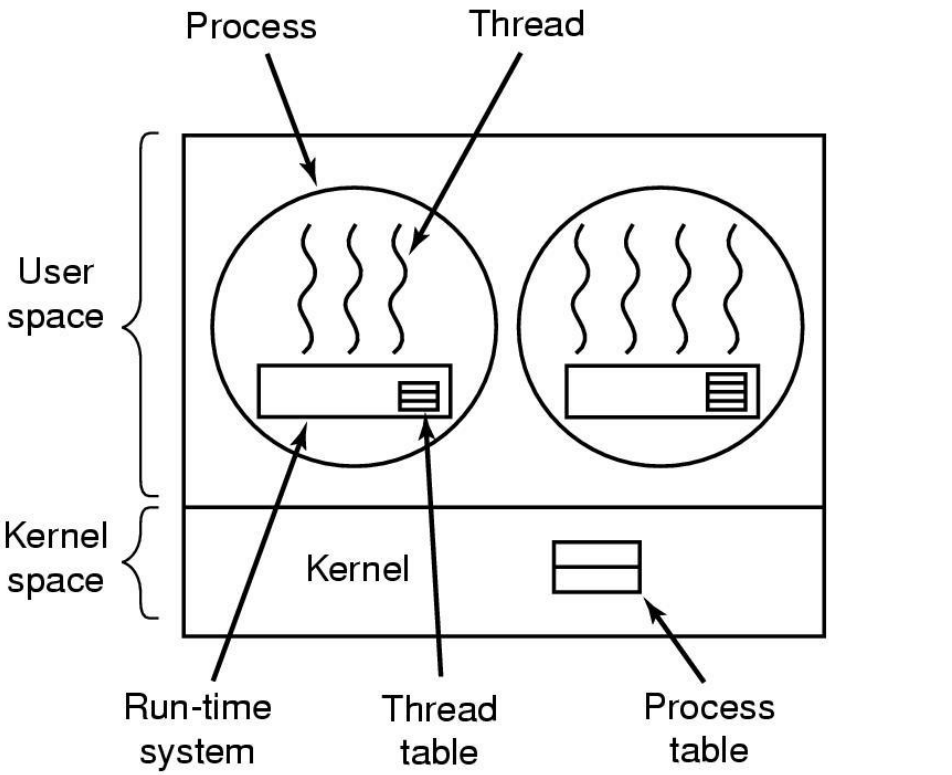
  在内核中实现

- 混合—两者结合方法

  在内核中实现，支持用户线程

1.用户级线程（User Level Thread）

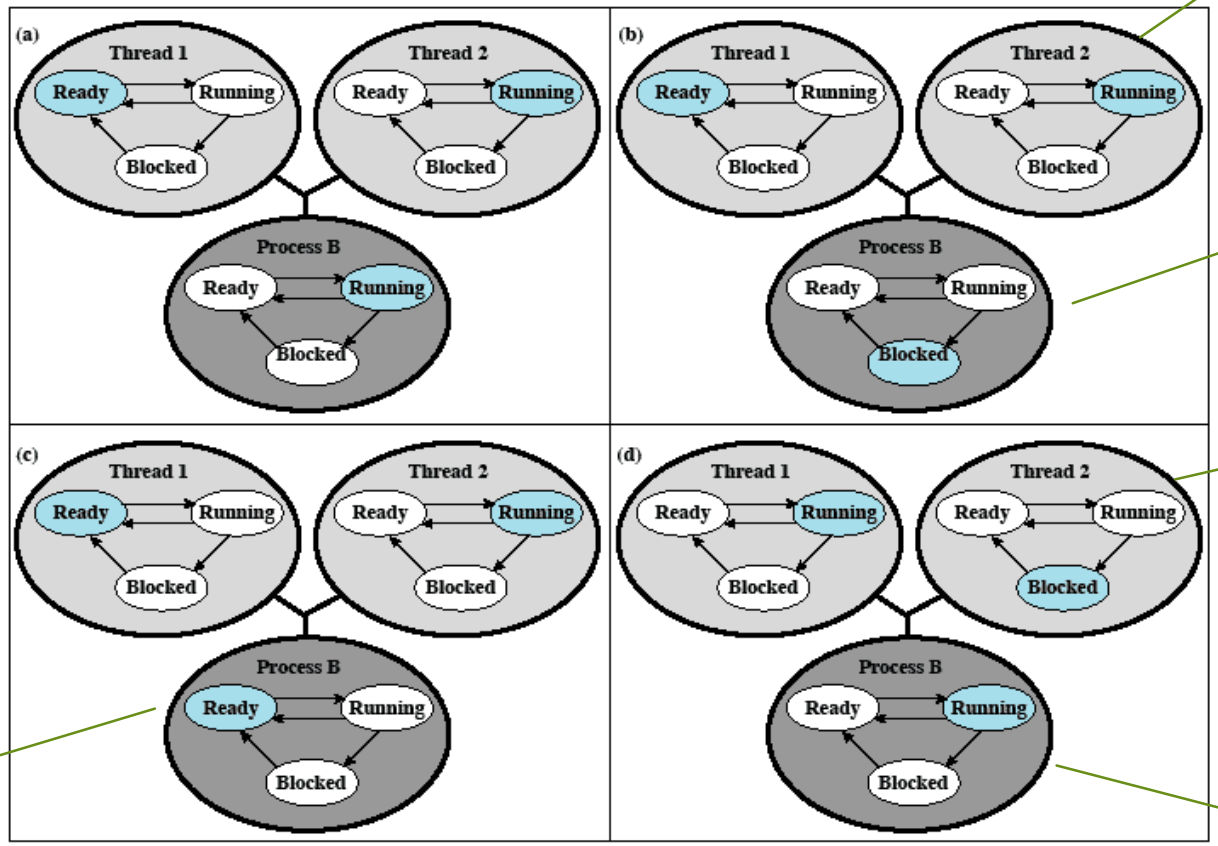
- 在用户空间建立线程库：提供一组管理线程的函数
- 运行时系统：完成线程的管理工作（操作、线程表）
- 内核管理的是进程，不知道线程的存在
- 线程切换不需要内核态特权
- 例子：**POSIX Pthreads**



进程状态与线程状态

典型的场景

case 1



case 2

case 3

Colored state is current state

Examples of the Relationships Between User-Level Thread States and Process States

线程2执行系统调用

进程B阻塞

线程1执行；线程2被阻塞

进程B被切换

进程B再次被调度

W. Stallings

例子：POSIX线程库——Pthreads

- POSIX(Portable Operating System Interface)1003.4a
- 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，为什么？

用户级线程小结

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：

- 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
- 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

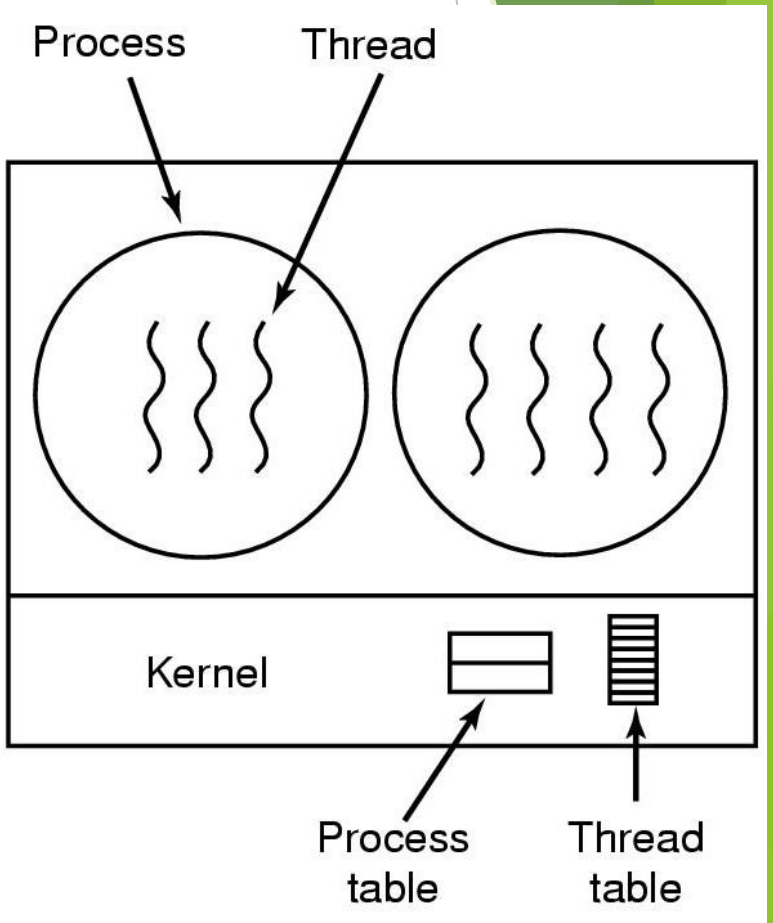
Jacketing/ wrapper

关于阻塞系统调用的处理

- 两种解决方案
  - 修改系统调用为非阻塞的
  - 重新实现对应系统调用的I/O库函数

2. 核心级线程（Kernel Level Thread）

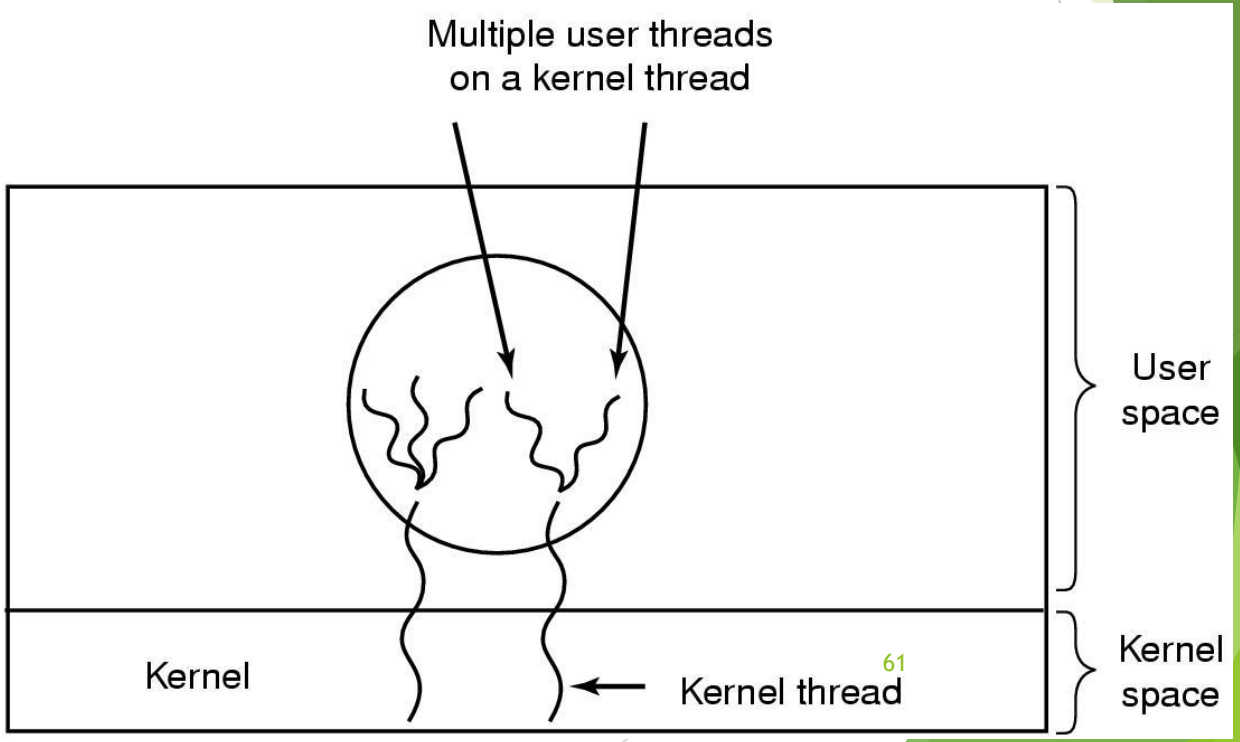
- 内核管理所有线程管理，并向应用程序提供API接口
- 内核维护进程和线程的上下文
- 线程的切换需要内核支持
- 以线程为基础进行调度
- 例子：Windows



3. 混合模型

多个用户级线程
多路复用
多个内核级线程

- 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

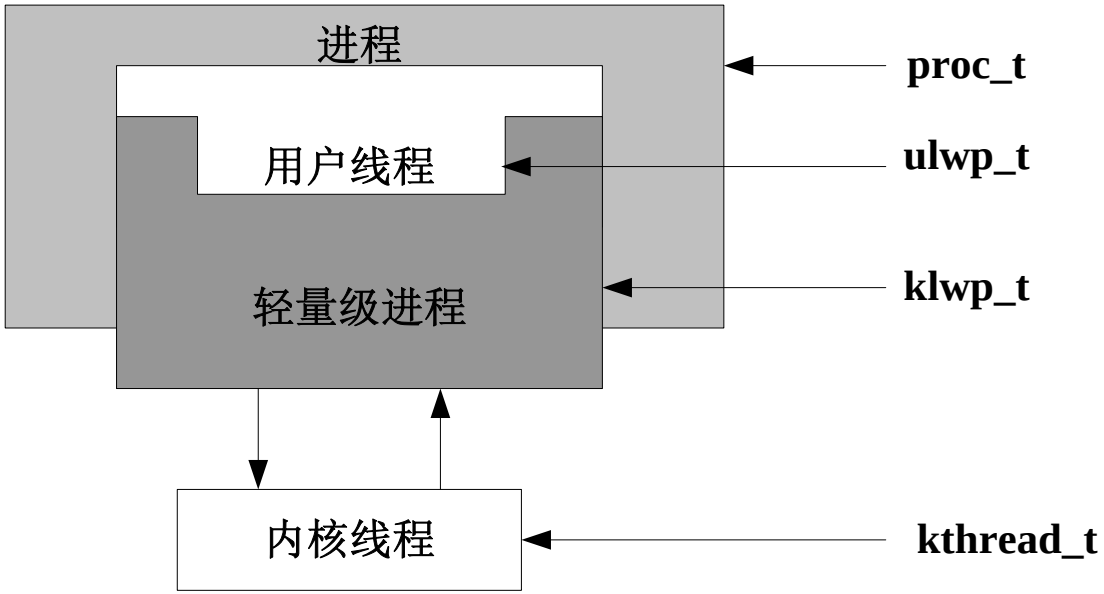


Solaris进程线程模型

- Solaris的多线程模型中包括四种实体：进程，内核线程，用户线程和轻量级进程（LWP）

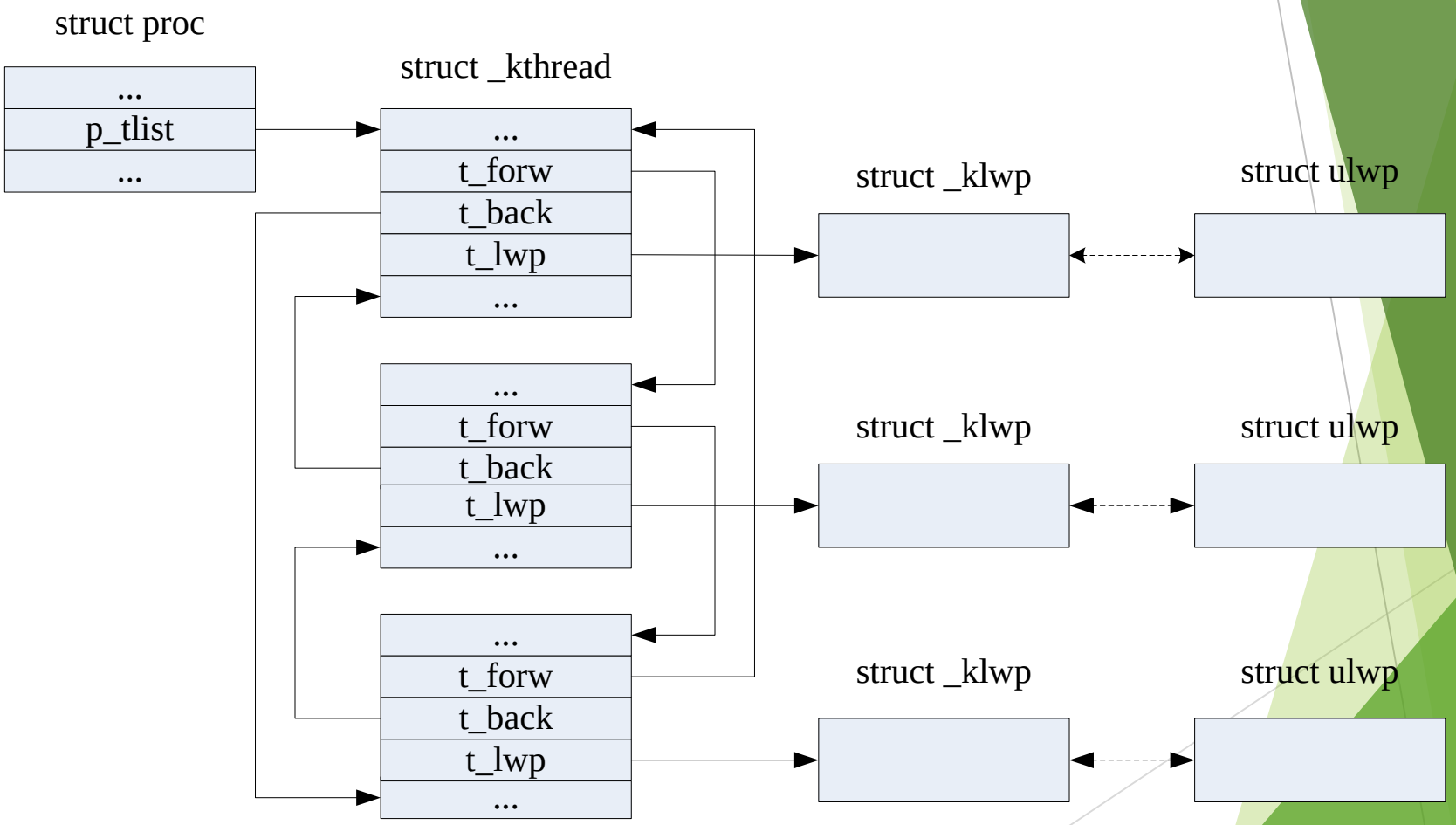
- Solaris内核是多线程的
  - 进程是资源分配和管理的单元
  - 内核级线程是内核的调度单元
  - 用户级线程是程序执行在用户态的抽象
  - LWP把用户线程和内核线程绑定到一起

重要的数据结构

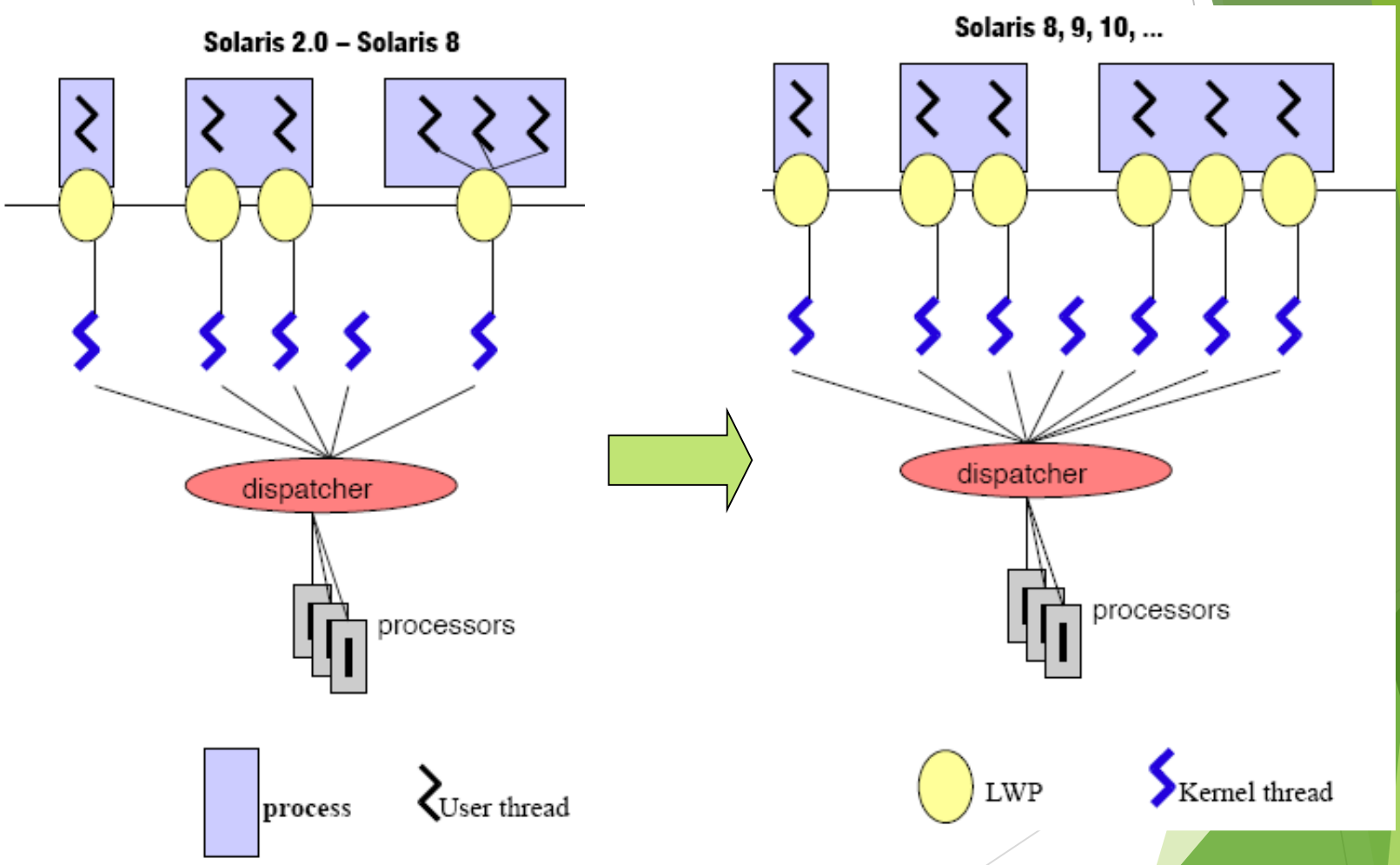


对象	定　　义	名　　字	包含数据结构定义的头文件
进程	一个执行环境——用于执行线程的状态容器	proc_t	uts/common/sys/proc.h
用户线程	在进程内由用户提供内核状态的对象	ulpw_t	lib/libc/_inc/thr_uberdata.h
轻量级进程	为用户线程提供内核状态的对象	klwp_t	uts/common/sys/klwp.h
内核线程	内核中调度和执行的基本单位	kthread_t	uts/common/sys/thread.h

用户线程、内核线程、lwp三者之间的关系



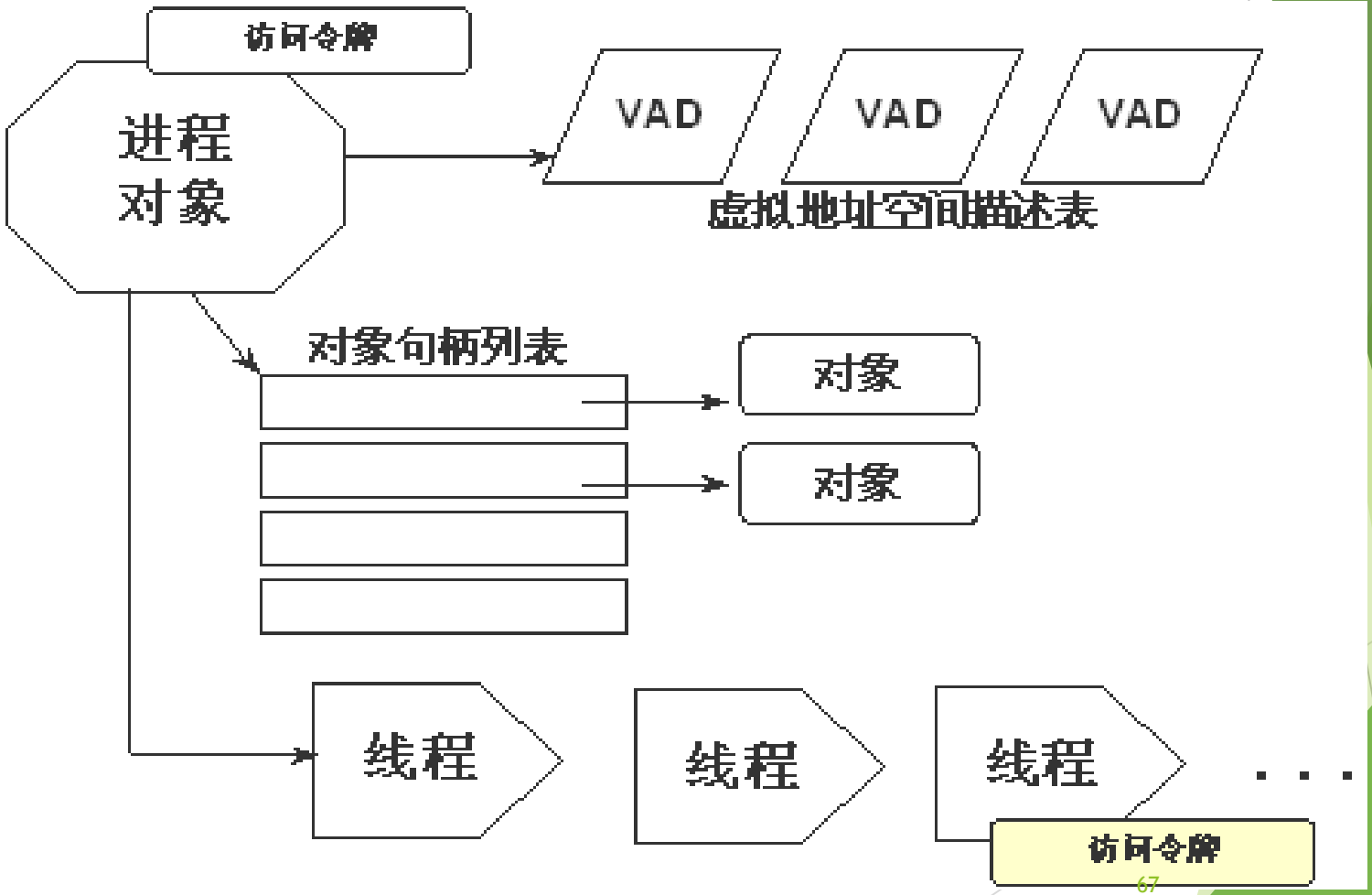
多线程模型的演化


Solaris 2.0 – Solaris 8
dispatcher
processors
process
User thread
Solaris 8, 9, 10, ...
dispatcher
processors
LWP
Kernel thread

四、Windows进程线程模型

- Windows进程是作为对象来管理的，进程对象的属性包括：
  - 进程标识(PID)
  - 资源访问令牌(Access Token)
  - 进程的基本优先级(Base Priority)
  - 默认亲合处理机集合(Processor Affinity)等

Windows中的进程及其资源

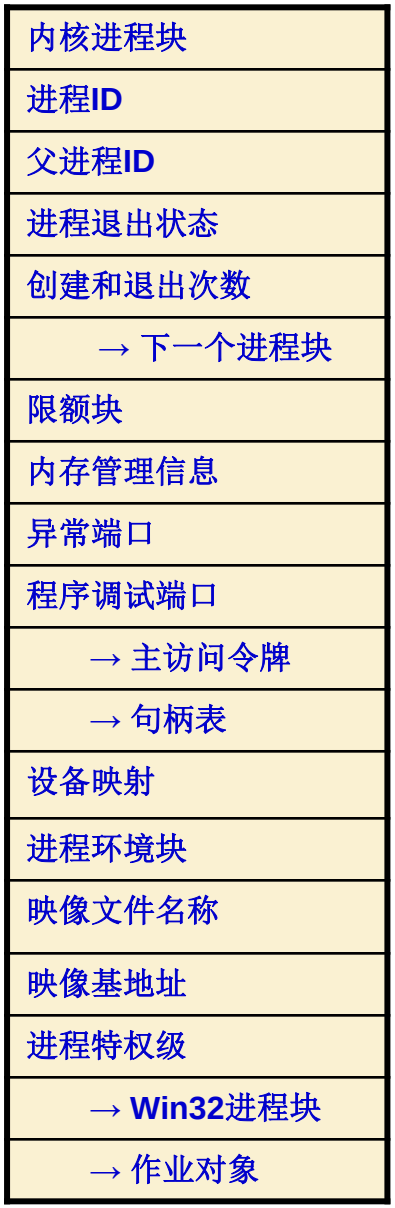


相关说明

- Windows 中的每个Win32进程都由一个执行体进程块（EPROCESS）表示，执行体进程块描述进程的基本信息，并指向其他与进程控制相关的数据结构
- EPROCESS的主要内容:
  - 线程块列表：描述属于该进程的所有线程的相关信息，以便线程调度器进行处理机资源的分配和回收
  - 虚拟地址空间描述表：描述进程地址空间各部分属性，用于虚拟存储管理
  - 对象句柄列表：当进程创建或打开一个对象时，就会得到一个代表该对象的句柄，用于对象访问

Windows 的进程

- Windows 的进程由执行体进程块EPROCESS表示，EPROCESS即执行体进程对象
- 进程对象的属性：PID，PCB， Access Token，Base Priority，句柄表，指向进程环境块PEB的指针，默认亲和处理器集合等
- 在Windows 中，PCB也称为内核进程块KPROCESS KPROCESS即内核进程对象
- EPROCESS和KPROCESS位于内核空间，PEB位于用户空间



EPROCESS

KPROCESS

PEB

Win32子系统的进程控制系统调用

- Win32子系统的进程控制系统调用：CreateProcess、ExitProcess和TerminateProcess

- CreateProcess用于进程创建
- ExitProcess和TerminateProcess用于进程退出

Windows 的进程——进程创建

- CreateProcess()函数用于创建新进程及其主线程，以执行指定的程序
- 新进程可以继承：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

Windows的进程——进程退出

- ExitProcess()或TerminateProcess()，则进程包含的线程全部终止
- ExitProcess()终止一个进程和它的所有线程；它的终止操作是完整的，包括关闭所有对象句柄、它的所有线程等
- TerminateProcess()终止指定的进程和它的所有线程；它的终止操作是不完整的（如：不向相关DLL通报关闭情况），通常只用于异常情况下对进程的终止

Windows的线程

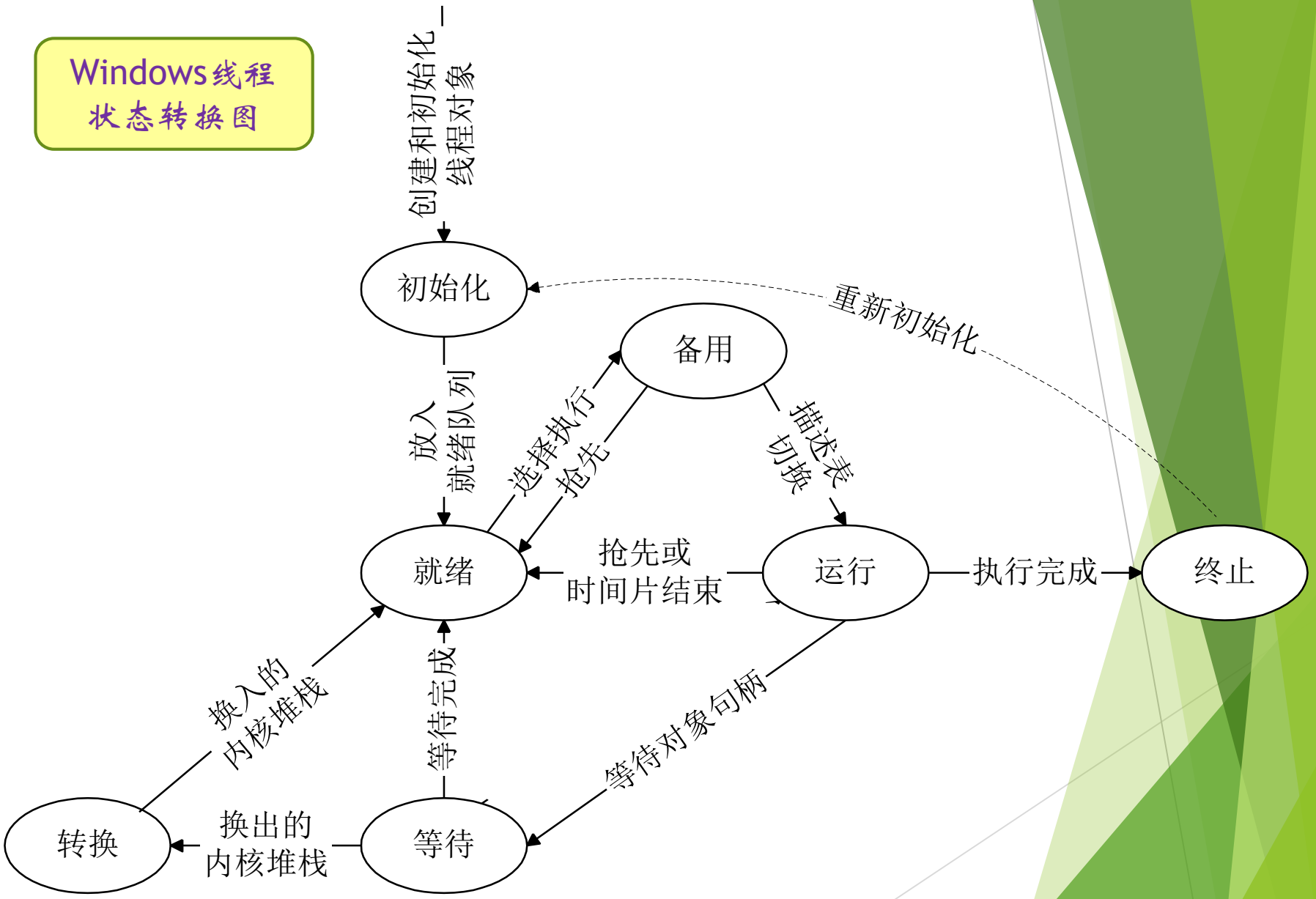
- Windows线程由执行体线程块ETHREAD表示，即执行体线程对象，其中包含
  - 内核线程块KTHREAD，即线程控制块TCB，KTHREAD即内核线程对象
  - 指向线程环境块TEB的指针

- ETHREAD和KTHREAD位于内核空间，TEB位于用户空间

Windows线程状态

- 就绪状态(Ready)：进程已获得除处理机外的所需资源，等待执行
- 备用状态(Standby)：特定处理器的执行对象，系统中每个处理器上只能有一个处于备用状态的线程
- 运行状态(Running)：完成描述表切换，线程进入运行状态，直到内核抢先、时间片用完、线程终止或进行等待状态
- 等待状态(Waiting)：线程等待对象句柄，以同步它的执行；等待结束时，根据优先级进入运行、就绪状态
- 转换状态(Transition)：线程在准备执行而其内核栈被换出内存时，线程进入转换状态；当其内核栈调回内存，线程进入就绪状态
- 终止状态(Terminated)：线程执行完就进入终止状态；如执行体有一指向线程对象的指针，可将线程对象重新初始化，并再次使用
- 初始化状态(Initialized)：线程创建过程中的内部状态

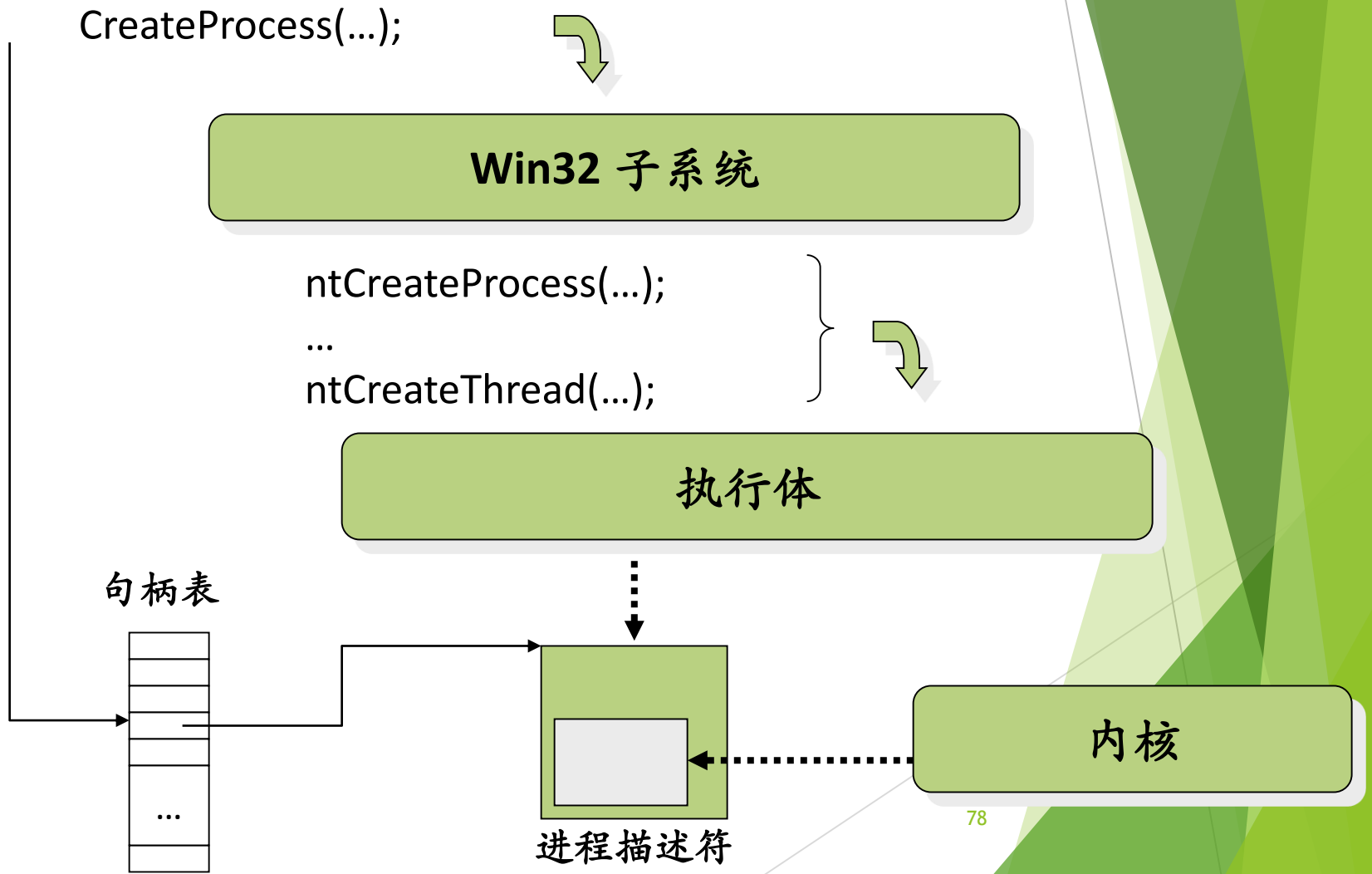
Windows线程状态转换图



Windows线程的有关API

- CreateThread()函数在调用进程的地址空间上创建一个线程，以执行指定的函数；返回值为所创建线程的句柄
- ExitThread()函数用于结束本线程
- SuspendThread()函数用于挂起指定的线程
- ResumeThread()函数递减指定线程的挂起计数，挂起计数为0时，线程恢复执行

例子


CreateProcess(…);
Win32 子系统
ntCreateProcess(…);
…
ntCreateThread(…);
执行体
句柄表
…
进程描述符
内核

重点小结(1/2)

进程

- 并发性
  任何进程都可以同其他进程一起向前推进
- 动态性
  进程是正在执行程序的实例
  - ✓ 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
  - ✓ 动态的地址空间
- 独立性
  进程是资源分配的一个独立单位，不同进程的工作不相互影响
  例如：各进程的地址空间相互独立
- 制约性
  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，因访问共享数据/资源或进程间同步而产生制约
- 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 + PCB

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

作业3

1、谈谈你对“进程是对CPU的抽象”这句话的理解。

2、当中断或系统调用把控制交给操作系统时，通常使用了与被中断进程运行栈不同的内核栈，为什么？

3、分析总结Linux的进程控制块task_struct。

4、本课件45页给出了一个多线程Web服务器。如果读取文件只能使用阻塞的read系统调用，那么Web服务器应该使用用户级线程还是内核级线程？为什么？

5、假设一个用户在命令行执行了程序A，A程序中调用了系统调用sleep(3)。请描述系统执行A程序的全过程。

提交时间：2020年10月25日晚23:30

XV6源代码阅读要求

阅读XV源代码之进程模型部分，结合代码写出阅读报告。

提交时间：2020年10月25日晚23:30

Thanks

The End