**实验六\_14011907\_2019302789\_梁芮槐**

**实验要求：**

（1）编写程序，实现父子进程通过共享内存进行数据通信。父子进程通过竞争方式来创建一个共享内存单元，然后子进程接受用户输入的信息（通过argv[1]输入），并将其写入到共享内存单元；父进程则从共享内存单元将该信息读出，并显示信息的个数。具体步骤为：

1. 创建子进程；
2. 将运行参数argv[1]字符串写入共享内存；
3. 并打印写入字符串;
4. 在父进程中读出所写入的字符串并打印。
5. 最后练习memcpy和memmove内存段数据处理函数的使用

process\_shared\_mem.c:

// 创建共享内存，子进程写入argv[1]，父进程读取

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#include <unistd.h>

void write\_unit(int shmid, char\*);

void read\_unit(int shmid);

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

{

pid\_t fpid;

int shmid;

shmid = shmget(IPC\_PRIVATE, 1024, IPC\_CREAT | 0600);

printf("shmid: %d\n", shmid);

if(shmid < 0)

{

perror("shmget error!\n");

exit(1);

}

printf("shmget OK. Create shared memory successfully.\n");

fpid = fork();

if(fpid < 0)

{

printf("create child process error!\n");

exit(1);

}

else if(fpid == 0)

{

write\_unit(shmid, argv[1]);

}

else

{

sleep(2);

read\_unit(shmid);

}

return 0;

}

void write\_unit(int shmid, char\* arg)

{

char\* shmaddr = (char\*)shmat(shmid, NULL, 0);

if((int)shmaddr == -1)

{

perror("shmaddr error.\n");

exit(1);

}

//strcpy(shmaddr, arg); // strcpy方式

//memcpy(shmaddr, arg, strlen(arg) + 1); // memcpy方式

memmove(shmaddr, arg, strlen(arg) + 1); // memmove方式

printf("write shared memory: %s\n", shmaddr);

shmdt(shmaddr);

}

void read\_unit(int shmid)

{

char\* shmaddr = (char\*)shmat(shmid, NULL, 0);

if((int)shmaddr == -1)

{

perror("shmaddr error.\n");

exit(1);

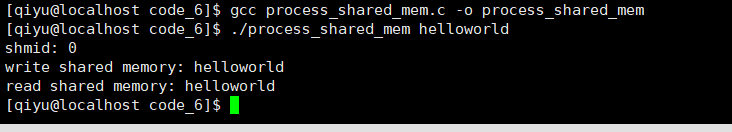
}

printf("read shared memory: %s\n", shmaddr);

shmdt(shmaddr);

}

运行结果：



笔记：

// 开辟或使用一块共享内存，返回shmid

int shmget(key\_t key, int size, int shmflg);

// 将参数shmid所指向的共享内存与当前进程连接。当使用某共享内存时，需要先使用shmat，达成连接

void \*shmat(int shmid, const void \*shmaddr, int shmflg);

// 将先前用shmat连接的共享内存与当前进程解除连接。参数shmaddr为shmat返回的共享内存的地址

int shmdt(const void \*shmaddr);

// 控制内存的操作。当cmd为IPC\_RMID时，删除shmid所指的共享内存

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

// memcpy()从源src所指的内存地址的起始位置开始拷贝N个字节到目标dst所指的内存地址的起始位置中

void \*memcpy(void \*dst, const void \*src, size\_t count);

// memmove()功能同memcpy基本一致，但是当src区域和dst内存区域重叠时，memcpy可能会出现错误，而memmove能正确进行拷贝

void \*memmove(void \*dst, const void \*src, size\_t count);

// 可见memcpy的速度比memmove快一点，如果使用者可以确定内存不会重叠，则可以选用memcpy，否则memmove更安全一些。

（2）查阅相关资料，了解Linux多线程编程，并完成以下实验

1、编写程序实现如下功能：编写程序，在主线程中创建一个新线程。要求在新线程中输出运行信息，在结束时返回主线程。使用函数pthread\_exit（0）退出线程。再试试exit(0)退出线程，看看是什么结果。比较一下二者的不同。

new\_thread\_output.c:

// 主线程创建新线程，新线程输出运行信息，分别试用pthread\_exit()和exit()退出新线程

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <time.h>

#include <string.h>

void output(void)

{

static char s[30]={0};

char YMD[15] = {0};

char HMS[10] = {0};

time\_t current\_time;

struct tm\* now\_time;

char \*cur\_time = (char \*)malloc(21\*sizeof(char));

time(&current\_time);

now\_time = localtime(&current\_time);

strftime(YMD, sizeof(YMD), "%F ", now\_time);

strftime(HMS, sizeof(HMS), "%T", now\_time);

strncat(cur\_time, YMD, 11);

strncat(cur\_time, HMS, 8);

printf("\nchild thread current time: %s\n", cur\_time);

memcpy(s, cur\_time, strlen(cur\_time)+1);

free(cur\_time);

cur\_time = NULL;

//pthread\_exit(NULL);

exit(0);

}

int main()

{

int ret;

pthread\_t thread\_1;

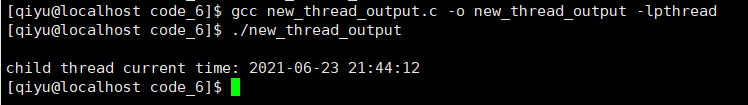
ret = pthread\_create(&thread\_1, NULL, (void\*)output, NULL);

pthread\_join(thread\_1, NULL);

printf("main thread end\n");

}

运行结果：



分析：

pthread\_exit用于线程退出，可以指定返回值，以便其他线程通过pthread\_join函数获取该线程的返回值。

exit是进程退出，如果在线程函数中调用exit，则该线程的进程也就退出了,会导致该线程所在进程的其他线程也错误退出，引发问题。

2、在两个线程之间实现交替执行输出。要求：例如主线程先输出“this is a thread0”,然后新线程输出“this is thread 2”一直交替直到结束。同时访问全局变量，修改变量的值，并打印看看输出结果。

thread\_alternate\_output.c:

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <time.h>

#include <string.h>

int shared\_sign = 0;

pthread\_mutex\_t father\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t child\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t father\_cond = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t child\_cond = PTHREAD\_COND\_INITIALIZER;

void child(void)

{

int count = 4;

pthread\_cond\_signal(&child\_cond);

while(count--)

{

pthread\_cond\_wait(&father\_cond, &father\_mutex);

pthread\_mutex\_trylock(&child\_mutex);

shared\_sign--;

printf("this is thread 2\n");

printf("current num is: -------->%d\n", shared\_sign);

printf("--------------------------------------\n");

pthread\_mutex\_unlock(&child\_mutex);

pthread\_cond\_signal(&child\_cond);

}

pthread\_exit(NULL);

}

void father(void)

{

int count = 4;

while(count--)

{

pthread\_cond\_wait(&child\_cond, &child\_mutex);

pthread\_mutex\_lock(&father\_mutex);

shared\_sign++;

printf("this is thread 1\n");

printf("current num is: -------->%d\n", shared\_sign);

printf("--------------------------------------\n");

pthread\_mutex\_unlock(&father\_mutex);

pthread\_cond\_signal(&father\_cond);

}

pthread\_cond\_signal(&father\_cond);

pthread\_exit(NULL);

}

int main()

{

int ret;

pthread\_t thread\_1, thread\_2;

ret = pthread\_create(&thread\_1, NULL, (void\*)father, NULL);

ret = pthread\_create(&thread\_2, NULL, (void\*)child, NULL);

pthread\_join(thread\_1, NULL);

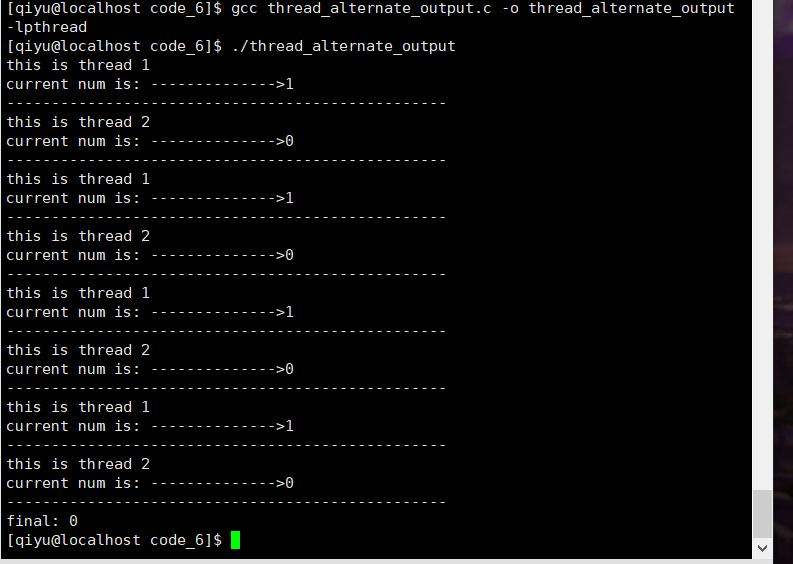
pthread\_join(thread\_2, NULL);

printf("final: %d\n", shared\_sign);

return 0;

}

运行结果：



分析：

我加上了两个pthread\_cond\_t条件信号让两个线程通过等待对方完成的信号量来实现交替输出。

主要是遇到了一个奇怪的bug——父线程最后一次循环结束时子线程的锁却被它锁上了，而拿出来用gdb调试的结果是子线程的锁被已经exited的父线程锁了。导致子线程最后一次循环尝试给自己的锁上锁时，因为这个锁已经锁上了而卡死。尝试在父线程结束时给子线程的锁解锁也不能解决。所以我换成了pthread\_mutex\_trylock来给子线程上锁，这样它不会尝试锁已经存在的锁，从而不会卡死。

3、编写程序实现使用信号量进行同步。编码实现输入字符串，统计每行的字符个数，以“end”结束输入。

semaphore\_synchronize\_strlen.c:

// 读取每行输入并输出该行字符长度

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <time.h>

#include <unistd.h>

#include <semaphore.h>

#include <string.h>

sem\_t sem\_1;

sem\_t sem\_2;

void count\_strlen(char\*);

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

{

int ret;

pthread\_t thread\_1;

char\* msg;

// 初始化

ret = sem\_init(&sem\_1, 0, 0);

if(ret == -1)

{

perror("semaphore initialization failed\n");

exit(EXIT\_FAILURE);

}

ret = sem\_init(&sem\_2, 0, 1);

if(ret == -1)

{

perror("semaphore initialization failed\n");

exit(EXIT\_FAILURE);

}

ret = pthread\_create(&thread\_1, NULL, (void\*)count\_strlen, msg);

if(ret != 0)

{

perror("pthread\_create failed\n");

exit(EXIT\_FAILURE);

}

// 开始输入

printf("Input line by line and type 'quit' to finish\n");

sem\_wait(&sem\_2);

while(strcmp("quit\n", msg) != 0)

{

fgets(msg, 512, stdin);

sem\_post(&sem\_1);

sem\_wait(&sem\_2);

}

printf("\nending...\n");

ret = pthread\_join(thread\_1, NULL);

if(ret != 0)

{

perror("pthread\_join failed\n");

exit(EXIT\_FAILURE);

}

sem\_destroy(&sem\_1);

sem\_destroy(&sem\_2);

printf("end successfully");

return 0;

}

void count\_strlen(char\* msg)

{

sem\_wait(&sem\_1);

while(strcmp("quit\n", msg) != 0)

{

printf("last line has %ld characters\n\n", strlen(msg) - 1);

sem\_post(&sem\_2);

sem\_wait(&sem\_1);

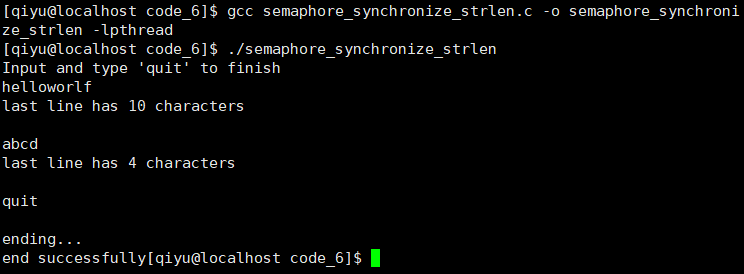
}

sem\_post(&sem\_2);

pthread\_exit(NULL);

}

运行结果：



分析：

我采用两个sem\_t信号量来实现多线程信息同步，读取输入的主线程和输出字符串长度的子线程分别控制一个信号量，只有在另一个线程的信号量等待不阻塞时才能继续运行。