**《操作系统》实验报告**

徽标, 公司名称

描述已自动生成

**题目: 实验二 线程创建管理与通信**

**学 院 计算机学院**

**班 级 2021211304**

**学 号 2021212171、2021212484**

**姓 名 杨晨、张梓良**

**2023年 11月**

## 目 录

第一章 实验概述 1

1.1 实验目的 1

1.2 实验内容 1

1.2.1 线程创建与管理 1

1.2.2 线程通信 1

1.3 实验设计原理 1

1.3.1 实验2-1Pthread 线程库背景知识 1

1.3.2 实验2-2线程创建与管理 1

第二章 实验步骤 2

2.1 Pthread线程库的基础知识 2

2.1.1 概述 2

2.1.2 Pthead中的函数 2

2.2 实验过程 5

2.2.1 使用pthread\_create()函数创建线程 5

2.2.2 使用pthread\_exit()函数退出线程的举例 6

2.2.3 用pthread\_join()实现线程等待 7

2.2.4 使用pthread\_self()获取线程ID 8

2.2.5 线程清理函数的使用 9

2.2.6 并行执行的多线程任务模拟 11

2.2.7 整数指针传递给新创建的线程函数 13

2.2.8 字符串传递给新创建的线程函数 14

2.2.9 结构体传递给新创建的线程函数 15

2.2.10 新建立的线程可以共享进程中的数据 16

第三章 实验结果分析 18

3.1 thread\_create.c 18

3.1.1 执行结果 18

3.1.2 结果分析 18

3.2 thread\_exit.c 18

3.2.1 执行结果 18

3.2.2 结果分析 18

3.3 thread\_join.c 19

3.3.1 执行结果 19

3.3.2 结果分析 19

3.4 thread\_id.c 20

3.4.1 执行结果 20

3.4.2 结果分析 20

3.5 thread\_clean.c 21

3.5.1 执行结果 21

3.5.2 结果分析 21

3.6 thread.c 22

3.6.1 执行结果 22

3.6.2 结果分析 23

3.7 int\_ptr\_thread.c 23

3.6.1 执行结果 23

3.6.2 结果分析 23

3.8 char\_ptr\_thread.c 24

3.6.1 执行结果 24

3.6.2 结果分析 24

3.9 struct\_ptr\_thread.c 25

3.6.1 执行结果 25

3.6.2 结果分析 25

3.10 shared\_data\_thread.c 25

3.6.1 执行结果 25

3.6.2 结果分析 25

第四章 总结 27

# **第一章 实验概述**

##### 1.1 实验目的

理解进程和内核级、用户级线程的概念，掌握在Linux 操作系统环境下，采用Pthread线程库创建和管理线程的方法，观察分析线程并发执行和通信行为。

##### 1.2 实验内容

# **1.2.1 线程创建与管理**

查阅Linux 和Pthread 线程库相关资料，参照相关示例程序，设计父进程/线程和子线程的业务处理逻辑；利用pthread\_create 、pthread\_exit 、pthread\_cancel 、pthread\_join 、pthread\_self等线程管理函数，创建和管理线程，观察父子进程 线程的结构和并发行为，掌握等待、退出、撤销等线程控制方法。

要求：

* 在一个进程内创建多个子线程（如不少于3 个子线程），父子线程具有各自不同的业务处理逻辑，子线程执行自身特定的线程函数。父子线程间通过pthread\_join() 函数实现同步和资源释放；
* 一个进程内的多个子线程分别以不同方式终止或退出执行，如通过pthread\_exit() 和return 自己主动终止，或被其它线程通过pthread\_cancel 被动终止，观察对比以不同方式退出执行的子线程的行为差异；
* 在创建的父子进程/线程代码中的不同位置处增加数十到数百毫秒的随机延迟（使用sleep()），使得进程和线程的执行横跨多个时间片，保证线程执行时间不少于3 个时间片。

# **1.2.2 线程通信**

属于同一个进程中的多个线程使用相同的地址空间，共享大部分数据，一个线程的数据可以直接为其它线程所用，这些线程相互间可以方便快捷地利用共享数据结构进行通信。

编写程序，创建线程，实现主线程与子线程间、子线程相互间通过共享数据类型，如整型变量、字符串、结构体等传递信息，进行通信。

##### 1.3 实验设计原理

# **1.3.1 实验2-1Pthread 线程库背景知识**

了解POSIX 线程标准库（Pthread 线程库）相关知识，分析Pthread 线程结构，掌握所提供的线程管理API，如pthread\_create, pthread\_join, pthread\_t pthread\_self, pthread\_detach, pthread\_exit；

# **1.3.2 实验2-2线程创建与管理**

参照 “【实验指导】 2. 线程创建及管理程序示例”，查阅参考资料，利用Pthread API，创建和管理线程，观察线程的结构和并发执行行为；

要求：本组实验至少用到pthread\_create, pthread\_exit, pthread\_join, pthread\_self 等四个API

# **第二章 实验步骤**

##### 2.1 Pthread线程库的基础知识

# **2.1.1 概述**

线程也称为轻质进程、轻量级进程LWP。线程是多线程程序运行中的基本调度单位，是进程中一个单一顺序的控制流，一个进程内部可以有很多线程。同一进程中的多个线程共享分配给该进程的系统资源，比如文件描述符和信号处理等。

Linux系统中的多线程遵循POSIX线程接口标准，称为pthread，通过Pthread 线程库来实现线程管理。编写Linux下的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a。

Linux下pthread的实现是通过系统调用clone()来实现的，clone()是 Linux所特有的系统调用，其使用方式类似fork。

线程与进程比较：

* 在Linux系统中，启动一个新的进程必须分配给它独立的地址空间，建立数据结构以维护其代码段、堆栈段和数据段，管理代价非常“昂贵”。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且线程间彼此切换所需要时间也远远小于进程间切换所需要的时间。
* 线程间方便的通信机制。对不同进程来说它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，费时且不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其他线程所用，方便快捷。

Pthread线程库提供的基本线程管理函数有：

* 创建线程的pthread\_create，
* 线程退出pthread\_exit，线程取消pthread\_cancel，线程脱离pthread\_detach
* 线程等待pthread\_join，
* 线程标识获取pthread\_self

# 

# **2.1.2 Pthread中的函数**

1. **线程创建**pthread\_create()

**调用格式**：

#include<pthread.h>

int pthread\_create(pthread\_t \*thread，pthread\_attr\_t\*attr，void\*(\*start\_routine)(void \*), void \*arg)

**参数**：

thread：指向所创建线程的标识符的指针，线程创建成功时，返回被创建线程的ID

attr：用于指定被创建线程的属性，NULL表示使用默认属性

start\_routine: 函数指针，指向线程创建后要调用的函数，是一个以指向void的指针作 为参数和返回值的函数指针，这个被线程调用的函数也被称为线程函数

arg：指向传递给线程函数的参数

**返回值**：

创建成功：0

创建失败：返回错误码

2. **线程退出**pthread\_exit()

**调用格式**：

#include<pthread.h>

void pthread\_exit(void \*retval)

**参数**：

retval：线程退出时的返回值，可被pthread\_join()等其它函数获取

当一个进程中包括多个线程时，使用进程退出函数exit()将终止进程内全部线程，而使用pthread\_exit()可以终止进程内特定线程。

如果被退出/终止的线程未脱离，则其线程ID和退出状态将一直保留到某个线程调pthread\_join为止。

3. **线程脱离**pthread\_detach()

调用格式：

#include<pthread.h>

int pthread\_detach(pthread\_t thread)  
**参数**：

thread：被脱离的线程的ID

函数pthread\_detach将指定的线程脱离，脱离的线程类似于守护进程。线程可以通过执行pthread\_detach(pthread\_self())将自己主动脱离。

4. **线程取消**pthread\_cancel()

**调用格式**：

#include<pthread.h>

int pthread\_cancel(pthread\_t thread)  
**参数**：

thread：要取消的线程的标识符ID

返回值：

取消成功：0

取消失败：返回错误码

在一个线程中调用pthread\_cancel()函数可以终止另一个线程的执行，但在被取消的线程的内部需要调用pthread\_setcancel()函数和pthread\_setcanceltype()函数设置自己的取消状态。

例如，被取消的线程接收到来自另一个线程的取消请求之后，可以忽略或接受这个请求；如果是接受取消请求，再判断立刻采取终止操作还是等待某个函数的调用等。

5. **线程清除**pthread\_cleanup\_push()/pthread\_cleanup\_pop()

**调用格式**：

#include<pthread.h>

void pthread\_cleanup\_push(void (\*rtn)(void \*), void \*arg)  
**参数**：

rtn：清除函数

arg：清除函数的参数

**功能说明**：

将清除函数压入清除栈

**调用格式**：

#include<pthread.h>

void pthread\_cleanup\_pop(int execute)  
**参数**：

execute：执行到pthread\_cleanup\_pop()时是否在弹出清除函数的同时执行该函数。

非0：执行；0：不执行

**功能说明**：

将清除函数弹出清除栈

线程终止有两种情况：正常终止和非正常终止。线程主动调用pthread\_exit()或者从线程函数中return都将使线程正常退出，属于正常的、可预见的退出方式；

如果线程在其它线程的干预下，或者由于自身运行出错（例如访问非法地址）而退出，对线程自身而言，这种退出方式是非正常、不可预见的。

对线程可预见的正常终止和不可预见异常终止，通过两个线程清除函数，可以保证线程终止时能顺利地释放掉自己所占用的资源。

在线程函数体内先后调用pthread\_cleanup\_push()、pthread\_cleanup\_pop()后，在从pthread\_cleanup\_push()的调用点到pthread\_cleanup\_pop()之间的程序段中的各种线程终止操作（包括pthread\_exit()和异常终止，但不包括return）都将执行pthread\_cleanup\_push()所指定的清理函数。

6. **线程等待**pthread\_join()

**调用格式**：

#include<pthread.h>

int pthread\_join(pthread\_t thread，void \*\*thread\_return)  
**参数**：

thread：等待退出的线程ID

thread\_return：用于定义的指针，存储被等待线程结束时的返回值（不为NULL时）

**返回值**：

等待成功：0

等待失败：返回错误码

一个进程内部的多个线程共享进程内的数据段。当进程内一个线程退出后，退出线程所占用的资源并不会随着该线程的终止而得到释放。类似于进程之间使用wait()系统调用来同步终止并释放资源，线程之间pthread\_join()函数实现线程同步和资源释放。

pthread\_join()将当前进程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

7. **线程标识获取**pthread\_self()

**调用格式**：

#include<pthread.h>

Pthread\_t pthread\_self(void)  
**参数**：

返回值：

返回调用该函数的线程的标识ID

##### 2.2 实验过程

# **2.2.1 使用pthread\_create()函数创建线程**

首先创建文件thread\_create.c，编写以下代码 **vi thread\_create.c**

/\* thread\_create.c \*/  
#include<stdio.h>  
#include<stdlib.h>  
#include<pthread.h>  
   
/\*线程函数1\*/  
void \*mythread1(void)  
{  
    int i;  
    for(i=0;i<5;i++)  
    {  
        printf("I am the 1st pthread,created by mybeilef321\n");  
        sleep(2);  
    }  
}  
/\*线程函数2\*/  
void \*mythread2(void)  
{  
    int i;  
    for(i=0;i<5;i++)  
    {  
        printf("I am the 2st pthread,created by mybelief321\n");  
        sleep(2);  
    }  
}  
   
int main()  
{  
    pthread\_t id1,id2; /\*线程ID\*/  
    int res;  
    /\*创建一个线程，并使得该线程执行mythread1函数\*/  
    res=pthread\_create(&id1,NULL,(void \*)mythread1,NULL);  
    if(res)  
    {  
        printf("create pthread error! \n");  
        return 1;  
    }  
    /\*创建一个线程，并使得该线程执行mythread2函数\*/  
    res=pthread\_create(&id2,NULL,(void \*)mythread2,NULL);  
    if(res)  
    {  
        printf("Create pthread error! n");  
        return 1;  
    }  
    /\*等待两个线程均退出后main()函数再退出\*/  
    pthread\_join(id1,NULL);  
    pthread\_join(id2,NULL);  
   
    return 1;  
}

使用命令：**gcc thread\_create.c -o thread\_create -lpthread**编译

之后，运行可执行文件 **./thread\_create**

运行结果如下

文本

描述已自动生成

# 

# **2.2.2 使用pthread\_exit()函数退出线程的举例**

首先创建文件thread\_exit.c，编写以下代码 **vi thread\_exit.c**

/\* thread\_exit.c文件 \*/  
#include<stdio.h>  
#include<pthread.h>  
/\*进程函数\*/  
void \*create(void \*arg)  
{  
    printf("New thread is created...\n");  
    pthread\_exit((void \*)6); /\*这里的6也可以设置成其它数值\*/  
}  
   
int main()  
{  
    pthread\_t tid;  
    int res;  
    void \*temp;  
    res=pthread\_create(&tid,NULL,create,NULL);  
    printf("I am the main thread!\n");  
    if(res)  
    {  
        printf("thread is not created...\n");  
        return -1;  
    }  
   
    res=pthread\_join(tid,&temp);  
    if(res)  
    {  
        printf("Thread is not exit...\n");  
        return -2;  
    }  
    printf("Thread is exit code %d \n",(int)temp);  
    return 0;  
}

编译**gcc thread\_exit.c -o thread\_exit -lpthread**

执行**./thread\_exit**

运行结果如下

文本

描述已自动生成

# 

# **2.2.3** **用pthread\_join()实现线程等待**

首先创建文件thread\_join.c，编写以下代码 **vi thread\_join.c**

/\*thread\_join.c\*/  
#include<pthread.h>  
#include<stdio.h>  
/\*线程函数\*/  
void \*thread(void \*str)  
{  
    int i;  
    for(i=0;i<4;++i)  
    {  
        sleep(2);  
        printf("This is the thread:%d\n",i);  
    }  
    return NULL;  
}  
   
int main()  
{  
    pthread\_t pth; /\*线程ID\*/  
    int i;  
    int ret=pthread\_create(&pth,NULL,thread,(void \*)(i));  
    pthread\_join(pth ,NULL);  
    printf("123\n");  
    for(i=0;i<3;++i)  
    {  
        sleep(1);  
        printf("This is the main:%d\n",i);  
    }  
    return 0;  
}

编译 **gcc thread\_join.c -o thread\_join -lpthread**

执行 **./thread\_join**

运行结果如下

文本

描述已自动生成

# **2.2.4 使用pthread\_self()获取线程ID**

首先创建文件thread\_id.c，编写以下代码 **vi thread\_id.c**

/\*thread\_id.c\*/  
#include<stdio.h>  
#include<pthread.h>  
#include<unistd.h>  
void \*create(void \*arg)  
{  
    printf("New thread......\n");  
    printf("This thread's id is %u \n",(unsigned int)pthread\_self());  
    printf("This thread process pid is %d \n",getpid());  
    return NULL;  
}  
   
int main()  
{  
    pthread\_t tid;  
    int res;  
    printf("Main thread is starting...\n");  
    res=pthread\_create(&tid,NULL,create,NULL);  
    if(res)  
    {  
        printf("thread is not created...\n");  
        return -1;  
    }  
    printf("The main process pid is %d \n",getpid());  
    sleep(1);  
    return 0;  
}

编译 **gcc thread\_id.c -o thread\_id -lpthread**

执行 **./thread\_id**

运行结果如下

文本

描述已自动生成

# **2.2.5 线程清理函数的使用**

首先创建文件thread\_clean.c，编写以下代码 **vi thread\_clean.c**

/\* thread\_clean.c \*/  
#include<stdio.h>  
#include<pthread.h>  
/\*清除函数\*/  
void \*clean(void \*arg)  
{  
    printf("cleanup:%s  \n",(char \*)arg);  
    return (void\*)0;  
}  
   
/\*线程函数1\*/  
void \*thr\_fn1(void \*arg)  
{  
    printf("Thread1 start \n");  
    /\*注意我这里是将清除函数压入了两次清除栈，  
    两次的清除函数的参数不同\*/  
    pthread\_cleanup\_push((void \*)clean,"thread1 first  handler");  
    pthread\_cleanup\_push((void \*)clean,"thread1 second handler");  
    printf("thread1 push complete \n");  
    if(arg)    /\*main()函数中如果传递的参数arg非0,则退出此线程\*/  
    {  
        return ((void \*)1);  
    }  
    pthread\_cleanup\_pop(1); /\*弹出清理函数的时候执行清理函数，  
                            一定要注意弹出栈的顺序，  
                            是先将最后压入栈的函数先弹出来\*/  
    pthread\_cleanup\_pop(1); /\*弹出清理函数的时候执行清理函数\*/  
    return (void \*)2;  
}  
   
void \*thr\_fn2(void \*arg)  
{  
    printf("Thread2 start \n");  
    pthread\_cleanup\_push((void \*)clean,"thread2 first  handler");  
    pthread\_cleanup\_push((void \*)clean,"thread2 second handler");  
    printf("thread2 push complete \n");  
    if(arg)  
    {  
        return ((void \*)3);  
    }  
    pthread\_cleanup\_pop(1);/\*弹出清理函数的时候执行清理函数\*/  
    pthread\_cleanup\_pop(0);/\*弹出清理函数的时候不执行清理函数\*/  
    return (void \*)4;  
}  
   
int main()  
{  
    int res;  
    pthread\_t tid1,tid2;  
    void \*tret;  
    /\*创建一个线程，注意传递的参数是1\*/  
    res=pthread\_create(&tid1,NULL,thr\_fn1,(void \*)1);  
    if(res!=0)  
    {  
        printf("Create thread error...\n");  
        return-1;  
    }  
    /\*创建一个线程，注意传递的参数是0\*/  
    res=pthread\_create(&tid2,NULL,thr\_fn2,(void \*)0);  
    if(res!=0)  
    {  
        printf("create thread error...\n");  
        return -1;  
    }  
    /\*等待线程1结束\*/  
    res=pthread\_join(tid1,&tret);  
    if(res!=0)  
    {  
        printf("Thread join error...\n");  
        return -1;  
    }  
    printf("Thread1 exit code: %d\n",(int)tret);  
    /\*等待线程2结束\*/  
    res=pthread\_join(tid2,&tret);  
    if(res!=0)  
    {  
        printf("Thread join error...\n");  
        return -1;  
    }  
    printf("Thread2 exit code: %d\n",(int)tret);  
    return 0;  
   
}

编译 **gcc thread\_clean.c -o thread\_clean -lpthread**

执行 **./thread\_clean**

运行结果如下

文本

描述已自动生成

# **2.2.6 并行执行的多线程任务模拟**

本次创建了3个进程，为了更好的描述线程之间的并行执行，让3个线程共用同一个执行函数。每个线程都有5次循环(可以看成5个小任务)，每次循环之间会随机等待1～10s的时间，意义在于模拟每个任务的到达时间是随机的，并没有任何特定的规律。

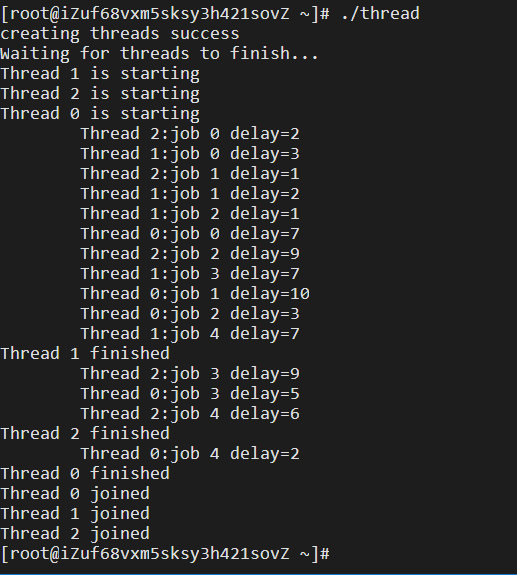
首先创建文件thread.c，编写以下代码 **vi thread.c**

/\* thread.c \*/  
#include<stdio.h>  
#include<stdlib.h>  
#include<pthread.h>  
#define THREAD\_NUMBER 3  /\*线程数\*/  
#define REPEAT\_NUMBER 5  /\*每个线程中的小任务数\*/  
#define DELAY\_TIME\_LEVELS 10.0 /\*小任务之间的最大时间间隔\*/  
   
/\*线程函数例程\*/  
void \*thrd\_func(void \*arg)  
{  
    int thrd\_num=(int)arg;  
    int delay\_time=0;  
    int count=0;  
   
    printf("Thread %d is starting\n",thrd\_num);  
    for(count=0;count<REPEAT\_NUMBER;count++)  
    {  
        delay\_time=(int)(rand()\*DELAY\_TIME\_LEVELS/(RAND\_MAX))+1;  
        sleep(delay\_time);  
        printf("\tThread %d:job %d delay=%d\n",thrd\_num,count,delay\_time);  
    }  
    printf("Thread %d finished\n",thrd\_num);  
    pthread\_exit(NULL); /\*退出线程\*/  
}  
   
int main(void)  
{  
    pthread\_t thread[THREAD\_NUMBER]; /\*存放线程ID\*/  
    int no,res;  
    void \*thrd\_ret;  
   
    srand(time(NULL)); /\*随机数发生器的初始化函数\*/  
   
    for(no=0;no<THREAD\_NUMBER;no++)  
    {  
        /\*创建多线程\*/  
        res=pthread\_create(&thread[no],NULL,(void \*)thrd\_func,(void \*)(no));  
        if(res!=0)  
        {  
            printf("create thread %d failed\n",no);  
            exit(res);  
        }  
    }  
   
    printf("creating threads success\nWaiting for threads to finish...\n");  
    for(no=0;no<THREAD\_NUMBER;no++)  
    {  
        /\*等待线程结束\*/  
        res=pthread\_join(thread[no],&thrd\_ret);  
        if(!res)  
        {  
            printf("Thread %d joined\n" ,no);  
        }  
        else  
        {  
            printf("Thread %d join failed\n",no);  
        }  
    }  
   
    return 0;  
}

编译 **gcc thread.c -o thread -lpthread**

执行 **./thread**

运行结果如下



# **2.2.7 整数指针传递给新创建的线程函数**

在main函数中传递int指针，传递到新建的线程函数中。

首先创建文件int\_ptr\_thread.c，编写以下代码 **vi int\_ptr\_thread.c**

#include <stdio.h>  
#include <pthread.h>  
   
void \*create(void \*arg)  
{  
    int \*num;  
    num = (int \*)arg;  
    printf("Create parameter is %d\n",\*num);  
    return (void \*)0;  
}  
   
int main(int argc, const char \*argv[])  
{  
    pthread\_t id1;  
    int error;  
   
    int test = 4;  
    int \*attr = &test;  
   
    error = pthread\_create(&id1,NULL,create,(void \*)attr);  
   
    if(error)  
    {  
        printf("Pthread\_create is not created!\n");  
        return -1;  
    }  
    sleep(1);  
   
    printf("Pthread\_create is created..\n");  
    return 0;  
}

编译 **gcc int\_ptr\_thread.c -o int\_ptr\_thread -lpthread**

执行 **./int\_ptr\_thread**

运行结果如下

文本

描述已自动生成

# **2.2.8 字符串传递给新创建的线程函数**

在main函数中传递char指针，传递到新建的线程函数中。

首先创建文件char\_ptr\_thread.c，编写以下代码 **vi char\_ptr\_thread.c**

#include <stdio.h>  
#include <pthread.h>  
   
void \*create(char \*arg)  
{  
    char \*str;  
    str = arg;  
    printf("The parameter passed from main is %s\n",str);  
   
    return (void \*)0;  
}  
   
int main()  
{  
    int error;  
    pthread\_t id1;  
    char \*str1 = "Hello!";  
    char \*attr = str1;  
    error = pthread\_create(&id1, NULL, create, (void \*)attr);  
   
    if(error != 0)  
    {  
        printf("This pthread is not created!\n");  
        return -1;  
    }  
    sleep(1);  
   
    printf("pthread is created..\n");  
    return 0;  
}

编译 **gcc char\_ptr\_thread.c -o char\_ptr\_thread -lpthread**

执行 **./char\_ptr\_thread**

运行结果如下

文本

描述已自动生成

# **2.2.9 结构体传递给新创建的线程函数**

在main函数中传递结构体指针，传递到新建的线程函数中。

首先创建文件struct\_ptr\_thread.c，编写以下代码 **vi struct\_ptr\_thread.c**

#include <stdio.h>  
#include <pthread.h>  
#include <stdlib.h>  
   
struct menber  
{  
    int a;  
    char \*s;  
};  
   
void \*create(void \*arg)  
{  
    struct menber \*temp;  
    temp = (struct menber \*)arg;  
    printf("menber->a = %d\n",temp->a);  
    printf("menber->s = %s\n",temp->s);  
   
    return (void \*)0;  
}  
   
int main()  
{  
    int error;  
    pthread\_t id1;  
    struct menber \*p;  
    p = (struct menber \*)malloc(sizeof(struct menber));  
    p->a = 1;  
    p->s = "Robben!";  
   
    error = pthread\_create(&id1,NULL,create,(void \*)p);  
   
    if(error)  
    {  
        printf("pthread is not created!\n");  
        return -1;  
    }  
    sleep(1);  
    printf("pthread is created!\n");  
   
    free(p);  
    p = NULL;  
    return 0;  
}

编译 **gcc struct\_ptr\_thread.c -o struct\_ptr\_thread -lpthread**

执行 **./struct\_ptr\_thread**

运行结果如下

图片包含 文本

描述已自动生成

# **2.2.10 新建立的线程可以共享进程中的数据**

在main函数中传递结构体指针，传递到新建的线程函数中。

首先创建文件shared\_data\_thread.c，编写以下代码 **vi shared\_data\_thread.c**

#include <stdio.h>  
#include <pthread.h>  
   
static int a = 5;  
   
void \*create(void \*arg)  
{  
    printf("New pthread...\n");  
    printf("a = %d\n",a);  
   
    return (void \*)0;  
}  
   
int main(int argc, const char \*argv[])  
{  
    int error;  
    pthread\_t id1;  
   
    error = pthread\_create(&id1, NULL, create, NULL);  
    if(error != 0)  
    {  
        printf("new thread is not created!\n");  
        return -1;  
    }  
    sleep(1);  
    printf("New thread is created...\n");  
   
    return 0;  
}

编译 **gcc shared\_data\_thread.c -o shared\_data\_thread -lpthread**

执行 **./** **shared\_data\_thread**

运行结果如下

文本

描述已自动生成

# **第三章 实验结果分析**

##### 3.1 thread\_create.c

# **3.1.1 执行结果**

文本

描述已自动生成

# **3.1.2 结果分析**

在给定的代码中，创建了两个线程：mythread1和mythread2。这两个线程在主函数中被分别创建，并且在每个线程的函数内部，通过循环打印一些文本信息。每次打印后，线程会休眠2秒钟。

根据运行结果，可以看出以下情况：

线程2（mythread2）的输出先于线程1（mythread1）的输出。这是因为线程2先被创建，然后才是线程1。

每个线程的输出在交替进行。线程2打印一条消息，然后线程1打印一条消息，然后再次轮到线程2，依此类推。这是因为每个线程在循环中打印5次消息，然后休眠2秒钟，然后切换到另一个线程。

循环交替执行直到每个线程完成其5次迭代。因此，总共会有10条消息打印出来，交替出现。

需要注意的是，线程的执行顺序和调度是由操作系统决定的，并且可能受到系统负载和其他因素的影响。因此，每次运行代码时，输出的顺序可能会有所不同。

##### 3.2 thread\_exit.c

# **3.2.1 执行结果**

文本

描述已自动生成

# **3.2.2 结果分析**

在主函数中，创建了一个新线程 tid，并通过 pthread\_create 函数将线程函数 create 分配给该线程。

在主线程中，打印了一条消息 "I am the main thread!"。

如果成功创建线程，pthread\_create 函数返回0，否则返回非零值。在代码中添加了错误处理，如果创建线程失败，则会打印一条消息 "thread is not created..." 并返回 -1。

在线程函数 create 中，打印了一条消息 "New thread is created..."。

然后，在线程函数中，使用 pthread\_exit 函数退出线程，并将退出码设置为 6。退出码可以是任意整数值，这里设置为 6。

主线程使用 pthread\_join 函数等待线程 tid 的退出，并将退出码存储在 temp 变量中。如果成功等待线程退出，pthread\_join 函数返回0，否则返回非零值。同样也添加了错误处理，如果等待线程退出失败，则会打印一条消息 "Thread is not exit..." 并返回 -2。

最后，在主线程中，打印了一条消息 "Thread is exit code 6"，将 temp 变量的值转换为整数，并打印出来。

根据以上分析，每条运行结果的解释如下：

* + 首先，主线程打印了消息 "I am the main thread!"。
  + 然后，新线程被创建，并打印了消息 "New thread is created..."。
  + 新线程执行完毕后，使用 pthread\_exit 函数退出，退出码设置为 6。
  + 主线程通过 pthread\_join 等待新线程退出，并获取到退出码 6。
  + 最后，主线程打印了消息 "Thread is exit code 6"。

这表明成功创建了一个新线程，并且在新线程中使用 pthread\_exit 函数退出，并返回退出码 6。主线程通过 pthread\_join 等待新线程退出，并获取到退出码。

##### 3.3 thread\_join.c

# **3.3.1 执行结果**

文本

描述已自动生成

# **3.3.2 结果分析**

在主函数中，创建了一个新线程 pth，并通过 pthread\_create 函数将线程函数 thread 分配给该线程。

在主线程中，使用 pthread\_join 函数等待线程 pth 的退出。pthread\_join 函数的第二个参数是一个指向线程返回值的指针，这里设置为 NULL，表示不关心线程的返回值。

如果成功创建线程，pthread\_create 函数返回0，否则返回非零值。

在线程函数 thread 中，使用循环打印了4次消息。每次循环后，线程会休眠2秒钟（使用 sleep 函数）。打印的消息中包含了线程的循环索引 i。

在主线程中，打印了一条消息 "123"。

然后，在主线程中，使用循环打印了3次消息。每次循环后，主线程会休眠1秒钟，并打印消息中包含了主线程的循环索引 i。

根据以上分析，运行结果的解释如下：

* + 首先，新线程开始执行，在循环中打印了4条消息，消息中包含了线程的循环索引 i。每次循环后，线程会休眠2秒钟。
  + 新线程执行完毕后，主线程继续执行，打印了一条消息 "123"。
  + 接着，主线程使用循环打印了3条消息，消息中包含了主线程的循环索引 i。每次循环后，主线程会休眠1秒钟。
  + 最后，程序执行完毕，主线程退出。

这表明新线程在主线程之前开始执行，打印了4条消息。然后，主线程打印了一条消息 "123"，接着使用循环打印了3条消息。线程和主线程的输出交替进行。

##### 3.4 thread\_id.c

# **3.4.1 执行结果**

文本

描述已自动生成

# **3.4.2 结果分析**

在主函数中，创建了一个新线程 tid，并通过 pthread\_create 函数将线程函数 create 分配给该线程。

在主线程中，打印了一条消息 "Main thread is starting..."。

如果成功创建线程，pthread\_create 函数返回0，否则返回非零值。在代码中添加了错误处理，如果创建线程失败，则会打印一条消息 "thread is not created..." 并返回 -1。

在线程函数 create 中，打印了两条消息：

- 第一条消息 "New thread......" 表示新线程的启动。

- 第二条消息使用 pthread\_self 函数获取当前线程的 ID，并打印出来。pthread\_self 函数返回一个 pthread\_t 类型的值，表示当前线程的 ID。

- 第三条消息使用 getpid 函数获取当前线程所属进程的 ID，并打印出来。

主线程在创建新线程后，打印了一条消息 "The main process pid is 2695"，使用 getpid 函数获取主线程所属进程的 ID，并打印出来。

然后，主线程通过 sleep(1) 函数休眠1秒钟。

最后，程序执行完毕，主线程返回。

根据以上分析，运行结果的解释如下：

* + 首先，主线程打印了消息 "Main thread is starting..."，并获取到主线程所属进程的 ID 2695，打印了消息 "The main process pid is 2695"。
  + 然后，新线程开始执行，打印了消息 "New thread......"，并获取到新线程的 ID 3925812992，打印了消息 "This thread's id is 3925812992"，同时获取到新线程所属进程的 ID 2695，打印了消息 "This thread process pid is 2695"。
  + 程序经过1秒钟的休眠后，主线程返回，程序执行完毕。

这表明主线程首先启动，并打印了主线程所属进程的 ID。然后，新线程开始执行，并打印了新线程的 ID 和所属进程的 ID。注意，线程的 ID 是一个无符号整数值，可能在不同的运行环境中具有不同的表示形式。

##### 3.5 thread\_clean.c

# **3.5.1 执行结果**

文本

描述已自动生成

# **3.5.2 结果分析**

在主函数中，创建了两个新线程 tid1 和 tid2，并通过 pthread\_create 函数将线程函数 thr\_fn1 和 thr\_fn2 分配给这两个线程。

在线程函数 thr\_fn1 中，首先打印了一条消息 "Thread1 start"，然后使用 pthread\_cleanup\_push 函数将两个清理函数 clean 压入清理栈，并分别传递不同的参数作为清理函数的参数。

在线程函数 thr\_fn2 中，首先打印了一条消息 "Thread2 start"，然后使用 pthread\_cleanup\_push 函数将两个清理函数 clean 压入清理栈，并分别传递不同的参数作为清理函数的参数。

在主函数中，使用 pthread\_join 函数等待线程 tid1 和 tid2 的退出，并将退出值保存在变量 tret 中。

然后，根据返回的退出值，打印了线程 tid1 和 tid2 的退出码。

根据以上分析，运行结果的解释如下：

* + 首先，线程 tid2 开始执行，打印了消息 "Thread2 start"，然后将两个清理函数 clean 压入清理栈，并分别传递不同的参数作为清理函数的参数。接着，打印了消息 "thread2 push complete"。
  + 然后，线程 tid1 开始执行，打印了消息 "Thread1 start"，然后将两个清理函数 clean 压入清理栈，并分别传递不同的参数作为清理函数的参数。接着，打印了消息 "thread1 push complete"。
  + 在线程函数 thr\_fn1 中，如果传递的参数 arg 非零，则返回值为 1。
  + 在线程函数 thr\_fn2 中，如果传递的参数 arg 非零，则返回值为 3。
  + 主线程使用 pthread\_join 函数等待线程 tid1 的退出，并将退出值保存在变量 tret 中。根据程序的运行结果，线程 tid1的退出码为 1。
  + 主线程使用 pthread\_join 函数等待线程 tid2 的退出，并将退出值保存在变量 tret 中。根据程序的运行结果，线程 tid2 的退出码为 4。
  + 最后，主线程打印了线程 tid1 和 tid2 的退出码。

这表明线程 tid2 先于线程 tid1 执行。线程 tid2 在执行过程中，将两个清理函数压入清理栈，并打印了消息。然后线程 tid1 执行，也将两个清理函数压入清理栈，并打印了消息。线程 tid1 的退出码为 1，线程 tid2 的退出码为 4。

##### 3.6 thread.c

# **3.6.1 执行结果**

图形用户界面, 文本

描述已自动生成

# **3.6.2 结果分析**

在主函数中，定义了常量 THREAD\_NUMBER 表示线程数，REPEAT\_NUMBER 表示每个线程中的小任务数，DELAY\_TIME\_LEVELS 表示小任务之间的最大时间间隔。

定义了线程函数 thrd\_func，该函数接收一个参数 arg，表示线程编号。线程函数中使用随机数生成一个延迟时间，并进行一定次数的循环，每次循环打印线程编号、任务编号和延迟时间。最后，线程函数通过 pthread\_exit 函数退出线程。

在 main 函数中，先使用 srand 函数初始化随机数发生器。

使用 pthread\_create 函数创建多个线程，并将线程函数 thrd\_func 分配给这些线程。每个线程的编号从 0 到 THREAD\_NUMBER-1。

主函数打印了一条消息表示线程创建成功，并进入等待线程结束的状态。

使用 pthread\_join 函数等待每个线程的结束，并打印相应的消息。

根据以上分析，运行结果的解释如下：

* + 首先，主函数打印了一条消息 "creating threads success\nWaiting for threads to finish..." 表示线程创建成功，并等待线程结束。
  + 然后，线程 1 开始执行，打印消息 "Thread 1 is starting"，并进入循环。在循环中，打印了任务编号和延迟时间的消息，然后使用 sleep 函数延迟一段时间。循环结束后，线程 1 打印了消息 "Thread 1 finished"，然后通过 pthread\_exit 函数退出线程。
  + 线程 2 和线程 0 也按照相同的方式执行，分别打印了线程开始、任务执行和线程结束的消息。
  + 主函数使用 pthread\_join 函数等待线程 1、2 和 0 的结束，并根据返回值判断线程是否成功结束。根据程序的运行结果，线程 1、2 和 0 都成功结束。
  + 最后，主函数打印了线程结束的消息。

这表明线程 1、2 和 0 同时开始执行，每个线程按照循环次数进行任务执行，并打印了任务执行的消息。最后，线程 1、2 和 0 分别结束，并打印了线程结束的消息。注意，具体的任务执行顺序和延迟时间是随机生成的，因此每次运行的结果可能会有所不同。

##### 3.7 int\_ptr\_thread.c

# **3.7.1 执行结果**

文本

描述已自动生成

# **3.7.2 结果分析**

在主线程中，定义了一个整型变量test并初始化为4。然后将其地址赋值给整型指针attr。

主线程调用pthread\_create函数创建一个新的线程。pthread\_create函数的第四个参数是一个void指针，可以用于向新线程传递参数。在这里，attr被强制转换为void指针类型传递给新线程。

新线程被创建后，它执行create函数。在create函数中，传递给该函数的参数被解释为指向整型值的指针。因此，通过将void指针强制转换为int指针，将参数值赋给整型指针num。

在新线程中，使用printf函数打印出传递进来的整型值，即"Create parameter is 4"。

主线程中使用sleep函数延迟1秒，以确保新线程有足够的时间执行。

最后，主线程打印出"Pthread\_create is created.."。

综上所述，运行结果表明成功创建了新线程，并且通过向新线程传递整型值，新线程打印出了正确的参数值。

##### 3.8 char\_ptr\_thread.c

# **3.8.1 执行结果**

文本

描述已自动生成

# **3.8.2 结果分析**

在主线程中，定义了一个字符指针str1，并将字符串"Hello!"的地址赋值给它。然后将str1赋值给字符指针attr。

主线程调用pthread\_create函数创建一个新的线程。pthread\_create函数的第四个参数是一个void指针，可以用于向新线程传递参数。在这里，attr被强制转换为void指针类型传递给新线程。

新线程被创建后，它执行create函数。在create函数中，传递给该函数的参数被解释为字符指针。因此，通过将void指针强制转换为字符指针，将参数值赋给字符指针str。

在新线程中，使用printf函数打印出传递进来的字符串参数，即"The parameter passed from main is Hello!"。

主线程中使用sleep函数延迟1秒，以确保新线程有足够的时间执行。

最后，主线程打印出"pthread is created.."。

综上所述，运行结果表明成功创建了新线程，并且通过向新线程传递字符串参数，新线程打印出了正确的参数值。

##### 3.9 struct\_ptr\_thread.c

# **3.9.1 执行结果**

图片包含 文本

描述已自动生成

# **3.9.2 结果分析**

在主线程中，定义了一个结构体member，其中包含一个整型成员a和一个字符指针成员s。

主线程中使用malloc函数为结构体指针p分配内存，大小为sizeof(struct member)。然后，将整型成员a赋值为1，字符指针成员s赋值为指向字符串"Robben!"的地址。

主线程调用pthread\_create函数创建一个新的线程。pthread\_create函数的第四个参数是一个void指针，可以用于向新线程传递参数。在这里，将结构体指针p强制转换为void指针类型传递给新线程。

新线程被创建后，它执行create函数。在create函数中，传递给该函数的参数被解释为结构体指针。因此，通过将void指针强制转换为结构体指针，将参数值赋给结构体指针temp。

在新线程中，使用printf函数打印出传递进来的结构体参数的成员值。首先打印出整型成员a的值为1，然后打印出字符指针成员s的值为"Robben!"。

主线程中使用sleep函数延迟1秒，以确保新线程有足够的时间执行。

最后，主线程打印出"pthread is created!"。

在程序结束前，主线程使用free函数释放结构体指针p所指向的内存，并将指针p设为NULL。

综上所述，运行结果表明成功创建了新线程，并且通过向新线程传递结构体参数，新线程打印出了正确的成员值。同时，在程序结束前，主线程正确释放了动态分配的内存。

##### 3.10 shared\_data\_thread.c

# **3.10.1 执行结果**

文本

描述已自动生成

# **3.10.2 结果分析**

在全局作用域中定义了一个静态整型变量a，初始化为5。这意味着该变量在进程的整个生命周期内都是可见的。

主线程中调用pthread\_create函数创建一个新的线程。pthread\_create函数的第四个参数是一个void指针，可以用于向新线程传递参数。在这里，传递了NULL作为参数，表示不向新线程传递任何数据。

新线程被创建后，它执行create函数。在create函数中，打印出"New pthread..."和全局变量a的值。由于全局变量a对所有线程可见，因此新线程能够访问并打印出其值为5。

主线程中使用sleep函数延迟1秒，以确保新线程有足够的时间执行。

最后，主线程打印出"New thread is created..."。

综上所述，运行结果表明新线程可以访问并共享进程中的数据。在这种情况下，新线程能够正确访问并打印出全局变量a的值为5。

# **第四章 总结**

本实验主要探究了Pthread线程库的基本概念和使用方法，以及线程创建、管理、通信和清理的相关技术。实验分为六个子实验，分别演示了不同的线程操作和效果。实验的目的是加深对线程编程的理解和掌握，提高并行程序设计的能力。

实验中，我们使用了Pthread线程库，它是一种跨平台的线程标准，提供了一系列的函数来创建、终止、等待、同步和控制线程。我们学习了如何使用pthread\_create()函数来创建线程，如何使用pthread\_exit()函数来退出线程，如何使用pthread\_join()函数来实现线程等待，如何使用pthread\_self()函数来获取线程ID，如何使用pthread\_cleanup\_push()和pthread\_cleanup\_pop()函数来设置和执行线程清理函数，以及如何使用互斥锁和条件变量来实现线程间的同步和通信。

实验结果表明，线程是一种轻量级的进程，它可以共享进程的地址空间和资源，但拥有自己的栈和寄存器。线程的创建和销毁比进程的创建和销毁更快，线程的切换也比进程的切换更快。线程可以利用多核处理器的并行性，提高程序的执行效率和响应速度。但线程也存在一些问题，如线程安全、死锁、竞态条件等，需要使用合适的同步机制来避免或解决。

通过本实验，我对线程编程有了更深入的认识和体会，也发现了自己在编程技巧和逻辑思维方面的不足，需要在今后的学习中继续努力和改进。