

# APUE - 1

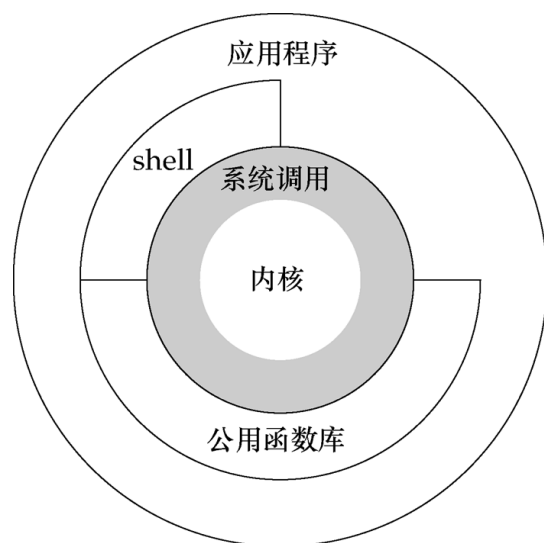
阅读目的：熟悉 Unix 基础知识，了解 Unix/Linux系统底层实现原理

阅读时间：预计 40 天

阅读概况：详读第 1 章、略读第 2 章

## 第 1 章 UNIX 基础知识

### 1. UNIX 体系结构



Unix/Linux 操作系统的体系架构分为用户态和内核态（或者用户空间和内核）。内核从本质上看是一种软件——控制计算机的硬件资源，并提供上层应用程序运行的环境。用户态即上层应用程序的活动空间，应用程序的执行必须依托于内核提供的资源，包括 CPU 资源、存储资源、I/O 资源等。

#### 内核态与用户态

**内核态:** CPU 可以访问内存所有数据, 包括外围设备, 例如硬盘, 网卡。CPU 也可以将自己从一个程序切换到另一个程序。

**用户态:** 只能受限的访问内存, 且不允许访问外围设备。 占用 CPU 的能力被剥夺, CPU 资源可以被其他程序获取。

#### 为什么要有用户态和内核态？

由于需要限制不同的程序之间的访问能力, 防止他们获取别的程序的内存数据, 或者获取外围设备的数据, 并发送到网络, CPU 划分出两个权限等级 -- **用户态** 和 **内核态**。

用户态的应用程序可以通过三种方式来访问内核态的资源：

- 系统调用
- 库函数
- Shell脚本

### 2. 文件和目录

Unix 文件系统是目录和文件的一种层次结构，所有东西的起点成为 **根（root）** 的目录。

**目录（directory）** 是一个包含目录项的文件。在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该 **文件属性** 的信息。

**文件属性** 是指文件类型（普通文件还是目录等）、文件大小、文件所有者、文件权限以及文件最后的修改时间等。

目录中的各个名字称为 **文件名（filename）**。POSIX.1推荐将文件名限制在以下字符集之内：字母（a~z、A~Z）、数字（0~9）、句点（.）、短横线（-）和下划线（\_）。

### 3. 程序和进程

程序（program） 是一个存储在磁盘上某个目录中的可执行文件。

程序的执行实例被称为 进程（process）。

Unix 系统确保每个进程都有一个唯一的数字标识符，称为 进程ID（process ID）。进程 ID 总是一个非负整数。

一个进程内的所有线程共享同一地址空间、文件描述符、栈以及进程相关的属性。尽管任何线程都可以在同一进程中访问其他线程的堆栈，但每个线程都在自己的堆栈上执行。

### 4. 信号

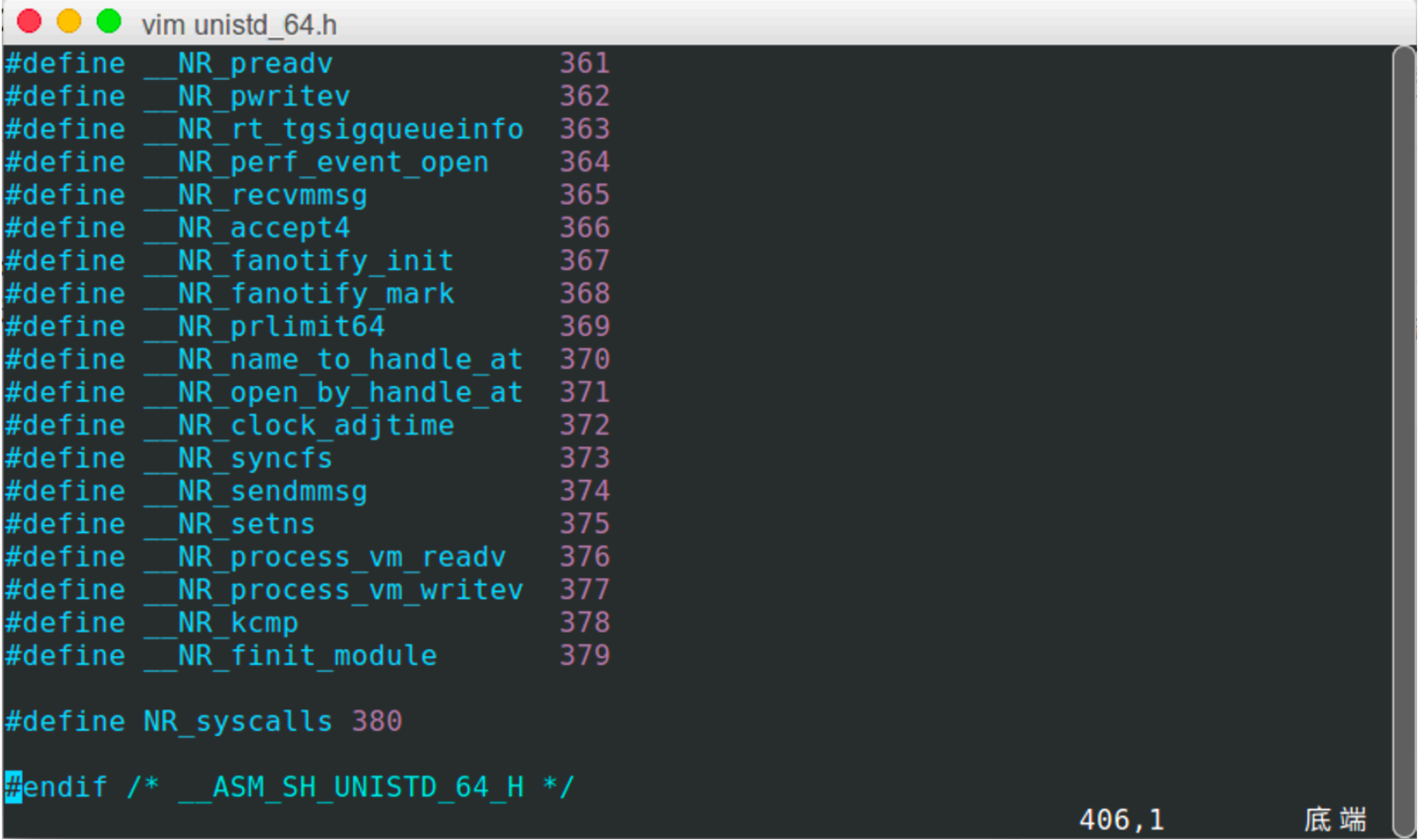
信号（signal）用于通知进程发生了某种情况。进程有以下三种处理信号的方式：

- 忽略信号。对该信号不做任何处理，就象未发生过一样。
- 提供一个函数。信号发生时调用该函数，这被称为捕捉该信号。进程可以指定处理函数，由该函数来处理。
- 按系统默认方式处理。这种缺省操作，对大部分的信号的缺省操作是使得进程终止。进程通过系统调用 signal 来指定进程对某个信号的处理行为。

### 5. 系统调用

系统调用是操作系统的最小功能单位，这些系统调用根据不同的应用场景可以进行扩展和裁剪，现在各种版本的Unix实现都提供了不同数量的系统调用，如Linux的不同版本提供了240-260个系统调用，FreeBSD大约提供了320个。

在 linux-source-4.4.0 内核版本中系统调用数量为379个。



```
vim unistd_64.h
#define __NR_preadv 361
#define __NR_pwritev 362
#define __NR_rt_tgsigqueueinfo 363
#define __NR_perf_event_open 364
#define __NR_recvmmsg 365
#define __NR_accept4 366
#define __NR_fanotify_init 367
#define __NR_fanotify_mark 368
#define __NR_prlimit64 369
#define __NR_name_to_handle_at 370
#define __NR_open_by_handle_at 371
#define __NR_clock_adjtime 372
#define __NR_syncfs 373
#define __NR_sendmmsg 374
#define __NR_setns 375
#define __NR_process_vm_readv 376
#define __NR_process_vm_writev 377
#define __NR_kcmp 378
#define __NR_finit_module 379

#define NR_syscalls 380

#endif /* __ASM_SH_UNISTD_64_H */
406,1 底端
```

路径：/usr/src/linux-source-4.4.0/linux-source-4.4.0/arch/sh/include/uapi/asm

系统调用的本质其实是中断，相对于外围设备的硬件中断，这种中断称为软件中断，x86 系统上的软件中断由 int \$0x80 指令产生。

Linux 中每个系统调用都有相应的系统调用号作为唯一的标识，内核维护一张系统调用表，**sys\_call\_table**，表中的元素是系统调用函数的起始地址，而系统调用号就是系统调用在调用表的偏移量。在x86上，系统调用号是通过 eax 寄存器传递给内核的。（reference：Linux 内核设计与实现）

## 1. shell/库函数/syscall三者之间是什么关系？

- 运行环境：shell && 库函数 运行在用户态 系统调用运行在内核态。
- 对应关系：一个库函数可以调用多个 syscall， 一个 syscall 可被多个 库函数/shell 调用。
- 各自作用：
  - syscall 的作用是屏蔽系统底层硬件，是内核提供给应用程序的接口，是面向底层硬件的。eg：read
  - 库函数是将经常用到的函数放到一文件中，是提供一个方法，是面向应用开发的。eg：printf
  - shell是一个特殊的应用程序，它下通系统调用，上通各种应用，通常充当着一种“胶水”的角色，来连接各个小功能程序，让不同程序能够以一个清晰的接口协同工作，从而增强各个程序的功能。

## 2. 同一个进程内的线程是如何共享栈的？

中文翻译：一个进程内的所有线程共享同一地址空间、文件描述符、栈以及与进程相关的属性。因为它们能访问同一存储区，所以各线程在访问共享数据时需要采取同步措施以避免不一致性。

英文原文：All threads within a process share the same address space, file descriptors, stacks, and process-related attributes. **Each thread executes on its own stack, although any thread can access the stacks of other threads in the same process.** Because they can access the same memory, the threads need to synchronize access to shared data among themselves to avoid inconsistencies.

怪自己偷懒看的中文版。。。。翻译直接删除后面那句话。。。。

栈存储临时变量，以及每次调用函数时保存的信息。

一个线程真正拥有的唯一私有储存是处理器寄存器。因为共享同一地址空间，线程栈可以通过暴露栈地址的方式与其它线程进行共享。

```

630
631 int
632 __pthread_create_2_1 (pthread_t *newthread, const pthread_attr_t *attr,
633                      void *(*start_routine) (void *), void *arg)
634 {
635     STACK_VARIABLES;
636
637     const struct pthread_attr *iattr = (struct pthread_attr *) attr;
638     struct pthread_attr default_attr;
639     bool free_cpuset = false;
640     bool c11 = (attr == ATTR_C11_THREAD);
641     if (iattr == NULL || c11)
642     {
643         lll_lock (__default_pthread_attr_lock, LLL_PRIVATE);
644         default_attr = __default_pthread_attr;
645         size_t cpusetsize = default_attr.cpusetsize;
646         if (cpusetsize > 0)
647         {
648             cpu_set_t *cpuset;
649             if (__glibc_likely (__libc_use_alloca (cpusetsize)))
650                 cpuset = __alloca (cpusetsize);
651             else
652             {
653                 cpuset = malloc (cpusetsize);
654                 if (cpuset == NULL)
655                 {
656                     lll_unlock (__default_pthread_attr_lock, LLL_PRIVATE);
657                     return ENOMEM;
658                 }
659                 free_cpuset = true;
660             }
661             memcpy (cpuset, default_attr.cpuset, cpusetsize);
662             default_attr.cpuset = cpuset;
663         }
664         lll_unlock (__default_pthread_attr_lock, LLL_PRIVATE);
665         iattr = &default_attr;
666     }
667
668     struct pthread *pd = NULL;
669     int err = ALLOCATE_STACK (iattr, &pd);
670     int retval = 0;
671
672     if (__glibc_unlikely (err != 0))
673     /* Something went wrong. Maybe a parameter of the attributes is
674        invalid or we could not allocate memory. Note we have to
675        translate error codes. */
676     {
677         retval = err == ENOMEM ? EAGAIN : err;
678         goto out;
679     }
680
681
682     /* Initialize the TCB. All initializations with zero should be
683        performed in 'get_cached_stack'. This way we avoid doing this if
684        the stack freshly allocated with 'mmap'. */
685
686     #if TLS_TCB_AT_TP
687     /* Reference to the TCB itself. */
688     pd->header.self = pd;
689
690     /* Self-reference for TLS. */
691     pd->header.tcb = pd;
692     #endif
693

```

路径: glibc-2.28/nptl/pthread\_create.c

在 glibc 文件中可见调用了 ALLOCATE\_STACK 函数，在创建线程是新开辟出栈空间的。

### 3. 中断是一个什么样的机制？

中断是硬件与处理器进行通信的方式。

操作系统需要管理硬件设备，所以需要进行通信。

硬件的速度较 CPU 相比是很慢的，如果内核采取让处理器向硬件发出一个请求，然后专门等待回应（轮询）的办法效率会很低，终端是将主动变被动（相对内核来说），让硬件在需要的时候再向内核发出信号。