嵌入式开发

Boyuan Zhao

2023 年 5 月 30 日

前言

临时抱佛脚.

Boyuan Zhao 2023 年 5 月 30 日

目录

第一章　STM32 开发

在这里可以输入笔记的内容.

1.1　STM32 的基本知识

这是笔记的正文部分.

第二章　Linux 开发

2.1　Linux 操作基本指令

2.1.1　目录操作

```
pwd             # 查看当前目录
cd ..           # 返回上一层目录
cd <directory>  # 进入特定目录
cd ~            # 返回用户主目录

mkdir <dir_name> # 创建新目录
rmdir <dir_name> # 删除空目录
rm -r <dir_name> # 删除非空目录

mv <old_dir> <new_dir> # 重命名或移动目录
```

2.1.2　文件操作

```
touch <filename>  # 创建新文件
ls                # 列出目录内容
ls -l             # 详细列出目录内容
```

```
# 以下是一个文件权限的示例:
# -rw-rw-r--. 1 test test  0 Mar 25 01:44 file1
# drwxr-xr-x. 2 test test  6 Mar 25  2022 Music

# 用户, 组, 其他
# rwx   rwx   rwx
# r: read, w: write, x: execute

ls -a             # 列出所有文件, 包括隐藏文件
ls -la            # 详细列出所有文件, 包括隐藏文件

cat <filename>    # 查看文件内容
rm <filename>     # 删除文件
mv <old_file> <new_file> # 重命名或移动文件

cp <src_file> <dest_file>   # 复制文件
cp -r <src_dir> <dest_dir>  # 复制目录
```

2.1.3 系统命令

```
su                  # 切换用户
yum install openssh-server  # 安装 OpenSSH 服务器
useradd <username>  # 添加新用户
passwd [username]   # 设置或更改用户的密码
ps -aux | more      # 查看进程
kill <process_id>   # 杀死进程
top                 # 显示系统任务, 按q退出
uptime              # 显示系统运行时间
```

```
# 改变文件权限
# chmod [ugoa][+-][rwx] <filename>
# u: owner, g: group, o: other, a: all (ugo)
# 或者
# chmod <numeric_permissions> <filename>
# r:4, w:2, x:1
find . -name <name> # 在当前目录下查找具有特定名称的文件或目录
tar -zcvf test1.tar.gz test1/  # 创建gzip压缩包
tar -zxvf test1.tar.gz  # 解压gzip压缩包
```

2.1.4 网络命令

```
ifconfig  # 显示网络配置
ping <domain/ip_address>  # 发送 ICMP 请求以测试网络连接
netstat -ant  # 显示网络统计信息
```

2.1.5 Vim 操作

```
i:  Enter editing mode, insert at cursor.        # 进入编辑模式，在光标处插入
x:  Delete a character.                          # 删除一个字符
dd: Delete a line.                               # 删除一行
ndd:Delete n lines.                              # 删除n行
yy: Copy a line.                                 # 复制一行
nyy:Copy n lines.                                # 复制n行
p:  Paste.                                       # 粘贴
nG: Move cursor to line n.                       # 移动光标到n行
G:  Move to the last line.                       # 移动到最后一行
$:  Move to the end of this line.                # 移动到这一行的末尾
^:  Move to the start of this line.              # 移动到这一行的开始
```

```
/string: Search for string.                    # 查找string
?string: Search for string upwards.            # 向上查找string

Edit vim configuration:
~/.vimrc
set tabstop=4    # 设置tab宽度为4个空格
set expandtab    # 将tab转化为空格
```

2.2 Linux 基本概念

2.2.1 系统调用

系统调用是操作系统提供的用户程序向操作系统请求服务的接口。在 Linux 中，它们是用来使进程能够访问硬件设备和操作系统服务的基础。例如，当一个进程需要打开一个文件或者创建一个新进程时，它会执行一个系统调用。这些系统调用的接口在 Linux 内核中实现，当一个进程需要使用某个系统服务时，它将执行一个特定的系统调用，然后内核将执行所需的服务，然后将结果返回给进程。

2.2.2 进程

进程是 Linux 中的基本执行实体，它包含执行程序所需的所有资源，如 CPU 时间、内存空间等。每个进程都有一个唯一的进程 ID，以及一套完整的系统资源，如打开的文件、挂起的信号、内部状态、优先级等。Linux 支持多进程并发执行，因此可以在同一时间运行多个任务。

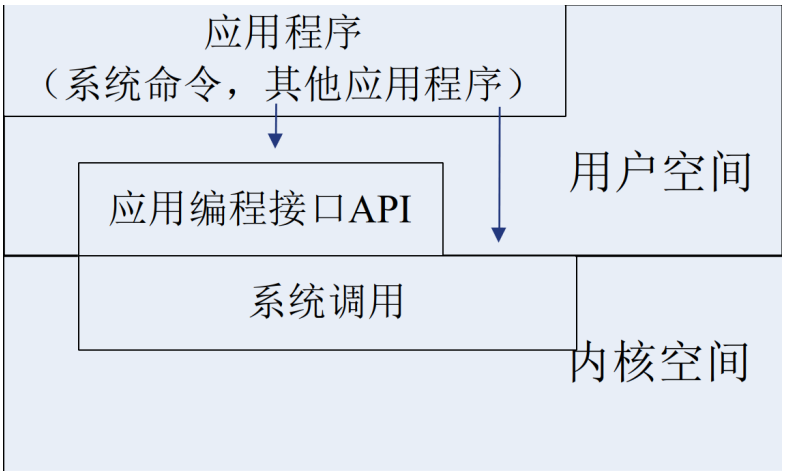

图 2.2.1:

2.2.3　进程间通信

进程间通信 (IPC, Inter-Process Communication) 指的是在同一操作系统中运行的不同进程之间传递信息或信号的机制。在 Linux 系统编程中，常见的 IPC 机制包括管道、信号、消息队列、共享内存、信号量等。

2.2.4　线程

线程是程序中的单一顺序控制流程，是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。同一进程内的多个线程将共享该进程的资源。多线程能提高程序的并发性，以提高应用程序的响应速度。在 Linux 中，线程是轻量级的进程，因此也拥有一些类似于进程的属性，但由于线程共享了进程的资源，所以线程之间的切换比进程之间的切换更加高效。

2.2.5 Linux 应用编程接口 API

Linux 应用编程接口 (API，Application Programming Interface) 是一系列函数和过程，允许创建的应用程序可以访问操作系统的特定功能和资源。Linux API 提供了大量的库函数，用于实现各种任务，如文件操作、网络通信、内存管理等。这些库函数在程序调用时，最终都会被映射到具体的系统调用。

请求机制

请求 API 函数的过程通常包括创建一个包含特定参数的数据结构，然后调用与任务相对应的 API 函数。API 函数会将参数数据结构传递给操作系统内核，内核再执行相应的系统调用。

标准

Linux API 主要遵循两个标准：POSIX 和 GNU。POSIX（Portable Operating System Interface）是 IEEE 为了实现软件在 UNIX 操作系统间的可移植性而制定的一系列 API 标准。GNU 库 (Glibc) 是 GNU 项目发布的用于提供系统调用的 C 库，它基本上遵循 POSIX 标准，但同时也提供了一些额外的 API。

实现方法

Linux API 的实现涉及到用户空间和内核空间的交互。当程序调用一个 API 函数时，该函数会创建一个系统调用，并将控制权交给操作系统。操作系统会切换到内核模式，执行系统调用，然后返回结果，最后再切换回用户模式。

2.2.6 Linux 系统命令

Linux 系统命令是在命令行接口（CLI）中执行操作的文本输入。Linux 系统提供了大量的系统命令，用于执行各种任务，如文件操作（如 ls、cd、mv 等）、系统监控（如 top、ps 等）、网络工具（如 ping、netstat 等）等。在 Linux 系统编程中，可以通过系统调用来执行这些系统命令。每个系统命令都有一系列的选项和参数，可以用来定制命令的行为。

2.3 文件及文件描述符

2.3.1 文件

在 Linux 系统中，几乎所有的事物都可以被视为文件。这些包括硬件设备、目录，甚至是网络通信，它们在 Linux 中都可以被当作文件来处理。文件是一个包含数据的容器，数据可以以任何格式存储在文件中。文件包含两部分信息：用户数据和元数据。用户数据是文件的主体部分，元数据则包含了关于文件的信息，如文件的大小、创建和修改时间、所有者和权限等。

Linux 文件类型: 普通文件、目录文件、链接文件、设备文件

2.3.2 文件描述符

文件描述符是一个用于访问文件或者其他输入/输出资源，如管道或网络套接字的非负整数。在 Linux 系统中，文件描述符是在进程中创建的，并在进程的生命周期内使用。它们是由内核自动分配的，并且总是返回最小的未使用的文件描述符。例如，当进程打开一个现有文件或者创建一个新文件时，操作系统将返回一个文件描述符以供进程在后续操作中使用。

在每个进程启动时，它都会有三个已经打开的文件描述符。标准输入 (STDIN) 的文件描述符是 0，标准输出 (STDOUT) 的文件描述符是 1，以

及标准错误 (STDERR) 的文件描述符是 2。这些文件描述符通常连接到用户的终端。

在 Linux 系统编程中，有多种系统调用用于操作文件描述符，如 open, read, write, close 等。

2.4 底层文件 I/O 操作

2.4.1 打开文件

在 Linux 中，可以使用系统调用 open() 打开一个文件。该函数需要文件名和打开模式作为参数，如只读、只写或读写等。如果操作成功，open() 将返回一个文件描述符，它是一个小的非负整数，可以在程序中用于引用该打开的文件。如果出错，open() 将返回-1。

```
#include <fcntl.h>
int open(const char *path, int flags);
int open(const char *path, int flags, mode_t mode);
```

2.4.2 读取文件

读取文件是通过 read() 系统调用完成的。该函数需要文件描述符、数据缓冲区的地址和要读取的字节数作为参数。read() 返回实际读取的字节数，如果已经到达文件末尾，那么返回 0，如果出现错误则返回-1。

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

2.4.3 写入文件

向文件中写入数据是通过 write() 系统调用完成的。该函数需要文件描述符、数据缓冲区的地址和要写入的字节数作为参数。write() 返回实际写

入的字节数，如果出现错误则返回-1。

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

2.4.4 关闭文件

关闭文件是通过 close() 系统调用完成的。该函数需要文件描述符作为参数，如果关闭操作成功，那么返回 0，如果出现错误则返回-1。

```
#include <unistd.h>
int close(int fd);
```

2.4.5 定位文件读写位置

通过 lseek() 系统调用可以在文件中改变当前的读写位置。该函数需要文件描述符、偏移量和基准点作为参数。如果操作成功，返回新的读写位置，如果出现错误则返回-1。

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

2.4.6 文件锁

在 Linux 中，可以使用文件锁（File Lock）来实现对文件的互斥访问。文件锁分为共享锁（Shared Lock）和独占锁（Exclusive Lock），用于控制对文件的读写操作。

设置文件锁

文件锁的设置使用 fcntl() 系统调用。通过设置 F_SETLK 或 F_SETLKW 命令，可以获取或释放文件锁。

```
#include <fcntl.h>
int fcntl(int fd, int cmd, struct flock *lock);
```

其中，fd 是文件描述符，cmd 是命令，lock 是指向 struct flock 结构的指针，该结构定义了文件锁的类型和位置。在设置文件锁时，需要设置以下字段：

- l_type：锁的类型，可以是 F_RDLCK（共享锁）或 F_WRLCK（独占锁）。
- l_whence：偏移量的基准点，可以是 SEEK_SET、SEEK_CUR 或 SEEK_END。
- l_start：锁定区域的起始偏移量。
- l_len：锁定区域的长度。

测试文件锁

文件锁的测试使用 fcntl() 系统调用。通过设置 F_GETLK 命令，可以检测给定的锁是否会阻塞。

```
#include <fcntl.h>
int fcntl(int fd, int cmd, struct flock *lock);
```

其中，fd 是文件描述符，cmd 是命令，lock 是指向 struct flock 结构的指针。在测试文件锁时，需要设置以下字段：

- l_type：锁的类型，可以是 F_RDLCK（共享锁）或 F_WRLCK（独占锁）。
- l_whence：偏移量的基准点，可以是 SEEK_SET、SEEK_CUR 或 SEEK_END。

- l_start：锁定区域的起始偏移量。
- l_len：锁定区域的长度。

释放文件锁

文件锁的释放使用 fcntl() 系统调用。通过设置 F_SETLK 命令，并将锁的类型设置为 F_UNLCK，可以释放文件锁。

```
#include <fcntl.h>
int fcntl(int fd, int cmd, struct flock *lock);
```

其中，fd 是文件描述符，cmd 是命令，lock 是指向 struct flock 结构的指针。在释放文件锁时，需要设置以下字段：

- l_type：锁的类型，设置为 F_UNLCK。
- l_whence：偏移量的基准点，可以是 SEEK_SET、SEEK_CUR 或 SEEK_END。
- l_start：锁定区域的起始偏移量。
- l_len：锁定区域的长度。

文件锁对于控制多个进程对同一文件的并发访问非常有用，可以防止数据损坏或不一致性。

2.5 标准 I/O 编程

2.5.1 概述

Linux 标准 I/O 库提供了一组函数，用于读写流，这些函数比系统调用更易于使用。在 C 语言中，标准 I/O 库被包含在 stdio.h 头文件中。

2.5.2 标准 I/O 函数

读函数

- `fgetc`: 从流中获取下一个字符。
- `fgets`: 从流中获取下一行。
- `fread`: 从流中读取数据。

写函数

- `fputc`: 向流中写入一个字符。
- `fputs`: 向流中写入一个字符串。
- `fwrite`: 向流中写入数据。

2.5.3 缓冲

标准 I/O 库使用缓冲来减少读写操作的数量。这意味着当你调用一个写函数时，数据可能不会立即写入磁盘，而是存储在缓冲区中，等待缓冲区满了或者你显式地刷新缓冲区。

标准 I/O 提供 3 种类型的缓冲存储：

全缓冲

在这种阻塞 I/O 模型中，填满标准 I/O 缓存后才进行实际 I/O 操作。

行缓冲

这是一种非阻塞模型，在输入和输出中遇到行结束符时，标准 I/O 库执行 I/O 操作。

不带缓冲

相当于用系统调用 write() 函数将这些字符全写到被打开的文件上。例如，stderr 是不带缓冲的。

2.5.4　文件定位

可以使用 `fseek`、`ftell` 和 `rewind` 函数来在流中更改和查询位置。

2.5.5　错误处理

如果一个标准 I/O 函数遇到错误，它会设置全局变量 `errno`，你可以使用 `perror` 或 `strerror` 函数来获取错误信息。

2.6　例程:copyfile.c

该程序复制源文件 (src_file) 的最后 1024 字节（或少于 1024 字节，如果源文件小于 1024 字节）到目标文件 (dest_file)。

```c
#include <unistd.h>  // 提供对 POSIX 操作系统 API 的访问，如文件处理和进程管理。
#include <fcntl.h>   // 提供文件控制操作，如打开和修改文件。
#include <stdio.h>   // 提供基本的输入输出功能。
#include <stdlib.h>  // 提供了一些通用函数，如内存管理，程序的退出等。

#define OFFSET 1024       // 定义偏移量，用于指定需要复制的文件部分的大小。
#define SRC_FILE "src"    // 定义源文件名。
#define DEST_FILE "dest" // 定义目标文件名。
#define BUFFER_SIZE 1024 // 定义缓冲区的大小。

int main(){
```

```
int src_fd, dest_fd;  // 定义源文件和目标文件的文件描述符。
unsigned char buff[BUFFER_SIZE];  // 定义缓冲区，用于存储文件读取的数据。
int real_read_len;  // 定义实际读取的字节数。
// 以只读方式打开源文件。
src_fd = open(SRC_FILE, O_RDONLY);
// 以写入方式打开目标文件，如果文件不存在则创建，如果文件已存在则清空文件内容。
dest_fd = open(DEST_FILE, O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|
// 检查文件是否正确打开。
if (src_fd<0||dest_fd<0){
    // 如果打开文件失败，输出错误信息并退出程序。
    perror("Open file error");
    exit(1);
}
// 输出源文件和目标文件的文件描述符。
printf("src fd:%d, dest fd:%d\n",src_fd,dest_fd);

// 将源文件的读写位置定位到文件的末尾减去 OFFSET 处。
if (lseek(src_fd, -OFFSET, SEEK_END)<0){
    // 如果定位失败，输出错误信息并退出程序。
    perror("Lseek error");
    exit(-1);
}
// 从源文件中读取数据，并写入到目标文件中。
while((real_read_len=read(src_fd,buff,sizeof(buff)))>0){
    // 如果写入的字节数不等于读取的字节数，输出错误信息并退出程序。
    if (write(dest_fd, buff, real_read_len)!=real_read_len){
        perror("Write file error");
```

```
            exit(-1);
        }
    }
    // 关闭源文件和目标文件。
    close(src_fd);
    close(dest_fd);
    return 0;  // 程序正常结束。
}
```