

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

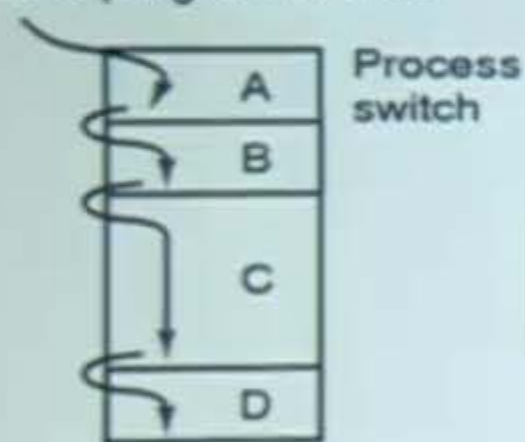


多道程序设计 (MULTIPROGRAMMING)

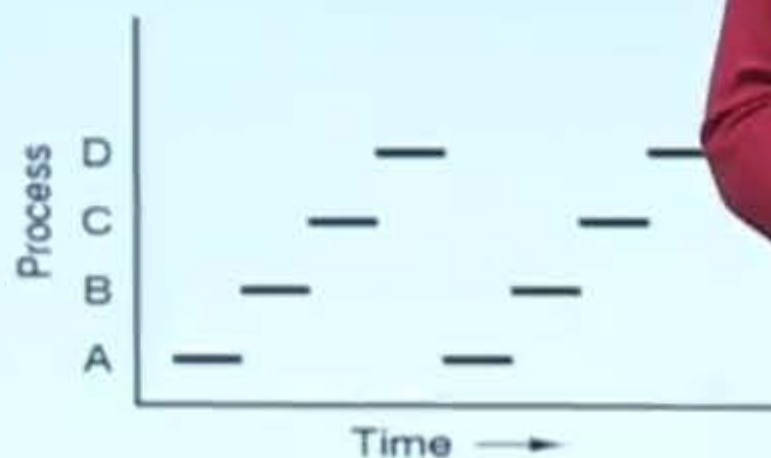
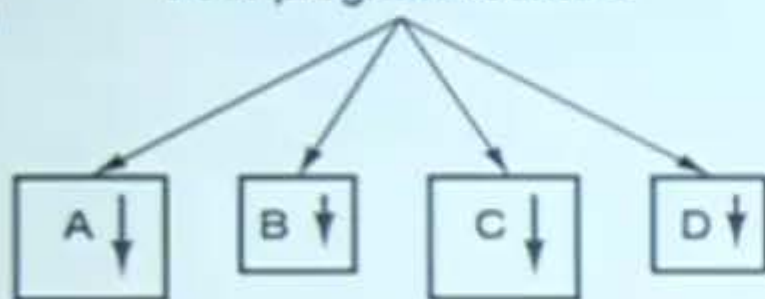
多道程序设计

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



4个程序计数器
Four program counters



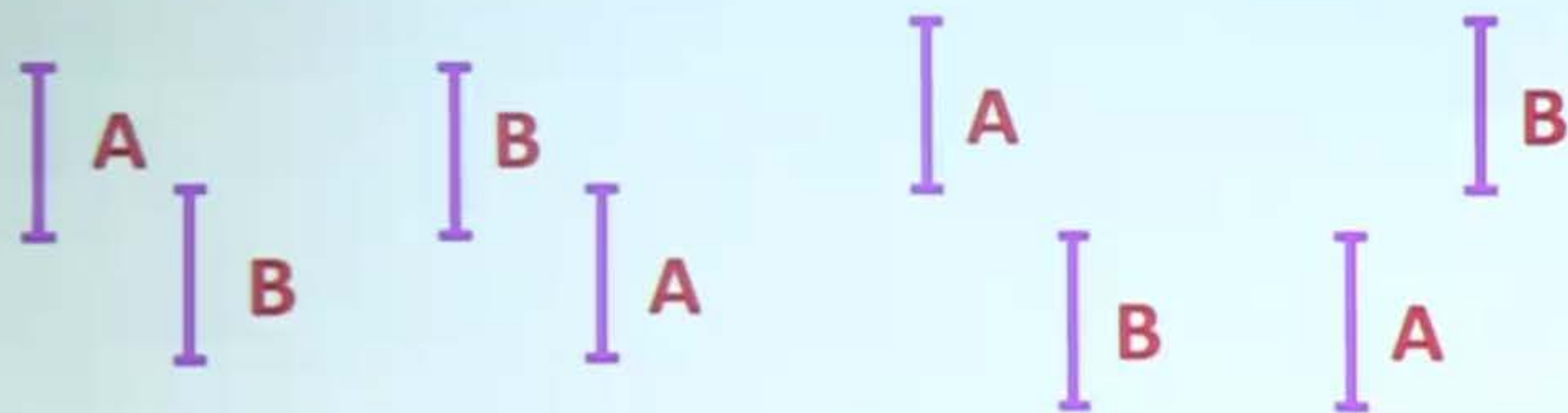
轮流执行



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



B 呢 并发执行 那么在一个并发环境下

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)

- 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

如何查看当前系统中有多少个进程?



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**

那么有的时候我们会说，这就是操作系统的并发度 最多有多少个进程可以执行



PCB的内容应
该包括什么
呢？



都应该包含什么内容呢？ 我们让小明同学来做这样的设计






CPU 的现场信息 下面我们从四类信息当中分别介绍 第一类呢

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

子进程, 有父进程, 或者兄弟姐妹进程呢, 也可以把他们的进程的之间的关系记录下来



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

另外进程呢，还有很多的队列，所以呢这里头有一些队列的指针啊，就所以进程控制信息是比较多的



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

进程在使用过程中会用到 存储空间，会用到一些打开的文件

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

好，这就是进程控制块应该包括的几类信息

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

进程控制块里要保存什么信息？ 从文件管理进程控制块要保存什么信息？

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		

EPROCESS、KPROCESS 和 PEB 好，我们来简单地看一下

Linux 2.6.x

```
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
```

```
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 utime, stime, ctime, cstime;
```

而具体的内容呢

一页都放不下，我们还有两页具体的内容呢，我们不做介绍，感兴趣的

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 minflt, majflt, cminflt, cmajflt;
```

```
/* 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;
```

```
/*
```

```
/*
```

```
/*
```

```
/*
```

```
/*
```

```
/*
```

```
/*
```

```
/*
```

```
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
```

```
/*
```

```
/*
```

```
/*
```

```
/*
```

```
/*
```

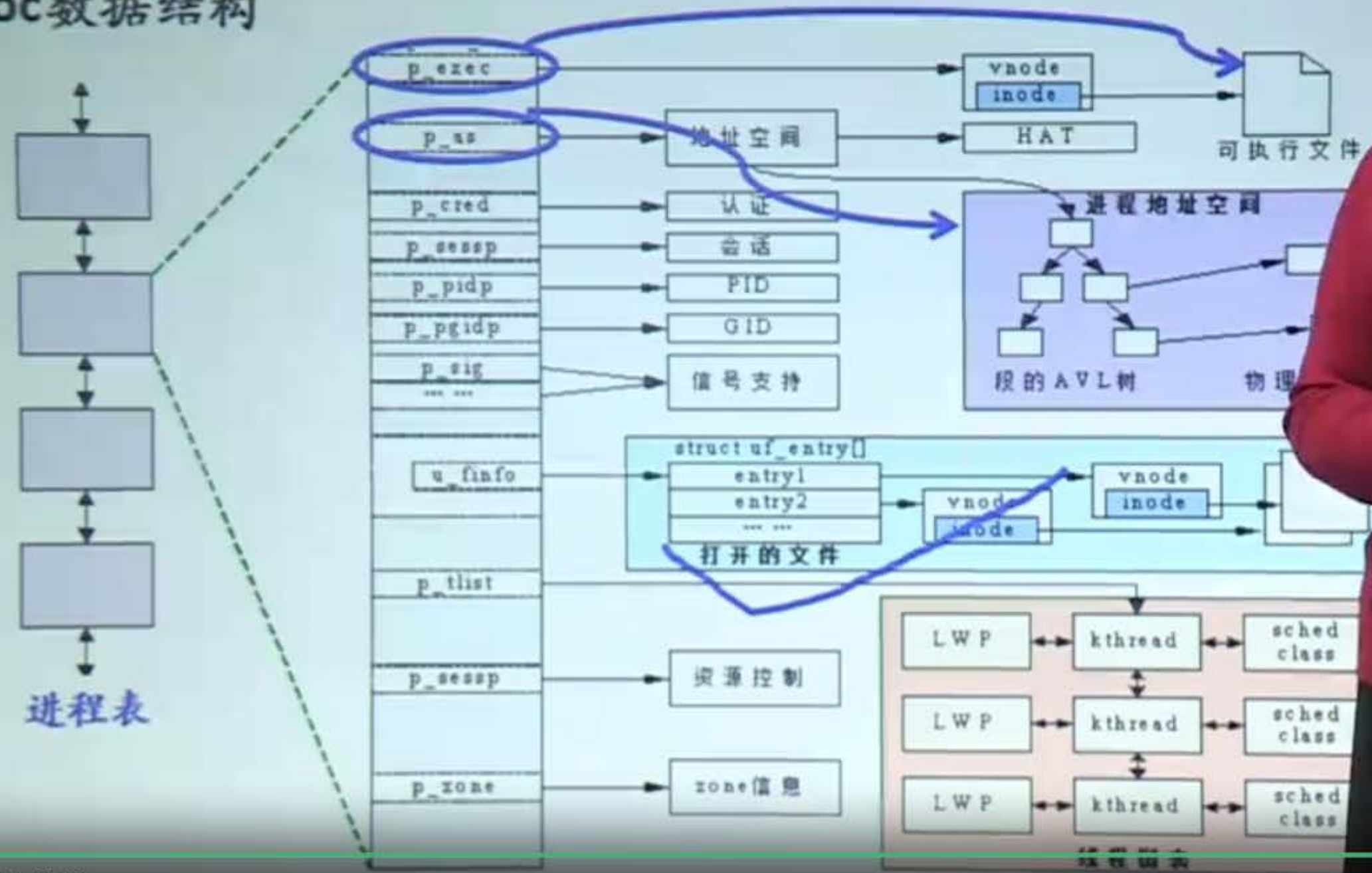
```
/*
```



一页都放不下，我们有两页 具体的内容呢，我们不做介绍，感兴趣的

SOLARIS的进程控制块与进程表

proc数据结构



大家好，今天我给大家带来的是操作系统原理的第三讲 进程/线程模型，这一讲主要有两部分内容 第一个是进程模型，什么是进程？操作系统在设计进程模型的时候 主要考虑哪些问题？第二部分是线程模型 为什么引入线程？操作系统在支持线程方面 都要做哪些工作？首先我们来介绍一下进程的基本概念 我们从多道程序设计技术这个概念入手 多道程序设计技术是操作系统最早引入的软件技术 它的基本思想是，允许多个程序同时进入内存并运行 为什么要有这个技术呢？主要是为了提高 CPU 的利用率 进而提高整个系统的效率 我们来看一个例子 在第一张图当中 在内存里头呢，有四个程序 这四个程序呢是串行执行的，为什么呢？因为我们只有一个物理的程序计数器，所以 A 程序执行完了，B 程序才能执行 那么有了多道程序设计技术之后 就得到了这样一个场景 每个程序呢 变换成了一个独立的控制流，占用一个 逻辑的程序计数器 这也是操作系统虚拟性的一个体现 把一个物理的程序计数器，给它变换成多个 逻辑的程序计数器，实际上每个程序都有自己的程序计数器 那么由于物理上只有一个程序计数器，所以每个程序真正的上 CPU 就把逻辑程序计数器的内容，推送到物理程序计数器里头 那么通过这种变换，达到了 在内存中同时有多个程序，他们又能够并发执行的效果 我们来看一下第三张图 第三张图呢，表示出在一个时间间隔内，每一个程序 A B C D 都执行过了 那么由于只有一个物理 CPU 所以这些程序呢 是轮流在 CPU 上执行 但是呢 从宏观上讲呢 它们都在并发执行 因此，在这样一个计算环境下 多个程序并发执行，就给我们带来了一个新的挑战 如何管理在并发环境下 同时执行的这些程序，那么我们来看一下什么是并发环境？所谓并发环境就指的是在一段时间间隔内 在物理机器上，有两个或者两个以上的程序 它们同时处于开始运行，但尚未 结束的状态，也就是说一个程序已经开始执行了，但 另一个程序呢 也，接着开始执行 第一个程序没有结束的时候，第二个程序又开始执行了 那么在这个并发环境下还有一个特征 也就是说这些程序它们谁先执行，谁后执行，它的 次序是不确定的，是没法预测的 好，这就是说并发环境下 若干程序处于开始执行，但尚未结束的状态 它们的次序是事先不确定的 那么在并发环境下执行的程序，我们就把它称之为并发程序 我们来看这就是几个并发程序的例子 那么 A 和 B 并发，或者是 B 和 A 并发 那么这两个程序呢，这两个例子当中呢，实际上它们都有一些交错的、重叠的部分 那么像后面两个例子，大家可以看到 A 执行完了，B 才执行 或者是 B 执行完了 A 才执行，由于这两个程序的执行顺序是不可预测的 所以呢 这两种情况都可能在系统中发生，所以我们可以说 A 和 B 呢 并发执行 那么在一个并发

环境下执行的并发程序，我们怎么样来刻画这样的程序呢？于是呢，进程的定义就应运而生了。什么是一个进程？进程这个概念实际上是非常重要的，而且是非常伟大的概念，它准确地刻画了一个并发环境下的，并行程序的执行。那么什么是一个进程呢？我们来看一下，进程是具有独立功能的程序关于某个数据集上的一次运行活动。进程呢，是资源分配的单位，也是 CPU 调度的单位，从这个描述当中，实际上我们看到了以下非常重要的几个特点。首先，进程是程序的一次执行过程。那么一个程序执行了两次，执行了三次，那就是不同的进程。进程呢，又是运行程序的一个抽象，它代表了所运行的那个环境，也就是它代表了一个 CPU。因此，我们有时候说进程是对 CPU 的一个抽象。正是因为有了虚拟化技术，所以将一个 CPU 把它变换成多个虚拟的 CPU。每个进程好像都在跑在自己的 CPU 上。啊，这就是一个抽象的结果，那么作为进程，它在运行过程中，需要各种各样的资源。所以操作系统的资源是以进程为单位来分配的，比如说内存、文件等等。最重要的一个资源呢，实际就是地址空间。操作系统为每一个进程分配了一个独立的地址空间。关于这个概念，我们后面还会非常详细的去介绍。那么操作系统把 CPU 的控制权，交给了某一个进程，让这个进程上去运行，那么这就称之为一个调度。所以操作系统通过调度把 CPU 的控制权交给某个进程。好，那么我们怎么知道这个系统中，到底有多少个进程在运行呢？我们可以通过 Windows 下，任务管理器，或者是在 linux 下，用 PS 命令，我们就大概知道系统中，有多少个进程在运行了。那么在操作系统的这个执行过程中，会有很多的程序向操作系统提出申请来运行，那么操作系统怎么知道这些进程是存在，还是不存在呢？这里头我们就介绍操作系统为了管理进程所设计的一个非常重要的数据结构，进程控制块 PCB。那么进程控制块其实又有一些其他的名称，比如说进程描述符，比如说进程属性，那么这个数据结构实际上是专门用于控制和管理进程的，操作系统设计这个数据结构保存控制和管理进程所需要的所有的信息，所以它是一个专门的数据结构，也是非常重要的数据结构。那么这些数据结构里头，记录了什么信息呢？主要是记录了进程的各种属性，并且描述出进程的运动变化过程，因为进程不断的往前进，所以它进展到什么程度了呢？也记录在这个数据结构里头。操作系统是通过这个数据结构 PCB 来管理控制进程的，因此这个数据结构就是操作系统感知进程存在的一个标志。有一个进程存在，就有一个 PCB。所以它们是一一对应的。那么操作系统管理了很多的进程，为了便于管理，就把所有进程的每个进程的 PCB，把它集中在一

起，放在了内存的固定区域 那么这就是形成了进程表，啊，也就是 所有进程的 PCB 的一个集合，就是进程表 那么这个进程表呢，大小呢，往往是固定的 也就是它的大小，确定了 在一个操作系统中，最多支持多少个进程 那么有的时候我们会说，这就是操作系统的并发度 最多有多少个进程可以执行 下面我们来讨论 PCB 都应该包含什么内容呢？ 我们让小明同学来做这样的设计 好，那我们来看看首先 要对进程有一些基本信息的描述 那么进程是运动变化的，所以呢，我们需要一些控制操作，需要一些控制信息 进程在运行过程中需要用到资源，那么 资源的使用情况，我们要记录下来，另外 在进程控制块当中，还要保存 CPU 的现场信息 下面我们从四类信息当中分别介绍 第一类呢 是进程描述信息 比如说创建了一个新的进程，就要给这个进程一个标识，唯一的一个 ID 就像每个学生有一个学号一样，它是唯一的，通常呢 是一个整数 那么也可以给进程，它的谁创建这个进程的用户的记录在这里 如果这个进程有子进程，有父进程，或者兄弟姐妹进程呢，也可以把他们的进程的之间的关系记录下来 那么进程的控制信息呢，主要包括当前进程处于什么样的状态，这个进程 为了调度它的优先级是多少，进程在执行过程 当中的代码执行的入口地址，或者是可执行文件在磁盘上的位置 另外进程呢，还有很多的队列，所以呢这里头有一些队列的指针啊，就所以进程控制信息是比较多的 进程在使用过程中会用到 存储空间，会用到一些打开的文件 这些信息呢，要记录在资源和使用的这个情况的这个 类里头。那么 CPU 的现场信息是指当进程不运行的时候 操作系统要把一些重要的信息，硬件 执行的状态信息，保存在这个 PCB 里头 那么这些信息呢，包括了一些通用寄存器 然后程序计数器、栈指针、程序状态字等等 还有一个比较重要的就是 跟地址空间相关的一个页表的指针 好，这就是进程控制块应该包括的几类信息 我们也可以从另外一个角度来看 PCB 的内容 比如说如果是进程管理，它又用到哪些信息？ 这里头很多项我们前面都介绍过了 如果从存储管理 进程控制块里要保存什么信息？ 从文件管理进程控制块要保存什么信息？ 所以这是从另外一个角度看，按不同的操作系统的功能 来分类说哪些信息应该保存在进程控制块当中 PCB 实际上是一个通用的名字 啊，它表示了这个数据结构的主要的特征 但实际上在一个实际的操作系统当中 进程控制块的名字呢是不一样的，比如说在 Linux 当中，进程控制块的名字呢就是 `task_struct` 在 Windows 当中，进程控制块呢由几部分组成 `EPROCESS`、`KPROCESS` 和 `PEB` 好，我们来简单地看一下 Linux 操作系统当中

的进程控制块 我们这里选取了 Linux 2.6.x 版本 那么我们也不详细来介绍，大家可以看一下一个实际操作系统进程控制块有哪些信息非常之多 一页都放不下，我们有两页 具体的内容呢，我们不做介绍，感兴趣的 同学呢，可以自己找到相关的材料去浏览一下 那么我们这里得举一个 SOLARIS 的进程控制块和进程表的例子 我们来看 SOLARIS 的进程表 因为 SOLARIS 是基于 Unix 操作系统的 所以它的进程控制块的名字呢，一般叫 Proc 结构 每一个 Proc 结构代表一个 PCB 把所有的 Proc 结构，把它组织成一个链，那么这就是一个进程表 我们看一个 Proc 结构应该保存什么信息。这里头呢我们重点介绍三个 首先呢，是第一个是 可执行文件，也就是通过这样一个记录信息，可以找到 这个进程所对应的可执行文件，在磁盘上的位置 第二个，进程的地址空间 进程的地址空间，怎么样把它记录在 进程控制块当中，或者叫做 Proc 结构中，因为进程地址空间呢 放了很多内容，每一项内容都放在一段里头。所以我们呢是通过段来把进程地址空间，把它描述清楚的，那么把这些段呢 按照地址大小的顺序，把它 建立成一个 AVL 树，那么便于以后的查找 这个以后我们还会去介绍，大家只知道说一个地址空间呢，啊，分成很多的段 每个段放了一些内容，那么怎么找到这些段呢，要通过一个 AVL 树去找 另外呢，进程运行过程中，需要用到文件，所以这里头呢，有一张表叫做打开文件表 通过这张表可以把所有打开的文件都能找到 我们通过这样一个例子呢，把一些在 PCB 里头非常重要的信息呢介绍一下，同时给大家 看一下啊，一个实际操作系统当中，PCB 里头有哪些内容