操作系统原理

理论课实验2 - 进程通信、命令解释器、线程实验

学院	专业	班级	学号	姓名
数据科学与计算机学院	软件工程	教务2班	17343131	许滨楠

实验目的及具体要求

- 进程间共享内存实验，了解共享内存形式的进程通信；
- 实现简单的Shell命令解释器，了解程序运行的大致过程；
- 完成线程实验，了解线程的创建和运行：
    - 利用线程生成Fibonacci数列；
    - 实现多线程矩阵乘法。

实验环境及工具准备

实验环境

- 主系统为MacOS X 10.14 Mojave
- 虚拟机系统为Linux Ubuntu 14.04 LTS

实验工具

- 在Parallels Desktop软件上运行虚拟机
- 代码文件编辑器：Visual Studio Code、Gedit Text Editor
- 终端：Mac下iTerm、Ubuntu下Terminal
- 终端Shell：zsh、其他环境包括gcc编译器、链接器以及一些小工具
- 实验报告用Markdown语言在Typora软件上编写并导出pdf

实验流程及结果验证

1. 进程通信实验

具体流程

使用fork()创建子进程，计算产生Fibonacci前n个数（n <= 10），存入共享内存区，由父进程输出。

程序中使用的相关函数包括：

- int shmget(key_t key, size_t size, int flag);
  - 用于创建或打开共享存储区。
  - key为共享存储区规则的标识符，此次程序中未使用ftok而是直接根据实验指导中示例使用了IPC_PRIVATE保证使用唯一ID；
  - size为共享存储区大小，程序中直接用sizeof(shmptr)来表示程序中指定的结构大小作为存储区大小；
  - flag为共享存储区的读写权限，此次程序中用了S_IRUSR | S_IWUSR，表示存储区为用户可读可写。
- void * shmat(int shmid, const void * addr, int flag);
  - 用于连接共享存储区。
  - shmid为共享存储区标识，在使用shmget创建时保存在句柄中；
  - addr和flag一般表示为0即可正常使用。
- int shmdt(void * addr); 和 int shmctl(int shmid, int cmd, struct shmid_ds * buffer);
  - 用于在使用完毕后拆除连接、回收删除共享存储区。

根据以上基本的函数，参照实验指导中的示例，完成实验代码编写。（源代码见附件中 **/src/1/** ）基本流程为获得用户输入后，声明共享存储区，之后根据pid执行相关代码：父进程挂起等待，子进程中完成计算写入数据，父进程访问数据进行输出。

结果验证

代码及Makefile编写完毕后，在Ubuntu终端cd进入代码目录，执行 **make** 语句完成代码编译，接着键入命令： **./1-ShareMemory** 执行程序，按照提示输入1-10的任意数字即可查看结果。

```
 pie@ubuntu  src/1  make clean
rm 1-ShareMemory
 pie@ubuntu  src/1  make
gcc -std=c99 -D_XOPEN_SOURCE=600 1-ShareMemory.c -o 1-ShareMemory
 pie@ubuntu  src/1  ./1-ShareMemory
Please enter a positive integer less than 11: 10
[result]: 0 1 1 2 3 5 8 13 21 34
 pie@ubuntu  src/1
```

结果满足实验要求。

2. 命令解释器实验

具体流程

参考实验指导中"main函数实现的建议"、"execvp()的使用"、"setup函数的实现"，加上历史记录相关处理，实现一个简陋的shell - 命令解释器。其实主要是模拟系统shell的前端，主要负责将命令读入，划分命令和参数，然后还是通过execvp()来执行命令，跟系统本身执行无异，添加了历史命令的罗列和执行、后台运行的功能而已。

历史命令功能利用queue相关操作实现，后台运行通过指定代码运行时主线程是否挂起来实现。用系统库<signal.h>来处理命令。（源代码见附件中 **/src/2/** ）

参照指导可见的main函数框架和初始化处理如下：

```
1  int main() {
```

```
    char buffer[MAX_LINE];
    char * args[MAX_LINE/2+1];
    int isbg;
    pid_t pid;
    struct sigaction handler;

    queue.front = -1;
    queue.rear = 0;

    while(1) {
        isbg = 0;
        printf("COMMAND -> ");
        fflush(stdout);
        setup(buffer, args, &isbg);

        if (-1 == (pid = fork())) printf("[error] when forking.\n");
        if (0 == pid) {
            execvp(args[0], args);
            exit(0);
        }
        if (!isbg) wait(0);

        handler.sa_handler = (void (*)(int))handle_SIGINT;
        sigaction(SIGINT, &handler, NULL);
    }

    return 0;
}
```

结果验证

由于此项实验代码文件稍多，用Makefile的优势更加体现。根据依赖关系编写Makefile如下：

```
2-Shell: 2-Shell.o 2-Setup.o 2-Queue.o
gcc -std=c99 2-Shell.o 2-Setup.o 2-Queue.o -o 2-Shell

2-Shell.o: 2-Shell.c
gcc -c -std=c99 -D_XOPEN_SOURCE=600 2-Shell.c

2-Setup.o: 2-Setup.c 2-Setup.h
gcc -c -std=c99 -D_XOPEN_SOURCE=600 2-Setup.c

2-Queue.o: 2-Queue.c 2-Queue.h
gcc -c 2-Queue.c

clean:
rm 2-Shell *.o
```

同1中，进入代码目录， **make** 执行编译链接， **./2-Shell** 运行，即可检验程序运行情况。

```
 pie@ubuntu  src/2  make clean
rm 2-Shell *.o
 pie@ubuntu  src/2  make
gcc -c -std=c99 -D_XOPEN_SOURCE=600 2-Shell.c
gcc -c -std=c99 -D_XOPEN_SOURCE=600 2-Setup.c
gcc -c 2-Queue.c
gcc -std=c99 2-Shell.o 2-Setup.o 2-Queue.o -o 2-Shell
 pie@ubuntu  src/2  ./2-Shell
COMMAND ->
```

```
 pie@ubuntu  src/2  ./2-Shell
COMMAND -> ls
2-Queue.c  2-Queue.o  2-Setup.h  2-Shell    2-Shell.o
2-Queue.h  2-Setup.c  2-Setup.o  2-Shell.c  makefile
COMMAND -> ps
  PID TTY          TIME CMD
 2628 pts/8    00:00:00 zsh
20987 pts/8    00:00:00 2-Shell
21150 pts/8    00:00:00 ps
COMMAND -> ^C
ls
ps
COMMAND -> r1
2-Queue.c  2-Queue.o  2-Setup.h  2-Shell    2-Shell.o
2-Queue.h  2-Setup.c  2-Setup.o  2-Shell.c  makefile
COMMAND ->
```

可以看到基本完成实验要求。其他多种命令，包括后台运行，ping命令，新建/删除文件夹或文件的命令，都可以正常执行。结果满足实验要求。

3.1 线程实验 - 利用线程生成Fibonacci数列实验

具体流程

用c语言pthread线程相关的系统调用（库文件：<pthread.h>），创建一个子线程计算Fibonacci序列，同时父进程挂起等待子进程执行完毕，输出结果。其中使用到的pthread相关系统调用包括：

- int pthread_create(pthread_t * tid, const pthread_attr_t * attr, void * (*start_routine)(void), void * arg);
  - 用于创建线程，保存调用该线程函数的入口。
  - tid用于标识线程id；attr用于设置线程属性；start_routine用于标记线程运行函数的起始地址；arg用于设置线程运行函数的参数，当参数有多个时，通过设置struct结构体等方法进行参数传递。
- int pthread_join(pthread_t * tid, (void *)value);
  - 用于阻塞当前线程，直到tid对应的线程退出，实验中用于阻塞父线程等待子线程计算完毕。
  - value为退出线程的返回值，实验中无需用到，设置为NULL/0即可。

根据以上基础知识和API，按照前述思路编写代码。（源代码见附件中 **/src/3/3.1/** ）主要代码：

```
1  // 库文件包含与宏定义
2  // ……
3
4  typedef struct {
5      int num;
```

```
    long long fibo[MAX_SEQUENCE];
} Data;

void fib(void *data) {
    Data * t = (Data *) data;
    int i;
    t->fibo[0] = 0;
    t->fibo[1] = 1;
    if (0 == t->num || 1 == t->num) return;
    for (i = 2; i < t->num; ++i)
        t->fibo[i] = t->fibo[i-1] + t->fibo[i-2];
    pthread_exit(NULL);
}

int main() {
    // 读入和合法性判定处理
    // ……

    if ((ret = pthread_create(&th, 0, (void *)&fib, (void *)&data)) != 0)
{
        printf("[error] when creating thread.\n");
        return -1;
    }

    pthread_join(th, 0);

    // ……
    // 结果输出
}
```

结果验证

进入代码目录，用 **make** 命令执行编译链接，键入 **./3-Fibonacci** 运行，即可检验程序运行情况。

```
 pie@ubuntu  3/3.1  make clean
rm 3-Fibonacci
 pie@ubuntu  3/3.1  make
gcc 3-Fibonacci.c -o 3-Fibonacci -pthread
 pie@ubuntu  3/3.1  ./3-Fibonacci
Please enter a positive integer in [1, 50]: 0
[invalid] Please input a positive integer: 100
[invalid] Please input a positive integer less than 51: 30
[result]: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 1
0946 17711 28657 46368 75025 121393 196418 317811 514229
 pie@ubuntu  3/3.1
```

结果满足实验要求。

3.2 线程实验 - 利用多线程完成矩阵乘法

具体流程

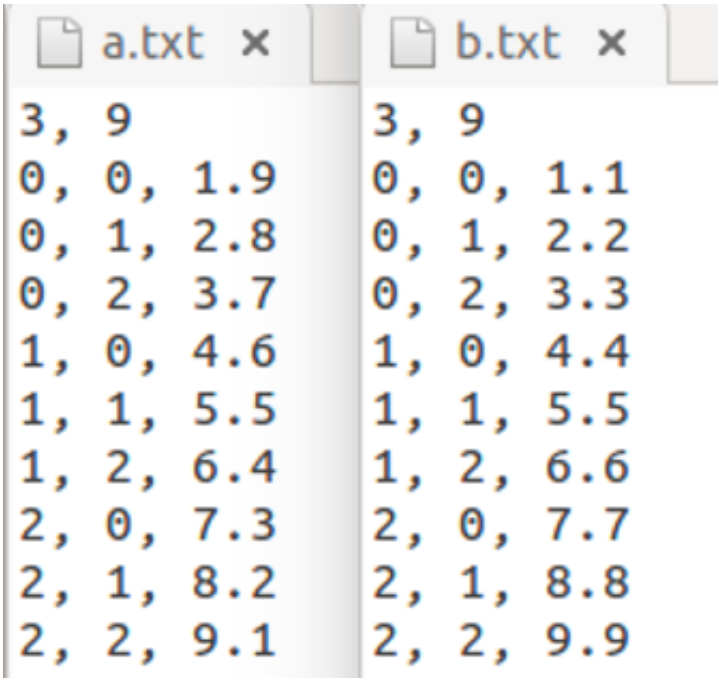
使用的pthread API和上一个实验相同。不同的是这个实验程序中需要产生多个线程来计算各个矩阵点的结果，以及增加了文件读写的操作。程序大概流程为：

- 打开两个初始矩阵的存储文件（文件中格式按照指导可见中的方阵三元组存储格式存储） —>
- 判断是否合法（同阶方阵，数据规模合适） —>
- 创建线程指定函数和参数 —>
- 启动线程执行，将运算结果存在公共的全局变量中 —>
- 输出运行结果，提示结果文件将生成 —>
- 将结果以方阵三元组的格式存储到文件中 —>
- 关闭文件，程序退出。

（源代码见附件中 **/src/3/3.2/** ）

结果验证

准备好初始两个矩阵的存储文件如下：



cd进入目录，**make** 命令编译， **./3-Matrix** 命令运行，查看结果：

```
 pie@ubuntu  src/3  cd 3.2
 pie@ubuntu  3/3.2  make clean
rm 3-Matrix
 pie@ubuntu  3/3.2  make
gcc 3-Matrix.c -o 3-Matrix -pthread
 pie@ubuntu  3/3.2  ./3-Matrix
The result of A * B:
     42.90       52.14       61.38
     78.54       96.69      114.84
    114.18      141.24      168.30
You can also check the result in c.txt
 pie@ubuntu  3/3.2
```

a.txt

```
3, 9
0, 0, 1.9
0, 1, 2.8
0, 2, 3.7
1, 0, 4.6
1, 1, 5.5
1, 2, 6.4
2, 0, 7.3
2, 1, 8.2
2, 2, 9.1
```

b.txt

```
3, 9
0, 0, 1.1
0, 1, 2.2
0, 2, 3.3
1, 0, 4.4
1, 1, 5.5
1, 2, 6.6
2, 0, 7.7
2, 1, 8.8
2, 2, 9.9
```

c.txt

```
3, 9
0, 0, 42.90
0, 1, 52.14
0, 2, 61.38
1, 0, 78.54
1, 1, 96.69
1, 2, 114.84
2, 0, 114.18
2, 1, 141.24
2, 2, 168.30
```

经过验证，程序计算结果与Matlab计算结果一致：

```
>> A=[1.9 2.8 3.7;4.6 5.5 6.4;7.3 8.2 9.1]

A =

    1.9000    2.8000    3.7000
    4.6000    5.5000    6.4000
    7.3000    8.2000    9.1000

>> B=[1.1 2.2 3.3;4.4 5.5 6.6;7.7 8.8 9.9]

B =

    1.1000    2.2000    3.3000
    4.4000    5.5000    6.6000
    7.7000    8.8000    9.9000

>> C=A*B

C =

   42.9000   52.1400   61.3800
   78.5400   96.6900  114.8400
  114.1800  141.2400  168.3000
```

结果满足实验要求。

思考与总结

本次实验是理论课的第二次实验，对比之前的fork()创建子进程的实验，这次的内容和难度都提高了很多，花费了不少时间。但由于主要内容其实还是相关知识的运用特别是一些API的调用，实验使用的语言也是比较熟悉的C语言，加上老师的上课讲解和指导参考资料还算充足，所以还算是没走太多弯路地完成了下来。理论课的实验还是非常有必要的，因为这让我们在自己实践的过程中从操作和使用的角度理解了操作系统理论的一些知识，不管是对于加深印象还是更好地理解，都有很大的作用。（尽管要花费很多时间，理论课实验课两个实验加上书面作业多管齐下着实给我们带了了不小的压力）能学到东西还是好的！

共享内存这样的通信方式抽象层次上理解起来不难，但要真的去实现还是需要好好琢磨的，对于各个API的理解和使用都要花一些时间，但如果真的理解了，其实当成一个概念上的数组或者结构体的存储型变量来做也并不困难；命令解释器虽然入学以来接触得越来越多，日常也都会使用，但是自己实现还是第一次，虽然是通过execvp()函数来"伪实现"命令解释器的功能，但是了解了一直很好奇的历史命令机制的简单实现，和对命令的处理分解传参执行，还是很有收获的；线程方面在上学期学习JAVA已经有接触过一些，在C中还是第一次使用，虽然有两个小项目但是对比前两个实验，这两个程序实现起来还是比较直观明了的，结果也顺利验证成功，虽然还没有深入，但作为pthread API的接触了解的实验，目的还是达到了的。

花了不少时间，有了不少收获，本次实验还是很有意义和价值的。但希望下次不要再多作业"批处理"，多作业"并行"了。但其实也是学好这门重要课程不得不做的一些事情吧。