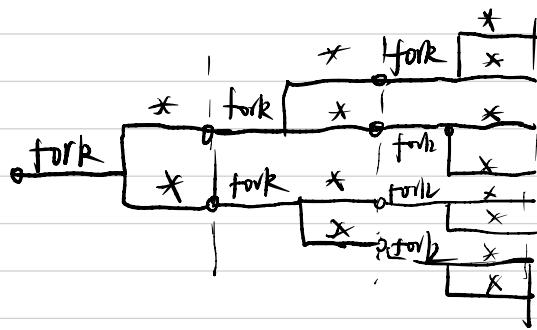




嵌入式程序设计



一、Linux 操作系统概述

1. Linux 的优势：

- ① 系统稳定
- ② 开源支持
- ③ 可裁剪，可移植
适合嵌入式系统

2. Linux 的应用

- ① 桌面应用
- ② 服务器应用
- ③ 嵌入式系统
- ④ 工作站应用
- ⑤ 集群计算机

3. Linux Shell (命令解释器)

系统的用户界面，提供用户与内核交互操作的接口。
它解释由用户输入的命令并把它们送到内核执行。

Shell Script：解释型的程序设计语言。

| 内部命令：shell 负责执行相应的动作

| 外部命令：默认路径下查找(配环境)

包含要用到的系列命令

一次执行

4. 系统调用与 API

系统调用：一定发生内核切换 状态 → 管态

API：包括系统调用，但不限于

5. Shell Script 基本语法

- ① 创建 shell 脚本： vi sh01.sh

② 标识拿什么解释：#!/bin/bash，接着写脚本内容

③ \$0 代表 程序/脚本名

\$1 代表 第一个参数 \$@, \$* 表示所有参数

\$2 代表 第二个参数

\$# 代表 参数个数

\$! 代表 当前进程的PID号

\$? 上一条命令执行后的状态，0表示正常。

echo：显示到终端，显示变量 echo \$command.

date：表示字符串，'date' 中 \$ (date) 表示命令不在if中时 || 表示：出现第一个真判断时结束执行
&& 表示：出现第一个假判断时结束执行

-eq, -le, -ge

等子 小于 大于

6. SSH 是 Telnet 的升级版，客户端与服务器之间的通信被加密

7. Linux 文件系统：树型结构，根结点为 /

1文件系统类型：Ext3, Ext4, vfat, NTFS 等

① hd: IDE硬盘 sd: SATA, SCSI, USB硬盘

a: 第一块硬盘 b: 第二块硬盘，依次类推 ...

1, 2, 3, 4 主分区 5-16 逻辑分区

②. Linux 中的文件目录结构

②

/ 根目录 → | /bin - 二进制可执行基本命令 (cd, pwd 等)
| /boot - 操作系统启动的引导文件
| /dev - 设备文件
| /etc → 配置文件
| /usr - 用户自己安装的程序和数据 (用户自己用)
| /home - 每个用户的主目录 (私人空间)
| /sbin - 系统级别的管理命令 (root 权限执行)
| /sys → 虚拟目录
| /proc → 真拟目录
| lib - 库文件 | /mnt : 挂载点

将磁盘分区挂载到子目录上:

mount -t vfat /dev/sdb1 /mnt/usb
③ 查看文件内容 (当前目录) ls -al

drwxr-x--	root	root	4096	Sep 8 14:06
文件夹 ↓	拥有者 ↓	拥有者 ↓	文件大小 ↓	
拥有者 ↓	所属用户组 ↓	其他用户 ↓	所在用户组 ↓	
权限				

Linux 中 - 动态文件.

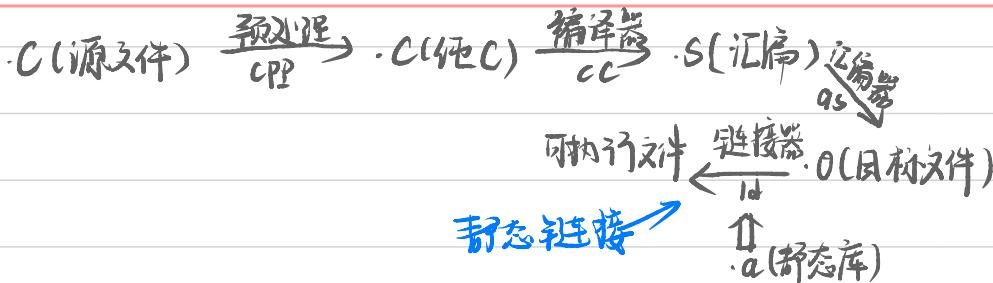
二、Linux C语言开发工具链

1. vim 编辑器

- 标准模式：命令模式，输入 i 进入编辑模式
- 编辑模式：输入正文且回显
- 底行模式：底行回显，Enter 有效，如：wq

2. GCC (GNU Compiler Collection)

一般情况，gcc 的 C 程序编译过程包括
预处理 → 编译 → 汇编 → 链接



① 预处理

```
gcc -E -o gctest.i gctest.c
```

主要工作：

- 文件包含，#include 处理
- 布局控制
- 宏替换

③ 编译成汇编代码

gcc -S gcctest.c ; 转出为 .s

主要将 C 代码处理为对应体系结构的汇编代码

④ 汇编成目标文件

gcc -c gcctest.c ; 生成为 .o

或使用汇编器

as -O gcctest.o gcctest.s

④ 得到可执行文件 (链接)

gcc -O program gcctest.o

或
[root] ./program

3. 链接

① 代码中使用到了外部函数或变量

A. include 相应的 .h 文件声明

B. extern 相应的 变量 / 函数

② 链接多个文件，整合成一个完整的代码段，以此执行

gcc -O program t1.o t2.o t3.o

③ 对于经常用到的函数，封装成库来链接

优点：模块化，可重用性强，编译速度快，保护知识产权

A. 静态库 (.a 后缀)：编译时链接，链接过程中，相应函数

体会被复制到程序的可执行文件中去。
生成和链接静态库：

gcc -c hello.c

生成步

ar -crsv libhello.a hello.o

指定库文件名

gcc -o test test.c -L. -lhello

指定静态库文件路径

B. 动态库(.SO后缀)：运行时链接，动态库文件在内存中只有
1份拷贝，也称共享库。

3步

gcc -fPIC -c hello.c

生成

检查
必须是

gcc -shared -D libhello.so hello.o

○与连接一个步骤

gcc -o test test.c -lhello

△) 指定动态库位置，
便于运行时加载

动态链接的优点：

① 1份拷贝，节省内存

② 可维护性和可扩展性好，更新时只要改库文件，不用全
部重新编译

三、GDB入门

1. GDB的基本使用

- ① 程序编译为调试版 `gcc -g`
- ② 启动 `gdb` `gdb program`
- ③ 运行命令

- 查看源文件 `list`

- 设置断点 `b 10` (`break 10`)

- 查看断点处情况 `info b`

- 运行代码 `r`

- 查看变量

- { `print n` : 断点或某一步结束后，主动查看

- `display n` : 设定后每一步都自动显示变量值 `n`

- `watch n` : 变量发生变化时显示 `n`.

- 单步运行: `stop` (进入函数体)

- `next` (不进入函数体)

- 恢复程序运行 `c` (`continue`)

2. 库打桩机制

对共享库代码进行截取，以执行自己的代码或加入调试信息（代替原来的库函数）

① 编译时打桩

在预处理器时发生，本质为宏替换

注意：编译时加上 -DMYMOCK 参数，定义 MYMOCK 宏
分开编译打桩函数和外部调用代码。

② 链接时打桩 — 静态库

在链接器阶段打桩，代码插入到时的替换

f 被解析为 --wrap-f，read-f 解析为 f

③ 运行时打桩 — 动态库 该库会先于标准库加载

重定义 malloc 函数，修改 LD_PRELDRD 环境变量，
这样便可以覆盖原来的函数，实现运行时打桩。
* export LD_PRELDRD=； 这样设置的环境变量
为临时，只在当前 shell 中有效

△ ppt 中的例程为将原本的 malloc 函数改为 malloc
+ 打印分配内存地址；

计数器 calltimes 的作用：避免反义调用

四、make 工具

1. 功能

- ① 多个源文件的一键全自动编译.
- ② 编译时不编译修改后的源文件，提高编译效率.

2. 步骤

- ① 编写 makefile 文件.
- ② make clean : 构成 clean 目标
- ③ make : 默认构建构造第一个目标.

3. 变量

- ① makefile 中用变量指代一个字符串.

如 $OBJ = main.o \ kbd.o \ command.o$
edit: \$(Objects)

- ② 预定义变量：指定了用法，值可变.

如 CFLAGS: C 编译的参数 如 $= -C -g -O$

- ③ 自动变量：指定了值和用法

\$@：指代目标文件. \$₁：所有不带@的依赖文件名.

\$<：第一个依赖文件名

如 yul.o: yul.c yul.h

(Tab) \$(CC) \$(CFLAGS) -O \$@ \$<

十. 隐式规则

如 .c → .o 有隐式规则

只要写依赖 main.o : main.c calc.h
不用写构建命令，隐式推导。

越来越简化

* gcc -MM \$(SRC)

-MM 指输出所有文件的依赖关系

make -C dir : 构建指定目录下的 makefile

make -n : 只打印构建的命令，但不执行

五. 文件 I/O 编程

1. 用户地址空间与内核地址空间

进程私有的代码和 DS 内核代码与数据

↑ 数据 ↑
↓ 访问 ↓

2. 用户态与核心态

CPU 的两种指令执行级别 (对进程而言)

用户态 → 核心态
系统调用

3. Linux 的 API

*主要是通过系统 C 库 (libc) 实现，包括：

标准 C, POSIX, SYSTEM V 等类型 API

4. Linux 底层 I/O 函数

① int open (const char* pathname, int flags, int perms)
文件描述符 路径+文件名 打开方式

② int close (int fd);
0 成功, -1 失败 文件描述符

③ ssize_t read (int fd, void* buf, size_t count)
实际读到的字节数 指定读取的位置 计划读字节数

0: 到文件尾

-1: 失败

④. write() 写文件，格式与 read 完全一致

⑤. off_t lseek (int fd, off_t offset, int whence)

修改文件当前偏移

偏移字节数

偏移起始点

⑥. int fcntl (int fd, int cmd, struct flock *lock)

控制参数

与锁的类型

5. VFS 是虚拟文件系统（第一层），第二层是各种不同的具体文件系统，VFS 屏蔽了底层具体文件系统的实现细节与差异。

通用文件模型 包括超级块对象，索引节点对象，目录项对象以及文件对象

6. I/O 处理模型

阻塞式 I/O：文件没准备好则阻塞进程，去干其他事

非阻塞式 I/O：I/O 操作立即返回，不保证成功，循环等待

多路复用 I/O 模型：一个进程同时监视多个文件描述符，实现了并发的效果

✓ select 函数 要监视的文件描述符最大值 读文件描述符集

int select (int numfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout)

异常文件描述符集

立即检测

超时：0

出错：-1

成功：符合条件的文件描述符数

标准步骤与聚：

① 创建 / 打开 文件 / 设备 / socket | 做道等

② 创建 select 监视的文件描述符集合

fd-set read-fds;

③ 初始化 FD_ZERO (&read-fds);

④ 加入要监视的文件：FD_SET (fd, &read-fds);

⑤ 在 while() 环 select 中间集 tmp-fds (存储是 read-fds)
判断返回值和某文件是否就绪，执行相应的 I/O。
如 FD_ISSET (fd, &tmp-fds)

✓ poll 函数

out)

int poll (struct pollfd *fds, int numfds, int time)

struct pollfd

↑ int fd; short events; short rvents;

监听事件：可读 | 可写 ↓ 用于判断是否
pollin | pollout 发生 | 未发生

poll 相比 select 效率更高

fds[1].events & POLLIN: 检测是否已准备好

用亲检测是否包含 POLLIN

7. 嵌入式 Linux 并口应用编程 — 设备也是文件

① 设备描述 - /dev / ttySO - /dev / ttyUSBO
串口 USB → 串口

SET-com-config

②. 对串口的参数设置：波特率(115200)，起始位比特数(1)，数据位比特数(8bit)，停止位比特数(1bit)

串口设置是设置 struct termios 结构体成员。

③. 终端有三种工作模式

规范模式，非规范模式，原始模式

↓ 回显，用户与机器间 不回显，机器之间
以字节为单位

支持行编辑

不支持行编辑

④ 串口使用

|D-NOCM

> Open 串口 fd = open (" /dev/ttyS0 ", O_RDWR | O_NDELAY)

>> 将交叉串口为阻塞态，等待数据

fcntl (fd, F_SETFL, O_)

>>> 确认串口是否正确打开 isatty (fd);

对象为 Open - PORT ()

⑤. 读写串口，与读写文件一致。

⑥. 标准 I/O 编程

全缓冲、行缓冲、不带缓冲

打开： fopen , fopen , freopen : 返回 FILE 类型

读： fread ,getc , gets() , scanf()

写： fwrite , putc , puts() , printf()
字节 字符 字符串 行 格式化

++：文件 不加：标准输入 | 车辆号 +S: 字符权限

堆：动态分配的内存区
栈：用于函数调用保护现场

六、Linux 中的进程控制

进程主要包括
PCB：进程控制块
地址空间：程序、数据、堆栈
资源：如文件描述符表

1. 进程

进程是程序的一次执行过程，是资源分配的最小单位

PCB 描述进程信息，Linux 下任务等同于进程

2. 线程

它是进程中独立的一条执行路径，调度的最小单位，线程共享进程的内存空间和资源。线程上下文切换的开销比进程小得多。

64位系统

• 线程与进程间的关系



47位地址空间

3. 进程标识符 PID 一般标识一个进程，存在 PCB 中。

pid_t getpid(); // 获取当前进程的 pid 号

pid_t getppid(); // 获取父进程的 pid 号

4. Linux 进程的五种状态

① R(TASK_RUNNING)：可执行状态（运行+就绪）

② S(TASK_INTERRUPTIBLE)：可中断的睡眠态

当所求资源如 socket 连接、信号量而等待时，进入该状态，而资源就绪，则立即唤醒

PS：查看进程列表

③ D(TASK - UNINTERRUPTIBLE)：不可中断的睡眠状态，
不可被异步信号打断

④ T(TASK - STOPPED or TASK - TRACED) 软停或跟踪状态
软停：进程响应信号而进入软停状态
跟踪：断点处，调试时用

⑤ Z：僵尸状态 (TASK_ZOMBIE)

进程死亡但僵尸未注销，进程 struct 结构体仍保留

5. fork() 函数	有独立的地址空间，但内容一样，“复制”行为
pid_t fork(void)	子进程复制父进程的一切资源
返回值 0：子进程	独有的只有进程号、资源使用以及计时器
大于 0：子进程 ID	封装
-1：出错	复制后的子进程和父进程都从 fork 处开始执行

子进程复制了父进程的用户 ID，用户组 ID，进程组 ID，会话 ID，工作目录，文件、资源权限，信号屏蔽位，文件描述符表，进程地址空间（程序、数据、堆栈）

6. exec 函数

exec 创建的新进程“占据了”原进程的大部分资源，进程代码段装入了新的可执行程序，exec 系列成功后，原进程就消失了。
代替逻辑 在接占用原进程

例子：由登陆界面 → 主界面 的地址空间
“替换”行为

`int exec(const char * path, const char * arg, ...)`

7. 写时拷贝机制 (cow)

使用 `fork()` 创建子进程时，内核只为子进程创建虚拟空间，不分配物理内存，当父进程有更改相应段行为发生时，才为子进程分配物理空间。

优点：相对旧版 `fork()` 效率高，节省内存空间。

`vfork()`，父子进程始终共享同一内存空间，`vfork`调用时，子进程会被阻塞。`vfork`的子进程修改变量会影响父进程。

`Sleep(1)`: 进程休眠 1s 钟。

是深度睡眠

8. `exit()` 函数 — 调用后，进程终止运行

`void exit() -> exit(int status)` 返回结束状态

- `exit` 相对 `exit` 最大的区别是终止进程会要把文件缓冲区的内容写回文件。

0 为正常结束

通常为 `exit(0)`

9. 子进程

①. 产生条件：

-子进程执行完毕，但父进程没有回收其状态。

②. 子进程退出前会向父进程发送 SIGCHLD 信号。

③. 父进程用 wait 和 waitpid 回收

我的理解：当子进程调用 exit() 结束时，由于其父进程后续处理可能需要用到子进程退出的状态信息 (status)，因此进程纪录了它仍保留了状态信息，这就是僵尸进程。

当父进程 wait 或 waitpid 时，接收了进程的状态，子进程才真正

10. wait() 函数

pid_t wait (int * status)

wait 函数使调用 wait 的进程阻塞，直到一个子进程结束或接到了指定信号，如果没有子进程，则立即返回。

返回值为当前结束了进程的 id，status 保存子进程退出状态。这也是为什么会有僵尸状态的原因。

pid_t waitpid (pid_t pid, int * status, int options)

↑
只等待该子进程，如
果没有结束一直等待

↑
一般为 0，支持非阻塞
得到子进程
的退出状态

11. Linux 守护进程

/fork 问题

/打包

/NTP

基本步骤

- ①. 创建子进程，退出父进程 → 子进程被1号进程回收/init进程
- ⑤. 在子进程中创建新会话，使其成为进程组组长
与之前仍控制终端断联 `setsid`)
- ③. 改变当前目录为根目录
从父进程的当前目录 → 根目录
- ④. 重设文件权限掩码: 要求umask(0)，打开所有文件权限
文件权限: 文件: 最大666 目录: 最大777
- ⑤. 关闭文件描述符。

守护进程随着系统启动而启动，关闭而关闭，通常用来实现各种服务(如 Web, ftp)，是 Linux 的后台服务器进程。

守护进程出错处理: `openlog()`, `syslog()`, `closelog()`。
通过系统日志打印错误或提示信息。

ps -e 显示所有进程
ps 当前会话的进程

2. Linux 进程通信机制 (IPC)

1. Linux IPC 分类:

最初 Unix 的进程间通信：管道、FIFO、信号

System V: System V 的消息队列、信号量、共享内存

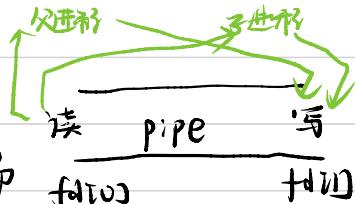
Posix: Posix 的消息队列、信号量、共享内存

Socket: 不同机器的进程间通信，相同机器也可以

2 有名管道

- 只能用于父子进程或兄弟进程

- 半双工模式：固定的读端和写端



①. pipe() 创建有名管道

int pipe(int fd[2]) 返回0: 成功 1: 失败

创建的描述符会保存到 fd[2] 中。

②. 标准流管道

FILE * popen(const char * command, const char * type)

command: 往入一个 shell 命令字符串，type 为 "r" 或 "w"

即，读还是写管道)

pclose(FILE * stream) 关闭管道

3. 命名管道 (fifo)

- 任意进程之间的通信

- 单向传输

创建有名管道

int mkfifo (const char* filename, mode_t mode)
 ↑
 管道名
 该驱动限带不为
 0777

4. Linux 中的信号机制

- 对中断的模拟，一种异步通信机制
- 每个进程都有自己的异步处理函数映射表（私有）
- 表中有 64 个表项，1-31（前 32 个）是有预定义的处理函数，后 32 个作为扩充。
- 前 32 个不可靠，不实时，可能会排队时丢失，后 32 个不会。

kill -l 可以查看所有信号及对应的编号。

- ① SIGHUP 终端挂起或控制进程结束时发出。
 - ② SIGINT 用户键入 Ctrl+C 发出，中断信号
 - ③ SIGQUIT 用户 Ctrl+\ 发出，退出信号
 - ④ SIGKILL 强制终止进程，杀死信号
 - ⑤ SIGCHLD 子进程状态改变信号，父进程会收到这个信号
- ⑥ 信号发送函数 int kill(pid_t pid, int sig)
 要发给的进程号 信号

② 信号捕捉

A. unsigned int alarm (unsigned int seconds)

系统经过 seconds 秒后向进程发 SIGALRM 信号（定时器），

成功：返回上一闹钟的剩余时间或 0

失败 -1

B. int pause(void) 用于将调用其的进程挂起，直到捕获到信号为止。

返回：-1, error值设为 EINTR

③ 信号的处理

A. signal 函数

typedef void (*sighandler_t)(int);

必须
成功返回以前的信号处理配置
失败：-1

信号
不返回

函数指针

B. sigaction 函数，更加健壮

int sigaction(int signum, const struct act, struct sigaction * oldact)

oldact 保存旧的对特定信号的处理

act 指向对该信号的处理函数

④ 一般信号处理

定义信号集合 → 设置信号屏蔽位 → 定义信号处理器 → 测试信号

5. 信号量

· 保护共享资源，如高亮代码段，一个时刻只能一个进程访问。

· 信号量非负，等于0没有资源，小于0有12个线程在等待

· P操作：-1 V操作：+1

① 创建信号量 (System V)

int semget(key_t key, int nsems, int semflg)

信号量集标识符 → 信号量集的键值 信号量数目 权限
② int semctl(int semid, int semnum, int cmd, union semun arg)
信号量集标识符 → 信号量集中信号量 一 般给LPC-SETRVAL
给信号量的初值 他的编号 单个, 连续为0

③ int semop(int semid, struct sembuf *ops, size_t nsops)
信号量集标识符 ↓
信号量集的操作 ↓
nsops: sops数组的个数

```
struct sembuf
{
    short sem-num; // 信号量编号
    short sem-op; // P:-1, V:+1
    short sem-flg; // SEM-UNDO
}
```

对于POSIX来说

初始化: 命名信号量

sem-open()

sem-init()

匿名信号量

PV操作

→ sem-wait():P

sem-post():V

释放信号量

sem-unlink()

sem-destroy()

互斥

P(S1)

临界1

V(S1)

S1=1

同步

P(S1)

临界1

V(S2)

初始:S1=1, S2=0

P(S2)

临界2

V(S1)

6. 共享内存

· 最高效的进程间通信机制，进程可以直接读写内存，不需要任何数据拷贝

`int shmget (key_t key, int size, int shmflg)`

键值，用于
多个进程锁定
共享内存

↓
大小

权限位，0666

返回共享内存区标识符

`shmflg)`

`char * shmat (int shmid, const void * shmaddr, int`

成功：返回被映射的段地址

失败：返回错误码

`int shmdt (void * shmaddr)` 撤销指映射

7. 消息队列

`msgget`

`msgsnd`

消息队列的实现已先创建或打开消息队列，添加消息。

读取消息和控制消息队列。

`msgrcv`

`msgctl`

与 FIFO 不同的是，消息队列可以读取指定的消息。

Linux 多线程编程

1. 实现该机制需要调用 Glibc 中的 pthread 库，编译时需要加上 -lpthread 参数

2. pthread_create()

int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *(*start_routine)(void *), void * arg)

线程属性设置
通常为 NULL
↓
线程函数输入
+地址
↓
参数
↓

成功：返回 0，失败：返回错误码

有关的 wait/waitpid

3. int pthread_join(pthread_t th, void ** thread_return)

th：等待线程的标识符

thread_return：存储被等待结束时的返回值

成功 0，失败：返回错误码

4. int pthread_cancel(pthread_t th) 其他线程使用

th：要取消的线程的标识符

5. void pthread_exit(void * retval) 已使用

retval 存线程结束的返回值

3> 线程主函数结束自动退出

6. 线程间的同步和互斥

互斥锁

```
int pthread_mutex_init(*mutex *mutexattr)  
int pthread_mutex_lock(mutex);  
int pthread_mutex_unlock(mutex);  
trylock, destroy
```

信号量机制

```
sem_init(), sem_wait()    sem_post()
```

7. 线程的属性

attr_attributes
†

*若设置 pthread_create() 函数的第二个参数 attr，
者为 NULL，则采用默认

·绑定属性

·分离属性

对属性进行初始化

```
int pthread_attr_init(pthread_attr_t *attr)
```

setscope 设置绑定属性

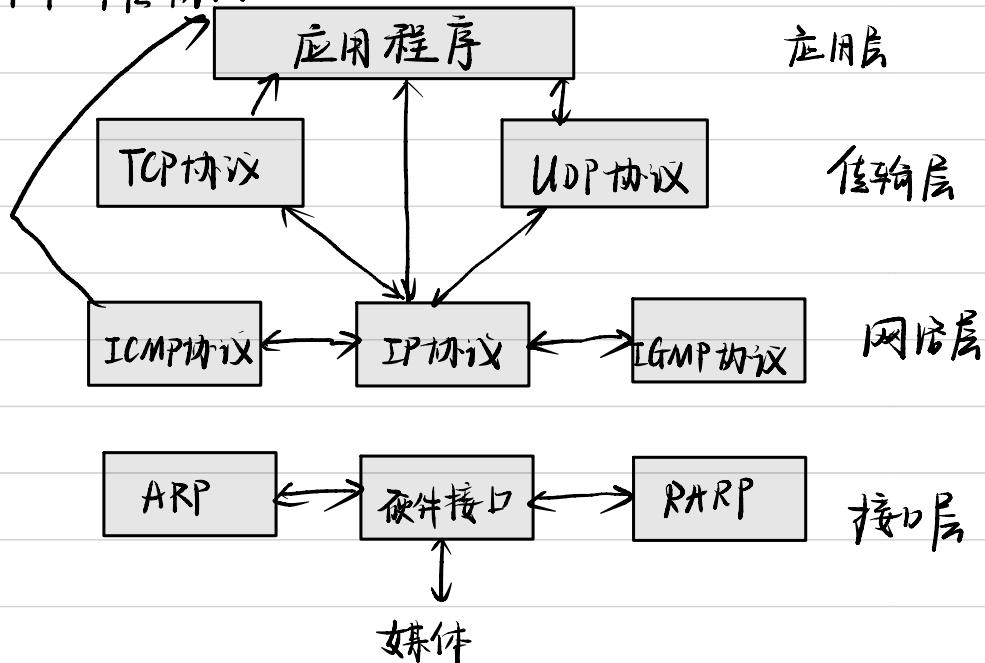
setdetachstate 设置分离属性

setschedparam() 设置线程优先级

getschedparam() 获得优先级

基于套接字的网络编程

1. TCP/IP 4层协议



2. socket (网络程序设计接口)

- ① 传输层和网络层提供给应用层的标准接口
- ② 类型 流式套接字 (Stream socket) : TCP协议:面向连接
数据报套接字 (Datagram socket) : UDP协议:无连接
原始套接字 (Raw socket)
- ③ socket 可认为是在因特网上进程间进行数据传输的一个端点，
应用程序间进行数据传输通过 socket 进行。
- ④ socket 连接: 五元组 < SIP, SPort, dIP, dPort, 端口 >

⑤. 套接字地址

A. struct sockaddr { // Linux通用socket地址格式 楼宇)

 u_short sa_family; // 协议 AF-INET(IPV4) AF-UNX(本地套
 char sa_data[14]); }; // 具体地址

B. TCP/IP协议的套接字地址 两种格式可以强制转换

struct sockaddr_in { (struct sockaddr*)&srraddr 大端序

 short sin_family; /* AF-INET */ ↗

 u_short sin_port; /* 端口号, 网络字节序 */

 struct in_addr sin_addr; /* IP地址, 网络字节序 */

 char sin_zero[8]; // 填充字节, 必须全0 |

}; struct in_addr { ↗
 u_long s_addr; }; // 无符号整型 IPV4.

C. 常用IP地址转换函数

[int inet_aton (const char *cp, struct in_addr *inp);

将点分字符串转为无符号整型: inet_aton("219.245.78.159", &addr.sin_addr)

- char * inet_ntoa (struct in_addr in);

将无符号整型转为点分字符串 printf ("%s", inet_ntoa(addr.sin_addr));

inet_pton 与 inet_ntop 类似, 只是支持了 IPv6 地址。

D. 服务器名字与地址转换 (DNS 服务)

int getaddrinfo (const char *node, const char *service,
 const struct addrinfo *hints, struct addrinfo **res);

域名+端口 → IP+端口

↙ 可重入

结果

node: 安解析的主机名, 如 www.example.com

service: 服务名或端口号: http/80

hints: 指向 addrinfo, 存放函数执行的指示信息

res: 指向 addrinfo 结构体指针的指针, 函数将通过这个指针返回一个链表 → 链表中存放所有符合条件的 socket 地址(即端口)

void freeaddrinfo() 回收生成这个链表空间

int getnameinfo() 反向操作, 地址→名字.

3 TCP 套接字编程模型

TCP 套接字编程典型模型



① 创建套接字

`int socket(int family, int type, int protocol)`

协议簇 类型

协议默认

失败：-1， 成功：socket描述符

创建socket意味着为一个socket数据结构分配五元组空间

②. 绑定服务器地址和端口。 0成功 -1失败

`int bind(int sockfd, struct sockaddr *myaddr,
int addrlen);` *addrlen 为地址结构长度，用sizeof*

功能：将socket描述符与服务器socket地址进行关联

若设置绑定的IP地址为INADDR_ANY(0)，则监听所有的本地网络接口（如网卡、虚拟网卡），适用于多IP主机

③. 监听端口

`int listen(int sockfd, int backlog)`

功能：建立一个连接请求队列，监听端口，监听到1个，就放入0成功，-1失败
队列1个

** 只用于面向连接方式的socket，即SOCK_STREAM，当连接到达时，accept()走*

请求队列中的连接已被TCP接受，即三次握手完成，放入队列，被应用层接受(accept())，才移出队列

④. 连接服务器(connect)

`int connect(int sockfd, struct sockaddr * servaddr,`

`int addrlen)`

结构长度

客户端sockfd

服务器地址

⑤ 接受客户端连接 (accept)

```
int accept(int sockfd, struct sockaddr *clientaddr,  
           int *addrlen)
```

返回值为新 socket 描述符，与 read 类似。accept 在无连接时将阻塞进程，新 socket 用于通信，原 socket 只用于侦听和连接。

* 返回 IP 地址(32位), 端口号(16位) 在赋值时, 必须用网络字节序, 即 htons 和 htonl.

* void * memset(void *s, int c, size_t n)

将参数 s 指定的内存区域的前 n 个字节设为 c.

⑥ 接收请求

A. int read(int fd, char *buf, int len);

// 读取文件, 按文件处理

B. recv(int fd, char *buf, int len, int flags)

// 只用于套接字, 有一些参数

C. int recvfrom(int fd, char *buf, int len, int flags,
 struct sockaddr *fromaddr, int *addrlen)

// 只适用于 UDP 套接字

⑦ 发送请求

A. int write(int fd, char *buf, int len);

B. send , send to 类似 , sendto 用于 UDP 通信

✓. accept, read, write 可能阻塞

4. Http 协议

①. 请求报问

request 方法 调用 URL 协议版本
↓ GET, POST 等

②. 响应报文

(404) (NOT Found)

HTTP 版本号 响应代码 响应主体
(空行)

资源内容

上，NTP 协议可用于时钟同步。

对工/0密集任务来说：

多路复用的性能相对最好

Linux 驱动程序设计

1. 内核模块

被动代码集合作，事件驱动模型，只能调用内核函数
可重入性、栈很小

2. 内核模块的加载与卸载

insmod /hello.ko

rmmmod hello.ko

3. 内核模块的程序结构

• 模块加载函数 ✓ 必有

```
static int __init initialization_function()  
{
```

/* 初始化代码 */

}

```
module_init(initialization_function);
```

• 模块卸载函数 ? 可无

```
static void __exit cleanup_function(void)
```

{ /* 释放内存，释放硬件资源， */ }

}

```
module_exit(cleanup_function);
```

insmod

- 模块参数 — 用于装载内核模块时，用户向内核模块传递参数（静态变量）
- 导出符号
- 模块声明描述

MODULE_AUTHOR (author);

• 模块许可证声明 ✓(必须)

insmod：调用模块加载函数

rmmod：调用模块卸载函数

4. 设备驱动程序特点、

内核模块，提供内核接口，可装载，可设置，动态性

分类 | 字符设备驱动：键盘、鼠标、触摸屏：非缓冲

| 块设备驱动：经过 VFS：磁盘、flash：缓冲

| 网络设备驱动：网卡

5. 设备文件与设备号

设备文件： /dev 子目录

设备号： 包括主设备号和次设备号

外设寄存器 | 位于 I/O 地址空间：I/O 端口，独立地址 X86
| 位于 内存地址空间：I/O 内存，统一地址 ARM

6. 字符设备驱动编程

重要数据结构：

- ▷ struct file_operations：自定义了文件相关操作。
- ▷ struct file：设备文件（/dev 下的）信息，包括模式（可读/可写），dev-t 和 rdev 设备号等。
- ▷ dev-t：32位，一般表示设备号，高 12 位为主设备号，低 20 位次设备号，通过表示设备类型，从而表示具体某一个设备。

✓ 设备注册与注销

设备注册：int register_chrdev(unsigned int major, const char *name, struct file_operations *fops)

要注册的主设备号（类），设备名，以及设备支持的操作，注册成功返回主设备号，且设备出现在 /proc/devices/name

通常在模块加载函数中调用

设备注销 int unregister_chrdev(major, name)

新版本注册与注销

这里的设备说明都是没有

struct cdev * cdev_alloc(void) → 动态内存，类型

void cdev_init(struct cdev *cdev, *fops) int count)

注册 int cdev_add (struct cdev *cdev, dev-t num, unsigned

卸载 int cdev_del (struct cdev *dev)

1. 嵌入式系统是一种在计算机硬件中嵌入软件的系统，该系统专用于某个应用或应用产品的特定部分，或大型系统的组件。

嵌入式系统的三个特性：嵌入性、专用性、计算机系统与通用计算机的区别与特点

① 专用性：专门的处理器，专用的功能算法，用于满足对客的要求
定制硬件

② 小型化：计算资源有限、结构紧凑

③ 软硬件设计一体化：包括应用软件与 OS 的一体化设计；硬件与软件的依赖性强

④ 需要交叉开发环境，开发向宿主机完成

⑤ 嵌入式系统对实时性要求严格，一般要求 RTOS

2. 嵌入式系统一般由硬件设备、嵌入式操作系统和应用软件三部分组成，常用的嵌入式操作系统有：

① 嵌入式 Linux：对标准 Linux 进行小型化裁剪，性能优良，衍生版本多，被广泛使用于嵌入式系统中

② μC/OS-II：开源，基于优先级的抢占式的硬 RTOS，占用空间小

③ VxWorks：不开源，RTOS，成本较高

3. 交叉编译：在宿主机上基于源代码生成可以在目标机上执行的程序

原因：①. 宿主机与目标机硬件平台差异，CPU 指令体系不同；②. 宿主机与目标机 OS 有差异，API 不同

4.

CISC

RISC

指令集

复杂指令集，变长
执行时间不同

精简指令集，定长
一周期一条指令

流水线

指令的执行需要调用
一个微程序

每周期前进一步

寄存器

通用寄存器数目少

通用寄存器多

Load/store

处理器能够直接处理
存储器中数据

独立 Load/Store 指令
在寄存器与内存传输

控制器

微程序

硬布线

5. 嵌入式开发一般采用交叉调试，包括软件方式和硬件方式

① 软件方式：优点：便于调试运行在目标机操作系统上的应用程序；缺点：不宜用来调试内核代码及启动代码

② 硬件方式：优点：对目标机性能影响小，可调试代码范围广；缺点：依赖于芯片厂商

