# linux系统编程-应用编程常识--专题讲座

## 计算机系统组成

### 1计算机系统硬件组成

|  |
| --- |
| * 计算机系统由软件硬件构成 |
| 总线   * 贯穿整个系统的一组电子管道称为总线 * 片内总线 * 系统总线   + 数据总线DB   + 地址总线CB   + 控制总线CB * 外部总线 |
| IO设备   * I/O设备是系统与外界联系的通道 * 键盘鼠标是输入设备 * 显式器是输出设备 * 磁盘既是输入设备也是输出设备 * 输入输出是相对于内存来说的。 |
| 内存   * 内存是一个重要的部件，它是与CPU进行沟通的桥梁。它用来存放程序以及程序要处理的数据，磁盘中的程序要加载到内存才能运行。   处理器 （运算器+控制器）   * 中央处理器（CPU），简称处理器。 * CPU主要有运算器、控制器、寄存器构成 * 取指 * 译码 * 执行 * 写回 * 跳转 |
| 计算机系统硬件组成 |
| 总结：  1、cpu操作内存步骤 通过地址总线写地址、写控制、操作内存  2、硬件角度看hello程序 |
| DMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 DMA 控制器 来实行和完成。典型的例子就是移动一个外存的区块到芯片内部更快的内存区。像是这样的操作并没有让处理器工作拖延，反而可以被重新安排去处理其他的工作。在实现DMA传输时，是由DMA控制器直接掌管总线，因此，存在着一个总线控制权转移问题。 |

### 2操作系统

|  |
| --- |
| 什么是操作系统   * 操作系统：有效地管理计算机系统中的资源，合理地管理计算机系统的工作流程，方便用户使用的程序的集合。 * 资源   + 软件资源   + 硬件资源 * 操作系统五大任务   + 文件管理   + 处理器管理 cpu 96   + 内存管理   + 设备管理   + 作业管理 |
| 操作系统三个基本抽象   * 操作系统通过三个基本抽象概念来实现这两个功能。（进程、虚拟存储器和文件）。文件是对I/O设备的抽象表示，虚拟存储器是对主存和磁盘I/O设备的抽象表示，进程则是对处理器、主存和I/O设备的抽象表示。 |
| 进程：   * 进程是操作系统对正在运行的程序的一种抽象。一个系统可以运行多个进程，而每个进程好像在独占使用硬件。 |
| 进程上下文切换 |
| **虚拟存储器**   * 虚拟存储器为每个进程提供了一个大的、一致的、私有的地址空间 * 它将内存看成是存储在磁盘上的地址空间的高速缓存，在主存中只保存活动区域，并根据需要在主存与磁盘中来回交换数据。 * 为每个进程提供一致的地址空间，简化了存储管理 * 保护每个进程的地址空间不被其他进程破坏。 |
| **进程虚地址空间** |
| **文件**   * **文件是一系列的字节序列** * **它向应用程序提供了一个统一的视角，来看待系统中各式各样的I/O设备。** |
| **虚拟文件系统VFS**   * **虚拟文件系统是内核实现的一种架构，为用户空间提供统一的文件操作接口，即文件系统调用。它在内核内部为不同的真实文件系统提供一致的抽象接口。** * **用户通过系统用与内核中的虚拟文件系统交互，进而操作实际的文件系统和设备。** |
|  |

### 3linux内核

|  |
| --- |
|  |
|  |
|  |
| 常识   Linux是最受欢迎的自由电脑[操作系统](http://baike.baidu.com/view/880.htm)内核。它是一个用C语言写成，符合POSIX标准的类Unix操作系统。Linux最早是由[芬兰](http://baike.baidu.com/view/20509.htm)[黑客](http://baike.baidu.com/view/1960.htm) Linus Torvalds为尝试在英特尔x86架构上提供自由免费的类Unix操作系统而开发的。该计划开始于1991年，在计划的早期有一些Minix [黑客](http://baike.baidu.com/view/1960.htm)提供了协助，而今天全球无数程序员正在为该计划无偿提供帮助。 技术上说Linux是一个内核。“内核”指的是一个提供[硬件抽象层](http://baike.baidu.com/view/500774.htm)、磁盘及文件系统控制、多任务等功能的[系统软件](http://baike.baidu.com/view/7860.htm)。一个内核不是一套完整的操作系统。一套基于Linux内核的完整操作系统叫作Linux操作系统，或是GNU/Linux。 |
| [Linux](http://baike.baidu.com/view/1634.htm)是一个一体化[内核](http://baike.baidu.com/view/1366.htm)（monolithic kernel）系统。[设备驱动程序](http://baike.baidu.com/view/15565.htm)可以完全访问硬件。Linux内的[设备驱动程序](http://baike.baidu.com/view/15565.htm)可以方便地以[模块化](http://baike.baidu.com/view/182267.htm)（modularize）的形式设置，并在系统运行期间可直接装载或[卸载](http://baike.baidu.com/view/386432.htm)。 |
| 架构  [Linux](http://baike.baidu.com/view/1634.htm)是一个一体化[内核](http://baike.baidu.com/view/1366.htm)（monolithic kernel）系统。[设备驱动程序](http://baike.baidu.com/view/15565.htm)可以完全访问硬件。Linux内的[设备驱动程序](http://baike.baidu.com/view/15565.htm)可以方便地以[模块化](http://baike.baidu.com/view/182267.htm)（modularize）的形式设置，并在系统运行期间可直接装载或[卸载](http://baike.baidu.com/view/386432.htm) 体系结构属性 在讨论大型而复杂的系统的[体系结构](http://baike.baidu.com/view/1188494.htm)时，可以从很多角度来审视系统。[体系结构](http://baike.baidu.com/view/1188494.htm)分析的一个目标是提供一种方法更好地理解[源代码](http://baike.baidu.com/view/60376.htm)。  Linux 内核实现了很多重要的[体系结构](http://baike.baidu.com/view/1188494.htm)属性。在或高或低的层次上，内核被划分为多个子系统。Linux 也可以看作是一个整体，因为它会将所有这些基本服务都集成到内核中。这与[微内核](http://baike.baidu.com/view/928365.htm)的[体系结构](http://baike.baidu.com/view/1188494.htm)不同，后者会提供一些基本的服务，例如[通信](http://baike.baidu.com/view/15007.htm)、[I/O](http://baike.baidu.com/view/300881.htm)、[内存](http://baike.baidu.com/view/1082.htm)和[进程管理](http://baike.baidu.com/view/364947.htm)，更具体的服务都是插入到微内核层中的。  随着时间的流逝，Linux 内核在[内存](http://baike.baidu.com/view/1082.htm)和 CPU 使用方面具有较高的效率，并且非常稳定。但是对于 Linux 来说，最为有趣的是在这种大小和复杂性的前提下，依然具有良好的[可移植性](http://baike.baidu.com/view/1936417.htm)。Linux 编译后可在大量处理器和具有不同[体系结构](http://baike.baidu.com/view/1188494.htm)约束和需求的平台上运行。一个例子是 Linux 可以在一个具有[内存管理](http://baike.baidu.com/view/4541016.htm)单元（MMU）的处理器上运行，也可以在那些不提供[MMU](http://baike.baidu.com/view/969924.htm)的处理器上运行。Linux 内核的[uClinux](http://baike.baidu.com/view/163694.htm)移植提供了对非 MMU 的支持 开发和规范 核心的开发和规范一直是由Linux社区控制着，版本也是唯一的。实际上，操作系统的[内核版本](http://baike.baidu.com/view/6744074.htm)指的是在Linux本人领导下的开发小组开发出的系统内核的版本号。自1994年3月14日发布了第一个正式版本Linux 1.0以来，每隔一段时间就有新的版本或其修订版公布。  Linux将标准的GNU许可协议改称Copyleft，以便与Copyright相对照。通用的公共许可（GPL）允许用户销售、拷贝和改变具有Copyleft的[应用程序](http://baike.baidu.com/view/330120.htm)。当然这些程序也可以是Copyright的，但是你必须允许进一步的销售、拷贝和对其代码进行改变，同时也必须使他人可以免费得到修改后的[源代码](http://baike.baidu.com/view/60376.htm)。事实证明，GPL对于Linux的成功起到了极大的作用。它启动了一个十分繁荣的商用Linux阶段，还为[编程](http://baike.baidu.com/view/3281.htm)人员提供了一种凝聚力，诱使大家加入这个充满了慈善精神的Linux运动。 主要子系统 **系统调用接口**：SCI 层提供了某些机制执行从[用户空间](http://baike.baidu.com/view/4274331.htm)到内核的[函数调用](http://baike.baidu.com/view/2369016.htm)。正如前面讨论的一样，这个接口依赖于[体系结构](http://baike.baidu.com/view/1188494.htm)，甚至在相同的处理器家族内也是如此。SCI 实际上是一个非常有用的[函数调用](http://baike.baidu.com/view/2369016.htm)多路复用和多路分解服务。在 ./[linux](http://baike.baidu.com/view/1634.htm)/kernel 中您可以找到 SCI 的实现，并在 ./[linux](http://baike.baidu.com/view/1634.htm)/arch 中找到依赖于[体系结构](http://baike.baidu.com/view/1188494.htm)的部分。  [进程管理](http://baike.baidu.com/view/364947.htm)：进程管理的重点是进程的执行。在内核中，这些进程称为线程，代表了单独的处理器虚拟化（线程代码、数据、[堆栈](http://baike.baidu.com/view/93201.htm)和 CPU[寄存器](http://baike.baidu.com/view/6159.htm)）。在[用户空间](http://baike.baidu.com/view/4274331.htm)，通常使用进程 这个术语，不过 Linux 实现并没有区分这两个概念（进程和线程）。内核通过 SCI 提供了一个[应用程序编程接口](http://baike.baidu.com/view/185287.htm)（[API](http://baike.baidu.com/view/16068.htm)）来创建一个新进程（fork、exec 或 Portable Operating System Interface [POSⅨ] 函数），停止进程（kill、exit），并在它们之间进行[通信](http://baike.baidu.com/view/15007.htm)和同步（signal 或者 POSⅨ 机制）。  [进程管理](http://baike.baidu.com/view/364947.htm)还包括处理活动进程之间共享 CPU 的需求。内核实现了一种新型的[调度算法](http://baike.baidu.com/view/2963962.htm)， 不管有多少个线程在竞争 CPU，这种算法都可以在固定时间内进行操作。这种算法就称为 O⑴ 调度程序，这个名字就表示它调度多个线程所使用的时间和调度一个线程所使用的时间是相同的。O⑴ 调度程序也可以支持多处理器（称为对称多处理器或 SMP）。您可以在 ./linux/kernel 中找到[进程管理](http://baike.baidu.com/view/364947.htm)的[源代码](http://baike.baidu.com/view/60376.htm)，在 ./linux/arch 中可以找到依赖于[体系结构](http://baike.baidu.com/view/1188494.htm)的源代码。  [内存管理](http://baike.baidu.com/view/4541016.htm)：内核所管理的另外一个重要资源是内存。为了提高效率，如果由硬  [VFS 在用户和文件系统之间提供了一个交换层](http://baike.baidu.com/picture/573460/573460/0/34bbf8cddce9b61c0fb3458a?fr=lemma&ct=single)  VFS 在用户和文件系统之间提供了一个交换层  管理[虚拟内存](http://baike.baidu.com/view/976.htm)，内存是按照所谓的内存页 方式进行管理的（对于大部分[体系结构](http://baike.baidu.com/view/1188494.htm)来说都是 4KB）。Linux 包括了管理可用[内存](http://baike.baidu.com/view/1082.htm)的方式，以及物理和虚拟映射所使用的硬件机制。  不过[内存管理](http://baike.baidu.com/view/4541016.htm)要管理的可不止 4KB[缓冲区](http://baike.baidu.com/view/266782.htm)。Linux 提供了对 4KB[缓冲区](http://baike.baidu.com/view/266782.htm)的抽象，例如 slab 分配器。这种[内存管理](http://baike.baidu.com/view/4541016.htm)模式使用 4KB[缓冲区](http://baike.baidu.com/view/266782.htm)为基数，然后从中分配结构，并跟踪内存页使用情况，[比如](http://baike.baidu.com/view/6814120.htm)哪些内存页是满的，哪些页面没有完全使用，哪些页面为空。这样就允许该模式根据系统需要来动态调整[内存](http://baike.baidu.com/view/1082.htm)使用。  为了支持多个用户使用[内存](http://baike.baidu.com/view/1082.htm)，有时会出现可用内存被消耗光的情况。由于这个原因，页面可以移出[内存](http://baike.baidu.com/view/1082.htm)并放入磁盘中。这个过程称为交换，因为页面会被从[内存](http://baike.baidu.com/view/1082.htm)交换到硬盘上。[内存管理](http://baike.baidu.com/view/4541016.htm)的[源代码](http://baike.baidu.com/view/60376.htm)可以在 ./[linux](http://baike.baidu.com/view/1634.htm)/mm 中找到。  [虚拟文件系统](http://baike.baidu.com/view/671797.htm)：虚拟文件系统（VFS）是 Linux 内核中非常有用的一个方面，因为它为文件系统提供了一个通用的接口抽象。VFS 在 SCI 和内核所支持的文件系统之间提供了一个交换层。  VFS 在用户和文件系统之间提供了一个交换层  在 VFS 上面，是对诸如 open、close、read 和 write 之类的函数的一个通用 API 抽象。在 VFS 下面是文件系统抽象，它定义了上层函数的实现方式。它们是给定文件系统（超过 50 个）的插件。文件系统的[源代码](http://baike.baidu.com/view/60376.htm)可以在 ./[linux](http://baike.baidu.com/view/1634.htm)/fs 中找到。  文件系统层之下是[缓冲区](http://baike.baidu.com/view/266782.htm)缓存，它为文件系统层提供了一个通用函数集（与具体文件系统无关）。这个缓存层通过将数据保留一段时间（或者随即预先读取数据以便在需要是就可用）优化了对[物理设备](http://baike.baidu.com/view/5059658.htm)的访问。[缓冲区](http://baike.baidu.com/view/266782.htm)缓存之下是[设备驱动程序](http://baike.baidu.com/view/15565.htm)，它实现了特定[物理设备](http://baike.baidu.com/view/5059658.htm)的接口。 特性 如果 Linux 内核的[可移植性](http://baike.baidu.com/view/1936417.htm)和效率还不够好，Linux 还提供了其他一些特性，它们无法划分到上面的分类中。  作为一个生产操作系统和[开源软件](http://baike.baidu.com/view/444964.htm)，Linux 是测试新协议及其增强的良好平台。Linux 支持大量[网络协议](http://baike.baidu.com/view/16603.htm)，包括典型的 TCP/IP，以及高速网络的扩展（大于 1 Gigabit Ethernet [GbE] 和 10 GbE）。Linux 也可以支持诸如[流控制传输协议](http://baike.baidu.com/view/1333903.htm)（SCTP）之类的协议，它提供了很多比 TCP 更高级的特性（是传输层协议的接替者）。  Linux 还是一个动态内核，支持动态添加或删除[软件组件](http://baike.baidu.com/view/551014.htm)。被称为动态可加载内核模块，它们可以在引导时根据需要（当前特定设备需要这个模块）或在任何时候由用户插入。  Linux 最新的一个增强是可以用作其他操作系统的操作系统（称为[系统管理](http://baike.baidu.com/view/635537.htm)程序）。该系统对内核进行了修改，称为基于内核的[虚拟机](http://baike.baidu.com/view/1132.htm)（KVM）。这个修改为[用户空间](http://baike.baidu.com/view/4274331.htm)启用了一个新的接口，它可以允许其他操作系统在启用了 KVM 的内核之上运行。除了运行 Linux 的其他实例之外， Microsoft&reg; Windows&reg; 也可以进行虚拟化。惟一的限制是底层处理器必须支持新的虚拟化指令[4] 。 |

## 系统编程和应用编程

|  |
| --- |
| * 系统编程   在操作系统之上利用系统调用、C库进行对系统资源进行访问。如apache 、gcc、gdb 等   * 应用编程   在更高层次的编程接口或者库之上构建应用程序。如android程序（android sdk）、iphone程序（iphone sdk）、QT程序设计（QT）MFC程序设计（MFC）等。 |
| **系统调用在系统中所处的位置**   * **所有操作系统都提供多种服务的入口点，由此程序向系统核请求服务。这些入口点被称之为系统调用(system call)，** |
| **C库**   * **这里我们所说的C库（libc），指的是标准C定义的Ｃ函数的集合。如标准输入输出函数、字符串处理函数、动态存储分配函数、日期时间函数、数学函数等。** * **GNU发布的libc称为glibc** |
| **系统调用与C库关系**   * 系统调用与Ｃ库从形式上来看都C函数 * C库函数有些是调用系统调用来实现的，比如说malloc、free调用brk，printf调用write系统用，有些函数不需要任何系统调用，比如abs、strcpy、atoi等，因为它并不是必需要使用内核服务 * 系统调用通常提供的是最小界面，而Ｃ库函数通常提供更复杂的功能。 |
| **内核如何处理系统调用**   * **每个系统调用被赋予一个系统调用号** * **在i386平台上，执行一个系统调用是通过int 0x80指令完成的。** * **eax存放系统调用号** * **ebx、ecx、edx、esi、edi存储系统调用参数，对于超过5个参数的系统调用，用一个寄存器指向用户空间存储所有系统调用参数的缓存。** |
| **错误处理**   * **在系统编程中错误通常通过函数返回值来表示，并通过特殊变量errno来描述。** * **errno这个全局变量在<errno.h>头文件中声明如下：extern int errno;** * **错误处理函数**   + **perror**   + **strerror** |

linux应用编程市场需求

|  |
| --- |
| 广州博庭诚招金融软件开发工程师    于2004年成立的广州博庭计算机科技有限公司位于广州天河广纺联创意园，是专注于证券金融软件与服务的技术公司。多年来，博庭科技在业界拥有优秀口碑。广州团队的产品**《飞狐交易师》**软件是众多技术分析发烧者心目中的标杆，至今拥有大量忠实用户。在台湾长期坐拥市占率第一。    博庭科技在量化投资交易领域尤其拥有强大实力。量化交易是指利用计算机结合一定的数据模型实现投资理念，执行投资策略的过程。量化交易在国际市场占交易量70%，是国际投资的主流先进方法，而在国内也是最热门的新兴技术，是未来的专业投资主流趋势。前景无可限量。    博 庭拥有广州和上海两个团队。博庭科技既开发面向大众的金融软件，也为高端客户订制交易软件。正在研发的《金魔方》软件在继承飞狐交易师强大功能的基础上， 融合先进的量化分析研发和交易功能，致力于成为中国期货、期权、股票、和现货市场领先的量化交易软件。上海团队拥有证券柜台系统/账户管理系统、风控系 统、高频交易系统、各大证券期货交易接口系统、手机客户端。    我们可以为客户提供从后台到前台的一个完整的交易服务全产品链。我们的客户包括各类交易所、券商、期货商、基金、证券投资人。    另外，广州团队和上海团队各自运营两个不同风格的量化的私募基金，均拥有骄人的投资业绩。    本次招聘员工主要工作于广州。同时，根据本人意愿和技能特点也可能在上海甚至海外工作。    我们提供的福利和待遇：薪资6000-2万。社保和公积金。带薪年假。文体活动。学习机会。以及私募基金投资收益等。    **招聘要求：**    招聘职位：软件开发工程师    工作职责：    1.负责证券量化分析和交易软件的客户端及服务器模块设计、编码、调试、测试等工作。  2.参与相关质量活动，确保设计、实现、测试工作按时保质完成。    职位要求：  1、对金融和证券投资拥有浓厚兴趣。平时关心相关知识。  2、计算机、通信、软件工程、自动化、数学或相关专业，本科及以上学历。  3、精通C/C++语言/JAVA软件编程，熟悉TCP/IP协议、Internet网络的基本知识、熟悉STL。  4、熟练使用windows或Linux开发、编译、调试环境，如VC、。  5、能够熟练阅读和理解英文资料。  6、深刻理解Windows/Linux/Unix内存管理机制、进程/线程、各种进程间通信方式、消息事件通知机制和异步机制者加分。  7、精通编译原理者加分。 |
| 初级linux开发工程师   |  | | --- | | **职位职能:**  软件工程师 | | **职位描述:**  岗位职责： 1）项目任务执行 l 完成开发经理下达的项目任务  l 积极配合开发经理，保证项目的质量和进度  2）技术任务执行 l 完成技术主管下达的技术研究任务  l 积极与同组员工分享技术知识；  l协助同事解决技术问题  3）领导交办的其他事项  任职资格： 1) 掌握c、c++语言基础和特性。  2) 掌握计算机相关技术基础，包括操作系统、数据结构、汇编语言、组成原理、计算机网络、编译原理、设计模式等。  3) 熟练应用gcc/g++/gdb等开发调试工具。  4) 熟悉shell脚本，能进行makefile编写。  5) 熟练使用常用的数据库sqlite、mysql、oracle等。  6) 熟练使用常用的网络工具。 |   [**北京鼎普科技股份有限公司**](http://search.51job.com/list/co,c,1284425,000000,10,1.html) |
| |  | | --- | |  | | **职位职能:**  高级软件工程师  软件工程师 | | **职位描述:**  岗位职责：  1.Linux平台下服务软件开发； 2.优化算法和性能； 3.脚本的自动化配置部署；   任职资格：  1.熟悉Linux，2年以上的开发经验； 2.精通C或Python或Perl开发，2年以上开发经验； 3.具备良好的分析解决问题能力； 4.学习能力强，善于沟通，富有团队合作精神，能承受一定的工作压力； 5.有大并发高性能服务器开发经验优先； |   [**北京中软冠群软件技术有限公司**](http://search.51job.com/list/co,c,235896,000000,10,1.html) |
|  |
| |  | | --- | |  | | **职位职能:**  高级软件工程师 | | **职位描述:**  岗位职责： 1.负责Wifi networking开发- Linux kernel/module, system programming。  任职资格： 1.3年以上软件开发相关经验, 熟悉Embedded system&RTOS； 2.熟悉C/C++ 开发； 3.具备Network Programming 经验； 4.熟悉linux内核开发。 |   高级linux开发工程师 [**360安全卫士**](http://search.51job.com/list/co,c,804886,000000,10,1.html) |

## 学到什么程度可以找工作

这个问题太难回答？

你要找什么样的工作

**linux内核开发**

linux内核常用开发模型

字符设备开发 块设备开发 网络设备开发usb设备开发。。。

驱动总线模型

linux内核 并发机制。。。。（中断、自旋锁、队列（任务））

linux内核内存管理。。。。。。。。。。。。。。。。。

linux内核移植

bootload /内核移植。。。。和硬件 绑定 + 2410 2440 6410 mini

linux应用开发

1多进程多线程的客户端 并发服务器。。。高效率的处理业务。。。。

多进程多线程机制

进程间锁机制。。。。。。线程锁 （同步和互斥 信号量的两种用法）

进程间通讯（共享内存、消息队列、socket、）

=======》死的知识点。。。

2、报文的打包机制

自定义报文

json xml der

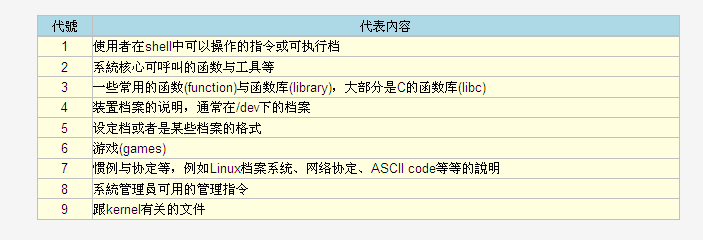
3、TCP/IP的深入编程

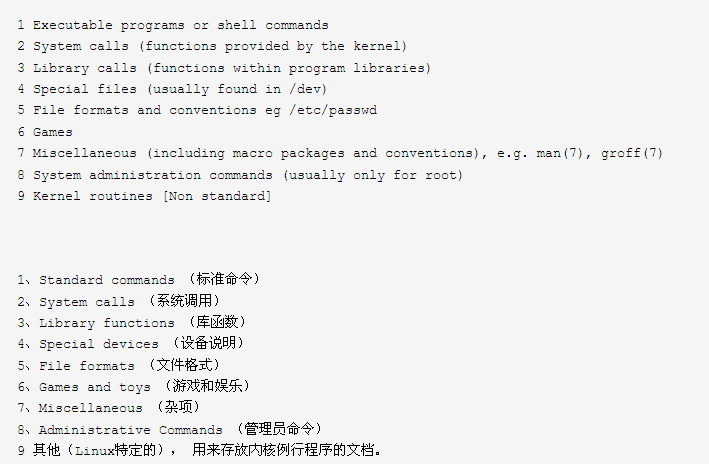
tcpip写入深入分析 （粘包无边界、连接数、5种服务器模型）

4、=======》商业代码级别的服务框架。。。

===============》

## man手册的使用





# linux进程-应用编程-专题讲座

## 进程基本概念

### 1程序pk进程

|  |
| --- |
| 什么是程序  程序是完成特定任务的一系列指令集合。 |
| 什么是进程  从用户的角度来看进程是程序的一次执行过程  从操作系统的核心来看，进程是操作系统分配的内存、CPU时间片等资源的基本单位。  进程是资源分配的最小单位  每一个进程都有自己独立的地址空间与执行状态。  像UNIX这样的多任务操作系统能够让许多程序同时运行，每一个运行着的程序就构成了一个进程 |
| 进程数据结构 |
| 进程的静态描述：由三部分组成:PCB、有关程序段和该程序段对其进行操作的数据结构集。  进程控制块：用于描述进程情况及控制进程运行所需的全部信息。  代码段：是进程中能被进程调度程序在CPU上执行的程序代码段。  数据段：一个进程的数据段，可以是进程对应的程序加工处理的原始数据，也可以是程序执行后产生的中间或最终数据 |
| 进程和程序的区别  进程是动态的，程序是静态的  进程的生命周期是相对短暂的，而程序是永久的。  进程数据结构PCB  一个进程只能对应一个程序，一个程序可以对应多个进程。 |
| 总结：  中断  早期是没有进程这个概念，当出现中断技术以后才出现进程这个概念  分式操作系统  基于时间片轮转  进程是操作系统对资源的一种抽象，一个进程：代码段、数据段、堆栈段、+进程控制块（PCB）  PCB是操作系统感知进程存在的一个重要数据结构（cpu通过进程控制块来控制进程） |

### 2进程状态

#### 进程状态

|  |
| --- |
| 操作系统经典三态   * 就绪 * 等待（阻塞） * 运行 |
| 总结状态条件和原因 |
| 因创建而就绪，因调度而执行；因时间片用完而重新就绪；  执行中因I/O请求而阻塞；  I/O完成而就绪  注意：阻塞以后不能直接执行，必须进入就绪状态。 |
| Linux内核中的进程状态   * 运行状态（TASK\_RUNNING） * 可中断睡眠状态（TASK\_INTERRUPTIBLE） * 不可中断睡眠状态（TASK\_UNINTERRUPTIBLE） * 暂停状态（TASK\_STOPPED） * 僵死状态（TASK\_ZOMBIE） |
|  |
| 备注：就绪态深入理解：内存中就绪和交换空间中就绪，操作系统支持虚拟内存。  虚拟内存实现需要操作系统支持：内存段式管理、段式管理、段页管理。 |
| 进程状态编程事件 |

#### 进程调度及调度算法

|  |
| --- |
|  |
|  |
|  |

### 3进程编程相关术语

|  |
| --- |
| 进程控制块 |
| * 进程描述信息   + 进程标识符用于唯一的标识一个进程。 * 进程控制信息   + 进程当前状态   + 进程优先级   + 程序开始地址   + 各种计时信息   + 通信信息 * 资源信息   + 占用内存大小及管理用数据结构指针   + 交换区相关信息   + I/O设备号、缓冲、设备相关的数结构   + 文件系统相关指针 * 现场保护信息   + 寄存器   + PC   + 程序状态字PSW   + 栈指针 |
| 进程标示 |
| * 每个进程都会分配到一个独一无二的数字编号，我们称之为“进程标识”(process identifier),或者就直接叫它PID. * 是一个正整数，取值范围从2到32768 * 当一个进程被启动时，它会顺序挑选下一个未使用的编号数字做为自己的PID * 数字1一般为特殊进程init保留的 |
| 进程创建 |
| * 不同的操作系统所提供的进程创建原语的名称和格式不尽相同，但执行创建进程原语后，操作系统所做的工作却大致相同，都包括以下几点： * 给新创建的进程分配一个内部标识，在内核中建立进程结构。 * 复制父进程的环境 * 为进程分配资源， 包括进程映像所需要的所有元素（程序、数据、用户栈等）， * 复制父进程地址空间的内容到该进程地址空间中。 * 置该进程的状态为就绪，插入就绪队列。 |
| 进程撤销 |
| * 进程终止时操作系统做以下工作： * **关闭软中断***:*因为进程即将终止而不再处理任何软中断信号； * **回收资源：**释放进程分配的所有资源，如关闭所有已打开文件，释放进程相应的数据结构等； * **写记帐信息**：将进程在运行过程中所产生的记帐数据（其中包括进程运行时的各种统计信息）记录到一个全局记帐文件中； * **置该进程为僵死状态**:向父进程发送子进程死的软中断信号，将终止信息status送到指定的存储单元中； * **转进程调度**:因为此时CPU已经被释放，需要由进程调度进行CPU再分配。 |
|  |
|  |
| 进程创建其他实践  1）Win PK Linux进程观察工具  0号进程（也称为）空闲进程  内存===》交换空间，支持虚拟内存。  1号进程 第一个用户进程。。。。  2）which init /sbin/  3）查看内核进程pid最大配置  [root@localhost ~]# cat /proc/sys/kernel/pid\_max  32768  4）终止进程方法中  SIGABORT |
|  |

## 进程编程实践

### 1fork系统调用相关说明

|  |
| --- |
| **复制一个进程映象fork**   * 使用fork函数得到的子进程从父进程的继承了整个进程的地址空间，包括：进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设置、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等。 * 子进程与父进程的区别在于：   + 1、父进程设置的锁，子进程不继承   + 2、各自的进程ID和父进程ID不同   + 3、子进程的未决告警被清除；   + 4、子进程的未决信号集设置为空集。   **fork系统调用**   * 包含头文件 <sys/types.h> 和 <unistd.h> * 函数功能:创建一个子进程 * 函数原型   pid\_t fork(void);   * 参数：无参数。 * 返回值:   + 如果成功创建一个子进程，对于父进程来说返回子进程ID   + 如果成功创建一个子进程，对于子进程来说返回值为0   + 如果为-1表示创建失败 |
|  |
| 注意点：   * 理解1：fork系统调用之后，父子进程将交替执行。 * 理解2：怎么样理解一次调用2次返回? * 理解3：怎么样理解，fork返回值大于零的是父进程，为什么这样设计:？ * 理解4：怎么样理解分支在fork之后，而不是父进程main函数的开始？ |
|  |
| 总结：  怎么样理解，一次调用，二次返回？  问题的本质是：两次返回，是在各自的进程空间中返回的。  子进程和父进程各有自己的内存空间 （fork：代码段、数据段、堆栈段、PCB进程控制块的copy）。 |
| 总结：  #include <sys/types.h>  #include <unistd.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <signal.h>  #include <errno.h>  #include <signal.h>  int main(void )  {  int fd;  pid\_t pid;  signal(SIGCHLD, SIG\_IGN);  printf("befor fork pid:%d \n", getpid());  int num = 10; //思考打印  fd = open("11.txt", O\_WRONLY);  if (fd == -1)  {  return 0;  }    pid = fork();  if (pid == -1)  {  printf("pid < 0 err.\n");  return -1;  }  if (pid > 0)  {  printf("parent: pid:%d \n", getpid());  write(fd, "parent", 6);  close(fd);  //sleep(1);  }  else if (pid == 0)  {  printf("child: %d, parent: %d \n", getpid(), getppid());  write(fd, "child", 5);  close(fd);  //sleep(100);  }  printf("fork after....\n");  return 0;  } |
| 强化训练  //请问打印几句hello….  int main()  {  fork();  fork();  fork();  printf("hello…..\n");  return 0;  } |
| **写时复制copy on write**   * 如果多个进程要读取它们自己的那部分资源的副本，那么复制是不必要的。 * 每个进程只要保存一个指向这个资源的指针就可以了。 * 如果一个进程要修改自己的那份资源的“副本”，那么就会复制那份资源。这就是写时复制的含义 |
| 原因分析：加快速度，linux内核是段页式管理机制（因段管理从0开始，），也可叫页式管理机制。只复制对应的页。缺页，在中断查询，再赋值。  概念要深入理解 |

### 2孤儿进程和僵尸进程

|  |
| --- |
|  |
| 孤儿进程和僵尸进程   * 如果父进程先退出，子进程还没退出那么子进程的父进程将变为init进程。（注：任何一个进程都必须有父进程） * 如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵进程。   孤儿进程  如果父亲进程先结束，子进程会托孤给1号进程 |
| root 14121 14089 0 22:20 pts/18 00:00:00 su - itcast01  itcast01 14122 14121 0 22:20 pts/18 00:00:00 -bash  itcast01 14300 1 0 22:38 pts/18 00:00:00 ./dm02\_fork  itcast01 14308 14028 0 22:38 pts/17 00:00:00 ps -ef  [itcast01@localhost 01process]$ ps –ef |
| 僵尸进程 |
| 如果子进程结束，父进程还没有查询子进程的状态，那么子进程就会是僵尸状态  [itcast01@localhost 01process]$ ps -u itcast01  PID TTY TIME CMD  14028 pts/17 00:00:00 bash  14122 pts/18 00:00:00 bash  14233 pts/18 00:00:00 dm02\_fork  14234 pts/18 00:00:00 dm02\_fork <defunct>  14235 pts/17 00:00:00 ps |
| 避免僵尸进程  #include <signal.h>  signal(SIGCHLD, SIG\_IGN); |

### 3fork之后父子进程共享文件

|  |
| --- |
| 父进程的文件描述符   * 如果父进程中打开了一个文件，那么子进程不需要在打开 * 会不会像int 赋值两份。。。。。赋值的文件描述符是什么意思？ |
|  |
| void main(void)  {  pid\_t pid;  signal(SIGCHLD, SIG\_IGN);  int fd;  fd = open("test.txt", O\_WRONLY);  if (fd == -1)  {  perror("open err");  exit(0);  }    pid = fork();  if (pid == -1)  {  perror("fork err");  return -1;  }  if (pid == 0)  {  printf("this is child pid:%d ppid:%d \n", getpid(), getppid());  write(fd, "child", 5);  //sleep(20);  }  if (pid > 0)  {  printf("this is parent pid:%d ppid:%d \n", getpid(), getppid());  write(fd, "parent", 5);  //sleep(20);  }  printf("fork() after\n");  sleep(1);  close(fd);  return 0;  } |
| fd = open("test.txt", O\_WRONLY);  write(fd, "parent", 5); |

### 4fork和vfork

|  |
| --- |
| 1）在fork还没实现copy on write之前。Unix设计者很关心fork之后立刻执行exec所造成的地址空间浪费，所以引入了vfork系统调用。  2）vfork有个限制，子进程必须立刻执行\_exit或者exec函数。  即使fork实现了copy on write，效率也没有vfork高，但是我们不推荐使用vfork，因为几乎每一个vfork的实现，都或多或少存在一定的问题。 |
| 结论：  1：fork子进程拷贝父进程的数据段  Vfork子进程与父进程共享数据段；  2：fork父、子进程的执行次序不确定  Vfork：子进程先运行，父进程后运行； |
|  |
| Vfork和exec函数族在一起  execve替换进程映像（加载程序）注意execve是一个系统调用;  替换意味着：代码段、数据段、堆栈段、进程控制块PCB全部替换。 |
| int main(void)  {  int fd;  pid\_t pid;  signal(SIGCHLD, SIG\_IGN);  printf("befor fork pid:%d \n", getpid());  g\_num = 10;  pid = vfork();  if (pid == -1)  {  printf("pid < 0 err.\n");  return -1;  }    if (pid > 0)  {  printf("parent: pid:%d \n", getpid());  }  else if (pid == 0)  {  printf("child: %d, parent: %d \n", getpid(), getppid());    // int execve(const char \*filename, char \*const argv[],  // char \*const envp[]);  char \*const argv[] = {"ls", "-lt", NULL};  execve("/bin/ls", argv, NULL);    //注意：此处有/bin/ls程序结束程序，不会出现coredump现象    printf("测试下面这句话还执行吗\n");  //exit(0);  //1 vfork只有需要用exit(0) \_exit(0)  //2 测试return 0; 区别  }  printf("fork after....\n");  return 0;  } |

### 5进程终止的5种方式

|  |
| --- |
| 进程终止的5种方式   * 正常退出   + 从main函数返回   + 调用exit   + 调用\_exit * 异常退出   + 调用abort 产生SIGABOUT信号   + 由信号终止 **ctrl+c SIGINT** |
|  |
| eg：  区别1：清空缓冲区的操作  int main(void)  {  printf("hello itcast");  //return 0;  //exit(0);  fflush(stdout);  \_exit(0);  }  区别2：exit会调用终止处理程序  有关终止处理程序 |
| * atexit可以注册终止处理程序，ANSI C规定最多可以注册32个终止处理程序。 * 终止处理程序的调用与注册次序相反 * man atexit   int atexit(void (\*function)(void));  man 2 atexit |
| 总结exit与\_exit区别  1）\_exit是一个系统调用，exit是一个c库函数  2）exit会执行清楚I/O缓存  3）exit会执行调用终止处理程序 |
|  |

### 6exec函数族替换进程印象

|  |
| --- |
| exec替换进程印象   * 在进程的创建上Unix采用了一个独特的方法，它将进程创建与加载一个新进程映象分离。这样的好处是有更多的余地对两种操作进行管理。 * 当我们创建了一个进程之后，通常将子进程替换成新的进程映象，这可以用exec系列的函数来进行。当然，exec系列的函数也可以将当前进程替换掉。 |
| exec关联的函数族   * 包含头文件<unistd.h> * 功能用exec函数可以把当前进程替换为一个新进程。exec名下是由多个关联函数组成的一个完整系列，头文件<unistd.h> * 原型   /\*  #include <unistd.h>  extern char \*\*environ;  int execl(const char \*path, const char \*arg, ...);  int execlp(const char \*file, const char \*arg, ...); PATH  int execle(const char \*path, const char \*arg,  ..., char \* const envp[]);  int execv(const char \*path, char \*const argv[]);  int execvp(const char \*file, char \*const argv[]);  \*/ |
| exec exec关联的函数族参考 man 手册 |
| 以execlp、execvp、和execle讲解   * path参数表示你要启动程序的名称包括路径名 * arg参数表示启动程序所带的参数 * 返回值:成功返回0,失败返回-1 |
| * execl，execlp，execle（都带“l”）的参数个数是可变的，参数以一个空指针结束。 * execv和execvp的第二个参数是一个字符串数组，新程序在启动时会把在argv数组中给定的参数传递到main * 这些函数通常都是用execve实现的，这是一种约定俗成的做法，并不是非这样不可。 |
| * 名字最后一个字母是“p”的函数会搜索PATH环境变量去查找新程序的可执行文件。如果可执行文件不在PATH定义的路径上，就必须把包括子目录在内的绝对文件名做为一个参数传递给这些函数   总结：l代表可变参数列表，p代表在path环境变量中搜索file文件。envp代表环境变量。 |
| int main(void)  {  //演示程序被完全替换  //替换以后，pid不会发生变化  //注意 printf后的\n不能忘记，不然main函数打印不出来  printf(“main getpid: %d\n”, getpid());    //execlp(“ls”, “ls”, “-lt”, NULL);  int ret = execlp(“./testpid2”, NULL, NULL);  if (ret == -1)  {  perror(“ERR: “);  }    printf(“fork after….\n”);  return 0;  } |
| 有关环境变量  int main(int argc, char \*argv[])  {  printf(“main egin…\n”);  //1)如果envp环境参数列表不填写，则testpidandenv程序会使用默认的环境参数列表  //2)如果envp环境参数列表填写，则testpidandenv程序会使用你填写的环境参数列表  char \*ars[] = {“aa=111”, “bb==222”, NULL};  int ret = execle(“./testpidandenv”,”testpidandenv”, NULL,NULL);  //int ret = execle(“./testpidandenv”,”testpidandenv”, NULL,ars);    if (ret == -1)  {  perror("main:");  }  printf("main end...\n");  return 0;  } |

### 7父进程wait和waitpid

|  |
| --- |
| 1、wait和waitpid出现的原因 |
| SIGCHLD   * 当子进程退出的时候，内核会向父进程发送SIGCHLD信号，子进程的退出是个异步事件（子进程可以在父进程运行的任何时刻终止） * 子进程退出时，内核将子进程置为僵尸状态，这个进程称为僵尸进程，它只保留最小的一些内核数据结构，以便父进程查询子进程的退出状态。   父进程查询子进程的退出状态可以用wait/waitpid函数 |
| 2、wait和waitpid函数用法 |
| Wait   * 头文件<sys/types.h>和<sys/wait.h> * 函数功能:当我们用fork启动一个进程时，子进程就有了自己的生命，并将独立地运行。有时，我们需要知道某个子进程是否已经结束了，我们可以通过wait安排父进程在子进程结束之后。 * 函数原型   + pid\_t wait(int \*status) * 函数参数   + status:该参数可以获得你等待子进程的信息 * 返回值：   + 成功等待子进程函数返回，返回值是等待子进程的ID * wait系统调用会使父进程暂停执行，直到它的一个子进程结束为止。 * 返回的是子进程的PID，它通常是结束的子进程 * 状态信息允许父进程判定子进程的退出状态，即从子进程的main函数返回的值或子进程中exit语句的退出码。 * 如果status不是一个空指针，状态信息将被写入它指向的位置 |
| Wait获取status后检测处理  宏定义 描述  WIFEXITED(status) 如果子进程正常结束，返回一个非零值  WEXITSTATUS(status) 如果WIFEXITED非零，返回子进程退出码  WIFSIGNALED(status) 子进程因为捕获信号而终止，返回非零值  WTERMSIG(status) 如果WIFSIGNALED非零，返回信号代码  WIFSTOPPED(status) 如果子进程被暂停，返回一个非零值  WSTOPSIG(status) 如果WIFSTOPPED非零，返回一个信号代码 |
| Waitpid  ❑函数功能:用来等待某个特定进程的结束  ❑函数原型:  **pid\_t waitpid(pid\_t pid, int \*status,int options)**  ❑参数:   * + status:如果不是空，会把状态信息写到它指向的位置   ❑options：允许改变waitpid的行为，最有用的一个选项是WNOHANG,它的作用是防止waitpid把调用者的执行挂起   * + 返回值：如果成功返回等待子进程的ID，失败返回-1 |
| 对于waitpid的*p I d*参数的解释与其值有关：   * *pid* == -1 等待任一子进程。于是在这一功能方面waitpid与wait等效。 * *pid* > 0 等待其进程I D与*p I d*相等的子进程。 * *pid* == 0 等待其组I D等于调用进程的组I D的任一子进程。换句话说是与调用者进程同在一个组的进程。 * *pid* < -1 等待其组I D等于*p I d*的绝对值的任一子进程。 |
| 3、wait pk waitpid |
| Wait和waitpid区别和联系   * 在一个子进程终止前， wait 使其调用者阻塞，而waitpid 有一选择项，可使调用者不阻塞。 * waitpid并不等待第一个终止的子进程—它有若干个选择项，可以控制它所等待的特定进程。 * 实际上wait函数是waitpid函数的一个特例。 |
| 僵尸进程   * 当一个子进程结束运行时，它与其父进程之间的关联还会保持到父进程也正常地结束运行或者父进程调用了wait才告终止。   进程表中代表子进程的数据项是不会立刻释放的，虽然不再活跃了，可子进程还停留在系统里，因为它的退出码还需要保存起来以备父进程中后续的wait调用使用。它将称为一个“僵进程” |
| 如何避免僵尸进程 |
| * 调用wait或者waitpid函数查询子进程退出状态，此方法父进程会被挂起。 * 如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG\_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的。 |

wait和waitpid的工程实践

|  |
| --- |
| abort() //会发出一个6号信号  man 7 signal  [itcast01@localhost 03exec]$ kill -l  1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL  5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE  9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2  13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT  17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP  21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU  25) SIGXFSZ 26) SIGVTALRM 27) SIGPROF 28) SIGWINCH  29) SIGIO 30) SIGPWR 31) SIGSYS 34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4  39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8  43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12  47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14  51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10  55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6  59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2  63) SIGRTMAX-1 64) SIGRTMAX |

利用man手册，大大减少开发难度，提高开发技能

|  |
| --- |
| Man 7 signal 查找有什么信号可以使应用程序暂停  Standard Signals  Linux supports the standard signals listed below. Several signal numbers are architecture dependent, as indicated in  the "Value" column. (Where three values are given, the first one is usually valid for alpha and sparc, the middle one  for i386, ppc and sh, and the last one for mips. A - denotes that a signal is absent on the corresponding architec-  ture.)  First the signals described in the original POSIX.1-1990 standard.  Signal Value Action Comment  -------------------------------------------------------------------------  SIGHUP 1 Term Hangup detected on controlling terminal  or death of controlling process  SIGINT 2 Term Interrupt from keyboard  SIGQUIT 3 Core Quit from keyboard  SIGILL 4 Core Illegal Instruction  SIGABRT 6 Core Abort signal from abort(3)  SIGFPE 8 Core Floating point exception  SIGKILL 9 Term Kill signal  SIGSEGV 11 Core Invalid memory reference  SIGPIPE 13 Term Broken pipe: write to pipe with no readers  SIGALRM 14 Term Timer signal from alarm(2)  SIGTERM 15 Term Termination signal  SIGUSR1 30,10,16 Term User-defined signal 1  SIGUSR2 31,12,17 Term User-defined signal 2  SIGCHLD 20,17,18 Ign Child stopped or terminated  SIGCONT 19,18,25 Cont Continue if stopped  SIGSTOP 17,19,23 Stop Stop process  SIGTSTP 18,20,24 Stop Stop typed at tty  SIGTTIN 21,21,26 Stop tty input for background process  SIGTTOU 22,22,27 Stop tty output for background process  The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored. |
| Man 2wait  SYNOPSIS  #include <sys/types.h>  #include <sys/wait.h>  pid\_t wait(int \*status);  pid\_t waitpid(pid\_t pid, int \*status, int options);  int waitid(idtype\_t idtype, id\_t id, siginfo\_t \*infop, int options);  在手册中清楚的写明了用法 |
| EXAMPLE  The following program demonstrates the use of fork(2) and waitpid(2). The program creates a child process. If no com-  mand-line argument is supplied to the program, then the child suspends its execution using pause(2), to allow the user  to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately,  using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors  the child using waitpid(2), and uses the W\*() macros described above to analyse the wait status value.  The following shell session demonstrates the use of the program:  例子：  $ ./a.out &  Child PID is 32360  [1] 32359  $ kill -STOP 32360  stopped by signal 19  $ kill -CONT 32360  continued  $ kill -TERM 32360  killed by signal 15  [1]+ Done ./a.out  $  #include <sys/wait.h>  #include <stdlib.h>  #include <unistd.h>  #include <stdio.h>  int  main(int argc, char \*argv[])  {  pid\_t cpid, w;  int status;  cpid = fork();  if (cpid == -1) { perror("fork"); exit(EXIT\_FAILURE); }  if (cpid == 0) { /\* Code executed by child \*/  printf("Child PID is %ld\n", (long) getpid());  if (argc == 1)  pause(); /\* Wait for signals \*/  \_exit(atoi(argv[1]));  } else { /\* Code executed by parent \*/  do {  w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);  if (w == -1) { perror("waitpid"); exit(EXIT\_FAILURE); }  if (WIFEXITED(status)) {  printf("exited, status=%d\n", WEXITSTATUS(status));  } else if (WIFSIGNALED(status)) {  printf("killed by signal %d\n", WTERMSIG(status));  } else if (WIFSTOPPED(status)) {  printf("stopped by signal %d\n", WSTOPSIG(status));  } else if (WIFCONTINUED(status)) {  printf("continued\n");  }  } while (!WIFEXITED(status) && !WIFSIGNALED(status));  exit(EXIT\_SUCCESS);  }  } |

### 8system C库函数

|  |
| --- |
| System   * 功能：system()函数调用“/bin/sh -c command”执行特定的命令，阻塞当前进程直到command命令执行完毕 * 原型：   int system(const char \*command);   * 返回值：   如果无法启动shell运行命令，system将返回127；出现不能执行system调用的其他错误时返回-1。如果system能够顺利执行，返回那个命令的退出码。   * system函数执行时，会调用fork、execve、waitpid等函数。 |
| 自动动手写system命令  int my\_system(const char \*command)  {  pid\_t pid;  int status;  if (command == NULL)  return 1;  if ((pid = fork()) < 0)  status = -1;  else if (pid == 0)  {  execl("/bin/sh", "sh", "-c", command, NULL);  exit(127);  }  else  {  while (waitpid(pid, &status, 0) < 0)  {  if (errno == EINTR)  continue;  status = -1;  break;  }  }  return status;  } |

### 9守护进程

守护进程概念

|  |
| --- |
|  |
| 什么是守护进程   * 守护进程是在后台运行不受控端控制的进程，通常情况下守护进程在系统启动时自动运行 * 守护进程的名称通常以d结尾，比如sshd、xinetd、crond等 |
| 创建守护进程步骤   * 调用fork(),创建新进程，它会是将来的守护进程 * 在父进程中调用exit，保证子进程不是进程组组长 * 调用setsid创建新的会话期 * 将当前目录改为根目录 （如果把当前目录作为守护进程的目录，当前目录不能被卸载，它作为守护进程的工作目录了。） * 将标准输入、标准输出、标准错误重定向到/dev/null |
| 守护进程api  **int daemon(int nochdir, int noclose);**   * 功能：创建一个守护进程 * 参数：   + nochdir：=0将当前目录更改至“/”   + noclose：=0将标准输入、标准输出、标准错误重定向至“/dev/null” |
|  |

守护进程工程实践

|  |
| --- |
|  |
| 1）  man setsid  DESCRIPTION  setsid() creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session, the process group leader of the new process group, and has no controlling tty. The process group ID and session ID of the calling process are set to the PID of the calling process. The calling process will be the only process in this new process group and in this new session. |
| Setsid创建一个新的会话；调用者进程会是这个会话期唯一的一个进程，是唯一组的组长；调用者进程id是组id，也是会话期的id。不能用进程组组长去调用setsid函数 |
| 2）  [itcast01@localhost 04wait]$ ps –ef  UID PID PPID C STIME TTY TIME CMD  itcast01 14122 14121 0 Jun25 pts/18 00:00:00 –bash  root 14431 3005 0 Jun25 ? 00:00:00 sshd: itcast01 [priv]  itcast01 14433 14431 0 Jun25 ? 00:00:00 sshd: itcast01@notty  itcast01 14434 14433 0 Jun25 ? 00:00:00 /usr/libexec/openssh/sftp-server  root 15064 1583 0 Jun25 pts/1 00:00:00 su – itcast01  itcast01 15065 15064 0 Jun25 pts/1 00:00:00 –bash  itcast01 25328 14028 0 12:56 pts/17 00:00:00 ps –ef  有的进程有tty，有的进程没有tty。带问号的是守护进程。 |
| 3）  可以利用killall 进程名字 来杀一族进程，命令很爽。。。。不需要 kill -9 pid  root 15064 1583 0 Jun25 pts/1 00:00:00 su – itcast01  itcast01 15065 15064 0 Jun25 pts/1 00:00:00 –bash  itcast01 25635 1 70 13:20 ? 00:00:16 ./dm01\_mydaemon  itcast01 25674 1 42 13:20 ? 00:00:03 ./dm01\_mydaemon  itcast01 25676 14028 0 13:20 pts/17 00:00:00 ps –ef  [itcast01@localhost 05daemon]$ killall dm01\_mydaemon  [itcast01@localhost 05daemon]$ pwd  /home/itcast01/05daemon |
| 4）守护进程，关闭和不关闭重定向之间的区别。。。  做实验。。。。。 |
| 5、其他知识延伸  会话期、进程组 |
|  |
| 会话期：是一个或者多个进程组的集合，通常一个会话期开始与用户登录，终止于用户退出。在此期间，该用户运行的所有进程都属于这个会话期。 |
| 请你思考  Telnet两个标签页登录linux服务器，会话期示意图应该如何画？  [itcast01@localhost 04wait]$ who  root :0 2014-06-25 11:50  root pts/1 2014-06-25 11:52 (:0.0)  root pts/17 2014-06-25 22:10 (192.168.6.253)  root pts/18 2014-06-25 22:20 (192.168.6.253)  root pts/10 2014-06-25 13:17 (:0.0)  root pts/4 2014-06-25 16:49 (:0.0)  root pts/5 2014-06-25 19:47 (:0.0)  root pts/16 2014-06-25 20:18 (:0.0) |
| int setup\_daemon(int nochdir, int noclose);  int main(int argc, char \*argv[])  {  mydaemon(1, 1);  //man daemon 可以看到  //0表示改变重定向 1表示不改变  //daemon(1, 1);  printf("test ...\n");  for (;;) ;  return 0;  }  int mydaemon(int nochdir, int noclose)  {  pid\_t pid;  pid = fork();  if (pid == -1)  ERR\_EXIT("fork error");  if (pid > 0)  exit(EXIT\_SUCCESS);  setsid();    if (nochdir == 0)  chdir("/");  if (noclose == 0)  {  int i;  for (i=0; i<3; ++i)  close(i);  //相当于把0号文件描述符之下/dev/null  open("/dev/null", O\_RDWR); //fd文件描述符fd-0的文件描述符指向 -16  dup(0); //把0号文件描述符 赋值给空闲的文件描述符 1  dup(0); //把0号文件描述符 赋值给空闲的文件描述符 2  }  return 0;  } |
|  |
|  |
|  |

|  |
| --- |
| UNIX下的网络服务程式，如Web　Server，FTP，Telnet一般都是由守护进程(Daemon)来实现的。守护进程不占用终端，在后台运 行。UNIX的守护进程一般都命名为　\*d　的形式，如httpd，telnetd等等。其实，守护进程的实现是非常简单的，在我的程式中，我使用一个 INIT\_DAEMON宏来实现守护进程的初始化工作，如图2.4所示。第一次调用fork函数，为避免挂起控制终端将守护进程放入后台执行。然后调用 setsid函数脱离控制终端，登录会话和进程组，使该进程成为会话组长，和原来的登录会话和进程组脱离，进程同时和控制终端脱离。进程已成为无终端的会 话组长。但他能重新申请打开一个控制终端。能通过使进程不再成为会话组长来禁止进程重新打开控制终端，这就需要第二次调用fork函数，父进程（会话组 长）退出，子进程继续执行，并不在拥有打开控制终端的能力。在正在执行的进程中调用INIT\_DAEMON后　进程将成 |
| #define INIT\_DAEMON \  { \  if(fork() >0) exit(0); \  setsid(); \  if(fork()>0) exit(0); \  } |
| 为守护进程，脱离控制终端进入后台执行。比如，我们的网络服务程式，能在完成创建套接口，绑定套接口，设置套接口为监听模式后，变成守护进程进入后台执行而不占用控制终端，这是网络服务程式的常用模式。　   　　INIT\_DAEMON宏只提供了非常简单的功能，守护进程一旦脱离了终端，退出就成了问题。我目前使用　PS　查出进程ID然后　kill　之。 |

## 进程项目开发进价

### 多进程测试框架搭建

### 多进程遇上动态库

# linux定时器中的时间

|  |
| --- |
| **三种不同精度的睡眠** |
| * unsigned int sleep(unsigned int seconds); 秒   若被中断打断，返回剩余时间   * int usleep(useconds\_t usec); 微妙   若被中断打断，返回剩余时间   * int nanosleep(const struct timespec \*req, struct timespec \*rem);纳秒时间   要睡眠的时间req；剩余睡眠时间，如果要中断，通过rem返回过来。 |
| **三种时间结构**  time\_t 秒  struct timeval {  long tv\_sec; /\* seconds \*/  long tv\_usec; /\* microseconds \*/微妙  };  struct timespec { 纳秒  time\_t tv\_sec; /\* seconds \*/  long tv\_nsec; /\* nanoseconds \*/  }; |
| setitime函数： 间隙性产生定时器   * 包含头文件<sys/time.h> * 功能setitimer()比alarm功能强大，支持3种类型的定时器 * 原型：   **int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue));**   * 参数   + 第一个参数which指定定时器类型   + 第二个参数是结构itimerval的一个实例，结构itimerval形式   + 第三个参数可不做处理。 * 返回值:成功返回0失败返回-1 * ITIMER\_REAL：经过指定的时间后，内核将发送SIGALRM信号给本进程 * ITIMER\_VIRTUAL ：程序在用户空间执行指定的时间后，内核将发送SIGVTALRM信号给本进程 * ITIMER\_PROF ：进程在内核空间中执行时，时间计数会减少，通常与ITIMER\_VIRTUAL共用，代表进程在用户空间与内核空间中运行指定时间后，内核将发送SIGPROF信号给本进程。 |
| 思考why ITIMER\_PROF存在？ |
| cpu运行时间：一般指（进程在用户空间和内核空间）共同的运行时间 |

setitmie函数用法

|  |
| --- |
|  |
| int main(int argc, char \*argv[])  {  struct timeval tv\_interval = {1, 0};  struct timeval tv\_value = {1, 0};  struct itimerval it;  it.it\_interval = tv\_interval; //产生定时器的时间间隔  it.it\_value = tv\_value; //第一次产生定时器信号的时间  setitimer(ITIMER\_REAL, &it, NULL);  int i;  for (i=0; i<10000; i++);  ;  struct itimerval oit;  setitimer(ITIMER\_REAL, &it, &oit);  printf("%d %d %d %d\n", (int)oit.it\_interval.tv\_sec, (int)oit.it\_interval.tv\_usec, (int)oit.it\_value.tv\_sec, (int)oit.it\_value.tv\_usec);  return 0;  } |
|  |

getitimer函数用法

|  |
| --- |
| int main(int argc, char \*argv[])  {  struct timeval tv\_interval = {1, 0};  struct timeval tv\_value = {1, 0};  struct itimerval it;  it.it\_interval = tv\_interval;  it.it\_value = tv\_value;  setitimer(ITIMER\_REAL, &it, NULL);  int i;  for (i=0; i<10000; i++);  ;  getitimer(ITIMER\_REAL, &it);  printf("%d %d %d %d\n", (int)it.it\_interval.tv\_sec, (int)it.it\_interval.tv\_usec, (int)it.it\_value.tv\_sec, (int)it.it\_value.tv\_usec);  return 0;  } |

# linux信号-应用编程-专题讲座

## 1 基本概念

|  |
| --- |
| 中断   * 中断是系统对于异步事件的响应 * 中断信号 * 中断源 * 现场信息 * 中断处理程序 * 中断向量表   异步事件的响应：进程执行代码的过程中可以随时被打断，然后去执行异常处理程序  生活中的中断和计算机系统中的中断   1. 无中断生活场景   张三看书，厨房烧水  2）有中断的生活场景  张三看书，设置闹钟，厨房烧水。  闹钟发出中断信号，张三把书合好(第20页)，去厨房把开水事情处理好，张三重新打开20页进行阅读。  3）计算机系统的中断场景  中断源发出中断信号，CPU判断中断是否屏蔽屏蔽、保护现场 ，cpu执行中断处理程序， cpu恢复现场，继续原来的任务。 4）中断的其他概念  中断向量表保存了中断处理程序的入口地址。  中断个数固定，操作系统启动时初始化中断向量表。  中断有优先级（有人敲门，有人打电话，有优先级）  中断可以屏蔽（张三可以屏蔽电话）。 |
| 中断分类   * 硬件中断（外部中断）   外部中断是指由外部设备通过硬件请求的方式产生的中断，也称为硬件中断   * 软件中断（内部中断）   内部中断是由CPU运行程序错误或执行内部程序调用引起的一种中断，也称为软件中断。   * x86平台INT指令 ARM软中断指令SWI |
| 信号概念 |
| * 信号是UNIX系统响应某些状况而产生的事件，进程在接收到信号时会采取相应的行动。 * 信号是因为某些错误条件而产生的，比如内存段冲突、浮点处理器错误或者非法指令等 * 信号是在软件层次上对中断的一种模拟，所以通常把它称为是软中断 |
| 信号和中断的区别   * 信号与中断的相似点：   + （1）采用了相同的异步通信方式；   + （2）当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序；   + （3）都在处理完毕后返回到原来的断点；   + （4）对信号或中断都可进行屏蔽。 * 信号与中断的区别：   + （1）中断有优先级，而信号没有优先级，所有的信号都是平等的；   + （2）信号处理程序是在用户态下运行的，而中断处理程序是在核心态下运行；   + （3）中断响应是及时的，而信号响应通常都有较大的时间延迟。 |

## 2 信号名称及常用信号

|  |
| --- |
| 信号名称 描述  SIGABRT 进程停止运行 6  SIGALRM 警告钟  SIGFPE 算述运算例外  SIGHUP 系统挂断  SIGILL 非法指令  SIGINT 终端中断 2  SIGKILL 停止进程(此信号不能被忽略或捕获)  SIGPIPE 向没有读者的管道写入数据  SIGSEGV 无效内存段访问  SIGQUIT 终端退出 3  SIGTERM 终止  SIGUSR1 用户定义信号1  SIGUSR2 用户定义信号2  SIGCHLD 子进程已经停止或退出  SIGCONT 如果被停止则继续执行  SIGSTOP 停止执行  SIGTSTP 终端停止信号  SIGTOUT 后台进程请求进行写操作  SIGTTIN 后台进程请求进行读操作 |
| 实验1：  kill –l 可以查看linux内核支持的信号  Man 7 signal 查看信号的默认动作、信号的含义 |

## 3 信号处理

|  |
| --- |
| 进程对信号的三种相应   * 忽略信号   不采取任何操作、有两个信号不能被忽略：SIGKILL（9号信号）和SIGSTOP。  思考1：为什么进程不能忽略SIGKILL、SIGSTOP信号。（如果应用程序可以忽略这2个信号，系统管理无法杀死、暂停进程，无法对系统进行管理。）。SIGKILL（9号信号）和SIGSTOP信号是不能被捕获的。   * 捕获并处理信号   内核中断正在执行的代码，转去执行先前注册过的处理程序。   * 执行默认操作   默认操作通常是终止进程，这取决于被发送的信号。  信号的默认操作：通过 man 7 signal 进程查看 |
| man 7 signal |
| 编程实践:让应用程序捕捉ctrl+c信号  以ctrl+c 会产生一个中断。当前应用程序捕捉ctrl+c中断信号。 |

## 4 信号signal函数编程实践

signal信号安装函数

|  |
| --- |
| signal函数，作用1：站在应用程序的角度，注册一个信号处理函数。  作用2：忽略信号、设置信号默认处理 信号的安装和恢复   * typedef void (\*\_\_sighandler\_t) (int); * #define SIG\_ERR ((\_\_sighandler\_t) -1) * #define SIG\_DFL ((\_\_sighandler\_t) 0) * #define SIG\_IGN ((\_\_sighandler\_t) 1) * 函数原型:   **\_\_sighandler\_t signal(int signum, \_\_sighandler\_t handler);**   * **参数**   + signal是一个带signum和handler两个参数的函数，准备捕捉或屏蔽的信号由参数signum给出，接收到指定信号时将要调用的函数由handler给出   + handler这个函数必须有一个int类型的参数（即接收到的信号代码），它本身的类型是void   + handler也可以是下面两个特殊值：   SIG\_IGN 屏蔽该信号  SIG\_DFL 恢复默认行为 |
| 编程实践 注册SIGINT、注册SIGQUIT  编程实践：信号的安装和恢复  //测试信号的安装与恢复  int main(void)  {  \_\_sighandler\_t oldHandle;  //sighandler\_t \*oldHandle;  printf("main ....begin\n");  oldHandle = signal(SIGINT, myhandle);  if (oldHandle == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }    printf("if u enter a, reset signal \n");  while(getchar() != 'a')  {  ;  }  //键入a以后，恢复默认函数  /\*  if (signal(SIGINT, oldHandle) == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }  \*/    if (signal(SIGINT, SIG\_DFL) == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }    while(1) ;  return 0;  }注意：signal函数函数返回值  RETURN VALUE  The signal() function returns the previous value of the signal handler, or SIG\_ERR on error. |

## 5 可靠信号、不可靠信号

|  |
| --- |
| 不可靠信号PK可靠信号   * linux信号机制基本上是从unix系统中继承过来的。早期unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，它的主要问题是：   + 进程每次处理信号后，就将对信号的响应设置为默认动作。在某些情况下，将导致对信号的错误处理；因此，用户如果不希望这样的操作，那么就要在信号处理函数结尾再一次调用signal()，重新安装该信号。   + 早期unix下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失。   + linux支持不可靠信号，但是对不可靠信号机制做了改进：在调用完信号处理函数后，不必重新调用该信号的安装函数（信号安装函数是在可靠机制上的实现）。因此，linux下的不可靠信号问题主要指的是信号可能丢失。 |
| 总结：unix信号机制不可靠地方，1）处理完信号以后，需要重新再注册信号；2）信号可能丢失。linux下已经对1做了优化。 |
| 可靠信号   * 随着时间的发展，实践证明，有必要对信号的原始机制加以改进和扩充。所以，后来出现的各种unix版本分别在这方面进行了研究，力图实现"可靠信号"。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失。同 时，信号的发送和安装也出现了新版本：信号发送函数sigqueue()及信号安装函数sigaction()。 |
| 总结：sigaction和signal函数，都是调用内核服务do\_signal函数；  内核服务函数（应用系统无法调用的函数） |
| 实时信号（都是可靠信号）和非实时信号（不可靠信号）   * 早期Unix系统只定义了32种信号，Ret hat7.2支持64种信号，编号0-63(SIGRTMIN=31，SIGRTMAX=63)，将来可能进一步增加，这需要得到内核的支持。前32种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的CTRL+C时，会产生SIGINT信号，对该信号的默认反应就是进程终止。后32个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收。实时信号是POSIX标准的一部分，可用于应用进程。 * 非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号。 |
| 实时信号和非实时信号   Man 7 signal 查看手册  非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号 |

## 6 信号发送

### kill函数

|  |
| --- |
| Kill基本用法 |
| 发送信号的函数有kill和raise  区别：kill既可以向自身发送信号，也可以向其他进程发送信号；  raise函数向进程自身发送信号。  Int kill(pid\_t pid, int siq)  int raise(int signo) |
| Int kill(pid\_t pid, int siq)  参数组合情况解释：  kill(pid\_t pid, int siq)  pid>0 将信号sig发给pid进程  pid=0 将信号sig发给同组进程  pid=-1 将信号sig发送给所有进程，调用者进程有权限发送的每一个进程（除了1号进程之外，还有它自身）  pid<-1 将信号sig发送给进程组是pid（绝对值）的每一个进程 |
| 实验：  //子进程向父进程发送信号  //子进程向同组进程发送信号 （getpgrp()函数获取进程组pid）  结论：注意，如果在fork之前安装信号，则子进程可以继承信号。 |
| kill和sleep在一起 |
| 子进程向父进程发送信号  sleep函数几点说明  1）sleep函数作用，让进程睡眠。  2）能被信号打断，然后处理信号函数以后，就不再睡眠了。直接向下执行代码  3）sleep函数的返回值，是剩余的秒数 |
|  |

### raise函数

|  |
| --- |
| raise   * raise   + 给自己发送信号。raise(sig)等价于kill(getpid(), sig); * killpg   + 给进程组发送信号。killpg(pgrp, sig)等价于kill(-pgrp, sig); * sigqueue   + 给进程发送信号，支持排队，可以附带额外数据信息。 |

### pause函数

|  |
| --- |
| pause()函数   * 将进程置为可中断睡眠状态。然后它调用内核函数schedule()，使linux进程调度器找到另一个进程来运行。 * pause使调用者进程挂起，直到一个信号被捕获 |

### alarm函数

|  |
| --- |
| alarm函数，设置一个闹钟延迟发送信号  告诉linux内核n秒中以后，发送SIGALRM信号；；  手册描述：  NAME  alarm - set an alarm clock for delivery of a signal  SYNOPSIS  #include <unistd.h>  unsigned int alarm(unsigned int seconds);  DESCRIPTION  alarm() arranges for a SIGALRM signal to be delivered to the process in seconds seconds.  If seconds is zero, no new alarm() is scheduled.  In any event any previously set alarm() is cancelled. |
| 实验1：手工发送信号alarm信号实践  kill -ALRM 21333  kill -14 213333  kill -SIGALRM 213333  命令发送信号实践  kill -alram ` ps -aux | grep 01aram | grep -v vi | awk '{print $2}' ` |
| 编程实践  void myhandle(int num)  {  printf("recv signal id num : %d \n", num);  //kill -alram ` ps -aux | grep 01aram | grep -v vi | awk '{print $2}' `  alarm(1);  }  int main(void)  {  printf("main ....begin\n");  //注册信号处理函数  if (signal(SIGALRM, myhandle) == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }    alarm(1);  while(1)  {  pause();  printf("pause return\n");  }  return 0;  } |

## 8 信号处理函数遇上可重入和不可重入函数

|  |
| --- |
| 可重入函数概念   * 为了增强程序的稳定性，在信号处理函数中应使用可重入函数。 * 所谓可重入函数是指一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错。因为进程在收到信号后，就将跳转到信号处理函数去接着执行。如果信号处理函数中使用了不可重入函数，那么信号处理函数可能会修改原来进程中不应该被修改的数据，这样进程从信号处理函数中返回接着执行时，可能会出现不可预料的后果。不可再入函数在信号处理函数中被视为不安全函数。 * 满足下列条件的函数多数是不可再入的：（1）使用静态的数据结构，如getlogin()，gmtime()，getgrgid()，getgrnam()，getpwuid()以及getpwnam()等等；（2）函数实现时，调用了malloc（）或者free()函数；（3）实现时使用了标准I/O函数的 |
| man手册实践  man 7 signal 查找可重入函数和不可重入函数，下一个内核用文件描述符来优化信号 |
| 结论：  在信号处理函数中，尽量不使用全局变量和静态变量的函数。特别是这个变量在程序中随时读写。 |

## 9 信号的阻塞和未达

### 信号在linux中的传递过程

|  |
| --- |
| 信号在内核中的表示   * 执行信号的处理动作称为信号*递达*（Delivery），信号从产生到递达之间的状态，称为信号*未决*（Pending）。进程可以选择阻塞（Block）某个信号。被阻塞的信号产生时将保持在未决状态，直到进程解除对此信号的阻塞，才执行递达的动作。 * 注意，阻塞和忽略是不同，只要信号被阻塞就不会递达，而忽略是在递达之后可选的一种处理动作。信号在内核中的表示可以看作是这样的： |
| 说明1）PCB进程控制块中结构体中有信号屏蔽状态字（block），信号未决状态字（pending）还有是否忽略标志；  说明2）信号屏蔽状态字（block），1代表阻塞、0代表不阻塞；信号未决状态字（pending）的1代表未决（表示有未达信号），0代表信号可以抵达了；  说明3）向进程发送SIGINT，内核首先判断信号屏蔽状态字是否阻塞，信号未决状态字（pending相应位制成1；若阻塞解除，信号未决状态字（pending）相应位制成0；表示信号可以抵达了。  说明4）block状态字、pending状态字 64bit；//socket select  说明5）block状态字用户可以读写，pending状态字用户只能读；这是信号设计机制。  思考1：状态字都64bit，编程时，如何表示状态字那？  思考2：block状态字信息如何获取或者操作那？哪些api？  思考3：pending状态字信息如何获取或者操作那？哪些api？ |
| 信号集操作函数（状态字表示）   * #include <signal.h> * int sigemptyset(sigset\_t \*set); 把信号集清空 64bit/8=8个字节 * int sigfillset(sigset\_t \*set); 把信号集置成1 * int sigaddset(sigset\_t \*set, int signo); 根据signo，把信号集中的对应bit置成1 * int sigdelset(sigset\_t \*set, int signo); 根据signo，把信号集中的对应bit置成0 * int sigismember(const sigset\_t \*set, int signo);//判断signo是否在信号集中 |
| sigprocmask读取或更改进程的信号屏蔽状态字（block）   * #include <signal.h> * int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oset); * 功能：读取或更改进程的信号屏蔽字。 * 返回值：若成功则为0，若出错则为-1 * 如果oset是非空指针，则读取进程的当前信号屏蔽字通过oset参数传出。如果set是非空指针，则更改进程的信号屏蔽字，参数how指示如何更改。如果oset和set都是非空指针，则先将原来的信号屏蔽字备份到oset里，然后根据set和how参数更改信号屏蔽字。假设当前的信号屏蔽字为mask，下表说明了how参数的可选值。   how含义    说明：SIG\_BLOCK ，讲信号集set添加到进程block状态字中。 |
| sigpending获取信号未决状态字（pending）信息  NAME  sigpending - examine pending signals  SYNOPSIS  #include <signal.h>  int sigpending(sigset\_t \*set);  DESCRIPTION  sigpending() returns the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). The mask of pending signals is returned in set. |

### 信号阻塞未达编程实践

|  |
| --- |
| 综合实验，  SIGINT信号未设置阻塞，查看未决关键字  SIGINT信号设置阻塞，查看未决关键字  SIGINT信号解除阻塞，查看未决关键字（通过ctrl+\ SIGQUIT 解除阻塞）  //演示信号从产生到抵达的整个过程  //信号的阻塞和解除阻塞综合实验  //设置信号阻塞和非阻塞，设置ctl+q来解除信号阻塞  void handler(int sig)  {  if (sig == SIGINT)  printf("recv a sig=%d\n", sig);  else if (sig == SIGQUIT)  {  sigset\_t uset;  sigemptyset(&uset);  sigaddset(&uset, SIGINT);  //ctr + \ 用来接触 SIGINT 信号  //解除阻塞  sigprocmask(SIG\_UNBLOCK, &uset, NULL);  }  }  void printsigset(sigset\_t \*set)  {  int i;  for (i=1; i<NSIG; ++i)  {  if (sigismember(set, i))  putchar('1');  else  putchar('0');  }  printf("\n");  }  //3 连续的按ctrl+c键盘，虽然发送了多个SIGINT信号，但是因为信号是不稳定的，只保留了一个。  //不支持排队  int main(int argc, char \*argv[])  {  sigset\_t pset; //用来打印的信号集  sigset\_t bset; //用来设置阻塞的信号集    sigemptyset(&bset);  sigaddset(&bset, SIGINT);    if (signal(SIGINT, handler) == SIG\_ERR)  ERR\_EXIT("signal error");    if (signal(SIGQUIT, handler) == SIG\_ERR)  ERR\_EXIT("signal error");  //读取或更改进程的信号屏蔽字 这里用来阻塞ctrl+c信号  //ctrl+c信号被设置成阻塞，即使用户按下ctl+c键盘，也不会抵达  sigprocmask(SIG\_BLOCK, &bset, NULL);    for (;;)  {  //获取未决 字信息  sigpending(&pset);    //打印信号未决 sigset\_t字  printsigset(&pset);  sleep(1);  }  return 0;  } |
| 杀死进程，命令行 |
| [itcast01@localhost ~]$ ps -u itcast01 | grep "dm01" | awk '{print $1}'  1496  [itcast01@localhost ~]$ kill -9 `ps -u itcast01 | grep "dm01" | awk '{print $1}' ` |

### 从linux信号机制编程实践说开去

没有看不懂的代码，只有看不懂的业务

学习linux应用编程、系统编程方法谈，重要的是

1：对各种linux各个机制搭建、深入理解；

2：思想兑现代码！（加强实践，多动手，加强考试！）===》逼迫自己强制使用！

## 10 信号的高级用法

### sigaction函数注册信号处理函数

|  |
| --- |
| **sigaction函数**   * 包含头文件<signal.h> * 功能:sigaction函数用于改变进程接收到特定信号后的行为。 * 原型：   **int sigaction(int signum,const struct sigaction \*act,const struct sigaction \*old);**   * 参数   + 该函数的第一个参数为信号的值，可以为除SIGKILL及SIGSTOP外的任何一 个特定有效的信号（为这两个信号定义自己的处理函数，将导致信号安装错误）   + 第二个参数是指向结构sigaction的一个实例的指针，在结构 sigaction的实例中，指定了对特定信号的处理，可以为空，进程会以缺省方式对信号处理   + 第三个参数oldact指向的对象用来保存原来对相应信号的处理，可指定oldact为NULL。 * 返回值:函数成功返回0，失败返回-1   signal(num., handle) |
| **sigaction结构体**   * 第二个参数最为重要，其中包含了对指定信号的处理、信号所传递的信息、信号处理函数执行过程中应屏蔽掉哪些函数等等   struct sigaction {  void (\*sa\_handler)(int); //信号处理程序 不接受额外数据  void (\*sa\_sigaction)(int, siginfo\_t \*, void \*); //信号处理程序 能接受额外数据，和sigqueue配合使用  sigset\_t sa\_mask; //  int sa\_flags; //影响信号的行为 SA\_SIGINFO表示能接受数据  void (\*sa\_restorer)(void); //废弃  };  注意1：回调函数句柄sa\_handler、sa\_sigaction只能任选其一。  注意2：The sigaction structure is defined as something like 思考如何测试？ |
| **会查找、会用man手册，是通往高手的必经之路。**  The siginfo\_t parameter to sa\_sigaction is a struct with the following elements  siginfo\_t {  int si\_signo; /\* Signal number \*/  int si\_errno; /\* An errno value \*/  int si\_code; /\* Signal code \*/  pid\_t si\_pid; /\* Sending process ID \*/  uid\_t si\_uid; /\* Real user ID of sending process \*/  int si\_status; /\* Exit value or signal \*/  clock\_t si\_utime; /\* User time consumed \*/  clock\_t si\_stime; /\* System time consumed \*/  sigval\_t si\_value; /\* Signal value \*/  int si\_int; /\* POSIX.1b signal \*/  void \* si\_ptr; /\* POSIX.1b signal \*/  void \* si\_addr; /\* Memory location which caused fault \*/  int si\_band; /\* Band event \*/  int si\_fd; /\* File descriptor \*/  } |
| **实验1：**sigaction的函数注册信号，基本用法  void handler(int sig)  {  printf("recv a sig=%d\n", sig);  }  \_\_sighandler\_t my\_signal(int sig, \_\_sighandler\_t handler)  {  struct sigaction act;  struct sigaction oldact;  act.sa\_handler = handler;  sigemptyset(&act.sa\_mask);  act.sa\_flags = 0;  if (sigaction(sig, &act, &oldact) < 0)  return SIG\_ERR;  return oldact.sa\_handler;  }  int main(int argc, char \*argv[])  {  struct sigaction act;  sigset\_t sa\_mask;    act.sa\_handler = handler;  act.sa\_flags = 0;  sigemptyset(&act.sa\_mask);  //测试信号安装函数  //sigaction(SIGINT, &act, NULL);    //模拟signal函数  my\_signal(SIGINT, handler);  for (;;)  {  pause();  }  return 0;  } |
| 实验2：测试sigaction结构体第三个参数sigset\_t sa\_mask的作用  /\*  struct sigaction {  void (\*sa\_handler)(int);  void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);  sigset\_t sa\_mask;  int sa\_flags;  void (\*sa\_restorer)(void);  } \*/  //测试sigaction结构体第三个参数sigset\_t sa\_mask的作用  //作用 sigaddset(&act.sa\_mask, SIGQUIT); 加入到sa\_mask中的信号，被阻塞（信号处理函数执行的过程中被阻塞）。  //注意：SIGQUIT信号最终还会抵达  int main(int argc, char \*argv[])  {  struct sigaction act;  act.sa\_handler = handler;    sigemptyset(&act.sa\_mask);  sigaddset(&act.sa\_mask, SIGQUIT);  act.sa\_flags = 0;  if (sigaction(SIGINT, &act, NULL) < 0)  ERR\_EXIT("sigaction error");  for (;;)  pause();  return 0;  }  void handler(int sig)  {  printf("recv a sig=%d 信号处理函数执行的时候，阻塞sa\_mask中的信号\n", sig);  sleep(5);  } |

### sigqueue新的信号发送函数

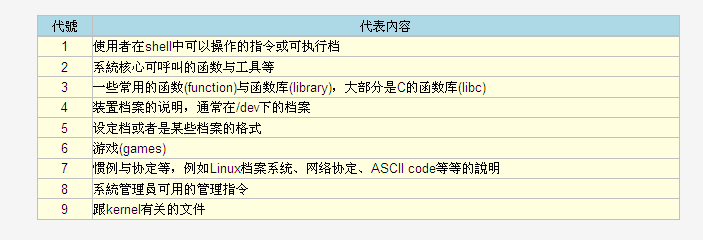
|  |
| --- |
| **sigqueue函数**   * 功能:新的发送信号系统调用,主要是针对实时信号提出的支持信号带有参数,与函数sigaction()配合使用。 * 注意：和kill函数相比Int kill(pid\_t pid, int siq)多了参数 * 原型:   **int sigqueue(pid\_t pid, int sig, const union sigval value);**   * 参数   + sigqueue的第1个参数是指定接收信号的进程id，第2个参数确定即将发送的信号，第3个参数是一个联合数据结构union sigval，指定了信号传递的参数，即通常所说的4字节值。 * 返回值成功返回0,失败返回-1 |
| * sigqueue()比kill()传递了更多的附加信息，但sigqueue()只能向一个进程发送信号，而不能发送信号给一个进程组。 * sigval联合体   typedef union sigval  {  int sival\_int;  void \*sival\_ptr;  }sigval\_t; |

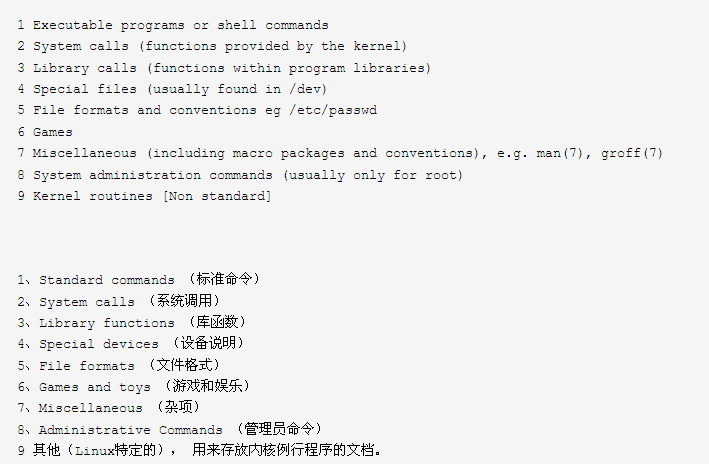
## 11 可靠信号与不可靠信号编程实践

|  |
| --- |
| 可靠信号、不可靠性信号的观察 |
| 综合案例   1. 创建子进程和父进程 2. 注册SIGINT非实时信号 SIGRTMIN 实时信号 添加到到进程阻塞中 3. 注册用户自定义信号SIGUSR1 4. 子进程发送3次非实时信号，发3次实时信号 5. 子进程发送 SIGUSR1解除信号阻塞； 6. 观察实时信号和非实时信号的表现与区别！ |
| man手册sigaction中  struct sigaction {  void (\*sa\_handler)(int);  void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);  sigset\_t sa\_mask;  int sa\_flags;  void (\*sa\_restorer)(void);  }  flags关键字选项很多，能让sigaction函数很强大。。。。  请你仔细阅读man手册看flags意思 |

|  |
| --- |
| 三种睡眠函数 |
| * unsigned int sleep(unsigned int seconds); 秒   若被中断打断，返回剩余时间   * int usleep(useconds\_t usec); 微妙   若被中断打断，返回剩余时间   * int nanosleep(const struct timespec \*req, struct timespec \*rem);纳秒时间   要睡眠的时间req；剩余睡眠时间，如果要中断，通过rem返回过来。 |
| 三种时间结构  time\_t 秒  struct timeval {  long tv\_sec; /\* seconds \*/  long tv\_usec; /\* microseconds \*/微妙  };  struct timespec { 纳秒  time\_t tv\_sec; /\* seconds \*/  long tv\_nsec; /\* nanoseconds \*/  }; |
| setitime函数： 间隙性产生   * 包含头文件<sys/time.h> * 功能setitimer()比alarm功能强大，支持3种类型的定时器 * 原型：   **int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue));**   * 参数   + 第一个参数which指定定时器类型   + 第二个参数是结构itimerval的一个实例，结构itimerval形式   + 第三个参数可不做处理。 * 返回值:成功返回0失败返回-1 * ITIMER\_REAL：经过指定的时间后，内核将发送SIGALRM信号给本进程 * ITIMER\_VIRTUAL ：程序在用户空间执行指定的时间后，内核将发送SIGVTALRM信号给本进程 * ITIMER\_PROF ：进程在内核空间中执行时，时间计数会减少，通常与ITIMER\_VIRTUAL共用，代表进程在用户空间与内核空间中运行指定时间后，内核将发送SIGPROF信号给本进程。 |
| 思考why ITIMER\_PROF存在？ |
| （备注：一般指cpu运行时间，比如，cpu在运行一段时间后，产生信号给本进程。。。。可见，是需要这样的信号）。  没有看不懂的代码，只有看不懂的业务。。 |

## 12 man手册的使用





# linux Socket-应用编程-专题讲座

## Socket编程实践

### 1 Socket Api基本概念

|  |
| --- |
| 什么是socket？   * socket可以看成是用户进程与内核网络协议栈的编程接口。 * socket不仅可以用于本机的进程间通信，还可以用于网络上不同主机的进程间通信。 |
| tcp/ip通讯模型 |
| **IPv4套接口地址结构**   * IPv4套接口地址结构通常也称为“网际套接字地址结构”，它以“sockaddr\_in”命名，定义在头文件<netinet/in.h>中   struct sockaddr\_in {  uint8\_t sin\_len;  sa\_family\_t sin\_family;  in\_port\_t sin\_port;  struct in\_addr sin\_addr;  char sin\_zero[8];  };   * sin\_len：整个sockaddr\_in结构体的长度，在4.3BSD-Reno版本之前的第一个成员是sin\_family. * sin\_family：指定该地址家族，在这里必须设为AF\_INET * sin\_port：端口 * sin\_addr：IPv4的地址； * sin\_zero：暂不使用，一般将其设置为0 |
| **通用地址结构**   * 通用地址结构用来指定与套接字关联的地址。   struct sockaddr {  uint8\_t sin\_len;  sa\_family\_t sin\_family;  char sa\_data[14];  };   * sin\_len：整个sockaddr结构体的长度 * sin\_family：指定该地址家族 * sa\_data：由sin\_family决定它的形式。 |
| **网络字节序**   * 字节序   大端字节序（Big Endian)  最高有效位（MSB：Most Significant Bit）存储于最低内存地址处，最低有效位（LSB：Lowest Significant Bit）存储于最高内存地址处。   * 小端字节序（Little Endian）   最高有效位（MSB：Most Significant Bit）存储于最高内存地址 处，最低有效位（LSB：Lowest Significant Bit）存储于最低内存地址处。   * 主机字节序   不同的主机有不同的字节序，如x86为小端字节序，Motorola 6800为大端字节序，ARM字节序是可配置的。   * 网络字节序   网络字节序规定为大端字节序 |
| **字节序转换函数**   * uint32\_t htonl(uint32\_t hostlong); * uint16\_t htons(uint16\_t hostshort); * uint32\_t ntohl(uint32\_t netlong); * uint16\_t ntohs(uint16\_t netshort); * 说明：在上述的函数中，h代表host；n代表network s代表short；l代表long |
| **地址转换函数**   * #include <netinet/in.h> * #include <arpa/inet.h> * int inet\_aton(const char \*cp, struct in\_addr \*inp); * in\_addr\_t inet\_addr(const char \*cp); * char \*inet\_ntoa(struct in\_addr in); |
| **套接字类型**   * 流式套接字(SOCK\_STREAM)   提供面向连接的、可靠的数据传输服务，数据无差错，无重复的发送，且按发送顺序接收。   * 数据报式套接字(SOCK\_DGRAM)   提供无连接服务。不提供无错保证，数据可能丢失或重复，并且接收顺序混乱。   * 原始套接字（SOCK\_RAW） |
|  |
|  |

### 2 SocketApi基本编程模型

|  |
| --- |
| **TCP客户/服务器模型**        **简单服务器模型** |
|  |

### 3 Socket Api基本实践

#### Socket API基本用法

|  |
| --- |
| **socket函数**   * 包含头文件<sys/socket.h> * 功能：创建一个套接字用于通信 * 原型   + int socket(int domain, int type, int protocol); * 参数   + domain ：指定通信协议族（protocol family）   + type：指定socket类型，流式套接字SOCK\_STREAM，数据报套接字SOCK\_DGRAM，原始套接字SOCK\_RAW   + protocol ：协议类型 * 返回值：成功返回非负整数， 它与文件描述符类似，我们把它称为套接口描述字，简称套接字。失败返回-1 |
| **bind函数**   * 包含头文件<sys/socket.h> * 功能：绑定一个本地地址到套接字 * 原型   + int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen); * 参数   + sockfd：socket函数返回的套接字   + addr：要绑定的地址   + addrlen：地址长度 * 返回值：成功返回0，失败返回-1 |
| **listen函数**   * 一般来说，listen函数应该在调用socket和bind函数之后，调用函数accept之前调用。 * 对于给定的监听套接口，内核要维护两个队列：   + 1、已由客户发出并到达服务器，服务器正在等待完成相应的TCP三路握手过程   + 2、已完成连接的队列 |
| **accept函数**   * 包含头文件<sys/socket.h> * 功能：从已完成连接队列返回第一个连接，如果已完成连接队列为空，则阻塞。 * 原型   + int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen); * 参数   + sockfd：服务器套接字   + addr：将返回对等方的套接字地址   + addrlen：返回对等方的套接字地址长度 * 返回值：成功返回非负整数，失败返回-1   **connect函数**   * 包含头文件<sys/socket.h> * 功能：建立一个连接至addr所指定的套接字 * 原型   + int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen); * 参数   + sockfd：未连接套接字   + addr：要连接的套接字地址   + addrlen：第二个参数addr长度 * 返回值：成功返回0，失败返回-1 |

#### Socket API 中的地址复用

|  |
| --- |
| **SO\_REUSEADDR**   * 服务器端尽可能使用SO\_REUSEADDR * 在绑定之前尽可能调用setsockopt来设置SO\_REUSEADDR套接字选项。   使用SO\_REUSEADDR选项可以使得不必等待TIME\_WAIT状态消失就可以重启服务器  int on = 1;  if (setsockopt(listenfd, SOL\_SOCKET, SO\_REUSEADDR, &on, sizeof(on)) < 0)  ERR\_EXIT("setsockopt"); |

#### Socket服务支持多并发（多客户端连接）

|  |
| --- |
| **分析最基本socket服务器客户端模型能否支持多客户端连接** |
|  |
|  |

#### 点对点聊天程序设计与实现

|  |
| --- |
| 点对点聊天程序，功能说明  点对点聊天程序，设计思想 |
|  |

### 4 Socket Api编程进价

#### 1流协议与粘包

|  |
| --- |
| **流协议与粘包** |
| 粘包产生的原因 |
| 说明  tcp 字节流 无边界  udp 消息、数据报 有边界  对等方，一次读操作，不能保证完全把消息读完。  对方接受数据包的个数是不确定的。 |
| 产生粘包问题的原因  1、SQ\_SNDBUF 套接字本身有缓冲区 （发送缓冲区、接受缓冲区）  2、tcp传送的端 mss大小限制  3、链路层也有MTU大小限制，如果数据包大于>MTU要在IP层进行分片，导致消息分割。  4、tcp的流量控制和拥塞控制，也可能导致粘包  5、tcp延迟发送机制 等等  结论：tcp/ip协议，在传输层没有处理粘包问题。 |
| **粘包解决方案**   * 本质上是要在应用层维护消息与消息的边界   + 定长包   + 包尾加\r\n（ftp）   + 包头加上包体长度   + 更复杂的应用层协议 |
| **编程实践**   * readn * writen |
|  |

#### 2包头加上包体长度编程实践

|  |
| --- |
| **包头加上包体长度**   * + 发报文时，前四个字节长度（转成网络字节序）+包体   + 收报文时，先读前四个字节，求出长度；根据长度读数据。 |
|  |

#### 3包尾加上\n编程实践

|  |
| --- |
| **\n作为协议的边界**   * ssize\_t recv(int s, void \*buf, size\_t len, int flags); * 与read相比，只能用于套接字文件描述符； * 多了一个flags   MSG\_OOB  This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at thehead of the normal data queue, and thus this flag cannot be used with such protocols.  带外数据 紧急指针  MSG\_PEEK  This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, asubsequent receive call will return the same data.  可以读数据，不从缓存区中读走，利用此特点可以方便的实现按行读取数据。  一个一个字符的读，方法不好；多次调用系统调用read方法 |
| recv函数会将套接字缓冲区中的内容读出，但不清空，与read函数的区别在此。此函数有一个flag标志位，设为MSG\_PEEK。  send函数会将缓冲区中的内容写入到套接字，也不清空，与write函数的区别在此。  用这两个函数可以先接收或发送缓冲区中的内容，然后再用readn（此时缓冲区中的内容依然存在）与write函数去继续判断换行符/n，对缓冲区内容实现换行输出。 |
| 参考例题： |

#### 4域名服务相关函数

getsockname 获取本地的地址 & getpeername（获取对方主机地址）

|  |
| --- |
| 获取本地的地址（注意是已连接以后的套接字）  NAME  getsockname - get socket name  SYNOPSIS  #include <sys/socket.h>  int getsockname(int s, struct sockaddr \*name, socklen\_t \*namelen); |
| struct sockaddr\_in localaddr;  socklen\_t addrlen = sizeof(localaddr);  //获取本地的地址  if (getsockname(sock, (struct sockaddr\*)&localaddr, &addrlen) < 0)  ERR\_EXIT("getsockname");  printf("ip=%s port=%d\n", inet\_ntoa(localaddr.sin\_addr), ntohs(localaddr.sin\_port)); |
|  |

gethostname（获取主机名）、gethostbyname、gethostbyaddr

|  |
| --- |
| * gethostname、gethostbyname、gethostbyaddr |
| The domain name queries carried out by gethostbyname() and gethostbyaddr() use a combination of any or all of the name server named(8), a broken out  line from /etc/hosts, and the Network Information Service (NIS or YP), depending upon the contents of the order line in /etc/host.conf. The default  action is to query named(8), followed by /etc/hosts.  The hostent structure is defined in <netdb.h> as follows:  struct hostent {  char \*h\_name; /\* official name of host \*/  char \*\*h\_aliases; /\* alias list \*/  int h\_addrtype; /\* host address type \*/  int h\_length; /\* length of address \*/  char \*\*h\_addr\_list; /\* list of addresses \*/  }  #define h\_addr h\_addr\_list[0] /\* for backward compatibility \*/  The members of the hostent structure are: |
| int getlocalip(char \*ip)  {  char host[100] = {0};  if (gethostname(host, sizeof(host)) < 0) //获取主机名  return -1;  struct hostent \*hp;  if ((hp = gethostbyname(host)) == NULL) //根据主机名获取主机的所有IP地址  return -1;  strcpy(ip, inet\_ntoa(\*(struct in\_addr\*)hp->h\_addr));  return 0;  }  int main(void)  {  char host[100] = {0};  if (gethostname(host, sizeof(host)) < 0)  ERR\_EXIT("gethostname");    printf("\n\n\nhost:%s \n", host);  struct hostent \*hp;  if ((hp = gethostbyname(host)) == NULL)  ERR\_EXIT("gethostbyname");  int i = 0;  while (hp->h\_addr\_list[i] != NULL) //根据主机名获取主机的所有IP地址  {  //char \*inet\_ntoa(struct in\_addr in); 要求填入一个结构体元素  printf("%s\n", inet\_ntoa(\*(struct in\_addr\*)hp->h\_addr\_list[i]));  i++;  }    char ip[16] = {0};  getlocalip(ip);  printf("localip=%s\n", ip);  return 0;  } |

#### 5Socket Api编程进价2

#### 服务器端多进程并发子进程退出问题

|  |
| --- |
| **TCP回射客户/服务器模型总结** |
| **TCP是个流协议**   * TCP是基于字节流传输的，只维护发送出去多少，确认了多少，没有维护消息与消息之间的边界，因而可能导致粘包问题。 * 粘包问题解决方法是在应用层维护消息边界。 |
|  |

**僵进程与SIGCHLD信号**

|  |
| --- |
| **服务器端避免僵尸进程的方法：**  **1）通过忽略SIGCHLD信号，解决僵尸进程**   * signal(SIGCHLD, SIG\_IGN) |
| **2）通过wait方法，解决僵尸进程**   * signal(SIGCHLD, handle\_sigchld); * wait(NULL) |
| **3）通过waitpid方法，解决僵尸进程**   * signal(SIGCHLD, handle\_sigchld); * wait(-1, NULL, WNOHANG) |
|  |
|  |

### 6 Socket Api 与TCP/IP的11种状态

#### TCP/IP协议的11种状态

|  |
| --- |
|  |
| 理解0：什么是主动套接字，什么是被动套接字？  理解1:为什么TCP/IP要三次握手，和四次断开？  理解2：客户端状态向前推进过程，服务器端状态向前推进过程  理解3：执行主动关闭的那一端，进入TIME\_WAIT状态  理解4：TIME\_WAIT 时间是多长2MSL （2倍的最大生命期时间）  原因：（ACK y+1）如果发送失败可以重发。  服务器端处于closed状态，不等于客户端也处于closed状态。。  理解5：图上几种状态，还有一种CLOSING状态  两端同时关闭 将产生closing状态，最后双方都进入TIME\_WAIT状态。 |
|  |
| 实验：   1. 关闭服务方子进程，观察TCP/IP状态 2. 关闭客户端，观察TCP/IP状态。 |

#### SIGPIPE

|  |
| --- |
| 如果对方socket已关闭，对等方再发写数据，则会产生SIGPIPE信号   * SIGPIPE信号会让进程终止（man 7 signal，阅读SIGPIPE默认ACT） * 往一个已经接收FIN的套接中写是允许的，接收到FIN仅仅代表对方不再发送数据。 * 在收到RST段之后，如果再调用write就会产生SIGPIPE信号，对于这个信号的处理我们通常忽略即可。   signal(SIGPIPE, SIG\_IGN); |
|  |
| 结论：对SIGPIPE处理方法：1）忽略该信号即可signal(SIGPIPE, SIG\_IGN);  2）捕捉。改变默认行为。 |
| TCP/IP 的RST段重置  1）服务器端启动、客户端启动  2）服务器端先kill与客户端通讯的子进程，服务器端会给客户端发送FIN分节  此时：只代表服务器端不发送数据了，不能代表客户端不能往套接字中写数据。  3）如果子进程此时写数据给服务器端（解除屏幕阻塞，输入字符aaaa），  将要导致TCP/IP协议重置，产生RST段；产生SIGPIPE信号。。  4）所以，一般情况下，需要我们处理SIGPIPE信号，忽略即可。 |

#### close与shutdown区别

|  |
| --- |
| * close终止了数据传送的两个方向。 * shutdown可以有选择的终止某个方向的数据传送或者终止数据传送的两个方向。 * shutdown how=1就可以保证对等方接收到一个EOF字符，而不管其他进程是否已经打开了套接字。而close不能保证，直到套接字引用计数减为0时才发送。也就是说直到所有的进程都关闭了套接字。 |
| 思考1  客户端向服务器发送：FIN(close) E D C B A，  问：服务器还能收到数据吗？服务器还可以向客户端回报文吗？  客户端想在关闭之后，仍然能接收到回射服务器应答（shutdown）。  思考2  父进程中close(conn);会不会向客户端发送FIN报文段那？  文件的引用计数-1，当减少为0，才会发送引用计数。  思考3：  客户端//shutdown(sock, SHUT\_WR);只关闭了写； |

### 7五种I/O模型

|  |
| --- |
| **阻塞I/O**    说明1：当上层应用app1调用recv系统调用时，如果对等方没有发送数据（缓冲区没有数据），上层应用app1将阻塞（默认行为，被linux内核阻塞）；  说明2：当对等方发送了数据，linux内核recv端缓冲区，有数据后，内核会把数据copy给用户空间。然后上层应用app1解除阻塞，执行下一步操作。 |
| **非阻塞I/O**    说明1： 上层应用程序app2将套接字设置成非阻塞模式。  说明2： 上层应用程序app2轮询调用recv函数，接受数据。若缓冲区没有数据，上层程序app2不会阻塞，recv返回值为-1，错误码是EWOULDBLOCK。  说明3：上层应用程序不断轮询有没有数据到来。会造成上层应用忙等待。大量消耗CPU。很少直接用。应用范围小，一般和selectIO复用配合使用。 |
|  |
| **I/O复用**    说明1： 上层应用程序app3调用select机制（该机制有linux内核支持，避免了app3忙等待。），进行轮询文件描述符的状态变化。  说明2：当select管理的文件描述符没有数据（或者状态没有变化时），上层应用程序app3也会阻塞。  说明3：好处select机制可以管理多个文件描述符  说明4：select可以看成一个管理者，用select来管理多个IO。  一旦检测到的一个I/O或者多个IO，有我们感兴事件，发生，select函数将返回，返回值为检测到的事件个数。进而可以利用select相关api函数，操作具体事件。  说明5：select函数可以设置等待时间，避免了上层应用程序app3，长期僵死。  说明6: 和阻塞IO模型相比，selectI/O复用模型相当于提前阻塞了。等到有数据到来时，再调用recv就不会发生阻塞。 |
| **信号驱动I/O**    说明1： 上层应用程序app4建立SIGIO信号处理程序。当缓冲区有数据到来，内核会发送信号告诉上层应用程序app4。  说明2：上层应用程序app4接收到信号后，调用recv函数，因缓冲区有数据，recv函数一般不会阻塞。  说明3：这种用于模型用的比较少，属于典型的“拉模式”。即：上层应用app4,需要调用recv函数把数据拉进来。 |
| **异步I/O**    说明1：上层应用程序app5调用aio\_read函数，同时提交一个应用层的缓冲区buf；调用完毕后，不会阻塞。上层应用程序app5可以继续其他任务。  说明2：当tcpip协议缓冲区有数据时，linux主动的把内核数据copy到用户空间。然后再给上层应用app5发送信号；告诉app5数据有了，赶快处理吧！  说明3：典型的“推模式”  说明4： 效率最高的一种形式，上层应用程序app5有异步处理的能力（在linux内核的支持下，言外之意：处理其他任务的同时，也可支持IO通讯）。异步I/O指的是什么？  上层应用程序app5，在也可以干别的活的时，可以接收数据（接受异步通信事件。===）异步命令来源）。与信号驱动IO模型，上层应用程序app5不需要调用recv函数。 |
| **结论：IO复用和异步IO是重点。** |
|  |
|  |

### 8 I/O复用模型

#### select基本概念

|  |
| --- |
| 其他重要概念补充   * 阻塞IO   + 数据没有准备好, 读操作就会阻塞   + 数据不能立即被收时, 写操作就会阻塞   + 打开文件时阻塞, 直到某些条件发生 * 非阻塞IO   + 立即返回, 并用错误值来表示当前的状态 * 指定非阻塞方式   + 打开时指定O\_NONBLOCK 标志   + 使用fcntl 打开或关闭非阻塞方式 * 网络编程时, 使用非阻塞, 用轮询方式发送 * 使用多线程可以避免使用非阻塞IO, 但是同步开销较大 |
| 多路IO   * 当程序需要同时从两个输入读数据时 * 使用多进程/多线程, 同步复杂, 进程线程开销 * 使用非阻塞IO, 交替轮询 * 通过信号使用异步IO, 无法判断哪个IO完成 * 多路IO: 把关心的IO放入一个列表, 调用多路函数 * 多路IO函数阻塞, 直到有一个IO数据准备好后返回 * 返回后告诉调用者哪个描述符准备好了 |
| select()实现说明   * 调用select时通过参数告诉内核用户感兴趣的IO描述符 * 关心的IO状态: 输入,输出或错误 * 调用者等待时间 * 返回之后内核告诉调用者多个描述符准备好了 * 哪些描述符发生了变化 * 调用返回后对准备好的描述符调用读写操作 * 不关心的描述符集合传NULL |
|  |
|  |
| **select()**  /\* According to POSIX 1003.1-2001 \*/  #include <sys/select.h>  /\* According to earlier standards \*/  #include <sys/time.h>  #include <sys/types.h>  #include <unistd.h>  int **select**(int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);  如果成功，返回所有sets中描述符的个数；如果超时，返回0；如果出错，返回-1。 |
| * 监视readfds来查看是否read的时候会被堵塞，注意，即便到了end-of-file，fd也是可读的。 * 监视writefds看写的时候会不会被堵塞。 * 监视exceptfd是否出现了异常。主要用来读取OOB数据，异常并不是指出错。 * 注意当一个套接口出错时，它会变得既可读又可写。 * 如果有了状态改变，会将其他fd清零，只有那些发生改变了的fd保持置位，以用来指示set中的哪一个改变了状态。 * 参数n是所有set里所有fd里，具有最大值的那个fd的值加1 |
| **fd\_set**  四个宏用来对fd\_set进行操作：  FD\_CLR(int fd, fd\_set \*set);  FD\_ISSET(int fd, fd\_set \*set);  FD\_SET(int fd, fd\_set \*set);  FD\_ZERO(fd\_set \*set);   * FD\_ZERO用来清空set； * FD\_SET和FD\_CLR用来对某个set添加和删除一个fd； * FD\_ISSET用来指示一个fd是不是一个set的一部分。他很有用，用来看select后哪一个fd可用了。 |
| **time\_out**   * timeout是从调用开始到select返回前，会经历的最大等待时间。 * 两种特殊情况：如果为值为0，会立刻返回。 * 如果timeout是NULL,会阻塞式等待。   struct timeval {  long tv\_sec; /\* seconds \*/  long tv\_usec; /\* microseconds \*/  };   * 一些调用使用3个空的set, n为zero, 一个非空的timeout来达到较为精确的sleep. * Linux中, select函数改变了timeout值，用来指示还剩下的时间，但很多实现并不改timeout。 * 为了较好的可移植性，timeout在循环中需要被重新赋初值。 * timeout== NULL   + 无限等待   + 被信号打断时返回1, errno 设置成 EINTR * timeout->tv\_sec == 0 && tvptr->tv\_usec == 0   + 不等待立即返回 * timeout->tv\_sec != 0 || tvptr->tv\_usec != 0   + 等待特定时间长度, 超时返回0 |
| **select示例**  man 手册 |

select实现原理说明

|  |
| --- |
| * fd\_set是一个位向量, 每位表示一个描述符      * int FD\_ISSET(int fd, fd\_set \*fdset);   + 测试某个描述符是否在集合内 * void FD\_CLR(int fd, fd\_set \*fdset);   + 从集合内把一个描述符移除 * void FD\_SET(int fd, fd\_set \*fdset);   + 把一个描述符加入集合 * void FD\_ZERO(fd\_set \*fdset);   + 清空描述符集合 |
| * 可以把同一个描述符同时放取读和写集合 * 当读和写者准备好时, 返回值的计数分别加1次 * 普通文件的三种状态总是返回准备好的状态 * 是否阻塞式IO不会影响select的结果 * 如果一个描述符到了文件结尾,select返回的状态是准备好 * 对一个准备好的描述符, 读出长度是0表示到达结尾 |
|  |

#### 简单select语法

#### 用select完善echoclient

#### 用select完善echoserver

#### 套接字IO超时设置方法

|  |
| --- |
| **套接字I/O超时设置方法**   * **alarm** * **套接字选项**    + **SO\_SNDTIMEO**   + **SO\_RCVTIMEO** * **select** |
| **闹钟方法alarm**  方案1  void handler(int sig)  {  return 0; //闹钟仅用来做打断read操作，无其他用途  }  //设置一个闹钟，5秒钟以后打断read函数，达到超时的目的  SIGALRM  signal(SIGALRM, handle);  alarm(5)  int ret = read(fd, buf, sizeof(buf));  if (ret==-1 && errno=EINTR)  {  errno = ETIMEDOUT;  }  else if (ret >=0)  {  alarm(0)';  } |
| **套接字选项**  SO\_SNDTIME 发送的时间  SO\_RCVTIME 接受的时间  setsockopt (sock, SOL\_SOCKET, SO\_RCVTIMEO, 5);  if (ret==-1 && errno=EWOULDBLOCK)  {  errno = ETIMEDOUT;  }  缺点：不易移植，有些内核没有实现。 |
| **select**   * **用select实现超时**   + **read\_timeout函数封装**   + **write\_timeout函数封装**   + **accept\_timeout函数封装**   + **connect\_timeout函数封装** |

### 9 POLL模型

|  |
| --- |
| POLL基本概念   * int poll(struct pollfd fdarray[ ], nfds\_t nfds, int timeout); * 所有文件描述符放在一个数组里, nfds是长度 |
| POLL参数   * POLLIN 优先不低于POLLRDNORM|POLLRDBAND的数据可读 * POLLRDNORM 普通优先级数据可读 * POLLRDBAND * POLLPRI 高优先级数据可读. * POLLOUT 普通优先级数据可写 * POLLWRBAND * POLLERR 出错 * POLLHUP 挂起, 目前还有数据等待操作 * POLLNVAL 描述符不是一个打开的文件 |
| * timeout == -1 永久等待 * timeout == 0 不等待 * timeout > 0 等待的毫秒数 |

### 10 socket性能测试

## UDP编程实践

|  |
| --- |
| **UDP的特点**   * 无连接 * 基于消息的数据传输服务 * 不可靠 * 一般情况下UDP更加高效 |
| **UDP客户/服务基本模型** |
| **UDP注意点**   * UDP报文可能会丢失、重复 * UDP报文可能会乱序 * UDP缺乏流量控制   udp缓冲区写满以后，没有流量控制机制，会覆盖缓冲区。   * UDP协议数据报文截断   如果接收到的数据报，大于缓冲区；报文可以被截断；后面的部分会丢失。   * recvfrom返回0，不代表连接关闭，因为udp是无连接的。   sendto可以发送数据0包。。。只含有udp头部。   * ICMP异步错误   观察现象： 关闭udp服务端，若启动udp客户端，从键盘接受数据后，再发送数据。udp客户端阻塞在sendto位置；  //说明1：udp发送报文的时，只把数据copy到发送缓冲区。在服务器没有起来的情况下，可以发送成功。  说明2：所谓ICMP异步错误是指：发送的报文的时候，没有错误，接受报文recvfrom的时候，回收到ICMP应答  说明3：异步的错误，是无法返回未连接的套接字。udp也可以调用connect   * UDP connect   说明1：//udp调用connet，并没有三次握手，只是维护了一个状态信息（和对等方的）。。。  说明2）//一但调用connect，就可以使用send函数 |
| 结论：客户端调用connet和不调connet的区别。   1. udp也可以调用connet 2. udp客户端调用了connect以后，不会阻塞在recvfrom函数这里。 3. 一但调用connect，就可以使用send函数 4. UDP协议数据报文截断   如果接收到的数据报，大于缓冲区；报文可以被截断；后面的部分会丢失。 |
| **实验1**  //数据报方式。。。。不是字节流  //如果接受数据时，指定的缓冲区的大小，较小；  //剩余部分将要截断，扔掉  **实验2：**  sendto可以发送0包，只含头部。 |

linux Socket-应用编程-理论概念-专题讲座

### ISO的开放互联网模型（OSI）

|  |
| --- |
| **TCP回射客户/服务器模型总结**   * OSI（open system interconnection）开放系统互联模型是由ISO（International Organization for Standardization）国际标准化组织定义的网络分层模型，共七层。 |
| * 物理层(Physical Layer)：物理层定义了所有电子及物理设备的规范，为上层的传输提供了一个物理介质，本层中数据传输的单位为比特（bit）。属于本层定义的规范有EIA/TIA RS-232、EIA/TIA RS-449、V.35、RJ-45等，实际使用中的设备如网卡等属于本层。 * 数据链路层（Data Link Layer）：对物理层收到的比特流进行数据成帧。提供可靠的数据传输服务，实现无差错数据传输。在数据链路层中数据的单位为帧（frame）。属于本层定义的规范有SDLC、HDLC、PPP、STP、帧中继等，实际使用中的设备如switch交换机属于本层。 * 网络层（Network Layer）：网络层负责将各个子网之间的数据进行路由选择，分组与重组。本层中数据传输的单位为数据包（packet）。属于本层定义的规范有IP、IPX、RIP、OSPF、ICMP、IGMP等。实际使用中的设备如路由器属于本层。 |
| * 传输层（Transport Layer）：提供可靠的数据传输服务，它检测路由器丢弃的包，然后产生一个重传请求，能够将乱序收到的数据包重新排序。 * 会话层（Session Layer）：管理主机之间会话过程，包括会话建立、终止和会话过程中的管理。 * 表示层（Presentation Layer）：表示层对网络传输的数据进行变换，使得多个主机之间传送的信息能够互相理解，包括数据的压缩、加密、格式转换等。 * 应用层（Application Layer）：应用层与应用程序界面沟通，以达至展示给用户的目的。 在此常见的协定有: HTTP，HTTPS，FTP，TELNET，SSH，SMTP，POP3等 * 结论：传输层能做差错控制。。对IP包进行排序 IP路由   **OSI和邮件发送** |

### TCP/IP四层模型

|  |
| --- |
| **和OSI七层的不同**    **TCP/IP主要协议**    **运行FTP的两台主机** |
| **通过路由器连接的网络**     * **（对等通信、封装、分用、端口）**   **对等通讯**    **封装（Encapsulation）**    **分用（Demultiplexing,解封）**    **端口**   * 众所周知端口（Well Known Ports）：从0到1023，这些端口由IANA分配和控制它们紧密绑定于一些服务。通常这些端口的通讯明确表明了某种服务的协议。例如：21端口为ftp服务端口。 * 注册端口（Registered Ports）：从1024到49151。它些端口不受IANA控制，但由IANA登记并提供使用情况清单。它们松散地绑定于一些服务。也就是说有许多服务绑定于这些端口，这些端口同样用于许多其它目的。例如：1433 Microsoft SQL服务端口 * 动态或私有端口（Dynamic or Private Ports）：从49152到65535。IANA不管这些端口。实际上，机器通常从1024起分配动态端口。但也有例外：SUN的RPC端口从32768开始。 |

### TCP/IP各层报文

#### 以太网帧格式

|  |
| --- |
| 说明1：链路层数据包，称为以太网帧。  说明2：链路层不识别IP地址（逻辑地址，公网IP），链路层识别物理网卡MAC地址。  说明3：需要根据IP地址找到对方的MAC地址。这个过程称为：地址解析，也就是ARP协议。（MAC—）IP地址方向地址解析 RARP）。  说明4：应用层根据对等方的IP地址进行通讯，在数据封装过程中，链路层有需要目的地址的MAC网卡地址？怎么样解决这个问题？ 需要将IP地址转换成MAC地址，也就是地址解析。 |
| ARP地址解析协议 |
|  |

#### 以太网的MTU和路径MTU

|  |
| --- |
| 以太网MTU   * 以太网和IEEE 802.3对数据帧的长度都有限制，其最大值分别是1500和1492字节，将这个限制称作最大传输单元（MTU，Maximum Transmission Unit） * 如果IP层有一个数据报要传，而且数据的长度比链路层的MTU还大，那么IP层就要进行分片（Fragmentation），把数据报分成若干片，这样每一片都小于MTU。 * 当网络上的两台主机互相进行通信时，两台主机之间要经过多个网络，每个网络的链路层可能有不同的MTU，其中两台通信主机路径中的最小MTU被称作路径MTU。 |

#### IP数据报

|  |
| --- |
|  |
| * 版本号，目前取值4 * 首部长度，4个字节为单位，取值范围5～15 * 服务类型，指定传输的优先级、传输速度、可靠性和吞吐量等 * 报文总长度，最大长度为65535字节 * 报文标识，唯一标识一个数据报，如果数据报分段，则每个分段的标识都一样 * 标志，最高位未使用，定义为0，其余两位为DF（不分段）和MF（更多分段） * 段偏移量，以8个字节为单位，指出该分段的第一个数据字在原始数据报中的偏移位置 |
| * 生存时间，取值0～255，每经过一个路由节点减1，为0时被丢弃 * 协议，指明该数据报的协议类型，如1为ICMP，4为IP，6为TCP，17为UDP等 * 首部校验和，每通过一次网关都要重新计算该值，用于保证IP首部的完整性 * 选项，长度可变，提供某些场合下需要的控制功能，IP首部的长度必须是4个字节的整数倍，如果选项长度不是4的整数倍，必须填充数据 |

#### ICMP协议

|  |
| --- |
| ICMP协议说明   * ICMP协议用于传递差错信息、时间、回显、网络信息等控制数据 * 当你ping 一台主机想看它是否运行时，就产生了一条ICMP 信息。远程主机将用它自己的ICMP 信息对ping 请求作出回应。   **ICMP协议**   * 本身是IP的一部分。 * 用途:网关或者目标机器利用ICMP与源通讯，当出现问题时，提供反馈信息 * 在IP协议栈中必须实现 * 特点：   + 其控制能力并不用于保证传输的可靠性   + 它本身也不是可靠传输的 * ICMP包含在IP数据包的净荷数据中，IP头中协议类型为1 * ICMP数据的第一个字节代表ICMP报文的类型，它决定了后续数据的格式   **ICMP数据包**   * ICMP包含在IP数据包的净荷数据中，IP头中协议类型为1 * ICMP数据的第一个字节代表ICMP报文的类型，它决定了后续数据的格式       **ICMP报文类型**      报文类型说明   * 0 Echo Reply * 3 Destination Unreachable * 4 Source Quench * 5 Redirect * 8 Echo * 11 Time Exceeded * 12 Parameter Problem * 13 Timestamp * 14 Timestamp Reply * 15 Information Request * 16 Information Reply * 17 Address Mask Request * 18 Address Mask Reply   报文分类   * ICMP差错报文   + 目的不可达报文（类型3）   + 超时报文（类型11）   + 参数出错报文（类型12） * ICMP控制报文   + 报源抑制报文（类型4）   + 重定向（类型5） * ICMP请求/应答报文   + 回送请求和响应报文（类型0和8）   + 时间戳请求和响应报文（类型13和14）   + 地址掩码请求和响应报文（类型17和18） * 其他 |
| **ICMP Echo报文**     * 类型：0表示Echo Reply，8表示Echo * 代码：0 * 标识符：标识一个会话，例如，用进程ID * 序号：例如每个请求增1 * 选项数据：回显 |
| **ICMP 目的不可达报文**     * 类型：3 * 代码：0表示网络不可达，1表示主机不可达； 2表示协议不可达；3表示端口不可达；等等 * 出错的IP包的IP首部+原始IP数据包中前8个字节 |

**UDP数据报**

|  |
| --- |
|  |

#### TCP报文段

|  |
| --- |
| **协议描述**     * 源端口号和目的端口号：源和目的主机的IP地址加上端口号构成一个TCP连接 * 序号和确认号：序号为该TCP数据包的第一个数据字在所发送的数据流中的偏移量；确认号为希望接收的下一个数据字的序号； * 首部长度，以4个字节为单位，通常为20个字节 * 6个标志位：   + URG：如果使用了紧急指针，URG置1，紧急指针为当前序号到紧急数据位置的偏移量   + ACK：为1表示确认号有效，为0表示该TCP数据包不包含确认信息   + PSH：表示是带有PUSH标志的数据，接收到数据后不必等缓冲区满再发送 |
| * + RST：用于连接复位，也可用于拒绝非法的数据或拒绝连接请求   + SYN：用于建立连接，连接请求时SYN＝1，ACK=0；响应连接请求时SYN=1，ACK=1   + FIN：用于释放连接，表示发送方已经没有供发送的数据 * 窗口大小：表示在确认字节后还可以发送字节数，用于流量控制 * 校验和：覆盖了整个数据包，包括对数据包的首部和数据 * 选项：常见的选项是MSS(Maximum Segment Size) |
| **连接建立三次握手** |
| **连接终止四次握手** |
| **TCP如何保证可靠性**   * 应用数据被分割成TCP认为最适合发送的数据块，称为段传递给IP层。 * 当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。 * 当TCP收到发自TCP连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒。 * TCP将保持它首部和数据的校验和。这是一个端到端的校验和，目的是检测数据在传输过程中的任何变化。如果收到段的校验和有差错，TCP将丢弃这个报文段并且不确认（导致对方超时重传） * TCP承载于IP数据报来传输，而IP数据报的到达可能会失序，因此TCP报文段的到达也可能会失序。TCP将对收到的数据进行重新排序。 * IP数据报会发生重复，TCP的接收端必须丢弃重复的数据。 * TCP还能提供流量控制。TCP连接的每一方都有一定大小的缓冲空间。 |

#### 综合案例

TCP/IP协议栈（linux内核中实现）、网卡驱动、网卡；

|  |
| --- |
| **数据在网络中传输过程**     * 步骤a：应用程序ping会判断发送的是主机名还是IP地址，调用函数gethostbyname()解析主机机B，将主机名转换成一个32位的IP地址。这个过程叫做DNS域名解析 * 步骤b：ping程序向目的IP地址发送一个ICMP的ECHO包 * 步骤c：将目标主机的IP地址转换为48位硬件地址，在局域网内发送ARP请求广播，查找主机B的硬件地址。 * 步骤d：主机B的ARP协议层接收到主机A的ARP请求后，将本机的硬件地址填充到应答包，发送ARP应答到主机A。 * 步骤e：发送ICMP数据包到主机B * 步骤f：主机B接收到主机A的ICMP包，发送响应包。 * 步骤g：主机A接收到主机B的ICMP包响应包。 |

#### 其他补充

|  |
| --- |
| **主要协议及其关系**    **TCP/IP协议栈** |
|  |

# linux进程间通信介绍

## 进程同步与进程互斥

|  |
| --- |
| **顺序程序与并发程序特征**   * 顺序程序特征 * 顺序性 * 封闭性：（运行环境的封闭性） * 确定性 * 可再现性 * 并发程序特征 * 共享性 * 并发性 * 随机性 |
| **进程互斥**   * 由于各进程要求共享资源，而且有些资源需要互斥使用，因此各进程间竞争使用这些资源，进程的这种关系为进程的互斥 * 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源。 * 在进程中涉及到互斥资源的程序段叫临界区 |
| **互斥示例** |
| **进程同步**   * 进程同步指的是多个进程需要相互配合共同完成一项任务。 |
| **同步示例** |
| **进程间通信目的**   * 数据传输：一个进程需要将它的数据发送给另一个进程 * 资源共享：多个进程之间共享同样的资源。 * 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。 * 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。 |

## 进程间通信发展

|  |
| --- |
| * 管道 * System V进程间通信 * POSIX进程间通信 |
| * Linux下的进程通信手段基本上是从UNIX平台上的进程通信手段继承而来的。而对UNIX发展做出重大贡献的两大主力AT&T的贝尔实验室及BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间的通信方面的侧重点有所不同。前者是对UNIX早期的进程间通信手段进行了系统的改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内；后者则跳过了该限制，形成了基于套接口（socket）的进程间通信机制。而Linux则把两者的优势都继承了下来 |
| • UNIX进程间通信（IPC）方式包括管道、FIFO以及信号。  • System V进程间通信（IPC）包括System V消息队列、System V信号量以及System V共享内存区。  • Posix 进程间通信（IPC）包括Posix消息队列、Posix信号量以及Posix共享内存区。 |
| **有关POSIX**  POSIX 表示可移植操作系统接口（Portable Operating System Interface ，缩写为 POSIX ），POSIX标准定义了操作系统应该为应用程序提供的接口标准，是IEEE为要在各种UNIX操作系统上运行的软件而定义的一系列API标准的总称，其正式称呼为IEEE 1003，而国际标准名称为ISO/IEC 9945。  POSIX标准意在期望获得源代码级别的软件可移植性。换句话说，为一个POSIX兼容的操作系统编写的程序，应该可以在任何其它的POSIX操作系统（即使是来自另一个厂商）上编译执行。  POSIX 并不局限于 UNIX。许多其它的操作系统，例如 DEC OpenVMS 支持 POSIX 标准，尤其是 IEEE Std. 1003.1-1990（1995 年修订）或 POSIX.1，POSIX.1 提供了源代码级别的 C 语言应用编程接口（API）给操作系统的服务程序，例如读写文件。POSIX.1 已经被国际标准化组织（International Standards Organization，ISO）所接受，被命名为 ISO/IEC 9945-1:1990 标准。  **有关POSIX2**  POSIX的诞生和Unix的发展是密不可分的，电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）最初开发 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。Unix于70年代诞生于贝尔实验室，并于80年代向美各大高校分发V7版的源码以做研究。加利福尼亚大学伯克利分校在V7的基础上开发了BSD Unix。后来很多商业厂家意识到Unix的价值也纷纷以贝尔实验室的System V或BSD为基础来开发自己的Unix，较著名的有Sun OS，AIX，VMS。  然而，POSIX 并不局限于 UNIX。许多其它的操作系统，例如 DEC OpenVMS 支持 POSIX 标准，尤其是 IEEE Std. 1003.1-1990（1995 年修订）或 POSIX.1，POSIX.1 提供了源代码级别的 C 语言应用编程接口（API）给操作系统的服务程序，例如读写文件。POSIX.1 已经被国际标准化组织（International Standards Organization，ISO）所接受，被命名为 ISO/IEC 9945-1:1990 标准。  POSIX 已发展成为一个非常庞大的标准族，某些部分正处在开发过程中。POSIX 与 IEEE 1003 和 2003 家族的标准是可互换的。  Windows NT-based 系统不能直接支持新版POSIX接口，仅支持第一版POSIX v1:ISO/IEC9945-1:1990[1] 版标准。由于仅支持第一版POSIX的Windows NT-based系统不能创建符合POSIX接口标准的线程和窗体、套接字，所以微软公司提供POSIX兼容层Microsoft POSIX subsystem软件包（Windows Services for UNIX）以支持新版POSIX接口，Windows 系统还可以运行其他POSIX兼容层例如Cygwin. |
| **进程间通信分类**   * 文件 * 文件锁 * 管道（pipe）和有名管理（FIFO） * 信号（signal） * 消息队列 * 共享内存 * 信号量 * 互斥量 * 条件变量 * 读写锁 * 套接字（socket）   **System V IPC & POSIX IPC**   * System V IPC   + System V 消息队列   + System V 共享内存   + System V 信号量 * POSIX IPC   + 消息队列   + 共享内存   + 信号量   + 互斥量   + 条件变量   + 读写锁 |
|  |
| **进程间共享信息的三种方式** |
| **IPC对象的持续性**   * 随进程持续：一直存在直到打开的最后一个进程结束。（如pipe和FIFO） * 随内核持续：一直存在直到内核自举或显式删除（如System V消息队列、共享内存、信号量） * 随文件系统持续：一直存在直到显式删除，即使内核自举还存在。（POSIX消息队列、共享内存、信号量如果是使用映射文件来实现） |
| **IPC对象是被linux内核管理** |
|  |

## 进程死锁及解决方案

|  |
| --- |
| **死锁**   * 死锁是指多个进程之间相互等待对方的资源，而在得到对方资源之前又不释放自己的资源，这样，造成循环等待的一种现象。如果所有进程都在等待一个不可能发生的事，则进程就死锁了。 |
| **死锁产生的必要条件**   * 互斥条件   + 进程对资源进行排它性使用，即在一段时间内某资源仅为一个进程所占用。 * 请求和保持条件   + 当进程因请求资源而阻塞时，对已获得的资源保持不放。 * 不可剥夺条件   + 进程已获得的资源在未使用完之前，不能被剥夺，只能在使用完时由自己释放。 * 环路等待条件   + 各个进程组成封闭的环形链，每个进程都等待下一个进程所占用的资源 |
| **防止死锁办法**   * 资源一次性分配：（破坏请求和保持条件） * 可剥夺资源：破坏不可剥夺条件） * 资源有序分配法：（破坏循环等待条件） |
| **死锁避免**   * 预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得较满意的系统性能。 * 由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁算法是银行家算法。 |
| **银行家算法**   * 为保证资金的安全，银行家规定： * (1) 当一个顾客对资金的最大需求量不超过银行家现有的资金时就可接纳该顾客； * (2) 顾客可以分期贷款，但贷款的总数不能超过最大需求量 * (3) 当银行家现有的资金不能满足顾客尚需的贷款数额时，对顾客的贷款可推迟支付，但总能使顾客在有限的时间里得到贷款 * (4) 当顾客得到所需的全部资金后，一定能在有限的时间里归还所有的资金. |
| **哲学家就餐问题**   * 五个哲学家围在一个圆桌就餐，每个人都必须拿起两把叉子才能用餐 * 哲学家就餐问题解法   + 服务生解法   + 最多4个哲学家   + 仅当一个哲学家两边筷子都可用时才允许他拿筷子   + 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之 |
| **信号量**   * 信号量和P、V原语由Dijkstra（迪杰斯特拉） * 信号量   + 互斥：P、V在同一个进程中   + 同步：P、V在不同进程中 * 信号量值含义   + S>0：S表示可用资源的个数   + S=0：表示无可用资源，无等待进程   + S<0：|S|表示等待队列中进程个数 |
| struct semaphore  {  int value;  pointer\_PCB queue;  }  **P原语**  P(s)  {  s.value = s.value--;  if (s.value < 0)  {  该进程状态置为等待状状态  将该进程的PCB插入相应的等待队列s.queue末尾  }  }  **V原语**  V(s)  {  s.value = s.value++;  if (s.value < =0)  {  唤醒相应等待队列s.queue中等待的一个进程  改变其状态为就绪态  并将其插入就绪队列  }  } |
| **用PV原语解决司机与售票员问题** |
| **用PV原语解决民航售票问题** |
| **用PV原语解决汽车租赁问题**   * 有一汽车租赁公司有两部敞篷车可以出租，假定同时来了四个顾客都要租敞篷车，那么肯定会有两个人租不到。   S(2)  P(S)  租车  还车  V(S) |
|  |

# linux消息队列-应用编程-专题讲座

## 1消息队列基本概念

|  |
| --- |
| **消息队列**   * 消息队列提供了一个从一个进程向另外一个进程发送一块数据的方法 * 每个数据块都被认为是有一个类型，接收者进程接收的数据块可以有不同的类型值 * 消息队列也有管道一样的不足，就是每个消息的最大长度是有上限的（MSGMAX），每个消息队列的总的字节数是有上限的（MSGMNB），系统上消息队列的总数也有一个上限（MSGMNI）   对比：  管道：流管道 消息：有边界  先进先出 可以后进入、先出来  消息大小三大限制  cat /proc/sys/kernel/msgmax 最大消息长度 限制  cat /proc/sys/kernel/msgmnb 消息队列总的字节数  cat /proc/sys/kernel/msgmni 消息条目数 |
| **IPC对象数据结构**   * 内核为每个IPC对象维护一个数据结构   struct ipc\_perm {  key\_t \_\_key; /\* Key supplied to xxxget(2) \*/  uid\_t uid; /\* Effective UID of owner \*/  gid\_t gid; /\* Effective GID of owner \*/  uid\_t cuid; /\* Effective UID of creator \*/  gid\_t cgid; /\* Effective GID of creator \*/  unsigned short mode; /\* Permissions \*/  unsigned short \_\_seq; /\* Sequence number \*/  };  struct msqid\_ds {  struct ipc\_perm msg\_perm; /\* Ownership and permissions \*/  time\_t msg\_stime; /\* Time of last msgsnd(2) \*/  time\_t msg\_rtime; /\* Time of last msgrcv(2) \*/  time\_t msg\_ctime; /\* Time of last change \*/  unsigned long \_\_msg\_cbytes; /\* Current number of bytes in  queue (nonstandard) \*/  msgqnum\_t msg\_qnum; /\* Current number of messages  in queue \*/  msglen\_t msg\_qbytes; /\* Maximum number of bytes  allowed in queue \*/  pid\_t msg\_lspid; /\* PID of last msgsnd(2) \*/  pid\_t msg\_lrpid; /\* PID of last msgrcv(2) \*/  }; |
| **消息队列在内核中的表示** |
| **消息队列函数**   * #include <sys/types.h> * #include <sys/ipc.h> * #include <sys/msg.h> * int msgget(key\_t key, int msgflg); * int msgctl(int msqid, int cmd, struct msqid\_ds \*buf); * int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg); * ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg); |
| **msgget函数**   * 功能：用来创建和访问一个消息队列 * 原型   + int msgget(key\_t key, int msgflg); * 参数   + key: 某个消息队列的名字   + msgflg:由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的 * 返回值：成功返回一个非负整数，即该消息队列的标识码；失败返回-1 |
| **msgget函数参数关系图** |
| **msgctl函数**   * 功能：消息队列的控制函数 * 原型   + int msgctl(int msqid, int cmd, struct msqid\_ds \*buf); * 参数   + msqid: 由msgget函数返回的消息队列标识码   + cmd:是将要采取的动作,（有三个可取值） * 返回值：成功返回0，失败返回-1   **cmd:将要采取的动作（有三个可取值），分别如下：** |

## 2消息队列基本API

### 消息队列的创建及控制

|  |
| --- |
| **消息队列函数**   * #include <sys/types.h> * #include <sys/ipc.h> * #include <sys/msg.h> * int msgget(key\_t key, int msgflg); * int msgctl(int msqid, int cmd, struct msqid\_ds \*buf); * int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg); * ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg); |
| **msgget函数**   * 功能：用来创建和访问一个消息队列 * 原型   + int msgget(key\_t key, int msgflg); * 参数   + key: 某个消息队列的名字   + msgflg:由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的 * 返回值：成功返回一个非负整数，即该消息队列的标识码；失败返回-1 |
| **msgget函数参数关系图** |
| **msgctl函数**   * 功能：消息队列的控制函数 * 原型   + int msgctl(int msqid, int cmd, struct msqid\_ds \*buf); * 参数   + msqid: 由msgget函数返回的消息队列标识码   + cmd:是将要采取的动作,（有三个可取值） * 返回值：成功返回0，失败返回-1   **cmd:将要采取的动作（有三个可取值），分别如下：** |

### 消息队列的发送和接受

|  |
| --- |
| **msgsnd函数**   * 功能：把一条消息添加到消息队列中 * 原型   + int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg); * 参数   + msgid: 由msgget函数返回的消息队列标识码   + msgp:是一个指针，指针指向准备发送的消息，   + msgsz:是msgp指向的消息长度，这个长度不含保存消息类型的那个long int长整型   + msgflg:控制着当前消息队列满或到达系统上限时将要发生的事情 * 返回值：成功返回0；失败返回-1 * msgflg=IPC\_NOWAIT表示队列满不等待，返回EAGAIN错误。 * 消息结构在两方面受到制约。首先，它必须小于系统规定的上限值；其次，它必须以一个long int长整数开始，接收者函数将利用这个长整数确定消息的类型 * 消息结构参考形式如下：   struct msgbuf {  long mtype;  char mtext[100];  } |
| **msgrcv函数**   * 功能：是从一个消息队列接收消息 * 原型   + ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg); * 参数   + msgid: 由msgget函数返回的消息队列标识码   + msgp:是一个指针，指针指向准备接收的消息，   + msgsz:是msgp指向的消息长度，这个长度不含保存消息类型的那个long int长整型   + msgtype:它可以实现接收优先级的简单形式   + msgflg:控制着队列中没有相应类型的消息可供接收时将要发生的事 * 返回值：成功返回实际放到接收缓冲区里去的字符个数，失败返回-1 |
| * msgtype=0返回队列第一条信息 * msgtype>0返回队列第一条类型等于msgtype的消息 * msgtype<0返回队列第一条类型小于等于msgtype绝对值的消息，并且是满足条件的消息类型最小的消息 * msgflg=IPC\_NOWAIT，队列没有可读消息不等待，返回ENOMSG错误。 * msgflg=MSG\_NOERROR，消息大小超过msgsz时被截断 * msgtype>0且msgflg=MSG\_EXCEPT，接收类型不等于msgtype的第一条消息。 |
|  |

## 3消息队列综合api使用

|  |
| --- |
| 同一个进程，使用消息队列 |
| struct msg\_buf  {  long mtype;  char data[255];  };    /\* 注意long 和 int 在32bit 和 64bit系统之下是不一样的  struct msg\_buf  {  long mtype;  char data[255];  };  \*/    int main()  {  key\_t key;  int msgid;  int ret;  struct msg\_buf msgbuf;  int msgtype = getpid();  key=ftok("./msgfile",'a');  printf("key =[%x]\n",key);    printf("sizeof(long):%ld, sizeof(int):%d \n", sizeof(long), sizeof(int));    msgid=msgget(key, IPC\_CREAT |IPC\_EXCL|0666); //通过文件对应  if(msgid==-1)  {  if (errno == EEXIST)  {  printf("EEXIST:.....\n");  key=ftok("./msgfile",'a');  msgid=msgget(key, IPC\_CREAT|0666); //通过文件对应  }  else  {  printf("create error\n");  perror("msget: \n");  return -1;  }    }  printf("msgid:%d \n", msgid);  msgbuf.mtype = msgtype; // getpid();  printf("getpid(): %d \n", getpid());  strcpy(msgbuf.data,"test haha");  ret = msgsnd(msgid,&msgbuf, sizeof(msgbuf.data), IPC\_NOWAIT);  if(ret==-1)  {  printf("send message err\n");  perror("senderr");  return -1;  }  sleep(1);  memset(&msgbuf,0,sizeof(msgbuf));    ret=msgrcv(msgid, &msgbuf, sizeof(msgbuf.data), msgtype, IPC\_NOWAIT);  if(ret==-1)  {  printf("recv message err\n");  perror("dd");  return -1;  }  printf("recv msg =[%s]\n",msgbuf.data);    } |
|  |

## 4 消息队列项目开发案例

|  |
| --- |
| 消息队列实现回射客户/服务器 |
|  |
|  |

# linux管道-应用编程-专题讲座

## linux管道基本概念

### 1管道基本概念及操作

|  |
| --- |
| 管道基本概念   * 管道是Unix中最古老的进程间通信的形式。 * 我们把从一个进程连接到另一个进程的一个数据流称为一个“管道”   比如：eg: ls | wc -l ps –u itcast01|grep “aa” |
| 管道的本质====》固定大小的内核缓冲区 |
| 管道限制   * 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道 * 只能用于具有共同祖先的进程（具有亲缘关系的进程）之间进行通信；通常，一个管道由一个进程创建，然后该进程调用fork，此后父、子进程之间就可应用该管道**。** |
| 匿名管道pipe   * 包含头文件<unistd.h> * 功能:创建一无名管道 * 原型   + int pipe(int fd[2]); * 参数   + fd：文件描述符数组,其中fd[0]表示读端, fd[1]表示写端 * 返回值:成功返回0，失败返回错误代码 |
| 管道创建后示意图 |
| 管道基本编程示例 |

### 2管道和文件描述符在一起

|  |
| --- |
| 管道命令的编程实现：  ls | wc –w |
| 文件描述符的深入理解 |
| #include <unistd.h>  #include <sys/stat.h>  #include <sys/wait.h>  #include <sys/types.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #include <errno.h>  #include <string.h>  #include <signal.h>  #define ERR\_EXIT(m) \  do \  { \  perror(m); \  exit(EXIT\_FAILURE); \  } while(0)  int main21(void )  {  int pipefd[2];  pid\_t pid;  if (pipe(pipefd) == -1 )  {  printf("pipe() err..\n");  return -1;  }  pid = fork();  if (pid == -1)  {  printf("fork err..\n");  return -1;  }  if (pid == 0)  {  close(pipefd[0]);  //复制文件描述符pipefd[1]，给标准输出，言外之意：execlp的ls命令输出到管道中  dup2(pipefd[1], STDOUT\_FILENO);  close(pipefd[1]);    execlp("ls", "ls", NULL);  //如果替换新的进程印象失败，则会执行下面一句话  sprintf(stderr, "execute the cmd ls err..\n");  exit(0);      }  else if (pid > 0 )  {  int len = 0;  char buf[100] = {0};  close(pipefd[1]);  //复制文件描述符pipefd[0]，给标准输入，言外之意：execlp的wc命令从管道中读  dup2(pipefd[0], STDIN\_FILENO);  close(pipefd[0]);  //len = read(pipefd[0], buf, 100);  execlp("wc", "wc", "-w", NULL);  printf("len:%d, buf:%s \n", len , buf);  //close(pipefd[0]);  }  wait(NULL);  printf("parent ..quit\n");  return 0;  }  int main(int argc, char \*argv[])  {  close(0); //关闭表示输入  open("makefile", O\_RDONLY); //makefile文件变成标准输入  close(1);//关闭标准输出  open("makefile2", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644); //maifle2变成标准输出  execlp("cat", "cat", NULL); //替换进程印象后，执行cat命令    //cat命名从标准输入中按行读，紧接着写到标准输出  return 0;  } |

## 2管道的读写规则

|  |
| --- |
| 管道读写规则研究（重点）   * 当没有数据可读时   + O\_NONBLOCK disable：read调用阻塞，即进程暂停执行，一直等到有数据来到为止。   + O\_NONBLOCK enable：read调用返回-1，errno值为EAGAIN。 * 当管道满的时候   + O\_NONBLOCK disable： write调用阻塞，直到有进程读走数据   + O\_NONBLOCK enable：调用返回-1，errno值为EAGAIN * 当管道不停的被写，写满的时候   + O\_NONBLOCK disable： write调用阻塞   + O\_NONBLOCK enable：调用返回-1，errno值为EAGAIN * 如果所有管道写端对应的文件描述符被关闭，则read返回0 * 如果所有管道读端对应的文件描述符被关闭，则write操作会产生信号SIGPIPE * 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。 * 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。 |
| 测试管道读(阻塞、非阻塞)  测试管道两端读、写均关闭  测试管道容量 |
| 编程实践  man fctl  File descriptor flags  The following commands manipulate the flags associated with a file descriptor. Currently, only one such  flag is defined: FD\_CLOEXEC, the close-on-exec flag. If the FD\_CLOEXEC bit is 0, the file descriptor will  remain open across an execve(2), otherwise it will be closed.  F\_GETFD  Read the file descriptor flags.  F\_SETFD  Set the file descriptor flags to the value specified by arg.  File status flags  Each open file description has certain associated status flags, initialized by open(2) and possibly modified  by fcntl(2). Duplicated file descriptors (made with dup(), fcntl(F\_DUPFD), fork(), etc.) refer to the same  open file description, and thus share the same file status flags.  The file status flags and their semantics are described in open(2).  F\_GETFL  Read the file status flags.  F\_SETFL  Set the file status flags to the value specified by arg. File access mode (O\_RDONLY, O\_WRONLY,  O\_RDWR) and file creation flags (i.e., O\_CREAT, O\_EXCL, O\_NOCTTY, O\_TRUNC) in arg are ignored. On  Linux this command can only change the O\_APPEND, O\_ASYNC, O\_DIRECT, O\_NOATIME, and O\_NONBLOCK flags. |
| 容量查询  man -6 pipe  Pipe Capacity  A pipe has a limited capacity. If the pipe is full, then a write(2) will block or fail, depending on  whether the O\_NONBLOCK flag is set (see below). Different implementations have different limits for the  pipe capacity. Applications should not rely on a particular capacity: an application should be designed so  that a reading process consumes data as soon as it is available, so that a writing process does not remain  blocked.  In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096  bytes on x86). Since Linux 2.6.11, the pipe capacity is 65536 bytes. |
| 读写规则  man -6 pipe  If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read(2)  from the pipe will see end-of-file (read(2) will return 0). If all file descriptors referring to the read  end of a pipe have been closed, then a write(2) will cause a SIGPIPE signal to be generated for the calling  process. If the calling process is ignoring this signal, then write(2) fails with the error EPIPE. An  application that uses pipe(2) and fork(2) should use suitable close(2) calls to close unnecessary duplicate  file descriptors; this ensures that end-of-file and SIGPIPE/EPIPE are delivered when appropriate.  It is not possible to apply lseek(2) to a pipe. |

|  |
| --- |
| #include <linux/limits.h>  PIPE\_BUF |
| [root@localhost ~]# find / -name "limits.h"  /usr/include/linux/limits.h  [root@localhost ~]# find / -name "limits.h" | cat  [root@localhost ~]# find / -name "limits.h" | sed -n "1p"  [root@localhost ~]# cat /usr/include/linux/limits.h  #ifndef \_LINUX\_LIMITS\_H  #define \_LINUX\_LIMITS\_H  #define NR\_OPEN 1024  #define NGROUPS\_MAX 65536 /\* supplemental group IDs are available \*/  #define ARG\_MAX 131072 /\* # bytes of args + environ for exec() \*/  #define CHILD\_MAX 999 /\* no limit :-) \*/  #define OPEN\_MAX 256 /\* # open files a process may have \*/  #define LINK\_MAX 127 /\* # links a file may have \*/  #define MAX\_CANON 255 /\* size of the canonical input queue \*/  #define MAX\_INPUT 255 /\* size of the type-ahead buffer \*/  #define NAME\_MAX 255 /\* # chars in a file name \*/  #define PATH\_MAX 4096 /\* # chars in a path name including nul \*/  #define PIPE\_BUF 4096 /\* # bytes in atomic write to a pipe \*/  #define XATTR\_NAME\_MAX 255 /\* # chars in an extended attribute name \*/  #define XATTR\_SIZE\_MAX 65536 /\* size of an extended attribute value (64k) \*/  #define XATTR\_LIST\_MAX 65536 /\* size of extended attribute namelist (64k) \*/  #define RTSIG\_MAX 32  #endif  [root@localhost ~]# |
| man 7 pipe  Pipe Capacity  A pipe has a limited capacity. If the pipe is full, then a write(2) will block or fail, depending on whether the O\_NONBLOCK flag is set (see below). Different implementations have different limits for the pipe capacity. Applications should not rely on a particular capacity: an application should be designed so that a reading process consumes data as soon as it is available, so that a writing process does not remain blocked.  In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096 bytes on x86). Since Linux 2.6.11, the pipe capacity is 65536 bytes. |

## 3 FIFO管道

|  |
| --- |
| 命名管道   * 管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。 * 如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。 * 命名管道是一种特殊类型的文件 |
| 命名管道和匿名管道区别   * 管道应用的一个限制就是只能在具有共同祖先（具有亲缘关系）的进程间通信。 * 如果我们想在不相关的进程之间交换数据，可以使用FIFO文件来做这项工作，它经常被称为命名管道。 * 命名管道是一种特殊类型的文件 |
| 创建一个命名管道   * 命名管道可以从命令行上创建，命令行方法是使用下面这个命令：   $ mkfifo filename   * 命名管道也可以从程序里创建，相关函数有：   int mkfifo(const char \*filename,mode\_t );   * man 3 mkfifo |
| 匿名管道与命名管道PK   * 匿名管道由pipe函数创建并打开。 * 命名管道由mkfifo函数创建，打开用open * FIFO（命名管道）与pipe（匿名管道）之间唯一的区别在它们创建与打开的方式不同，一但这些工作完成之后，它们具有相同的语义。 |
| 命名管道的打开规则   * 如果当前打开操作是为读而打开FIFO时   + O\_NONBLOCK disable：阻塞直到有相应进程为写而打开该FIFO   + O\_NONBLOCK enable：立刻返回成功 * 如果当前打开操作是为写而打开FIFO时   + O\_NONBLOCK disable：阻塞直到有相应进程为读而打开该FIFO   + O\_NONBLOCK enable：立刻返回失败，错误码为ENXIO   命名管道的读写规则   * 同匿名管道 |
| mknod  管道文件不能使用vi打开。。就是vi不能打开看管道内容 |
|  |

## 4FIFO管道编程实践

|  |
| --- |
| 利用管道，两个进程间进行文件复制。  fifow  读文件1.txt  写入管道  fifor  读管道  写入2.txt文件 |
| FIFO(7) Linux Programmer's Manual FIFO(7)  NAME  fifo - first-in first-out special file, named pipe |
| man 3 mkfifo  mkfifo - make a FIFO special file (a named pipe)  SYNOPSIS  #include <sys/types.h>  #include <sys/stat.h>  int mkfifo(const char \*pathname, mode\_t mode);  DESCRIPTION  mkfifo() makes a FIFO special file with name pathname. mode specifies the FIFO's permissions. It is modi-  fied by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).  A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an  anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo().  Once you have created a FIFO special file in this way, any process can open it for reading or writing, in  the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can  proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some  other process opens the same FIFO for writing, and vice versa. See fifo(7) for non-blocking handling of FIFO  special files.  RETURN VALUE  On success mkfifo() returns 0. In the case of an error, -1 is returned (in which case, errno is set appro-  priately). |
|  |

# linux共享内存-应用编程-专题讲座

## 1共享内存基本概念

|  |
| --- |
| **共享内存**   * 共享内存区是最快的IPC形式。一旦这样的内存映射到共享它的进程的地址空间，这些进程间数据传递不再涉及到内核，换句话说是进程不再通过执行进入内核的系统调用来传递彼此的数据。 |
| **用管道或者消息队列传递数据**   * 内核为每个IPC对象维护一个数据结构 |
| **用共享内存传递数据** |

## 2System V共享内存API

### 共享内存数据结构及基本API

|  |
| --- |
| struct shmid\_ds {  struct ipc\_perm shm\_perm; /\* Ownership and permissions \*/  size\_t shm\_segsz; /\* Size of segment (bytes) \*/  time\_t shm\_atime; /\* Last attach time \*/  time\_t shm\_dtime; /\* Last detach time \*/  time\_t shm\_ctime; /\* Last change time \*/  pid\_t shm\_cpid; /\* PID of creator \*/  pid\_t shm\_lpid; /\* PID of last shmat(2)/shmdt(2) \*/  shmatt\_t shm\_nattch; /\* No. of current attaches \*/  ...  }; |
| **共享内存函数**   * #include <sys/ipc.h> * #include <sys/shm.h> * int shmget(key\_t key, size\_t size, int shmflg); * void \*shmat(int shmid, const void \*shmaddr, int shmflg); * int shmdt(const void \*shmaddr); * int shmctl(int shmid, int cmd, struct shmid\_ds \*buf); |
| **shmget函数**   * 功能：用来创建共享内存 * 原型   + int shmget(key\_t key, size\_t size, int shmflg); * 参数   + key:这个共享内存段名字   + size:共享内存大小   + shmflg:由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的 * 返回值：成功返回一个非负整数，即该共享内存段的标识码；失败返回-1 |
| **shmat函数**   * 功能：将共享内存段连接到进程地址空间 * 原型   + void \*shmat(int shmid, const void \*shmaddr, int shmflg); 0xaa11   + 0xaa11== * 参数   + shmid: 共享内存标识   + shmaddr:指定连接的地址   + shmflg:它的两个可能取值是SHM\_RND和SHM\_RDONLY * 返回值：成功返回一个指针，指向共享内存第一个节；失败返回-1 |
| **shmat函数（续）**   * shmaddr为NULL，核心自动选择一个地址 * shmaddr不为NULL且shmflg无SHM\_RND标记，则以shmaddr为连接地址。 * shmaddr不为NULL且shmflg设置了SHM\_RND标记，则连接的地址会自动向下调整为SHMLBA的整数倍。公式：shmaddr - (shmaddr % SHMLBA) * shmflg=SHM\_RDONLY，表示连接操作用来只读共享内存 |
| **shmdt函数**   * 功能：将共享内存段与当前进程脱离 * 原型   + int shmdt(const void \*shmaddr); * 参数   + shmaddr: 由shmat所返回的指针 * 返回值：成功返回0；失败返回-1 * 注意：将共享内存段与当前进程脱离不等于删除共享内存段 |
| **shmctl函数**   * 功能：用于控制共享内存 * 原型   + N * 参数   + shmid:由shmget返回的共享内存标识码   + cmd:将要采取的动作（有三个可取值）   + buf:指向一个保存着共享内存的模式状态和访问权限的数据结构 * 返回值：成功返回0；失败返回-1 |
| **shmctl函数（续）** |

## 3共享内存API示例

|  |
| --- |
| 共享内存查看工具及共享内存命令 |
| //  //key shmid owner perms bytes nattch status  //0x00002234 131073 itcast01 666 68 0  //  int main(int argc, char \*argv[])  {  int ret = 0;  int shmid;  //相当于打开文件，文件不存  shmid = shmget(0x2234, sizeof(Teacher), IPC\_CREAT | 0666);  if (shmid == -1)  {  perror("shmget err");  return errno;  }  printf("shmid:%d \n", shmid);  Teacher \*p = NULL;  p = shmat(shmid, NULL, 0);  if (p == (void \*)-1 )  {  perror("shmget err");  return errno;  }    strcpy(p->name, "aaaa");  p->age = 33;    shmdt(p);    printf("键入1 删除共享内存，其他不删除\n");    int num;  scanf("%d", &num);  if (num == 1)  {  ret = shmctl(shmid, IPC\_RMID, NULL);  if (ret < 0)  {  perror("rm errr\n");  }  }  return 0;  } |
| int main(int argc, char \*argv[])  {  int ret = 0;  int shmid;  //相当于打开文件，文件不存  //shmid = shmget(0x2234, sizeof(Teacher), IPC\_CREAT |IPC\_EXCL | 0666);  shmid = shmget(0x2234, 0, 0);  if (shmid == -1)  {  perror("shmget err");  return errno;  }  printf("shmid:%d \n", shmid);  Teacher \*p = NULL;  p = shmat(shmid, NULL, 0);  if (p == (void \*)-1 )  {  perror("shmget err");  return errno;  }    printf("name:%s\n", p->name);  printf("age:%d \n", p->age);  shmdt(p);    //  //key shmid owner perms bytes nattch status  //0x00002234 131073 itcast01 666 68 0  //    printf("键入1 程序暂停，其他退出\n");    int num;  scanf("%d", &num);  if (num == 1)  {  pause();  }    return 0;  } |

# linux信号量-应用编程-专题讲座

## 1信号量基本概念

|  |
| --- |
| **信号量**   * 信号量和P、V原语由Dijkstra（迪杰斯特拉）提出 * 信号量 * 互斥：P、V在同一个进程中 * 同步：P、V在不同进程中 * 信号量值含义 * S>0：S表示可用资源的个数 * S=0：表示无可用资源，无等待进程 * S<0：|S|表示等待队列中进程个数   struct semaphore  {  int value;  pointer\_PCB queue;  } |
| **P原语**  P(s)  {  s.value = s.value--;  if (s.value < 0)  {  该进程状态置为等待状状态  将该进程的PCB插入相应的等待队列s.queue末尾  }  } |
| **V原语**  V(s)  {  s.value = s.value++;  if (s.value < =0)  {  唤醒相应等待队列s.queue中等待的一个进程  改变其状态为就绪态  并将其插入就绪队列  }  } |
|  |

## 2信号量API

### 信号量集

|  |
| --- |
| struct semid\_ds {  struct ipc\_perm sem\_perm; /\* Ownership and permissions \*/  time\_t sem\_otime; /\* Last semop time \*/  time\_t sem\_ctime; /\* Last change time \*/  unsigned short sem\_nsems; /\* No. of semaphores in set \*/  }; |
| **信号量集函数**   * #include <sys/types.h> * #include <sys/ipc.h> * #include <sys/sem.h> * int semget(key\_t key, int nsems, int semflg); * int semctl(int semid, int semnum, int cmd, ...); * int semop(int semid, struct sembuf \*sops, unsigned nsops); |
| **semget函数**   * 功能：用来创建和访问一个信号量集 * 原型 * int semget(key\_t key, int nsems, int semflg); * 参数 * key: 信号集的名字 * nsems:信号集中信号量的个数 * semflg: 由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的 * 返回值：成功返回一个非负整数，即该信号集的标识码；失败返回-1 |
| **shmctl函数**   * 功能：用于控制信号量集 * 原型 * int semctl(int semid, int semnum, int cmd, ...); * 参数 * semid:由semget返回的信号集标识码 * semnum:信号集中信号量的序号 * cmd:将要采取的动作（有三个可取值） * 最后一个参数根据命令不同而不同 * 返回值：成功返回0；失败返回-1 |
| **shmctl函数续** |
| **semop函数**   * 功能：用来创建和访问一个信号量集 * 原型 * int semop(int semid, struct sembuf \*sops, unsigned nsops); * 参数 * semid:是该信号量的标识码，也就是semget函数的返回值 * sops:是个指向一个结构数值的指针 * nsops:信号量的个数 * 返回值：成功返回0；失败返回-1 |
| **semop函数续**   * sembuf结构体：   struct sembuf {  short sem\_num;  short sem\_op;  short sem\_flg;  };   * sem\_num是信号量的编号。 * sem\_op是信号量一次PV操作时加减的数值，一般只会用到两个值，一个是“-1”，也就是P操作，等待信号量变得可用；另一个是“+1”，也就是我们的V操作，发出信号量已经变得可用 * sem\_flag的两个取值是IPC\_NOWAIT或SEM\_UNDO |
|  |

## 3信号量API示例及工具

|  |
| --- |
| //创建信号量  int sem\_create(key\_t key)  {  /\* 注意是 IPC\_CREAT | IPC\_EXCL不是 O\_CREAT|O\_EXCL);  int semid = 0;  semid = semget(key, 1, 0666 | O\_CREAT|O\_EXCL);  if (semid == -1)  ERR\_EXIT("semget");    return semid;  \*/      int semid = semget(key, 1, 0666 | IPC\_CREAT | IPC\_EXCL);  if (semid == -1)  ERR\_EXIT("semget");  return semid;  }  int sem\_open(key\_t key)  {  int semid = semget(key, 0, 0);  if (semid == -1)  ERR\_EXIT("semget");  return semid;  }  int sem\_setval(int semid, int val)  {  union semun su;  su.val = val;  int ret;  ret = semctl(semid, 0, SETVAL, su);  if (ret == -1)  ERR\_EXIT("sem\_setval");  return 0;  }  int sem\_getval(int semid, int val)  {  union semun su;  su.val = val;  int ret;  ret = semctl(semid, 0, GETVAL, su);  if (ret == -1)  ERR\_EXIT("sem\_setval");  return 0;  }  int sem\_d(int semid)  {  int ret;  ret = semctl(semid, 0, IPC\_RMID, 0);  if (ret == -1)  ERR\_EXIT("semctl");  return 0;  }  int sem\_p(int semid)  {  int ret = 0;    struct sembuf sp = {0, -1, 0};  ret = semop(semid, &sp, 1); //第三个参数是信号量的参数  if (ret == -1)  ERR\_EXIT("semctl");  return ret;  }  int sem\_v(int semid)  {    struct sembuf sp = {0, 1, 0};  int ret = semop(semid, &sp, 1); //第三个参数是信号量的参数  if (ret == -1)  ERR\_EXIT("semctl");  return ret;  }      int main(int argc, char \*argv[])  {  int semid;  //semid = sem\_create(0x3234);  semid = sem\_open(0x3234);  printf("semid:%d \n", semid);  sleep(5);  sem\_d(semid);    return 0;  } |

# linux线程-应用编程-专题讲座 线程概念

### 1线程概念

|  |
| --- |
| **什么是线程**   * 在一个程序里的一个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的控制序列” * 一切进程至少都有一个执行线程 |
| **进程与线程**   * 进程是资源竞争的基本单位 * 线程是程序执行的最小单位 * 线程共享进程数据，但也拥有自己的一部分数据   + 线程ID   + 程序计数器   + 寄存器组   + 栈   + errno * 一个进程内部的线程可以共享资源   + 代码段   + 数据段   + 打开文件和信号 |
| **fork和创建新线程的区别**   * 当一个进程执行一个fork调用的时候，会创建出进程的一个新拷贝，新进程将拥有它自己的变量和它自己的PID。这个新进程的运行时间是独立的，它在执行时几乎完全独立于创建它的进程 * 在进程里面创建一个新线程的时候，新的执行线程会拥有自己的堆栈（因此也就有自己的局部变量），但要与它的创建者共享全局变量、文件描述符、信号处理器和当前的工作目录状态 |
| **线程的优点**   * 创建一个新线程的代价要比创建一个新进程小得多 * 与进程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多 * 线程占用的资源要比进程少很多 * 能充分利用多处理器的可并行数量 * 在等待慢速I/O操作结束的同时，程序可执行其他的计算任务 * 计算密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现 * I/O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。 |
| **线程缺点**   * 性能损失   + 一个很少被外部事件阻塞的计算密集型线程往往无法与共它线程共享同一个处理器。如果计算密集型线程的数量比可用的处理器多，那么可能会有较大的性能损失，这里的性能损失指的是增加了额外的同步和调度开销，而可用的资源不变。 * 健壮性降低   + 编写多线程需要更全面更深入的考虑，在一个多线程程序里，因时间分配上的细微偏差或者因共享了不该共享的变量而造成不良影响的可能性是很大的，换句话说线程之间是缺乏保护的。 * 缺乏访问控制   + 进程是访问控制的基本粒度，在一个线程中调用某些OS函数会对整个进程造成影响。 * 编程难度提高   + 编写与调试一个多线程程序比单线程程序困难得多 |
| **线程调度竞争范围**   * 操作系统提供了各种模型，用来调度应用程序创建的线程。这些模型之间的主要不同是：在竞争系统资源（特别是CPU时间）时，线程调度竞争范围(thread-scheduling contention scope）不一样 * 进程竞争范围（process contention scope）：各个线程在同一进程竞争“被调度的CPU时间”（但不直接和其他进程中的线程竞争）。 * 系统竞争范围（system contention scope）：线程直接和“系统范围”内的其他线程竞争。 |

### 2线程模型

|  |
| --- |
| **N:1用户线程模型**   * “线程实现”建立在“进程控制”机制之上，由用户空间的程序库来管理。OS内核完全不知道线程信息。这些线程称为用户空间线程。 * 这些线程都工作在“进程竞争范围”     **N:1用户线程模型**   * 在N:1线程模型中，内核不干涉线程的任何生命活动，也不干涉同一进程中的线程环境切换。 * 在N:1线程模型中，一个进程中的多个线程只能调度到一个CPU，这种约束限制了可用的并行总量。   第二个缺点是如果某个线程执行了一个“阻塞式”操作（如read），那么，进程中的所有线程都会阻塞，直至那个操作结束。为此，一些线程的实现是为这些阻塞式函数提供包装器，用非阻塞版本替换这些系统调用，以消除这种限制。 |
|  |
| **1:1核心线程模型**   * 在1:1核心线程模型中，应用程序创建的每一个线程都由一个核心线程直接管理。 * OS内核将每一个核心线程都调到系统CPU上，因此，所有线程都工作在“系统竞争范围”。 * 这种线程的创建与调度由内核完成，因为这种线程的系统开销比较大（但一般来说，比进程开销小） |
| **N:M混合线程模型**   * N:M混合线程模型提供了两级控制，将用户线程映射为系统的可调度体以实现并行，这个可调度体称为轻量级进程（LWP:lightweight process），LWP再一一映射到核心线程 |
| **总结：**  （1）用户级线程  用户级线程主要解决的是上下文切换的问题，它的调度算法和调度过程全部由用户自行选择决定，在运行时不需要特定的内核支持。在这里，操作系统往往会提供一个用户空间的线程库，该线程库提供了线程的创建、调度和撤销等功能，而内核仍然仅对进程进行管理。如果一个进程中的某一个线程调用了一个阻塞的系统调用函数，那么该进程包括该进程中的其他所有线程也同时被阻塞。这种用户级线程的主要缺点是在一个进程中的多个线程的调度中无法发挥多处理器的优势。  （2）轻量级进程  轻量级进程是内核支持的用户线程，是内核线程的一种抽象对象。每个线程拥有一个或多个轻量级线程，而每个轻量级线程分别被绑定在一个内核线程上。  （3）内核线程  这种线程允许不同进程中的线程按照同一相对优先调度方法进行调度，这样就可以发挥多处理器的并发优势。  现在大多数系统都采用用户级线程与核心级线程并存的方法。一个用户级线程可以对应一个或几个核心级线程，也就是“一对一”或“多对一”模型。这样既可满足多处理机系统的需要，也可以最大限度地减少调度开销。  使用线程机制大大加快上下文切换速度而且节省很多资源。但是因为在用户态和内核态均要实现调度管理，所以会增加实现的复杂度和引起优先级翻转的可能性。一个多线程程序的同步设计与调试也会增加程序实现的难度 |
|  |
|  |

## 2线程API

### 线程基本posix api

|  |
| --- |
| **POSIX线程库**   * 与线程有关的函数构成了一个完整的系列，绝大多数函数的名字都是以“pthread\_”打头的 * 要使用这些函数库，要通过引入头文<pthread.h> * 链接这些线程函数库时要使用编译器命令的“-lpthread”选项 |
| **pthread\_create函数**   * 功能：创建一个新的线程 * 原型 * int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*), void \*arg); * 参数 * thread:返回线程ID * attr:设置线程的属性，attr为NULL表示使用默认属性 * start\_routine:是个函数地址，线程启动后要执行的函数 * arg:传给线程启动函数的参数 * 返回值：成功返回0；失败返回错误码 |
| **错误检查**   * 传统的一些函数是，成功返回0，失败返回-1，并且对全局变量errno赋值以指示错误。 * pthreads函数出错时不会设置全局变量errno（而大部分其他POSIX函数会这样做）。而是将错误代码通过返回值返回 * pthreads同样也提供了线程内的errno变量，以支持其它使用errno的代码。对于pthreads函数的错误，建议通过返回值业判定，因为读取返回值要比读取线程内的errno变量的开销更小 |
| **pthread\_exit函数**   * 功能：线程终止 * 原型 * void pthread\_exit(void \*value\_ptr); * 参数 * value\_ptr:value\_ptr不要指向一个局部变量。 * 返回值：无返回值，跟进程一样，线程结束的时候无法返回到它的调用者（自身） |
| **pthread\_join函数**   * 功能：等待线程结束 * 原型   + int pthread\_join(pthread\_t thread, void \*\*value\_ptr); * 参数   + thread:线程ID   + value\_ptr:它指向一个指针，后者指向线程的返回值 * 返回值：成功返回0；失败返回错误码 |
| **pthread\_self函数**   * 功能：返回线程ID * 原型 * pthread\_t pthread\_self(void); * 返回值：成功返回0 |
| **pthread\_cancel函数**   * 功能：取消一个执行中的线程 * 原型 * int pthread\_cancel(pthread\_t thread); * 参数 * thread:线程ID * 返回值：成功返回0；失败返回错误码 |
| **pthread\_detach函数**   * 功能：将一个线程分离 * 原型   + int thread); * 参数   + thread:线程ID * 返回值：成功返回0；失败返回错误码 |

### 线程客户端测试框架

### 多线程服务器框架

## 3线程属性

|  |
| --- |
| **线程属性**   * 初始化与销毁属性   + int pthread\_attr\_init(pthread\_attr\_t \*attr);   + int pthread\_attr\_destroy(pthread\_attr\_t \*attr); * 获取与设置分离属性   + int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*attr, int \*detachstate);   + int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate); * 获取与设置栈大小   + int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, size\_t stacksize);   + int pthread\_attr\_getstacksize(pthread\_attr\_t \*attr, size\_t \*stacksize); * 获取与设置栈溢出保护区大小   + int pthread\_attr\_setguardsize(pthread\_attr\_t \*attr, size\_t guardsize);   + int pthread\_attr\_getguardsize(pthread\_attr\_t \*attr, size\_t \*guardsize); * 获取与设置线程竞争范围   + int pthread\_attr\_getscope(const pthread\_attr\_t \*attr,int \*contentionscope);   + int pthread\_attr\_setscope(pthread\_attr\_t \*attr, int contentionscope); * 获取与设置调度策略   + int pthread\_attr\_getschedpolicy(const pthread\_attr\_t \*attr, int \*policy);   + int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*attr, int policy); * 获取与设置继承的调度策略   + int pthread\_attr\_getinheritsched(const pthread\_attr\_t \*attr, int \*inheritsched);   + int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr, int inheritsched); * 获取与设置调度参数   + int pthread\_attr\_getschedparam(const pthread\_attr\_t \*attr, struct sched\_param \*param);   + int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr, const struct sched\_param \*param); |
| **并发级别**   * 获取与设置并发级别   + int pthread\_setconcurrency(int new\_level);   + int pthread\_getconcurrency(void); * 仅在N:M线程模型中有效，设置并发级别，给内核一个提提示：表示提供给定级别数量的核心线程来映射用户线程是高效的。 |

其他说明

|  |
| --- |
| • 绑定属性  Linux中采用“一对一”的线程机制，也就是一个用户线程对应一个内核线程。绑定属性就是指一个用户线程固定地分配给一个内核线程，因为CPU时间片的调度是面向内核线程（也就是轻量级进程）的，因此具有绑定属性的线程可以保证在需要的时候总有一个内核线程与之对应。而与之对应的非绑定属性就是指用户线程和内核线程的关系不是始终固定的，而是由系统来控制分配的。  • 分离属性  分离属性是用来决定一个线程以什么样的方式来终止自己。在非分离情况下，当一个线程结束时，它所占用的系统资源并没有被释放，也就是没有真正的终止。只有当pthread\_join()函数返回时，创建的线程才能释放自己占用的系统资源。而在分离属性情况下，一个线程结束时立即释放它所占有的系统资源。这里要注意的一点是，如果设置一个线程的分离属性，而这个线程运行又非常快，那么它很可能在pthread\_create()函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用。 |

## 4线程互斥和同步

### 基本概念

|  |
| --- |
| 多线程同步问题：  线程共享进程的资源和地址空间  任何线程对系统资源的操作都会给其他线程带来影响  多线程同步方法：  互斥锁  信号量  条件变量 |

### posix互斥锁

|  |
| --- |
| posix互斥锁   * 互斥锁是用一种简单的加锁方法来控制对共享资源的原子操作。这个互斥锁只有两种状态，也就是上锁和解锁，可以把互斥锁看作某种意义上的全局变量。在同一时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经被上锁的互斥锁，则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。可以说，这把互斥锁保证让每个线程对共享资源按顺序进行原子操作。 |
| posix互斥锁api   * 互斥锁机制主要包括下面的基本函数。 * 互斥锁初始化：pthread\_mutex\_init() * 互斥锁上锁：pthread\_mutex\_lock() * 互斥锁判断上锁：pthread\_mutex\_trylock() * 互斥锁接锁：pthread\_mutex\_unlock() * 消除互斥锁：pthread\_mutex\_destroy() * 其中，互斥锁可以分为快速互斥锁、递归互斥锁和检错互斥锁。这三种锁的区别主要在于其他未占有互斥锁的线程在希望得到互斥锁时是否需要阻塞等待。快速锁是指调用线程会阻塞直至拥有互斥锁的线程解锁为止。递归互斥锁能够成功地返回，并且增加调用线程在互斥上加锁的次数，而检错互斥锁则为快速互斥锁的非阻塞版本，它会立即返回并返回一个错误信息。默认属性为快速互斥锁。 |
| 该函数用于C函数的多线程编程中，互斥锁的初始化。  头文件：#include <pthread.h>  函数原型：int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);  pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_mutex\_init()函数是以动态方式创建互斥锁的，参数attr指定了新建互斥锁的属性。如果参数attr为空，则使用默认的互斥锁属性，默认属性为快速互斥锁 。互斥锁的属性在创建锁的时候指定，在LinuxThreads实现中仅有一个锁类型属性，不同的锁类型在试图对一个已经被锁定的互斥锁加锁时表现不同。  pthread\_mutexattr\_init()函数成功完成之后会返回零，其他任何返回值都表示出现了错误。  函数成功执行后，互斥锁被初始化为未锁住态。 |
| 互斥锁属性  使用互斥锁（互斥）可以使线程按顺序执行。通常，互斥锁通过确保一次只有一个线程执行代码的临界段来同步多个线程。互斥锁还可以保护单线程代码。  要更改缺省的互斥锁属性，可以对属性对象进行声明和初始化。通常，互斥锁属性会设置在应用程序开头的某个位置，以便可以快速查找和轻松修改 |
| 销毁互斥锁对象  pthread\_mutexattr\_destroy(3C)可用来取消分配用于维护 pthread\_mutexattr\_init() 所创建的属性对象的存储空间。  pthread\_mutexattr\_destroy 语法  int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*mattr)  #include <pthread.h>  pthread\_mutexattr\_t mattr;  int ret;/\* destroy an attribute \*/  ret = pthread\_mutexattr\_destroy(&mattr);  pthread\_mutexattr\_destroy 返回值  pthread\_mutexattr\_destroy() 成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。  EINVAL 描述: 由 mattr 指定的值无效。 |

### posix条件变量

|  |
| --- |
| 基本概念   * 与互斥锁不同，条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况 发生为止。通常条件变量和互斥锁同时使用。 * 条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局变量进行同步的一种机制， 主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。 * 条 件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件 变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。 * api列表 * pthread\_cond\_init * pthread\_cond\_destroy * pthread\_cond\_wait * pthread\_cond\_signal * pthread\_cond\_broadcast |
| **条件变量使用规范**   * 等待条件代码   pthread\_mutex\_lock(&mutex);  while (条件为假）  pthread\_cond\_wait(cond, mutex);  修改条件  pthread\_mutex\_unlock(&mutex);   * 给条件发送信号代码   pthread\_mutex\_lock(&mutex);  设置条件为真  pthread\_cond\_signal(cond);  pthread\_mutex\_unlock(&mutex); |
| * 使用条件变量之前要先进行初始化。可以在单个语句中生成和初始化一个条件变量如： * pthread\_cond\_t my\_condition=PTHREAD\_COND\_INITIALIZER;（用于进程间线程的通信）。   也可以利用函数pthread\_cond\_init动态初始化  C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\SNAGHTML120212b.PNG |
| pthread\_cond\_destroy函数可以用来摧毁所指定的条件变量，同时将会释放所给它分配的资源。调用该函数的进程也并不要求等待 在参数所指定的条件变量上。 |
| 第一个参数\*cond是指向一个条件变量的指针。第二个参数\*mutex则是对相关的互斥锁的指针。函数pthread\_cond\_timedwait函数类型与函数pthread\_cond\_wait,区别在于，如果达到或是超过所引用的参数\*abstime,它将结束并返回错误ETIME.pthread\_cond\_timedwait函数的参数\*abstime指向一个timespec结构。该结构如下：  typedef struct timespec{         time\_t tv\_sec;         long tv\_nsex;  }timespec\_t; |
| 参数\*cond是对类型为pthread\_cond\_t 的一个条件变量的指针。当调用pthread\_cond\_signal时一个在相同条件变量上阻塞的线程将被解锁。如果同时有多个线程阻塞，则由调度策略确定接收通知的 线程。如果调用pthread\_cond\_broadcast,则将通知阻塞在这个条件变量上的所有线程。一旦被唤醒，线程仍然会要求互斥锁。如果当前没有线程等待通知，则上面两种调用实际上成为一个空操 作。如果参数\*cond指向非法地址，则返回值EINVAL。 |
|  |

### 综合案例1多线程互斥

### 综合案例2线程同步