

|  |
| --- |
| **实验报告** |
|  |

学年学期： 2023 - 2024学年第二学期

|  |  |
| --- | --- |
| 课程名称 | 嵌入式系统 |
| 指导老师 |  |
| 学院 | 计算机学院 |
| 班级 |  |
| 学号 |  |
| 学生姓名 |  |

实 验 报 告

|  |  |  |  |
| --- | --- | --- | --- |
| 实验课程 | 嵌入式技术 | | |
| 实验名称 | 嵌入式linux之文件IO的使用 | | |
| 指导老师 |  | | |
| 实验性质  （必修、选修） | 必修 | 实验类型  （验证、设计、创新、综合） | 验证 |
| 实验课时 | 2 | 实验日期、时间 | 2024.04.05 |
| 实验消耗材料 | 无 | | |
| 实验仪器设备  (实验软硬件要求) | （1）Ubuntu操作系统 | | |
| 实验目的 | 1. 掌握read函数 2. 掌握write函数 3. 掌握open函数 | | |
| 成绩评定 |  | | |

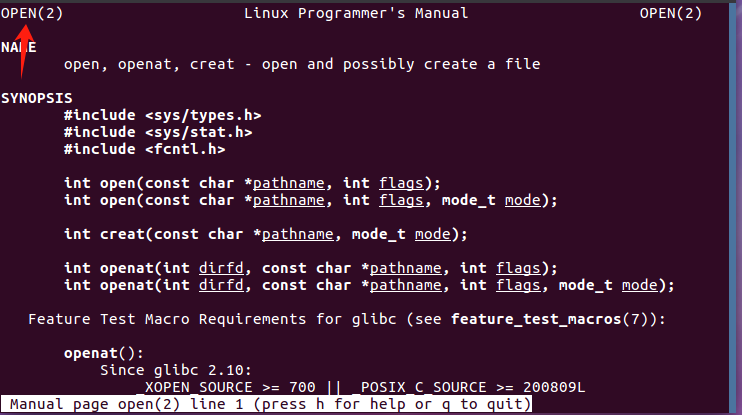
实验内容：

**一、概述**

**文件io主要有这几个函数：open、read、write、close。具体可以直接在ubantu上输入：**

man 2 open

来查看open的帮助手册



**二、open函数**

**1.理论部分**

open函数有三个需要引用的头文件和两个函数原型：

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags);

int open(const char \*pathname, int flags, mode\_t mode);

参数部分：

pathname：文件路径+文件名

flags：为标志位，不同的标志位打开文件的方式也不同，简单介绍几种：

O\_RDWR 以可读、可写模式打开文件

O\_RDONLY 以只读模式打开文件

O\_WRONLY 以可写模式打开文件

O\_CREAT 要打开的目标文件不存在时，创建一个并打开

O\_APPEND 以追加模式打开文件

O\_TRUNC

如果目标文件存在，会清除原先的文件内容

注：O\_RDWR、O\_RDONLY、O\_WRONLY这三种只能选一个，其它的可以的通过逻辑符‘|’结合使用。

mode：当flags标志位有O\_CREAT，也就是要创建文件的时候，这个参数就是拿来配置文件的权限，这里我输入是的值 0666 （第一个0表示这是一个八进制数） 最终创建的文件权限是 -rw-rw-r--（664），这个值是八进制数，跟用户掩码umask有关。 计算公式： mode值 &~ umask值， mode值就是我们自己给open函数的值，然后 ‘与’ 上 ‘取反’ umask的值，umask的值可以直接在ubantu输入umask查看。

返回值：

fd：返回的是文件描述符，一般从3开始，前3系统已占用：0（标准输入）1（标准输出）、2（标准出错）；如果返回的是-1，那么说明出错。

**2.实操部分**

举个例子：

创建一个open.c文件，这个程序是打开一个名为"nowfile.c"的文件，函数内使用了O\_RDWR | O\_CREAT这个两个标志位，意为如果没有名为"nowfile.c"的文件那就创建一个。

输入gedit open.c 输入以下程序，输入完成后，ctrl+s保存，ctrl+q退出

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main(void)

{

// define a variable, receive the return value of the funtion.

int fd;

fd = open("nowfile.c", O\_RDWR | O\_CREAT, 0666);

if(fd < 0)

{

printf("file creation failure\n");

}

else

{

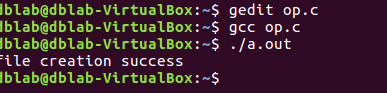
printf("file creation success\n");

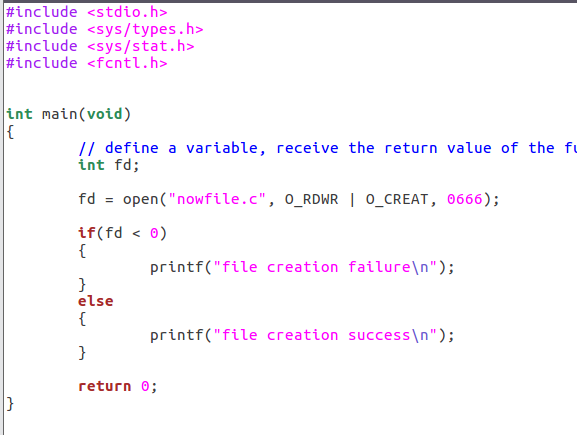
}

return 0;

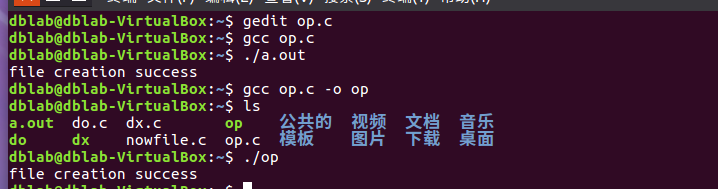
}

以下是操作方式和输出结果





也可以执行以下操作：



**三、read函数**

**1.理论部分**

下面是函数原型和其需要包含的头文件：

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

**参数部分：**

fd:文件描述符，也就是我们用open函数时返回的那个文件描述符。

buf：数据区，读到的数据将存放在这里。

count：要读得数据个数。

**返回值：**

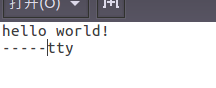
ssize\_t：返回值大于0时，返回值即为读到的数据数量；

返回值等于0时，表示读到了文件的末尾没有数据可读；

返回值小于0时，表示出错。

**2.实操**

首先在上一步创建好的nowfile文件里面写上点东西，我这里写“hello world！”，右下角可以看到共有12个字节大小，然后保存退出。



敲入命令，并输入以下代码



#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

int main(void)

{

// define a variable, receive the return value of the funtion.

int fd;

ssize\_t dsize;

char buff[64];

fd = open("nowfile.c", O\_RDWR | O\_CREAT, 0666);

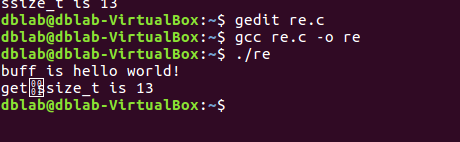
dsize = read(fd, buff, 64);

printf("buff is %s", buff);

printf("ssize\_t is %d\n", (int)dsize);

return 0;

}



读到的数据为 “hello world！”，没有问题；

后面的输出为 13，这是因为文件末尾带有一个结束符‘\0’。

**四、write函数**

**1.理论部分**

下面是函数原型和其需要包含的头文件：

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

**参数部分：**

fd:文件描述符，也就是我们用open函数时返回的那个文件描述符。

buf：数据区，将要写的数据放在这里。

count：要写入的数据个数。

**返回值：**

ssize\_t：返回值大于或等于0时，返回值即为写入的数据数量；

返回值小于0时，表示出错。

**2.实操**

先把代码敲上，打开/创建一个“datefile.c”的文件，在里面写入“hello world!”，后面再查看当前目录下的“datefile.c”文件是否成功写入。



输入以下代码：

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

int main(void)

{

// define a variable, receive the return value of the funtion.

int fd;

ssize\_t dsize;

char wbuff[]="hello world!";

fd = open("datafile.c", O\_RDWR | O\_CREAT, 0666);

if(fd < 0)

{

printf("open file failure!\n");

}

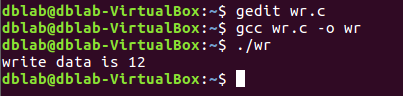
// write data

dsize = write(fd, wbuff, strlen(wbuff));

printf("write data is %d\n", (int)dsize);

return 0;

}



**五、close函数**

**1.理论部分**

这个函数很简单，就是当我们用open函数打开了文件之后，操作完是需要关闭文件的，close哈数就是关闭文件，下面是函数原型和其需要包含的头文件：

#include <unistd.h>

int close(int fd);

**参数部分：**

fd:文件描述符，也就是我们用open函数时返回的那个文件描述符。

**返回值：**

int：返回值0时，成功关闭；

返回值小于0时，表示出错。

**2.实操**

代码敲上，这里打开了之前写有“hello world!”的文件，然后再读取之前执行close函数关闭已打开的文件，这时候再read读取就报错了，因为文件已经关闭了。

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

int main(void)

{

// define a variable, receive the return value of the funtion.

int fd;

ssize\_t dsize;

char rbuff[64];

char wbuff[]="hello world!";

fd = open("datafile.c", O\_RDWR | O\_CREAT, 0666);

if(fd < 0)

{

printf("open file failure!\n");

}

close(fd);

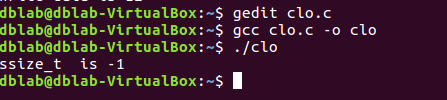
// read data

dsize = read(fd, rbuff, strlen(wbuff));

// print status

printf("ssize\_t is %d\n", (int)dsize);

return 0;}



实 验 报 告

|  |  |  |  |
| --- | --- | --- | --- |
| 实验课程 | 嵌入式技术 | | |
| 实验名称 | 嵌入式linux之进程管理 | | |
| 指导老师 | 王亚 | | |
| 实验性质  （必修、选修） | 必修 | 实验类型  （验证、设计、创新、综合） | 验证 |
| 实验课时 | 2 | 实验日期、时间 | 2024.04.19 |
| 实验消耗材料 | 无 | | |
| 实验仪器设备  (实验软硬件要求) | （1）Ubuntu操作系统 | | |
| 实验目的 | 1.掌握gcc编译器的使用  2.了解fork()程序 | | |
| 成绩评定 |  | | |

实验内容：

**实验一：**

在Linux环境下，用c语言编程，使用系统调用fork创建进程多个子进程。

（1）在终端里输入gedit main.c 在里面输入以下代码：

#include <unistd.h>

#include <stdio.h>

int main ()

{

pid\_t fpid; //fpid表示fork函数返回的值

fpid=fork();

printf(" you print me\n");

return 0;

}

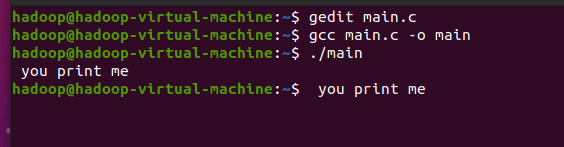
1. 用gcc编译器编译该程序（操作流程参见下图）

要求：

1. 请说出执行这个程序后，将一共运行几个进程。

执行这个程序后，将会运行两个进程：一个是父进程，一个是子进程。

1. 观察运行结果，并给出分析与解释。



分析：

由于调用fork()函数产生了一个新的进程，子进程复制了父进程的运行环境（上下文）,由于两个进程的执行，所以产生了两条输出语句。

**实验二：**

#include <unistd.h>

#include <stdio.h>

int main ()

{

pid\_t fpid; //fpid表示fork函数返回的值

int count=0;

fpid=fork();

if (fpid < 0)

printf("error infork!");

else if (fpid == 0) {

printf("i am the childprocess, my process id is %d\n",getpid());

count++;

}

else {

printf("i am the parentprocess, my process id is %d\n",getpid());

count++;

}

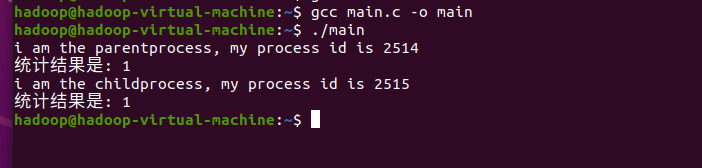
printf("统计结果是: %d\n",count);

return 0;

}

**要求：**

**观察运行结果，并给出分析与解释**



**分析：**

在主函数中，fork函数创建了一个子进程，创建成功后在main函数中返回子进程的pid，执行else语句，count++；子进程中，返回0执行elseif 语句，count++；由于二者的count是各自独立的，因此都是1。

**实验三：**

#include "stdio.h"

#include "sys/types.h"

#include "unistd.h"

int main()

{

pid\_t pid1,pid2;

pid1 = fork();

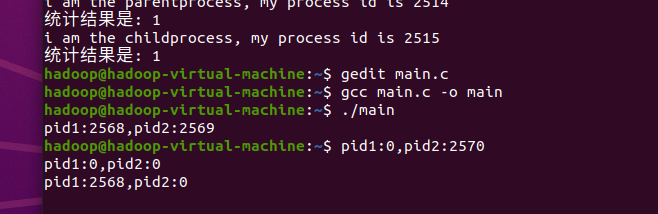
pid2 = fork();

printf("pid1:%d,pid2:%d\n",pid1,pid2);

}

**要求：**

**观察运行结果，并给出分析与解释**



这个程序运行后，一共将运行4个进程。这4个进程并没有严格的区分先后顺序。

实 验 报 告

|  |  |  |  |
| --- | --- | --- | --- |
| 实验课程 | 嵌入式技术 | | |
| 实验名称 | 嵌入式linux进程间通信 | | |
| 指导老师 |  | | |
| 实验性质  （必修、选修） | 必修 | 实验类型  （验证、设计、创新、综合） | 验证 |
| 实验课时 | 2 | 实验日期、时间 | 2024.05.11 |
| 实验消耗材料 | 无 | | |
| 实验仪器设备  (实验软硬件要求) | （1）Ubuntu操作系统 | | |
| 实验目的 | 1、熟悉信号的常用操作及信号处理函数；  2、熟悉Linux中管道的创建和读写；  3、掌握Linux中消息队列的使用 | | |
| 成绩评定 |  | | |

实验内容：

**一、信号的常用操作**

运行一下程序，并且查看输出结果

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

// 回调函数的声明

typedef void (\*sighandler\_t)(int);

void fun1(int signo)

{

printf("in fun1\n");

}

void fun2(int signo)

{

printf("in fun2\n");

}

int main(int argc, char \*argv[])

{

sighandler\_t previous = NULL;

// 第一次返回 NULL

previous = signal(SIGINT,fun1);

if(previous == NULL)

{

printf("return fun addr is NULL\n");

}

// 下一次返回此信号上一次注册的信号处理函数的地址。

previous = signal(SIGINT, fun2);

if(previous == fun1)

{

printf("return fun addr is fun1\n");

}

// 还是返回 NULL，因为处理的信号变了

previous = signal(SIGQUIT,fun1);

if(previous == NULL)

{

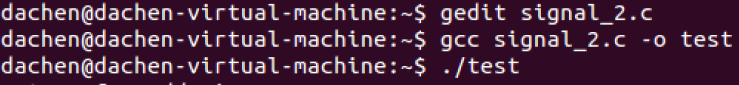
printf("return fun addr is NULL\n");

}

return 0;

}

在终端输入以下命令：



检验输出结果



**二、信号处理函数**

一个进程在发送信号，一个进程在接收信号的发送。

输入以下程序查看并分析结果

发送信号示例代码：

#include <stdio.h>

#include <signal.h>

#include <sys/types.h>

#include <unistd.h>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*功能： 发 SIGINT 信号及信号携带的值给指定的进程

\*参数： argv[1]：进程号

argv[2]：待发送的值（默认为100）

\*返回值： 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \*argv[])

{

if(argc >= 2)

{

pid\_t pid,pid\_self;

union sigval tmp;

pid = atoi(argv[1]); // 进程号

if( argc >= 3 )

{

tmp.sival\_int = atoi(argv[2]);

}

else

{

tmp.sival\_int = 100;

}

// 给进程 pid，发送 SIGINT 信号，并把 tmp 传递过去

sigqueue(pid, SIGINT, tmp);

pid\_self = getpid(); // 进程号

printf("pid = %d, pid\_self = %d\n", pid, pid\_self);

}

return 0;

}

接收信号示例代码：

#include <signal.h>

#include <stdio.h>

// 信号处理回电函数

void signal\_handler(int signum, siginfo\_t \*info, void \*ptr)

{

printf("signum = %d\n", signum); // 信号编号

printf("info->si\_pid = %d\n", info->si\_pid); // 对方的进程号

printf("info->si\_sigval = %d\n", info->si\_value.sival\_int); // 对方传递过来的信息

}

int main(int argc, char \*argv[])

{

struct sigaction act, oact;

act.sa\_sigaction = signal\_handler; //指定信号处理回调函数

sigemptyset(&act.sa\_mask); // 阻塞集为空

act.sa\_flags = SA\_SIGINFO; // 指定调用 signal\_handler

// 注册信号 SIGINT

sigaction(SIGINT, &act, &oact);

while(1)

{

printf("pid is %d\n", getpid()); // 进程号

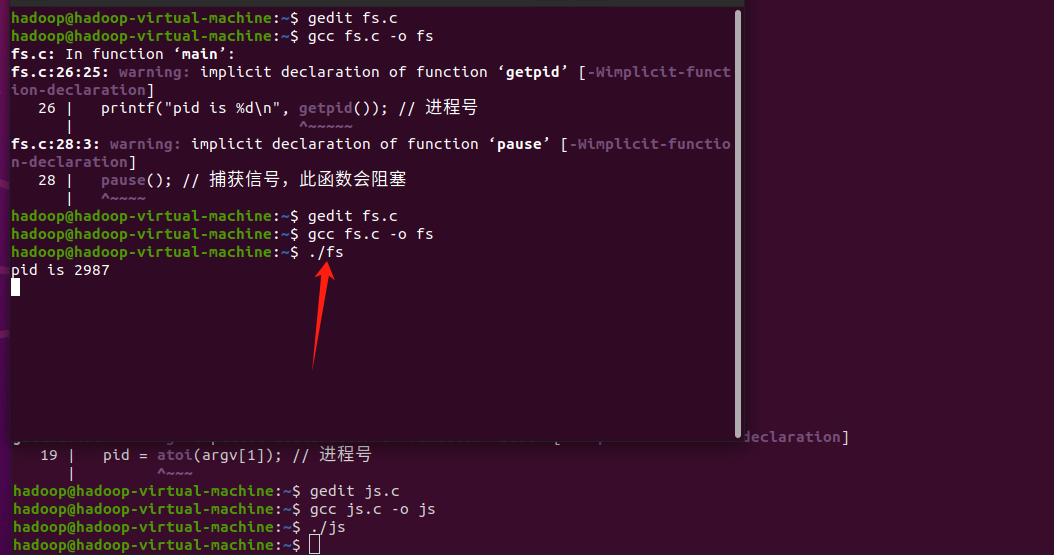
pause(); // 捕获信号，此函数会阻塞

}

return 0;

}

打开两个终端分别编译以上代码，一个进程接收，一个进程发送，查看结果。



**三、管道的操作**

管道也叫**无名管道，**它是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"。

**管道的操作：**  
所需头文件：

#include <unistd.h>

原型 : int pipe(int fd[2]);

功能  :  创建一个无名管道

参数 : fd：文件描述符数组 , 其中 fd[0] 表示读端  ,  fd[1] 表示写端

当一个管道建立时，它会创建两个文件描述符 fd[0] 和 fd[1]。其中 fd[0] 固定用于读管道，而 fd[1] 固定用于写管道 (0->嘴:读 ， 1->笔:写)。一般文件 I/O 的函数都可以用来操作管道( lseek() 除外)。

返回值： 成功：0 失败：-1

以下程序为子进程通过无名管道给父进程传递一个字符串数据：

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

int main(int argc, char \*argv[])

{

int fd\_pipe[2] = {0};

pid\_t pid;

if( pipe(fd\_pipe) < 0 ){// 创建无名管道

perror("pipe");

}

pid = fork(); // 创建进程

if( pid < 0 ){ // 出错

perror("fork");

exit(-1);

}

if( pid == 0 ){ // 子进程

char buf[] = "I am mike";

// 往管道写端写数据

write(fd\_pipe[1], buf, strlen(buf));

\_exit(0);

}else if( pid > 0){// 父进程

wait(NULL); // 等待子进程结束，回收其资源

char str[50] = {0};

// 从管道里读数据

read(fd\_pipe[0], str, sizeof(str));

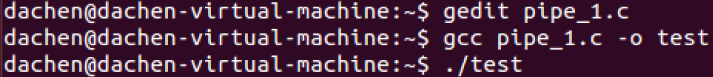
printf("str=[%s]\n", str); // 打印数据

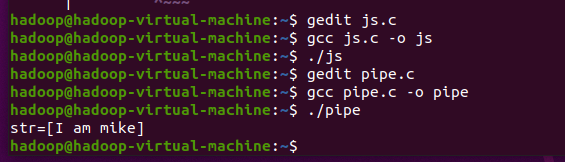
}

return 0;

}

输入以下命令查看并分析结果：





**四、消息队列的创建**

什么是[消息队列](https://so.csdn.net/so/search?q=%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_32172681/article/details/_blank)：

我们可以把消息队列比作是一个存放消息的容器，当我们需要使用消息的时候可以取出消息供自己使用。消息队列是分布式系统中重要的组件，使用消息队列主要是为了通过异步处理提高系统性能和削峰、降低系统耦合性。

## 创建消息队列

所需头文件：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

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，用法和创建文件时使用的mode模式标志是一样的，这里举两个来说明。
* IPC\_CREAT：如果消息队列对象不存在，则创建之，否则则进行打开操作
* IPC\_EXCL：如果消息对象不存在则创建之，否则产生一个错误并返回

返回值：

* 成功msgget将返回一个非负整数，即该消息队列的标识码；
* 失败则返回“-1”

那么如何获取key值？

* 通过宏定义key值
* 通过ftok函数生成key值。

**ftok函数的定义：**

系统建立IPC通讯 （[消息队列](http://baike.baidu.com/view/262473.htm" \t "https://www.likecs.com/_blank)、[信号量](http://baike.baidu.com/view/1303265.htm" \t "https://www.likecs.com/_blank)和[共享内存](http://baike.baidu.com/view/120892.htm" \t "https://www.likecs.com/_blank)） 时必须指定一个ID值。通常情况下，该id值通过ftok函数得到。

头文件

#include <[sys/types.h](http://baike.baidu.com/view/2956875.htm" \t "https://www.likecs.com/_blank)>

#include <sys/ipc.h>

函数原型：

key\_t ftok( const char \* fname, int id )

参数：

* fname就是你指定的文件名（已经存在的文件名），一般使用当前目录，如：key\_t key;
* key = ftok(".", 1); 这样就是将fname设为当前目录。
* id是子序号。虽然是int类型，但是只使用8bits(1-255）。

输入以下程序：

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

int main(int argc, char \*argv[])

{

key\_t key;

int msgqid;

key = ftok(".", 2012); // key 值

// 创建消息队列

msgqid = msgget(key, IPC\_CREAT|0666);

return 0;

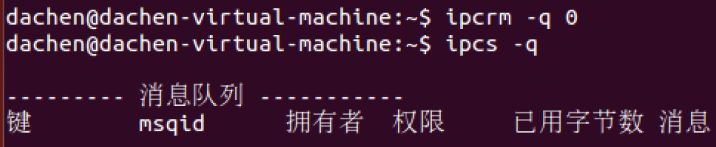
}

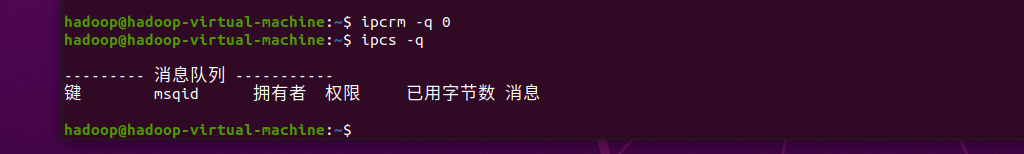
以下为输入命令，其中ipcs -q为查看消息队列。





ipcrm -q id ：删除消息队列标识 id和其相关的消息队列和数据结构，其中id即为msqid的值，删除后通过ipcs -q查看消息队列





实 验 报 告

|  |  |  |  |
| --- | --- | --- | --- |
| 实验课程 | 嵌入式技术 | | |
| 实验名称 | 多线程实现多任务 | | |
| 指导老师 | 王亚 | | |
| 实验性质  （必修、选修） | 必修 | 实验类型  （验证、设计、创新、综合） | 验证 |
| 实验课时 | 2 | 实验日期、时间 | 2024.05.17 |
| 实验消耗材料 | 无 | | |
| 实验仪器设备  (实验软硬件要求) | （1）Ubuntu操作系统 | | |
| 实验目的 | 1、熟悉线程的基本操作；  2、熟悉线程私有数据；  3、掌握线程的同步于互斥 | | |
| 成绩评定 |  | | |

实验内容：

**一、线程的创建**

运行一下程序，并且查看输出结果

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

int var = 8;

void \*thread\_1(void \*arg)

{

while(1)

{

printf("this is my new thread1: var++\n");

var++;

sleep(1);

}

return NULL;

}

void \*thread\_2(void \* arg)

{

while(1){

printf("this is my new thread2: var = %d\n", var);

sleep(1);

}

return NULL;

}

int main(int argc, char \*argv[])

{

pthread\_t tid1,tid2;

//创建两个线程

pthread\_create(&tid1, NULL, thread\_1, NULL);

pthread\_create(&tid2, NULL, thread\_2, NULL);

while(1){

printf("the main thread: var = %d\n", var);

sleep(1);

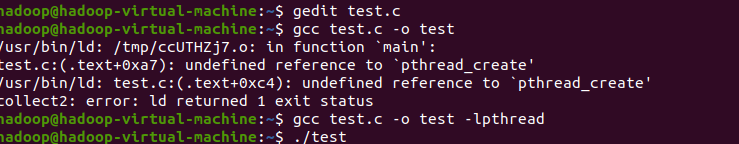
}

return 0;

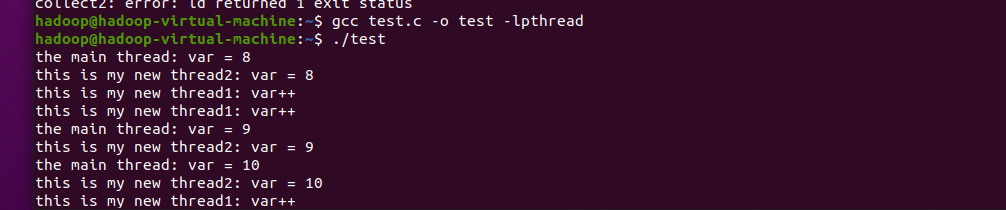
}

检验输出结果。

运行截图（含书写代码窗口截图、运行命令截图和运行结果截图）：



部分结果如下：



**二、线程私有数据**

进程内的所有线程共享进程的数据空间，所以全局变量为所有线程共有。在某些场景下，线程需要保存自己的私有数据，这时可以创建线程私有数据(Thread-specific Data)TSD来解决。在线程内部，私有数据可以被线程的各个接口访问，但对其他线程屏蔽。

　　TSD采用了一种被称为一键多值的技术，即一个键对应多个数值。访问数据时都是通过键值来访问，好像是对一个变量进行访问，其实是在访问不同的数据。使用线程私有数据时，首先要为每个线程数据创建一个相关联的键。在各个线程内部，都使用这个公用的键来指代线程数据，但是在不同的线程中，这个键代表的数据是不同的。

输入以下程序查看并分析结果

#include<stdio.h>

#include<pthread.h>

pthread\_key\_t key; //定义一个key

void \*pthread1(void \*arg){

int a = 1;

pthread\_setspecific(key, &a);//在线程1中，将这个key关联其私有数据

printf("thread1: %d\n", \*(int \*)pthread\_getspecific(key)); //获取其私有数据

}

void \*pthread2(void \*arg){

int a = 2;

pthread\_setspecific(key, &a); //在线程2中，将这个key关联其私有数据

printf("thread2: %d\n", \*(int \*)pthread\_getspecific(key)); //获取其私有数据

}

int main()

{

pthread\_key\_create(&key, NULL); //创建一个key

pthread\_t tid1, tid2;

pthread\_create(&tid1, NULL, &pthread1, NULL);

pthread\_create(&tid2, NULL, &pthread2, NULL);

pthread\_join(tid1, NULL);

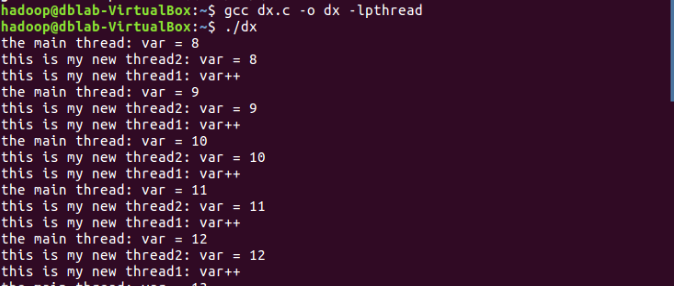
pthread\_join(tid2, NULL);

return 0;

}

检验输出结果。（从运行结果来看，各线程对自己的私有数据操作互不影响。也就是说，虽然 key 是同名且全局，但访问的内存空间并不是同一个。

运行截图（含书写代码窗口截图、运行命令截图和运行结果截图）：



**三、线程的同步与互斥**

（一）线程互斥

**1.1 相关概念介绍**

临界资源： 多线程执行流共享的资源叫做临界资源

临界区： 每个线程内部访问临界资源的代码，被称为临界区

互斥： 任何时刻，互斥保证有且只有一个执行流进入临界区访问临界资源，通常对临界资源起保护作用

原子性： 不会被任何调度机制打断的操作，该操作只有两态：要么完成，要么未完成

下面模拟实现一个抢票系统，将记录票的剩余张数的变量定义为全局变量，主线程创建四个新线程进行抢票，当票被抢完后这四个线程自动退出。

#include<stdio.h>

#include<unistd.h>

#include<pthread.h>

int thickets=100;//100张票

//thickets--表示抢票

void\* Routine(void\* arg)

{

while(1)

{

if(thickets>0)

{

usleep(30000);//抢票时间

printf("%s get a thickets, now thickets' number:%d\n",(char\*)arg,--thickets);

}

else

break;

}

return NULL;

}

int main()

{

pthread\_t tid1,tid2,tid3;

pthread\_create(&tid1,NULL,Routine,(void\*)"thread 1");

pthread\_create(&tid2,NULL,Routine,(void\*)"thread 2");

pthread\_create(&tid3,NULL,Routine,(void\*)"thread 3");

pthread\_join(tid1,NULL);

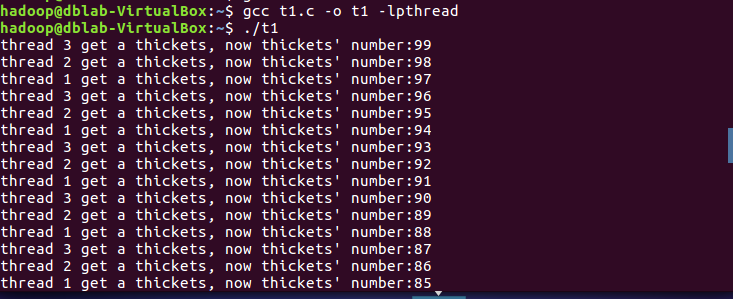
pthread\_join(tid2,NULL);

pthread\_join(tid3,NULL);

return 0;

}

编译并运行以下命令，部分结果如下：



**1.2互斥量mutex介绍**

概念：

大部分情况，线程使用的数据都是局部变量，变量的地址空间在线程栈空间内，这种情况变量归属单个线程，其他线程无法获得这种变量。

但有时候，很多变量都需要在线程间共享，这样的变量成为共享变量，可以通过数据的共享，完成线程之间的交互。

多个线程并发的操作共享变量，就会带来一些问题。

要解决以上问题需要做到三点：

代码必须要有互斥行为：当代码进入临界区执行时，不允许其他线程进入该临界区。

如果多个线程同时要求执行临界区的代码，并且临界区没有线程在执行，那么只能允许一个线程进入该临界区。

如果线程不在临界区中执行，那么该线程不能阻止其他线程进入临界区。

**1.3互斥量的使用**

初始化互斥量：

静态分配：

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER

动态分配：

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrictattr);

参数：mutex：要初始化的互斥量；attr：互斥量的属性，一般设置为NULL

销毁互斥量：

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex)；

注意：

1. 使用 PTHREAD\_ MUTEX\_ INITIALIZER 初始化的互斥量不需要销毁
2. 不要销毁一个已经加锁的互斥量
3. 已经销毁的互斥量，要确保后面不会有线程再尝试加锁

互斥量加锁和解锁:

1、int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

2、int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

返回值：成功返回0，失败返回错误号

调用 pthread\_ lock 时可能遇到的情况:

互斥量处于未锁状态，该函数会将互斥量锁定，同时返回成功

发起函数调用时，其他线程已经锁定互斥量，或者存在其他线程同时申请互斥量，但没有竞争到互斥量，那么pthread\_ lock调用会陷入阻塞(执行流被挂起)，等待互斥量解锁

改进抢票程序：

#include<stdio.h>

#include<unistd.h>

#include<pthread.h>

int thickets=100;//100张票

//thickets--表示抢票

pthread\_mutex\_t lock;//线程共用一个互斥锁

void\* Routine(void\* arg)

{

while(1)

{

pthread\_mutex\_lock(&lock);

if(thickets>0)

{

usleep(100000);//抢票时间

printf("%s get a thickets, now thickets' number:%d\n",(char\*)arg,--thickets);

pthread\_mutex\_unlock(&lock);

}

else

{

pthread\_mutex\_unlock(&lock);

break;

}

usleep(100000);

}

return NULL;

}

int main()

{

pthread\_mutex\_init(&lock,NULL);

pthread\_t tid1,tid2,tid3;

pthread\_create(&tid1,NULL,Routine,(void\*)"thread 1");

pthread\_create(&tid2,NULL,Routine,(void\*)"thread 2");

pthread\_create(&tid3,NULL,Routine,(void\*)"thread 3");

pthread\_join(tid1,NULL);

pthread\_join(tid2,NULL);

pthread\_join(tid3,NULL);

pthread\_mutex\_destroy(&lock);

return 0;

}

编译并运行以下命令，运行截图（含书写代码窗口截图、运行命令截图和运行结果截图）：

