June 2024

嵌入式软件期末复习讲义

林星辰

前言

本讲义主要功能在于考前学习与记忆，并不具备教材学习功能。学习本讲义内容可搭配讲解视频，视频发布在下方所述 B 站号。

特别鸣谢：兰总、白总、关总、柴总、王总、郭总等，对讲义完成的大力帮助，不一一。

本讲义正式动笔于考试前，至完成仅用两天，时间之仓促，难免有些纰漏，如读者发现，可反馈到 B 站 UP《林鹤鸣 ID》（ID 号：102399253）。

版本：0.2

修改记录：

- 新增 `Makefile` 三种改写形式
- 原第 10 页-e、-s 改为大写
- 原第 30 页 Listing23 代码第 6 行，`flout` 改为 `float`
- 新增 30 页 `Makefile` 代码解析
- 代码部分重新调整顺序

2024 年 6 月 30 日

目录

1 基础命令与 shell 脚本

1.1 目录结构

/bin : 存放常用命令
/boot: 存放启动程序
/dev: 存放设备文件
/etc: 存放启动，关闭，配置程序与文件
/home: 用户工作根目录
/lib: 存放共享链接库
/root: 超级用户的工作目录
/sbin: 系统管理员的常用管理程序
/tmp: 存放临时文件
/lost+found: 系统出现异常时，用于保存部分资料
/misc : 一些实用工具
/mnt: 光驱、硬盘等的挂载点
/media: 光驱的自动挂载点
/proc: 操作系统的实时信息
/sys: 系统中的硬件设备信息
/srv: 服务启动后需要提取的信息
/var: 主要存放系统日志
/usr: 存放用户程序
/tftpboot: tftp服务器的服务目录
/selinux: redhat提供的selinux安全程序

1.2 基本操作

```
useradd smb  # 添加名字为smb的用户
passwd smb  # 修改smb用户的密码
su root    # 切换到root用户

shutdown
shutdown now  # 立刻关机
shutdown -r +5 # 系统在5分钟后关机并且马上重新启动
shutdown -h now # 系统马上关机并且不重新启动

cp [选项] 源文件或目录 目标文件或目录
cp /home/test /tmp/   # 将/home目录下的test文件copy到/tmp目录
cp -r /home/dir1 /tmp/ # 将/home目录下的dir1目录copy到/tmp目录下

mv [选项] 源文件或目录 目标文件或目录
mv /home/dir1 /tmp/      # 将 /home目录下dir1目录移动（剪切）到/tmp目录下
mv /home/test /home/test1   # 将 /home目录下的test文件更名为test1
```

```
rm [选项] 文件或目录
rm /home/test    #  删除/home目录下的test文件
rm -r /home/dir   #  删除/home目录下的dir目录

mkdir [选项] 目录名
mkdir /home/workdir # 在/home目录下创建workdir目录
mkdir -p /home/dir1/dir2 # 创建/home/dir1/dir2目录，如果dir1不存在，先创建dir1

cd 目录名
cd /home/ # 进入/home目录

pwd  # 显示当前工作目录的绝对路径

ls [选项] [目录或文件]
ls /home # 显示/home目录下的文件与目录（不包含隐藏文件）
ls -a /home # 显示/home目录下的所有文件与目录（包含隐藏文件）
ls -l /home # 显示/home目录下的文件与目录的详细信息
ls -c /home # 显示/home目录下的文件与目录,按修改时间排序

tar [选项] 目录或文件 # 打包与压缩
tar cvf tmp.tar /home/tmp # 将/home/tmp目录下的所有文件和目录打包成一个tmp.tar文件
tar xvf tmp.tar   # 将打包文件tmp.tar在当前目录下解开
tar cvzf tmp.tar.gz /home/tmp # 将/home/tmp目录下的所有文件和目录打包并压缩成一个tmp.tar.gz文
tar xvzf tmp.tar.gz # 将打包压缩文件tmp.tar.gz在当前目录下解开
-c: 打包 -x: 解压 -z: gzip的属性压缩包 -v: 显示所有过程 -f: 后面接文件名

unzip tmp.zip # 解压tmp.zip文件

chmod [who] [+ | - | =] [mode] 文件名
who:
 u 表示文件的所有者。
 g 表示与文件所有者同组的用户。
 o 表示“其他用户”。
 a 表示“所有用户”。它是系统默认值。
mode:
 + 添加某个权限
 - 取消某个权限
 = 赋予给定权限
 r 可读 (4)
 w 可写 (2)
 x 可执行 (1)
chmod a=rwx file # 给所有用户赋予可读可写可执行file文件权限
chmod 777 file # 与上述效果相同
```

```
chmod 761 hello.c # 赋予User权限为rwx, group权限为rw-,赋予其他用户权限为--x

df -k # 以KB为单位显示磁盘使用情况
du -b ipc # 以字节为单位显示ipc这个目录的大小

ifconfig [选项] [网络接口] # 网络配置
ifconfig eth0 192.168.0.1 # 配置eth0这一网卡的ip地址为192.168.0.1
ifconfig eth0 down  # 暂停eth0这一网卡的工作
ifconfig eth0 up   # 恢复eth0这一网卡的工作

netstat -a # 查看系统中所有的网络监听端口。

rpm [选项] [安装文件]
rpm -ivh tftp.rpm # 安装名字为tftp的文件
rpm -qa      # 列出所有已安装rpm包
rpm -e name    # 卸载名字为name的rpm包

ps # 查看进程

kill [选项] 进程号
kill -s SIGKILL 4096 # 杀死4096号进程

top  # 查看系统中的进程对cpu、内存等的占用情况。
```

1.3 Shell 脚本

Shell 脚本是一个包含一系列命令序列的文本文件。当运行这个脚本文件时，文件中包含的命令序列将得到执行。

1.3.1 注释、变量、输出

- `$#`：传入脚本的命令行参数个数
- `$*`：所有命令行参数值，在各个参数值之间留有空格
- `$0`：命令本身（shell 文件名）
- `$1`：第一个命令行参数
- `$2`：第二个命令行参数

```
1 #!/bin/sh
2
3 name="lxc"
4 name='lxc'
5
6 echo "name is"
7 echo $name
```

```
echo ${name}er

# 思考执行./文件名 1 2 3 4 的输出结果
echo "number of vars:"$#
echo "values of vars:"$*
echo "file name:"$0
echo "value of var1:"$1
echo "value of var2:"$2
echo "value of var3:"$3
echo "value of var4:"$4
```

1.3.2 判断

Listing 1: 单层 if

```
if [ 条件 ]
then
        代码块
fi
```

Listing 2: 单层 if-else

```
if [ 条件 ]
then
        代码块
else
        代码块
fi
```

Listing 3: 多层 if-else if-else

```
if [ 条件 ]
then
        代码块
else if [ 条件 ]
        then
                代码块
        else
                代码块
        fi
fi
```

Listing 4: 多层 if-elif-elif-else

```
if [ 条件 ]
then
   代码块
elif [ 条件 ]
        then
                代码块
        elif [ 条件 ]
                then
                代码块
        else
                代码块
fi
```

比较操作	整数操作	字符串操作
相同	-eq	=
不同	-ne	!=
大于	-gt	>
小于	-lt	<
大于或等于	-ge	-
小于或等于	-le	-
为空	-z	-
不为空	-n	-

- 比较整数 a 和 b 是否相等:
  `if [ $a = $b ]` （也可用 `-eq`）
- 判断整数 a 是否大于整数 b:
  `if [ $a -gt $b ]`
- 比较字符串 a 和 b 是否相等:
  `if [ $a = $b ]`
- 判断字符串 a 是否为空:
  `if [ -z $a ]`
- 判断整数变量 a 是否大于 b:
  `if [ $a -gt $b ]`

- `-e` 文件已经存在
- `-f` 文件是普通文件

- -s 文件大小不为零
- -d 文件是一个目录
- -r 文件对当前用户可以读取
- -w 文件对当前用户可以写入
- -x 文件对当前用户可以执行

1.3.3 循环

```
for var in val1 val2
val3
do
语句 1
语句 2
...
done
```

```
while condition
do
语句 1
语句 2
...
done
```

```
until condition
do
语句 1
语句 2
...
done
```

2 文件编程

- 所有打开的文件都对应一个文件描述符。文件描述符的本质是一个非负整数。当打开一个文件时，该整数由系统来分配
- 系统调用-创建，`int creat(const char *filename, mode_t mode)`
  - 0：占位符（可忽略）
  - 7：文件的所有者可读，可写，可执行
  - 5：文件所有者所在的组，跟用户一组可读，可执行
  - 1：其他用户，可执行
  - `S_IRUSR` (Read permission, owner)
    文件所有者的读权限。
    等价数字：`0400`
    缩写含义：`S_IRUSR` 是"Set user read" 的缩写。
  - `S_IWUSR` (Write permission, owner)
    文件所有者的写权限。
    等价数字：`0200`
    缩写含义：`S_IWUSR` 是"Set user write" 的缩写。
  - `S_IXUSR` (Execute/search permission, owner)
    文件所有者的执行权限。
    等价数字：`0100`
    缩写含义：`S_IXUSR` 是"Set user execute" 的缩写。
  - `S_IRWXU` (Read, write, and execute/search by owner)
    文件所有者的读、写和执行权限。
    等价数字：`0700` (八进制)
    缩写含义：`S_IRWXU` 是"Set user read, write, execute" 的缩写。
- 系统调用-打开
  - `int open(const char *pathname, int flags)`
  - `int open(const char *pathname, int flags, mode_t mode)`
    - pathname：要打开的文件名 (包含路径，缺省为当前路径)
    - flags：打开标志
    - Mode: 访问权限
  - 常见的打开标志 flags
    - `O_RDONLY` 只读方式打开
    - `O_WRONLY` 只写方式打开
    - `O_RDWR` 读写方式打开
    - `O_APPEND` 追加方式打开
    - `O_CREAT` 创建一个文件（必须使用第二个 open）

- 系统调用-关闭：`int close(int fd)`，fd: 文件描述符
- 系统调用-读：`int read(int fd, const void *buf, size_t length)`
  - 从文件描述符 `fd` 所指定的文件中读取
  - `length` 个字节到 `buf` 所指向的缓冲区中
  - 返回值为实际读取的字节数
- `int write(int fd, const void *buf, size_t length)`
  - 把 `length` 个字节从 `buf` 指向的缓冲区中写到文件描述符 `fd` 所指向的文件中
  - 返回值为实际写入的字节数

3 Makefile

3.1 GCC

- gcc（GNU C Compiler）能将 C、C++ 语言源程序、汇编程序编译、链接成可执行文件
- gcc 的四个编译流程：预处理、编译、汇编、链接一步完成
- .c 为后缀的文件：C 语言源代码文件
- .o 为后缀的文件：是编译后的目标文件
- -E：只进行预处理，不做其他处理
- -S：只是编译不汇编，生成汇编代码
- -c：只是编译不连接，生成目标文件“.o”
- -o：将编译好了的.o 链接库，生成可执行文件（output_filename：确定可执行文件的名称）
- -g：产生调试工具 (GNU 的 gdb) 所必要的符号信息
- -O2：完成程序的优化工作。

  编译和运行 hello.c 文件

```
gcc hello.c -o hello

gcc -E hello.c -o hello.i
gcc -S hello.i -o hello.s
gcc -c hello.s -o hello.o
gcc hello.o -o hello
./hello

gcc -O2 hello.c -o hello1
```

3.2 GDB

GDB 是 GNU 发布的一款功能强大的程序调试工具，主要完成以下三个方面的功能：

1. 启动被调试程序。
2. 让被调试的程序在指定的位置停住。
3. 当程序被停住时，可以检查程序状态（如变量值）。

GDB 调试过程

1. 将 tst.c 文件编程成包含标准调试信息的文件 tst `gcc -g tst.c -o tst`
2. 启动 gdb 进行调试 `gdb tst`
3. l（list）命令用于查看文件
4. b（breakpoint）命令设置断点：b + 行号

5. info 命令查看断点情况：`info b`
6. r（run）命令：用于运行代码，从首行开始，到断点处停止
7. p（print）命令：查看变量的值，p + 变量名
8. s（step）、n（next）：它们的区别在于：如果有函数调用的时候，“s”会进入该函；数而“n”不会进入该函数
9. finish 命令，运行程序，直到当前函数结束

3.3 Makefile

- GNU 的 make 能够使整个软件工程的编译、链接只需要一个命令就可以完成。
- make 命令默认在当前目录下寻找名字为 makefile 或者 Makefile 的工程文件，当名字为其他时使用如下的方法
  `make -f` 文件名
- Makefile 文件描述了整个工程的编译，连接等规则编译，连接等规则。

```
目标: 依赖
	命令
```

Listing 5: Makefile 例

```
hello: main.o func1.o func2.o
	gcc main.o func1.o func2.o -o hello
main.o : main.c
	gcc -c main.c
func1.o : func1.c
	gcc -c func1.c
func2.o : func2.c
	gcc -c func2.c
.PHONY : clean
clean :
	rm -f hello main.o func1.o func2.o
```

  - 目标依赖命令解释
    - 目标 hello 依赖于三个对象文件：main.o、func1.o 和 func2.o。
    - 如果这些对象文件不存在或已经更新，Makefile 会运行第二行命令，使用 gcc 将这三个对象文件链接成一个可执行文件 hello。
    - 其他行同理
  - 清理生成的文件
    - 目标 clean 不依赖于任何文件，并标记为伪目标（.PHONY），这意味着即使存在名为 clean 的文件，它也不会影响这个目标。(这一行可以没有)
    - 当运行 make clean 时，Makefile 会运行第二行命令，删除可执行文件 hello 和所有的对象文件 main.o、func1.o 和 func2.o。

- 变量：`$(变量名)`

  如果要为 hello 目标添加一个依赖，如：func3.o，该如何修改?

```
#答案1:
hello: main.o func1.o func2.o func3.o
	gcc main.o func1.o func2.o func3.o -o hello

#答案2:
obj=main.o func1.o func2.o func3.o
hello: $(obj)
	gcc $(obj) -o hello
```

- 自动化变量
  - `$^` ： 代表所有的依赖文件
  - `$@`: 代表目标
  - `$<`: 代表第一个依赖文件

```
#改写前
hello: main.o func1.o func2.o
	gcc main.o func1.o func2.o -o hello

#改写后
hello: main.o func1.o func2.o
	gcc $^ -o $@
```

- 模式规则 `%.o:%.c`: 表示所有.o 文件都可以从对应的.c 文件生成。

4 进程控制

- 必备头文件：
  - `#include<sys/types.h>`
  - `#include <unistd.h>`
  - `#include <sys/wait.h>`
- 重要函数：
  - `pid_t getpid(void)`：获取本进程 ID
  - `pid_t getppid(void)`：获取父进程 ID
- `pid_t fork(void)`：创建子进程
  - 父进程中：fork() 返回子进程的进程 ID（PID）。这通常是一个正整数
  - 子进程中：fork() 返回 0
  - 错误时：fork() 返回 -1
  - 虽然子进程是父进程的副本，但它们有各自的进程 ID。
  - 子进程和父进程在相同的地址空间中运行，但它们的变量和资源是独立的。
- `pid_t vfork(void)`：创建子进程
  - 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
  - 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
- `pid_t wait (int * status)` 阻塞该进程，直到其某个子进程退出。
- 思考以下两程序运行结果

Listing 6: fork 创建子进程

```c
#include <unistd.h>
#include <stdio.h>
int main(void)
{
        pid_t pid;
        int count=0;

        pid = fork();

        count++;
        printf("count = %d\n", count );

        return 0;
}
```

Listing 7: vfork 创建子进程

```c
#include <unistd.h>
#include <stdio.h>
int main(void)
{
        pid_t pid;
        int count=0;

        pid = vfork();

        count++;
        printf("count = %d\n", count );

        return 0;
}
```

5 进程通信

为什么进程间需要通信?

- 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，通知它们发生了某种事件
- 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）

5.1 管道通信

- 管道是单向的、先进先出的
- 一个进程（写进程）在管道的尾部写入数据，filedes[1] 用于写管道
- 另一个进程（读进程）从管道的头部读出数据，filedes[0] 用于读管道
- 数据被一个进程读出后，将被从管道中删除
- 无名管道：用于父进程和子进程间的通信
- 有名管道：用于运行于同一系统中的任意两个进程间的通信

5.1.1 无名管道

1. 父进程调用 `pipe` 开辟管道，得到两个文件描述符指向管道的两端。
2. 父进程调用 `fork` 创建子进程，那么子进程也有两个文件描述符指向同一管道（`fork` 只复制父进程的文件描述符，不复制内核中的管道）。
3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从读端流出，这样就实现了进程间通信。
4. 必须在系统调用 fork( ) 前调用 pipe( )，否则子进程将不会继承文件描述符。

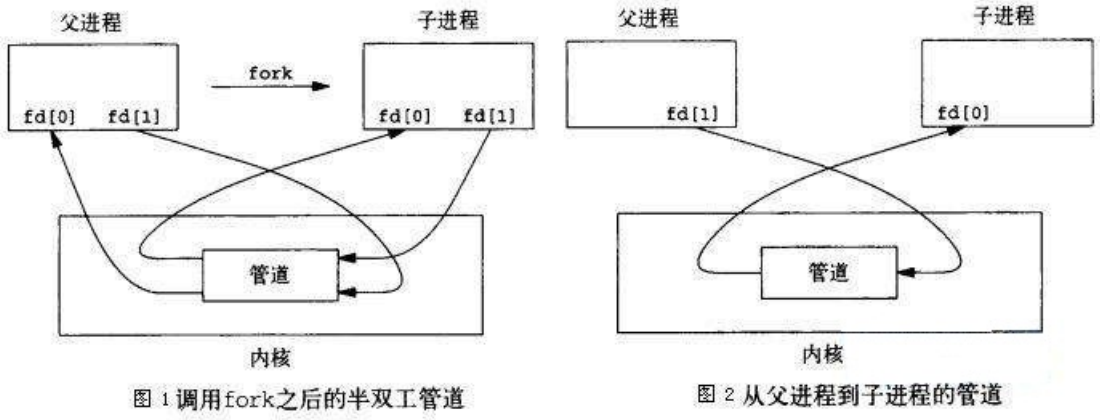

图 1 调用fork之后的半双工管道　　图 2 从父进程到子进程的管道

图 1: 子进程与父进程通过创建的管道进行通信

Listing 8: 管道通信示例代码

```c
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    int pipe_fd[2]; // 声明一个整数数组，用于存储管道的两个文件描述符

    // 创建管道，pipe_fd[0] 用于读端，pipe_fd[1] 用于写端
    if (pipe(pipe_fd) < 0) {
        // 如果管道创建失败，输出错误信息并返回 -1
        printf("pipe create error\n");
        return -1;
    } else {
        // 如果管道创建成功，输出成功信息
        printf("pipe create success\n");
    }

    // 关闭管道的读端和写端文件描述符
    close(pipe_fd[0]);
    close(pipe_fd[1]);

    return 0;
}
```

5.1.2 有名管道

```c
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char * pathname, mode_t mode);
```

- pathname：FIFO 文件名，mode：属性（见文件操作章节）
- 命名管道 FIFO 是一个设备文件，一般的文件访问函数都适用：open、close、read、write

5.2 信号通讯

- SIGINT 信号
  - 由用户通过 Ctrl+C 发出，表示希望中断（终止）当前运行的进程。
  - 当用户在终端中按下 Ctrl+C 时，会产生 **SIGINT 信号**。
  - 操作系统捕捉到这个信号并传递给当前前台进程，你的程序中定义的信号处理函数 my_func 会被调用，并打印 "I have get SIGINT".
- SIGQUIT 信号
  - 由用户通过 Ctrl+\ 发出，表示希望终止当前运行的进程并生成核心转储文件。
  - 当用户在终端中按下 Ctrl+\ 时，会产生 **SIGQUIT 信号**。
  - 操作系统捕捉到这个信号并传递给当前前台进程，你的程序中定义的信号处理函数 my_func 会被调用，并打印 "I have get SIGQUIT".

- kill 信号发送函数

Listing 9: 信号通信示例代码

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void my_func(int sign_no)
{
	if(sign_no==SIGINT)
		printf("I have get SIGINT\n");
	else if(sign_no==SIGQUIT)
		printf("I have get SIGQUIT\n");
}
int main()
{
	printf("Waiting for signal SIGINT or SIGQUIT \n ");
	signal(SIGINT, my_func);
	signal(SIGQUIT, my_func);
	pause();
	exit(0);
}
```

5.3 共享内存

- **创建共享内存**：使用 `shmget` 函数。

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

- **映射共享内存**：将这段创建的共享内存映射到具体的进程空间去，使用 `shmat` 函数。

```c
int shmat(int shmid, char *shmaddr, int flag);
```

6 内核

6.1 内核模块

内核题目

根据下列功能要求，编写程序、代码及操作命令。

- 编写内核模块，分别实现一个字符型，一个浮点型模块参数
- 编写（1）内核模块的 Makefile
- 安装、查看和卸载内核模块的操作命令

- 模块的安装、卸载、查看
  - 加载 insmod (insmod hello.ko)
  - 卸载 rmmod (rmmod hello)
  - 查看 lsmod
  - 依赖加载 modprobe （modprobe hello）
- 模块可选信息
  - 许可证申明：宏 `MODULE_LICENSE` 用来告知内核, 该模块带有一个许可证
  - 作者申明：`MODULE_AUTHOR("LXC");`
  - 模块参数：`module_param`，用于在加载模块时传递参数给模块。

    `module_param(name, type, perm);`
    - name 是模块参数的名称
    - type 是这个参数的类型：
      - bool: 布尔型 int: 整型 charp: 字符串型 float: 浮点型
    - perm 是模块参数的访问权限。
      - `S_IRUGO`：任何用户都对/sys/module 中出现的该参数具有读权限
      - `S_IWUSR`：允许 root 用户修改/sys/module 中出现的该参数

1. 编写 `param.c` 和 `Makefile` 文件
2. 编译内核模块：`make`
3. 加载内核模块
   (a) 不加内核参数，使用其默认值 `insmod param.ko`
   (b) 带参数传递 `insmod param.ko name="Tom" age=10`
4. 查看内核是否被加载：`lsmod | grep param`
5. 卸载模块：`rmmod param`（重新加载模块需要先卸载）
6. 查看到打印输出信息：`cat /var/log/message`

Listing 10: 样例的 param.c

```c
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");

static char *name = "DAVID";
static int age=30;
module_param(age, int ,S_IRUGO);
module_param(name, charp ,S_IRUGO);
static int __init hello_init(void)
{
	printk(KERN_EMERG "Name:%s\n",name);
	printk(KERN_EMERG "Age:%d\n",age);
	return 0;
}
static void __exit hello_exit(void)
{
	printk(KERN_EMERG"Module exit!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Listing 11: 样例的 Makefile

```
ifneq ($(KERNELRELEASE),)

obj-m := param.o

else

KDIR := /lib/modules/2.6.18-53.el5/build
all:
	make -C $(KDIR) M=$(PWD) modules
clean:
	rm -f *.ko *.o *.mod.o *.mod.c *.symvers

endif
```

6.2 Mini2440 开发板内核制作

1. 通过 SMB 服务将 `linux.2.6.29.tar.bz2` 内核拷贝到 Linux 虚拟机。
2. 解压内核到当前目录：`tar jxvf linux.2.6.29.tar.bz2`
3. 清理内核中间文件，配置文件：`cd linux-2.6.29 make distclean`
4. 选择参考配置文件：`cp config-mini2440 .config`
5. 安装 ncurses 支持包。
6. 配置内核：`make menuconfig ARCH=arm CROSS_COMPILE=arm-linux-`
7. 保存退出。
8. 从 u-boot 源代码(编译之后的)中把 `mkimage` 拷贝到 `/bin` 文件目录下：`cp ../u-boot-2008.10/tools/mki /bin`

9. 编译内核：`make uImage ARCH=arm CROSS_COMPILE=arm-linux-`
10. 编译完成后，内核映像 `uImage` 位于 `arch/arm/boot` 目录下。

6.3 X86 平台内核配置、编译和安装

1. 解压内核：`tar jxvf linux.2.6.29.tar.bz2`
2. 清理内核中间文件，配置文件：`cd linux-2.6.29 make distclean`
3. 使用正在运行的 RHEL5 的内核配置作为参考配置文件:`cp /boot/config-2.6.18-53.el5 .config`
4. 配置内核：`make menuconfig`
5. 保存退出。
6. 编译内核：`make bzImage` 生成的内核映像 `bzImage` 位于 `arch/x86/boot/` 目录下。
7. 编译内核模块：`make modules`
8. 安装内核模块：`make modules_install` 完成安装后，编译好的内核模块会从内核源代码目录拷贝至 `/lib/modules` 下。
9. 制作 init ramdisk(提供一种让内核可以简单使用 ramdisk 的能力):`mkinitrd initrd-2.6.29 2.6.29`
10. 安装内核
    (a) 将内核和 init ramdisk 拷贝至 `/boot` 目录:
        - `cp linux-2.6.29/arch/x86/boot/bzImage /boot/vmlinuz-2.6.29`
        - `cp /home/guoqian/4-1-1/initrd-2.6.29 /boot/`

    (b) 为了让 grub 在启动时能提供一项我们自己制作的 Linux 内核的选择项，修改 grub 的配置文件：`vi /etc/grub.conf` 在配置文件中添加如下代码:

```
/*************************************************************/
title my linux(2.6.29) /*选择项名字*/
   root (hd0,0)
   kernel /vmlinuz-2.6.29 ro root=LABEL=/ rhgb quiet
   initrd /initrd-2.6.29
/*************************************************************/
```

11. 重新启动虚拟机测试新内核是否能使用。

7 交叉编译与 uboot

内核题目

根据下列功能要求，编写程序、代码及操作命令。

- linux 交叉编译工具链的安装、配置、版本检测。
- 使用该工具链编译 helloworld 应用程序、查看可执行文件格式、对可执行文件反汇编
- 制作在 smdk4610 开发板使用的 u-boot
  工具链版本：arm-linux-gcc-4.7.7.tgz
  u-boot 版本：u-boot-2022.04.tar.bz2

7.1 交叉编译工具链的安装

1. 解压工具链到根目录下：`tar zxvf arm-linux-gcc-4.3.2.tgz -C /`
2. 修改`/etc/profile`，添加 `pathmunge /usr/local/arm/4.3.2/bin`
3. 执行 `source /etc/profile`
4. 版本查看 `arm-linux-gcc -v`

7.2 交叉编译器工具的使用

1. 编译器：`arm-linux-gcc`
2. 反汇编工具：`arm-linux-objdump`
3. ELF 文件查看工具：`arm-linux-readelf`

Listing 12: 样例的 Makefile

```
// 编译
arm-linux-gcc hello.c -o hello

// 查看可执行文件格式
file hello

// objdump反汇编命令的使用
arm-linux-objdump -D -S hello >log

// readelf命令的使用
arm-linux-readelf -a hello >log // 显示所有信息
arm-linux-readelf -d hello >log // 查看hello使用的动态库
```

7.3 uBoot

1. 解压：`tar jxvf u-boot-2008.10.tar.bz2`
2. 选择配置：`make smdk2410_config`
3. 编译内核：`make CROSS_COMPILE=arm-linux`
   - compile：编译，cross：交叉
   - 这条命令告诉 make 使用以 arm-linux- 为前缀的交叉编译工具链来编译目标代码

8 代码

8.1 Shell

Listing 13: 脚本编程-监视文件

```
#!/bin/bash

#判断命令行是否代带有两个文件名的参数
if [ "$1" = "" ] || [ "$2" = "" ]
then
    echo "Please enter file name"
    exit 1
fi

#判断目标文件是否存在
if [ -e $2 ]
then
        echo "The file already exists"
        until [ ! -f $2 ] #监视该文件是否被删除
        do
        sleep 1
    done
        echo "The file have been deleted"
fi

#执行源文件移动为目标文件的命令
if [ ! `mv $1 $2` ]
then
    echo "mv sucessful"
else
    echo "mv error"
fi
```

- `if [ "$1" = "" ] || [ "$2" = "" ]`: 判断传入的两个文件名是否为空
- `if [ -e $2 ]` 检查第二个命令行参数 $2 指定的文件是否存在
- `until [ ! -f $2` -f 是一个文件测试选项，用于检查文件是否为普通文件。循环会一直运行，直到指定的文件被删除
- 每次循环中，脚本会暂停 1 秒钟（避免占用过多 CPU 资源），然后再次检查文件是否存在。
- mv $1 $2 命令尝试将第一个命令行参数 $1（源文件）移动或重命名为第二个命令行参数 $2（目标文件）。
- mv 命令成功执行时返回值为 0，失败时返回非零值。

8.2 文件编程

Listing 14: 打开文件

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    int fd;

    // 如果程序的参数少于2个（即没有提供文件路径），输出提示信息并退出
    if (argc < 2) {
        puts("please input the open file pathname!\n");
        exit(1);
    }

    // 尝试打开或创建文件
    // O_CREAT: 如果文件不存在则创建
    // O_RDWR: 以可读写方式打开文件
    // 0755: 新创建文件的权限，表示文件所有者可以读写执行，其他用户可以读和执行
    if ((fd = open(argv[1], O_CREAT | O_RDWR, 0755)) < 0) {
        perror("open file failure!\n"); // 如果打开失败，输出错误信息并退出
        exit(1);
    } else {
        printf("open file %d success!\n", fd); // 如果成功打开文件，输出文件描述符
    }

    close(fd); // 关闭文件描述符
    exit(0); // 退出程序

    return 0;
}
```

8.3 Makefile

Listing 15: Makefile 练习：三个 C 文件

```c
// fun.c
#include <stdio.h>
extern int max_fun(int x,int y);

// fun.h
#include "fun.h"

int max_fun(int x,int y)
{
    if(x>=y) return x;
        else return y;
}

// main.c
#include "fun.h"

int main(void)
{
    int a,b;
        printf("Please enter the number a an b\n");
        scanf("%d%d",&a,&b);
        int max=0;
        max=max_fun(a,b);
        printf("The max number is %d\n",max);
        return 0;
}
```

Listing 16: Makefile 练习：基本 Makefile 文件

```
main:main.o fun.o
	gcc main.o fun.o -o main
main.o:main.c fun.h
	gcc -c main.c -o main.o
fun.o:fun.c fun.h
	gcc -c fun.c -o fun.o
clean:
	rm -f main *.o
```

```
OBJS=main.o fun.o
CC=gcc
CELAGS=-c
main:$(OBJS)
	$(CC) $(OBJS) -o main
main.o:main.c fun.h
	$(CC) $(CELAGS) main.c -o main.o
fun.o:fun.c fun.h
	$(CC) $(CELAGS) fun.c -o fun.o
clean:
	rm -f main *.o
```

```
OBJS=main.o fun.o
CC=gcc
CELAGS=-c
main:$(OBJS)
	$(CC) $^ -o $@
clean:
	rm -f main *.o
```

```
OBJS=main.o fun.o
CC=gcc
CELAGS=-c
main:$(OBJS)
	$(CC) $^ -o $@
%.o:%.c
	$(CC) $(CELAGS) $< -o $@
clean:
	rm -f main *.o
```

8.4 子进程

Listing 17: fork 函数

```c
#include <sys/types.h>
#include <unistd.h>
main()
{
	pid_t pid;

	/*此时仅有一个进程*/
	pid=fork();

		/*此时已经有两个进程在同时运行*/
	if(pid<0)
			printf("error in fork!");
	else if(pid==0)
			printf("I am the child process, ID is %d\n",getpid());
	else
			printf("I am the parent process,ID is %d\n",getpid());//父进程是谁?

	wait(NULL);//等待子进程结束
	return 0;
}
```

当一个子进程终止时，它的退出状态信息（如返回码）仍然保存在系统中，直到父进程读取它。这种终止了但仍在系统进程表中占据条目的进程称为僵尸进程。

父进程可以使用 wait() 或 waitpid() 来等待子进程结束，并读取其退出状态。这样可以清除子进程的信息，防止其成为僵尸进程。

Listing 18: wait 函数

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main()
{
	pid_t pc, pr;
	pc = fork();

	if (pc < 0) /* 如果出错 */
		printf("error ocurred!/n");
	else if (pc == 0)
	{ /* 如果是子进程 */
		printf("This is child process with pid of %d\n", getpid());
		sleep(10); /* 睡眠10秒钟 */
	}
	else
	{ /* 如果是父进程 */
		pr = wait(NULL); /* 在这里等待 */
		printf("I catched a child process with pid of %d\n", pr);
	}
	exit(0);
}
```

8.5 信号处理

Listing 19: 信号处理

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

/* 自定义信号处理函数 */
void my_func(int sign_no)
{
    if (sign_no == SIGBUS)
        printf("I have get SIGBUS\n");
}

int main()
{
    printf("Waiting for signal SIGBUS\n");

    /* 注册信号处理函数 */
    signal(SIGBUS, my_func);

    pause(); /* 将进程挂起直到捕捉到信号为止 */

    exit(0);
}
```

- 运行这个程序时，它会输出 Waiting for signal SIGBUS 并进入挂起状态
- 在另一终端输入 `kill -BUS 17778`：向进程发送 SIGBUS 信号（假设 17778 是这个程序的 PID）
- 程序会捕捉到这个信号，并调用 my_func 函数打印 I have get SIGBUS

8.6 共享内存

Listing 20: shm1.c

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"

int main(void) {
    int running = 1; // 用于控制循环的标志
    void *shared_memory = (void *)0; // 指向共享内存的指针
    struct shared_use_st *shared_stuff; // 共享内存结构体指针
    int shmid; // 共享内存标识符

    // 创建共享内存
    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }

    // 映射共享内存
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (int)shared_memory);

    // 让结构体指针指向这块共享内存
    shared_stuff = (struct shared_use_st *)shared_memory;

    // 初始化标志位
    shared_stuff->written_by_you = 0;

    // 循环读取输入数据并写入共享内存，直到输入 "end"
    while (running) {
        // 如果客户端尚未读取数据，则等待
        while (shared_stuff->written_by_you == 1) {
            sleep(1); // 等待1秒
            printf("waiting for client...\n");
        }
        // 获取输入数据
        printf("Enter some text: ");
        fgets(shared_stuff->some_text, TEXT_SZ, stdin);

        // 更新标志位
        shared_stuff->written_by_you = 1;

        // 检查是否输入了 "end"
        if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
            running = 0; // 结束循环
        }
    }

```

```
    // 分离共享内存
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

Listing 21: shm2.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shm_com.h"

int main(void) {
    int running = 1; // 用于控制循环的标志
    void *shared_memory = (void *)0; // 指向共享内存的指针
    struct shared_use_st *shared_stuff; // 共享内存结构体指针
    int shmid; // 共享内存标识符

    // 创建共享内存
    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }

    // 映射共享内存
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (int)shared_memory);

    // 让结构体指针指向这块共享内存
    shared_stuff = (struct shared_use_st *)shared_memory;

    // 初始化标志位
    shared_stuff->written_by_you = 0;

    // 循环读取共享内存中的数据，直到读取到 "end"
    while (running) {
        if (shared_stuff->written_by_you) {
            printf("You wrote: %s", shared_stuff->some_text);

            // 更新标志位
            shared_stuff->written_by_you = 0;

            // 检查是否读取到 "end"
            if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
                running = 0; // 结束循环
            }
        } else {
```

```
            sleep(1); // 等待1秒
        }
    }

    // 分离共享内存
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

Listing 22: shm_com.h

```
#ifndef SHM_COM_H
#define SHM_COM_H

#define TEXT_SZ 2048

struct shared_use_st {
    int written_by_you; // 标志位，指示数据是否已被写入
    char some_text[TEXT_SZ]; // 用于存储文本数据的共享内存区
};

#endif
```

8.7 内核

Listing 23: 例题的 hello.c

```c
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");

static char *name = "lxc";
static int heavy=4643; // 内核代码中不推荐直接使用浮点运算
                                        // 采用将浮点数放大成整数来进行处理的方法。
module_param(heavy, int , 0444);
module_param(name, charp , 0444);

static int __init hello_init(void)
{
        printk(KERN_EMERG "Name:%s\n",name);
        printk(KERN_EMERG "Heavy:%d.%02d\n",heavy / 100, heavy % 100);
        return 0;
}

static void __exit hello_exit(void)
{
        printk(KERN_EMERG"Module exit!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Listing 24: 例题的 Makefile

```
ifneq ($(KERNELRELEASE),)
    obj-m := hello.o
else
    KDIR := /lib/modules/2.6.18-53.el5/build
    all:
        make -C $(KDIR) M=$(PWD) modules
    clean:
        rm -f *.ko *.o *.mod.o *.mod.c *.symvers
endif
```

- `ifneq ($(KERNELRELEASE),)`
  这行检查 KERNELRELEASE 变量是否为空。KERNELRELEASE 是由内核构建系统在编译内核模块时自动设置的变量。通过检查这个变量，可以确定当前是否在内核构建环境中。
- 如果 KERNELRELEASE 不为空，表示我们正在内核构建环境中。
- `obj-m := hello.o`
  这行在内核构建环境中执行，表示编译 hello.c 文件生成 hello.o 目标模块。obj-m 是内核构建系统中用于指定要构建的模块文件的变量。hello.o 是编译 hello.c 源文件后生成的目标文件，最终会被打包成一个 .ko（内核对象）文件。
- `else`
  如果 KERNELRELEASE 为空，表示我们不在内核构建环境中，执行以下的用户空间命令。

- `KDIR := /lib/modules/2.6.18-53.el5/build`
  这行定义了一个变量 `KDIR`，它表示当前运行的内核版本的构建目录路径。在这个例子中，路径是 `/lib/modules/2.6.18-53.el5/build`。这个路径通常是一个符号链接，指向当前内核源码树的构建目录。

- `all:`
  这行定义了一个名为 `all` 的目标。`all` 通常是默认目标，即如果运行 `make` 命令而不指定目标，Makefile 将执行 `all` 目标中的命令。

- `make -C ${KDIR} M=${PWD} modules`
  这是 `all` 目标的命令。它解释如下：
  - `make -C ${KDIR}`：进入内核构建目录并执行那里的 Makefile。
  - `M=${PWD}`：指定要构建的模块源文件所在的目录，即当前目录（由 `${PWD}` 表示）。
  - `modules`：这个目标告诉内核构建系统编译模块。

  综上，这条命令会在内核构建目录中调用 Makefile，并在当前目录中根据 `obj-m` 变量中指定的目标文件编译模块。

- `clean:`
  这行定义了一个名为 `clean` 的目标。`clean` 目标通常用于删除编译过程中生成的临时文件和目标文件。

- `rm -f *.ko *.o *.mod.o *.mod.c *.symvers`
  这是 `clean` 目标的命令，解释如下：
  - `rm -f`：强制删除文件而不提示。
  - `*.ko`：删除所有扩展名为 `.ko` 的文件（内核模块文件）。
  - `*.o`：删除所有扩展名为 `.o` 的文件（目标文件）。
  - `*.mod.o`：删除所有扩展名为 `.mod.o` 的文件（模块目标文件）。
  - `*.mod.c`：删除所有扩展名为 `.mod.c` 的文件（模块源文件）。
  - `*.symvers`：删除所有扩展名为 `.symvers` 的文件（符号版本文件）。

- `endif`
  结束 `ifneq` 条件语句。