第一章 绪论

操作系统概念

- 操作系统是计算机系统中包含的基本程序集合
  - 内核　（这门功课的重点）
  - 实用程序

- 操作系统的2个主要的目标
  - 与硬件交互
  - 管理软件资源
  - 为APP提供执行环境

第一部分　Linux/Unix系统概貌

- Unix/Linux系统中，内核可以访问全部系统资源，应用程序只有部分权限，应用程序不可以执行的操作：
  - 访问硬件
  - 读写文件
  - 访问存放内核代码、内核变量的内存空间
  - 读写CPU内部，除通用寄存器之外的其它寄存器

- 支持Linux系统的CPU，必须具备2种执行模式
  - 用户态（user mode），CPU正在执行的是应用程序
  - 核心态（kernel mode），CPU正在执行的是内核

- 应用程序想要访问计算机资源时，执行系统调用
  内核执行系统调用，为应用程序 IO数据，控制外部设备

一、多用户系统

- 允许多个用户并发、独立使用的计算机系统

- 安全保护机制
  - 认证机制，核实用户身份 （登录）
  - 一个保护机制，防止有错误的用户程序妨碍其它应用程序在系统中运行
  - 一个保护机制，防止有恶意的用户程序干涉或窥视其他用户的活动
  - 记账机制，为每个用户提供资源使用的限额，限制用户使用的资源总数

- Linux是基于系统资源硬件保护的多用户系统

二、用户和用户组——用户

- 每个合法用户有唯一ID，这个数字是UID。

- 文件的UID

每个合法用户在机器上有其私有磁盘空间，用户ID号是私有空间中文件的UID号。该用户是这些文件的文件主。除非授权，不允许其他用户访问这些文件。

- 进程的UID

成功登录后至logout注销，是用户的一次会话过程（session）。其间，每执行一个应用程序，系统就会派发一个进程负责执行这个应用程序。成功登录后，系统将UID记录在每个进程的进程控制块task_struct中的uid字段。这就是进程的UID。

二、用户和用户组 —— 系统使用UID确定进程的文件访问权限

- 当进程想要访问文件时，如果进程的UID和文件的UID相等。这是文件主在操作，进程拥有这个文件的全部访问权限。否则，除非授权，进程不可以访问这个文件。

- 超级用户
  - 以root身份登陆
  - 可以处理用户账号，可以备份系统，可以升级程序。
  - 可以访问系统中的每个文件，执行所有应用程序，能干涉每个用户程序的活动。

二、用户和用户组 —— 用户组

- 组（group）
  - 用户分组，组内用户可共享文件。
  - 同一个用户可以参加不同的group。

- 每个进程有UID和GID。系统匹配进程和文件的UID、GID，判断前者有没有相应的文件访问权限。

三、进程

- 在Linux系统中，每当用户需要执行应用程序，系统就会创建一个新的进程，由它来承担执行应用程序的任务。

- 进程存在的价值：　执行应用程序 *

- 进程是APP的执行上下文

解释书中的一段话

- In traditional operating systems, a process executes a single sequence of instructions in an address space; **the address space** is **the set of memory addresses that the process is allowed to reference**.

- Modern operating systems allow processes with multiple execution flows that is, multiple sequences of instructions executed in the same address space.

Linux系统是多道进程并发的

- 现运行进程是CPU正在执行的进程
  - 单CPU系统中只有一个现运行进程。
  - 多CPU系统，有多少个CPU，就有多少个现运行进程。这些进程物理并行。
- 系统中进程的数量远远大于CPU的数量。除现运行进程之外，其它进程
  - 就绪　　等待使用CPU
  - 阻塞　　等待IO操作结束，或等待使用外部设备，或等待内存变量开锁~
- 就绪进程共享CPU。所以，同时执行的APP越多，每个APP跑得就越慢。

进程切换

- 进程切换：CPU换现运行进程
  - 进程PA将CPU让给进程PB
- 进程的硬件上下文：这个进程运行时，CPU内部寄存器的值

  PS：一颗CPU只有一套计算资源，为现运行进程服务。这些计算资源包括：所有的寄存器，FPU 和 其它计算资源。寄存器的值，有些反应的是**系统**当前工作状态。有些存放正在运行的**进程**的工作结果和运行环境。进程的硬件上下文指后者寄存器的值，是进程的CPU执行现场。
- 进程切换操作
  - 保护PA进程的CPU执行现场，恢复PB进程的CPU执行现场。

现运行进程的模式切换

- 用户态→核心态　　CPU暂停执行应用程序，开始跑内核代码
  - 现运行进程执行系统调用，访问内核变量，获取IO数据
  - 现运行进程执行中断处理程序
- 核心态→用户态
  - 系统调用执行完毕，现运行进程返回用户态，从系统调用的下条指令开始继续执行应用程序。
  - 中断处理程序执行完毕，现运行进程返回用户态，从哪条指令开始继续执行应用程序？？　一定是被中断的那条指令的下条指令嘛？

内核运行在进程上下文

内核运行在中断上下文

四、内核架构（Kernel Architecture）

- 单内核

  内核所有模块打包成一个程序，系统初始化的时候从磁盘加载入内存。

- 微内核

  内核=一个微内核+好多模块。这里是好多程序。模块之间，用消息传递的方式传数据。微内核负责消息传递。系统初始化的时候从磁盘加载微内核入内存。之后，内存按需加载其余模块。

- Linux

  内核=基本模块+LKM。基本模块是一个程序，集成了Linux运行不可或缺的进程管理，内存管理，ext2文件系统，VFS……模块。其它模块，动态链接。完成后，和基本模块组成一个程序，可以彼此调用。

五、UNIX 文件系统　略

- 一切皆文件
  - 正规文件
    存放在磁盘上，用来永久存储数据。
  - 进程通信工具
    socket、pipe、FIFO 文件
  - 字符设备文件　　终端（tty，键盘+屏幕）
    键盘是标准输入文件 STDIN　　（0#文件）
    屏幕是标准输出文件 STDOUT　（1#文件）
    标准错误输出文件 STDERR　　（2#文件）

- 符号链(软链接)

  是磁盘上的小文本文件，用来存放另一个文件的文件名，便于方便地引用其它文件。与windows系统中的快捷方式类似。
- 设备文件

  所有设备都是文件
- proc文件系统

  用户态进程可以访问内核数据结构

  /proc/cpuinfo　CPU型号

  /proc/<PID>/maps

  代码段、数据段、堆、栈……所有信息
- sys文件系统

  输出关于外设和系统总线的信息

进程使用文件描述符，读写文件，输入输出数据

- 键盘输入

  read（0，……，……）

- 屏幕输出字符

  write（1，……，……）

六、信号

- 信号向进程报告系统事件，这些系统事件会直接影响进程的运行过程，甚至杀死进程。有2种系统事件：

  - 异步通告

    用户按下ctrl+c，socket连接断了 ~
  - 同步错误（异常）

    应用程序段错误（访问非法地址）

Linux传统信号

- **Linux定义了32种不同的事件。每种事件对应固定信号，每种信号对应唯一整数 1~31**

- **Linux规定不同信号默认信号处理方式**
  - **终止进程（进程被信号杀死，这是绝大多数信号的默认处理方式）**
  - **core dump 将出错进程的虚地址空间和CPU上下文写入文件**
  - **忽略信号**
  - **挂起进程**
  - **恢复执行先前暂停的进程**

编号	信号名称	默认的信号处理方式	解释	POSIX支持
1	SIGHUP	Abort	控制 tty 断开连接（挂断）	是
2	SIGINT	Abort	来自键盘的中断（用户在键盘上按 CTRL_C）	是
3	SIGQUIT	Dump	来自键盘的退出（TTY 键盘上按 CTRL_\）	是
4	SIGILL	Dump	非法指令（异常）	是
5	SIGTRAP	Dump	跟踪断点（遇到 debug 断点，用于调试）	否
6	SIGABRT	Dump	异常结束（使进程流产）	是
6	SIGIOT	Dump	等价于 SIGABRT	否
7	SIGBUS	Abort	访问内存失败	否
8	SIGFPE	Dump	算术运算或浮点处理出错	是
9	SIGKILL	Abort	强迫进程终止（不可屏蔽）	是
10	SIGUSR1	Ignore？	开放给应用程序自行定义和使用	是
11	SIGSEGV	Dump	越界访问内存	是
12	SIGUSR2	Ignore？	开放给应用程序自行定义和使用	是
13	SIGPIPE	Abort	向无读者的管道（管道读端已关闭）写	是
14	SIGALRM	Ignore？	由 setitimer()设置的定时器到点	是
15	SIGTERM	Abort	进程终止	是
16	SIGSTKFLT	Abort	用于堆栈出错（尚未使用）	否
17	SIGCHLD	Ignore	子程序停止或结束	是
18	SIGCONT	Continue	令暂停的进程恢复执行(与 SIGSTOP 结合使用)	是
19	SIGSTOP	Stop	进程暂停运行，转入 TASK_STOPPED 状态	是
20	SIGTSTP	Stop	CTRL_Z，所有前台进程组中的进程挂起（进入 TASK_STOPPED 状态）	是
21	SIGTTIN	Stop	后台进程请求输入（读控制终端）	是
22	SIGTTOU	Stop	后台进程请求输出（写控制终端）	是
23	SIGURG	Ignore	收到带外数据（表示 socket 上的紧急事件）	否
24	SIGXCPU	Abort	进程使用 CPU 已超过限制	否
25	SIGXFSZ	Abort	进程使用文件大小超过限制	否
26	SIGVTALRM	Abort	由 setitimer()设置的“虚拟”定时器到点	否
27	SIGPROF	Abort	由 setitimer()设置的“统计”定时器到点	否
28	SIGWINCH	Ignore	控制终端窗口的大小发生改变	否
29	SIGIO	Abort	用于异步 IO	否
29	SIGPOLL	Abort	等价于 SIGIO	否
30	SIGPWR	Abort	电源供给失效	否
31	SIGUNUSED	Abort	没有使用	否

Linux传统信号

- 应用程序可以改变信号的处理方式
  - 忽略
  - 回用户态，执行程序员编写的信号处理函数
- 发送信号，就是向目标进程发送这个整数
- 每个进程，有一个数据结构，装它收到的信号。另一个数据结构，装每种信号的处理方式。

编号	信号名称	默认的信号处理方式	解释	POSIX支持
1	SIGHUP	Abort	控制 tty 断开连接（挂断）	是
2	SIGINT	Abort	来自键盘的中断（用户在键盘上按 CTRL_C）	是
3	SIGQUIT	Dump	来自键盘的退出（TTY 键盘上按 CTRL_\）	是
4	SIGILL	Dump	非法指令（异常）	是
5	SIGTRAP	Dump	跟踪断点（遇到 debug 断点，用于调试）	否
6	SIGABRT	Dump	异常结束（使进程流产）	是
6	SIGIOT	Dump	等价于 SIGABRT	否
7	SIGBUS	Abort	访问内存失败	否
8	SIGFPE	Dump	算术运算或浮点处理出错	是
9	SIGKILL	Abort	强迫进程终止（不可屏蔽）	是
10	SIGUSR1	Ignore？	开放给应用程序自行定义和使用	是
11	SIGSEGV	Dump	越界访问内存	是
12	SIGUSR2	Ignore？	开放给应用程序自行定义和使用	是
13	SIGPIPE	Abort	向无读者的管道（管道读端已关闭）写	是
14	SIGALRM	Ignore？	由 setitimer()设置的定时器到点	是
15	SIGTERM	Abort	进程终止	是
16	SIGSTKFLT	Abort	用于堆栈出错（尚未使用）	否
17	SIGCHLD	Ignore	子程序停止或结束	是
18	SIGCONT	Continue	令暂停的进程恢复执行(与 SIGSTOP 结合使用)	是
19	SIGSTOP	Stop	进程暂停运行，转入 TASK_STOPPED 状态	是
20	SIGTSTP	Stop	CTRL_Z，所有前台进程组中的进程挂起（进入 TASK_STOPPED 状态）	是
21	SIGTTIN	Stop	后台进程请求输入（读控制终端）	是
22	SIGTTOU	Stop	后台进程请求输出（写控制终端）	是
23	SIGURG	Ignore	收到带外数据（表示 socket 上的紧急事件）	否
24	SIGXCPU	Abort	进程使用 CPU 已超过限制	否
25	SIGXFSZ	Abort	进程使用文件大小超过限制	否
26	SIGVTALRM	Abort	由 setitimer()设置的“虚拟”定时器到点	否
27	SIGPROF	Abort	由 setitimer()设置的“统计”定时器到点	否
28	SIGWINCH	Ignore	控制终端窗口的大小发生改变	否
29	SIGIO	Abort	用于异步 IO	否
29	SIGPOLL	Abort	等价于 SIGIO	否
30	SIGPWR	Abort	电源供给失效	否
31	SIGUNUSED	Abort	没有使用	否

实时Linux信号（略）

- 32~63号信号

信号和失败的系统调用

- 应用程序执行时，如果收到了信号
  - 被杀死
  - 执行信号处理程序

- 如果应用程序在执行慢系统调用的时候收到了信号，这个系统调用立即失败返回，返回值是-1，错误号error=EINTR。应用程序应该重启系统调用。

七、进程间通信

- 基于文件读写的进程间通信
  - PIPE　无名管道
  - FIFO　有名管道

PS1：管道文件是进程之间的通信介质。一个读进程，一个写进程。写进程写入管道的消息只能读取一次。文件读写同步问题由内核负责，程序员比较happy。

PS2：PIPE，父子进程之间通信。

FIFO，非父子进程也可以通信。

进程间通信

- 基于内存的进程间通信（System V IPC）
  - 消息队列
  - 共享内存+信号量
  - socket

PS：消息队列、共享内存和信号量是IPC对象。
socket是VFS管理的文件。
这些对象，和文件一样，使用完必须释放。

进程间通信

- 消息队列
  - 消息的接收进程PA拥有消息队列PAsQueue，进程将消息送入PAsQueue供PA进程处理。
  - PS1：消息只能被接收一次
  - PS2：消息队列的同步问题由内核负责，程序员比较happy
  - PS3：可以将消息队列看作无界缓冲区，供PA暂存来不及处理的数据

进程间通信

- 共享内存+信号量
  - 共享内存用来存放进程间交流的信息。
    - 内核不限制借助共享内存进行通信的进程的数量。
    - 内核只是提供存放信息的场地，不限制应用系统使用信息的方式。
  - 信号量　程序员使用信号量同步并发进程对共享内存的使用。必须确保正确无误地
    - 在必要时上锁使用共享内存中的变量
    - 访问结束，唤醒等待开锁的进程
    - 杜绝死锁

进程PA和PB共享内存

物理内存上的一块区域，同时映射到PA和PB的地址空间。

PA和PB都可以直接访问其中的变量。

进程间通信

- socket
  - 用 收发 TCP/UDP 包的方式通信
    - 一台机器上，许多进程通信。
    - 多台机器间，许多进程通信。
      - 同构集群
      - 异构集群
  - 通信模式较灵活
    - 一对一
    - 组播
    - 广播
  - 适合于构造由多个APP组成的分布式系统。基于socket搭建的是分布式系统的通信架构。这种分布式系统的优点在于：每个APP组成独立功能模块，各个APP可独立开发维护。

八、进程管理

- 派发一个新的进程，让它执行应用程序 A

```
if( fork() ==0 ) {
    exec(A, …… ); }
else
    父进程做想做的事，
    比如 wait()睡眠等待子进程终止
```

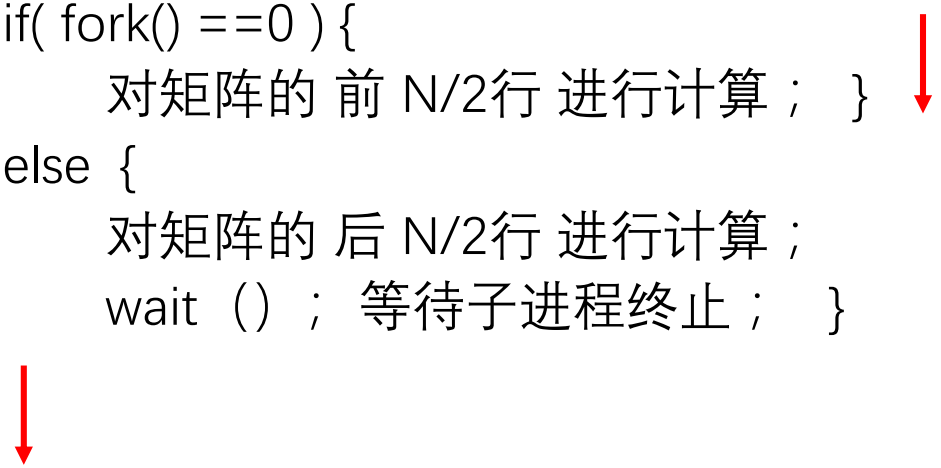
应用程序B中用来创建子进程的代码段

- fork完成的时候，子进程执行程序B。exec之后，执行程序A，从main函数的第一条指令开始运行。

- 用fork实现的并行计算

- 派发一个新的进程，让它和父进程一起执行应用程序 B

```
if( fork() ==0 ) {
        对矩阵的 前 N/2行 进行计算；  }
else  {
        对矩阵的 后 N/2行 进行计算；
        wait（）；  等待子进程终止；   }
```



应用程序B中用来
创建子进程的代码段

- 进程终止
  - 应用程序正常终止
    - main函数返回，exit（0）
    - 应用程序主动执行exit（n）
  - 应用程序异常终止
    - 执行应用程序的进程，运行时收到了信号。程序没有相应的信号处理函数，按系统默认的处理方式，进程便执行exit终止自己

- 进程终止后，释放其拥有的资源，只保留PCB，变成僵尸进程。向父进程发送SIGCHLD信号。
- 父进程执行wait系统调用回收子进程的PCB。

- 如果父进程终止前没有执行wait()。子进程的PCB由 1#进程（init进程）回收。

- 进程PCB回收的dilemma
  - 进程终止后，PCB不能马上释放。
    父进程需要读PID，系统需要将子进程的运行时间加在父进程的PCB中。
  - 系统需要尽早回收进程的PCB
    太多的僵尸进程，会使系统没有PCB可用，无法创建新进程

用信号处理程序回收终止子进程的PCB

```
2
3   void handler1(int sig)
4   {
5       int olderrno = errno;
6
7       if ((waitpid(-1, NULL, 0)) < 0)
8           sio_error("waitpid error");
9       Sio_puts("Handler reaped child\n");
10      Sleep(1);
11      errno = olderrno;
12  }
13
14  int main()
15  {
16      int i, n;
17      char buf[MAXBUF];
18
19      if (signal(SIGCHLD, handler1) == SIG_ERR)
20          unix_error("signal error");
```

```
22      /* Parent creates children */
23      for (i = 0; i < 3; i++) {
24          if (Fork() == 0) {
25              printf("Hello from child %d\n", (int)getpid());
26              exit(0);
27          }
28      }
29
30      /* Parent waits for terminal input and then processes it */
31      if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
32          unix_error("read");
33
34      printf("Parent processing input\n");
35      while (1)
36          ;
37
38      exit(0);
39  }
```

会少回收一个子进程的PCB

```
linux> ./signal1
Hello from child 14073
Hello from child 14074
Hello from child 14075
Handler reaped child
Handler reaped child
CR
Parent processing input
```

正确的代码

```
1   void handler2(int sig)
2   {
3       int olderrno = errno;
4
5       while (waitpid(-1, NULL, 0) > 0) {
6           Sio_puts("Handler reaped child\n");
7       }
8       if (errno != ECHILD)
9           Sio_error("waitpid error");
10      Sleep(1);
11      errno = olderrno;
12  }
```

九、进程组（process group）和 会话（session）

- 每执行一个应用程序，系统就会为它分配一个进程。

- Linux系统中，一个命令行就是一个作业job。承担作业执行任务的是进程组，包括为执行该命令行系统创建的所有进程。

  $ ls | sort | more

- 进程组中的所有进程
  - 利用管道传数据。前一个命令的标准输出是后一个命令的标准输入。
  - 利用管道同步。前一个命令执行完毕，后一个命令才会开始。

- 每个TTY只有一个前台进程组，其中的进程可以获得键盘输入，可以向屏幕输出。
- 每个TTY可以有多个后台进程组，其中的进程不能获得键盘输入，不可以向屏幕输出。

```
$ ls | sort | more &
```

- 进程组中创建时间最早的进程是进程组组长，是shell进程的子进程。进程组号，等于组长进程的PID号。
- shell进程wait睡眠等待前台进程组长终止。

- 会话是用户的一次上机过程，含本次上机过程中为用户服务的所有进程：一个shell进程 ∪ 一个前台进程组 ∪ 未运行结束的后台进程组
  ∪ 运行结束的所有进程组

- 每个TTY 1 个session，session ID是shell进程的PID。用户登录成功后，系统将用户的uid和使用的终端号tty记录在shell进程的task_struct中。

- session中所有其它进程是shell进程的子孙进程。fork执行时会将父进程task_struct的内容复制一份给子进程。所以，同一个session中所有进程的uid和tty与shell进程相等。
  tty的键盘是进程的0#文件，屏幕是1#文件

- ctrl+c 语义：终止前台作业
  - 实现：将信号SIGINT发送给前台进程组中的所有进程。用信号促其终止。
  - 后台作业不受ctrl+c影响。

八、内存管理

- 虚拟内存
  - APP指令和变量 存放在 进程的虚拟内存中。这个地址叫做**虚拟地址**，又叫做逻辑地址。比如~ 程序 EXM中，A变量的逻辑地址是10M。编译、链接完成后，它们的地址就固定了。

  - 指令和变量在物理内存中的地址，叫做**物理地址**。

  程序 EXM每次运行，占据不同的内存单元。所以程序 EXM这次运行和下次运行，变量A的物理地址是不同的，取决于内核为程序 EXM分配的内存单元的地址。

  - APP每次运行，需要访问变量A时，进程给出变量A的**逻辑地址**，**MMU**根据Linux内核提供的页表，**计算出**变量A的**物理地址**。发总线，读写A变量。

  MMU（Memory Management Unit），CPU中的单元，负责**地址映射**和内存保护

- 续　页式虚拟存储器
  - 物理内存分页，页框。每个页框4K字节
  - APP的代码和数据，分装在不相邻的页框中。每个进程一张页表，登记着存放APP代码和数据 的页框号。
  - 只要为main函数和main栈帧分配2个页框，APP就能够运行。剩余的代码和数据由缺页异常装入内存。

  - PS：进程运行时，内存中可能并没有APP全部的代码和数据。剩余的代码和数据在磁盘上，一些在交换区（swap area），一些在文件系统里。

进程运行过程中，始终没有访问到的~没必要为它们分配内存。

每次缺页，进程就会

- 去磁盘调数据。这是很慢的IO操作。
- 为进程分配内存，并刷TLB。

SO，进程运行时，缺页率越高，跑得越慢。

- 内存（RAM）的使用
  - 固定区域（若干兆字节）　　存放内核映像（内核代码和内核静态结构）
  - 其余部分
    - 存放内核动态数据结构，比如缓存（内核生成的一些信息），task_struct，……
    - 存放APP代码和数据，mmap的文件　这个叫做进程图像
    - 磁盘高速缓存和其余外设的缓存，存放IO数据
- 页框回收（Page-Frame-Reclaiming）
  - 当可用的内存不剩多少的时候，页框回收算法释放额外的内存。
- 碎片
  - 内核经常需要使用连续的主存页框，如果不存在，内存分配操作就会被拒绝。这就是碎片给系统带来的危害。

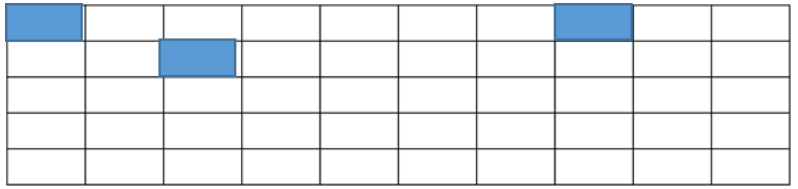
- KMA（Kernel Memory Allocator）子系统：分配动态内存。
  - 内核模块　　放内核数据结构。每次分配的尺寸：n个字节（n是这个数据结构的大小）
  - 进程（APP）　　放代码和数据（fork、exec、malloc分配得到的内存）每次分配的尺寸：1个物理页框，或整数个物理页框
  - 外设缓存　　存放IO数据的内核空间。尺寸：1个物理页框

- Linux的内存分配：　　伙伴系统 + slab分配算法。

- 伙伴系统管理**连续物理页框。**为整个系统提供空闲内存资源。

内核模块空间管理技术　slab

- 装动态内核数据结构的空间叫slab
  - 一个slab装的数据结构是同种类型的



可以装50个task_struct的slab
一个格子装一个数据结构

  - 每种数据结构，有自己的slab集合。叫cache。
  - 内核new数据结构的时侯，cache中找空格子。
    - 存在，空格子的起始地址是new的返回值
    - 不存在，向伙伴系统申请新slab，加入cache集合。

用户空间的空间管理技术

- 用户空间指为进程分配物理内存空间（APP用的内存）
  - 存放的是APP的代码和数据

- 空间管理技术
  - 基于缺页异常的请求调页
  - 写时复制（Copy On Write，COW）

外设缓存管理技术（磁盘）

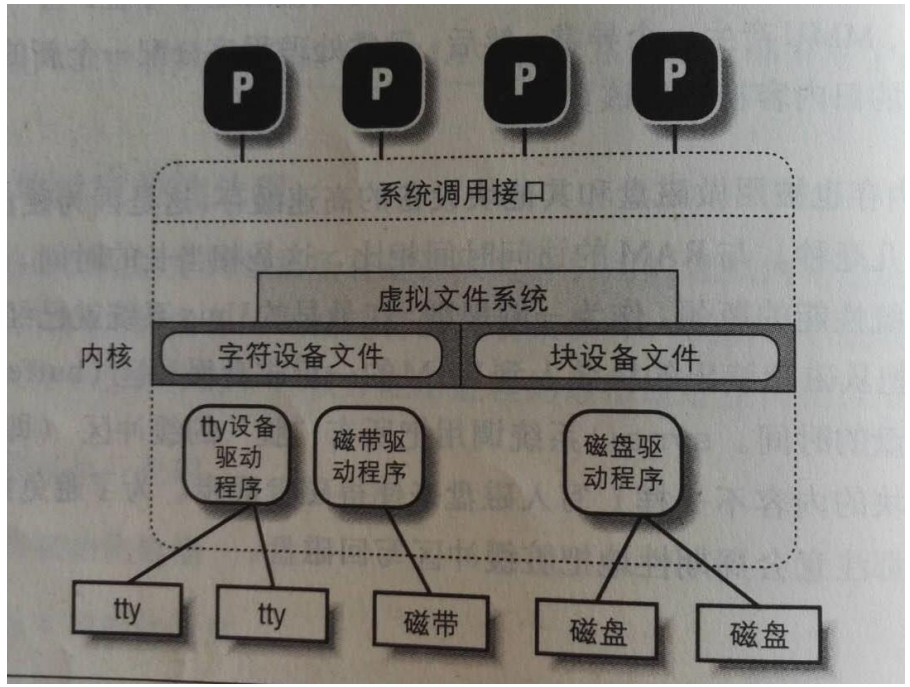
- 内存设置磁盘高速缓存保存最近访问过的磁盘数据。

- 读磁盘，数据取自磁盘高速缓存。 不命中，再访问磁盘。
  预读：为了提高IO效率，系统会将文件中包含数据的相邻8个数据块读入内存。∴ 读文件操作基本能在内存搞定。

- 写磁盘　数据入磁盘高速缓存，写操作就算结束。
  磁盘高速缓存中的数据，不会立即写盘。　**延迟写**策略。
  为了避免数据丢失，内核定期将脏缓存写入磁盘。
  ∴写文件结束后，数据在内存中，还没写盘。

- 磁盘IO，多数是写操作。应当针对写操作优化磁盘文件系统，Linux Ext3就是这样。

九、设备驱动

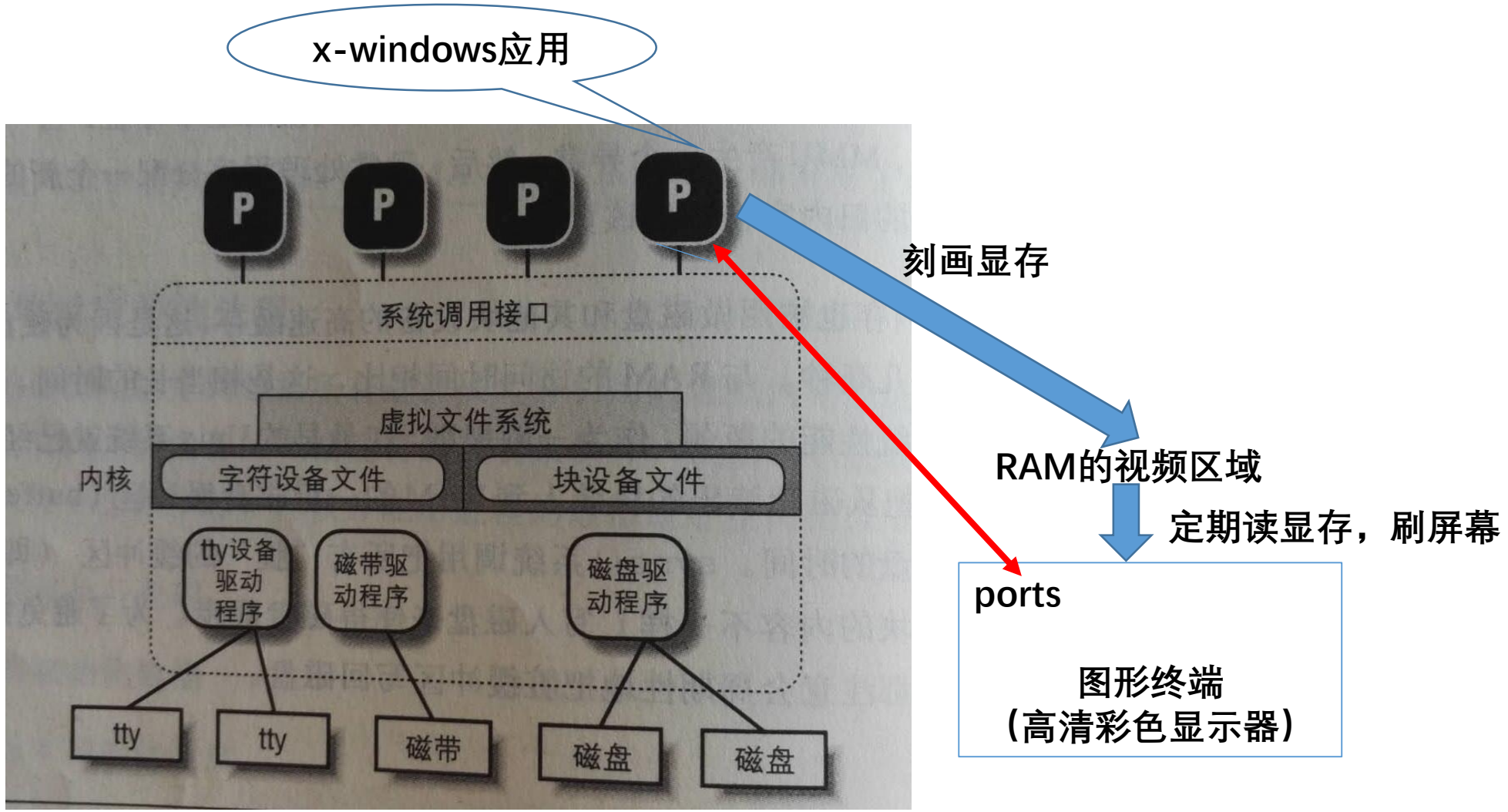


设备驱动是与外设交互的内核模块
含：若干控制外设的函数（应该有一个中断处理程序）
+ 数据结构

Linux用户如何使用设备?
/dev目录下有与之相对应的文件
读写这个文件（用VFS read、write系统调用）
具体而言：
VFS识别读写操作针对的外设，调用具体设备驱动程序实施数据传输。

实施 IO 的是内核。
应用不可以直接访问外设，需要执行系统调用。

- 设备驱动程序（device driver）：图形终端（GUI）



第二部分　　Linux/UNIX内核概述

- 用户进程
  - 有用户地址空间（[0,3G]装APP代码和数据）
  - 除守护进程外，APP执行完毕，进程终止。守护进程，启动后不再终止。
  - 除守护进程外， 其余进程有 tty。
    - tty上，用户按下ctrl+c，SIGINT信号将杀死所有前台用户进程。守护进程是杀不死的。
    - 默认，进程（APP）的标准输入输出文件是tty的键盘和屏幕。可以用输入、输出重定向技术改变进程的标准输入输出。

- 内核进程（线程）
  - 内核线程没有用户地址空间（因为它不执行APP，所以[0,3G]是空的）
  - 内核线程永不终止，在系统运行的时候，时刻提供系统维护服务
  - 内核线程没有tty（这样，任何tty的ctrl+c都杀不死它）

- 模式切换（mode switch）
  - 进程运行时，绝大多数时间跑在用户态。必需时，停止执行APP，陷入内核态运行，执行一些内核任务。 **用户态→核心态**
    - 执行系统调用
    - 异常
    - 被中断（时钟中断，外设IO中断）
  - 内核任务执行完毕后，进程返回用户态继续执行APP。 **核心态→用户态**

保证代码运行正确

- 可重入（reentrant）的子程序
  - 不访问全局变量的子程序
  - 你可以反复调用这些子程序，不管先前调用的它有没有return。
    也就是说，无论它们顺序执行还是交错执行，程序的运行结果都是正确的。

- 可重入的内核
  - 内核控制路径（kernel control path）

    内核为了处理 系统调用或中断 必须执行的指令序列 + 内核线程~
  - 什么时候会出现多条内核控制路径并发？
    - 内核运行在开中断的环境下，CPU响应中断
    - 执行系统调用，内核发现暂时无法满足 进程 继续运行的要求。现运行进程就会停下来，放弃CPU。其它内核控制路径就会开始跑~
    - 现运行进程异常，比如要从磁盘上调缺页~睡眠放弃CPU。~
    - 多核系统中，每个核都可能执行一条内核控制路径
    - 内核线程~
  - 可重入的内核是指

    多条内核控制路径同时运行，系统也不会出错的内核

- 如何构造可重入的内核
  - 内核可重入
    - 写可重入函数
    - 不可重入的函数　用锁保护共享变量（内核数据结构）
      - 共享变量访问的时间长　　用信号量
      - ~　多核系统用自旋锁
  - 关中断，保护单CPU变量
  - 不可剥夺的内核控制路径
    - 这个内核控制路径涉及非常容易出错的内核全局变量。现运行进程不放弃CPU+其它CPU也不可以跑内核~
    - 如果放弃CPU，下次运行需要检查先前访问过的数据结构的值。

- 死锁（deadlock）
  - 避免死锁的方式：破坏循环等待条件

    有限的内核信号量 + 内核按编号递增的顺序请求信号量