南京航空航天大学

**实 验 报 告**

**课程名称 操作系统实践**

**姓 名**

**学 号**

**班 级**

**日 期**

目录

**[一、Job6](#_Toc105779050)** [3](#_Toc105779050)

[1. 题目要求 3](#_Toc105779051)

[2. 实现思路 3](#_Toc105779052)

[3. 关键代码 3](#_Toc105779053)

[4.运行结果 6](#_Toc105779054)

**[二、job7](#_Toc105779055)** [6](#_Toc105779055)

[1. 题目要求 6](#_Toc105779056)

[2. 实现思路 7](#_Toc105779057)

[3. 关键代码 7](#_Toc105779058)

[4.运行结果 8](#_Toc105779059)

**[三、job8](#_Toc105779060)** [8](#_Toc105779060)

[1. 题目要求 8](#_Toc105779061)

[2. 实现思路 8](#_Toc105779062)

[3. 关键代码 9](#_Toc105779063)

[4.运行结果 9](#_Toc105779064)

**[四、job9](#_Toc105779065)** [10](#_Toc105779065)

[1. 题目要求 10](#_Toc105779066)

[2. 实现思路 10](#_Toc105779067)

[3. 关键代码 11](#_Toc105779068)

[4.运行结果 11](#_Toc105779069)

**[五、job10](#_Toc105779070)** [12](#_Toc105779070)

[1. 题目要求 12](#_Toc105779071)

[2. 实现思路 13](#_Toc105779072)

[3. 关键代码 13](#_Toc105779073)

[4.运行结果 15](#_Toc105779074)

**一、Job6**

1. 题目要求

·实现shell程序，要求支持基本命令、重定向命令、管道命令、后台命令

·使用结构体 tree 描述命令

·从命令行中读取一行命令，输出该命令的结构

1. 实现思路

我使用了给出的tree\_t结构体，在自己的parse.c编写的tree\_build()函数构建树，print\_t()函数输出树结构。在exec.c中编写了exec\_wrapper()，execute(),execute\_pipe(),exec\_back(),exec\_basic(),exec\_redirict()等函数实现shell功能。Sh3.c中对这些进行调用，实现系统。

对于tree\_build(),我的想法是，根据读入的命令，首先判断命令最后一位是否为后台命令，构建back结点，接着对剩下的命令从左向右进行读取，构建basic结点，遇到重定向命令便构建redirict结点，将已构建的basic结点作为子树连接进去，对于同时重定向输入输出，再构建一个redirict结点，将之前的redirict结点作为子树连接进去。而当遇到管道符”|”时，此时管道符左侧命令子树已经构建完成，构建pipe结点，然后从管道符的右侧开始，递归调用tree\_build()函数，构建其右命令子树。

但这样会出现管道符处理顺序问题，正常应使最右侧的pipe结点作根节点，而这样构建会使最左侧pipe结点为根节点，所以需要旋转。对于pipe1结点，如果其右结点为pipe2为pipe型，则将pipe1的右结点改为pipe2的左结点，而pipe2的左结点改为pipe1，同时，不再用pipe1改为用pipe2作为根节点，最后判断是否与back结点连接并返回根节点。打印树结构只要遍历树即可。

在exec.c中实现shell程序。用exwc\_wrapper()封装，首先判断是否为内置指令，是便执行，否则进行fork()来执行shell命令，这是因为内置命令中exit是要退出整个程序的，不能在子进程中执行，而shell命令都需要掉用execvp实现，会清空代码段，所以只能在子进程中执行。判断根节点是否为back类型来决定是否wait子进程结束。子进程中调用execute(),将不同类型的结点分派到其对应的exec函数中。

Back类型继续execute其子节点，basic类型则从子树取词调用execvp，redirict类型则先根据重定向类型，重定向输入或输出，接着继续调用execute执行其子树。对于pipe类型，需要先创建管道，再fork()创建子进程。在子进程中，将输出端重定向到管道的写端，然后执行左子树。在对应主进程中，等待子进程完成，将输入端重定向管道读端（记得关闭无用fd），接着执行右子树。

1. 关键代码

Tree\_build（）函数太长就不粘了，给出exec\_wrapper(),exec\_redirict(),exec\_pipe(),exec\_basic()的代码

void exec\_wrapper(*tree\_t* \**tree*)

{

    int tag = -1;

    if(*tree*->type == TREE\_BASIC) // 判断是否为内置指令

        tag = exec\_build\_in(*tree*);

    if(tag == 1)

        return;

*pid\_t* pid;

    pid = fork();

    if(pid <= 0)

    {

        execute(*tree*);

exit(1);

    }

    else

    {

        int status;

        if(*tree*->type == TREE\_BACK) // 后台指令不需要等待子进程结束

            return;

        else

            wait(&status);

    }

}

void exec\_redirict(*tree\_t* \**tree*)

{

    int fd;

    //puts(tree->child\_vector[1]->token);

    if(*tree*->child\_vector[1]->token[0] == '<') //重定向输入

    {

        fd = open(*tree*->child\_vector[2]->token, O\_RDONLY, 0777);

        dup2(fd,0);

        close(fd);

    }

    else if(strlen(*tree*->child\_vector[1]->token) == 1) //重定向输出

    {

        fd = open(*tree*->child\_vector[2]->token,O\_CREAT|O\_RDWR|O\_TRUNC, 0777);

        dup2(fd,1);

        close(fd);

    }

    else

    {

        fd = open(*tree*->child\_vector[2]->token,O\_RDWR|O\_APPEND, 0777);

        dup2(fd,1);

        close(fd);

    }

    execute(*tree*->child\_vector[0]);

}

void exec\_pipe(*tree\_t* \**tree*)

{

    int fd[2];

    pipe(fd);

*pid\_t* pid;

    pid = fork();

    if(pid == 0)

    {

        dup2(fd[1],1); // 重定向输出到管道写端

        close(fd[1]);

        close(fd[0]);

        execute(*tree*->child\_vector[0]); //执行左子树

        exit(1);

    }

    int status;

    wait(&status); // 等待左子树指令执行完成

    dup2(fd[0],0); // 重定向输入到管道读端

    close(fd[0]);

    close(fd[1]);

    execute(*tree*->child\_vector[1]);

}

void execute(*tree\_t* \**tree*)

{

    switch (*tree*->type)

    {

    case TREE\_BACK:

        exec\_back(*tree*);

        break;

    case TREE\_BASIC:

        exec\_basic(*tree*);

        break;

    case TREE\_PIPE:

        exec\_pipe(*tree*);

        break;

    case TREE\_REDIRICT:

        exec\_redirict(*tree*);

        break;

    default:

        break;

    }

}

void exec\_basic(*tree\_t* \**tree*)

{

    char \*\*argv = (char \*\*)malloc(128\*sizeof(char \*));

    get\_argv(*tree*, argv); // 从子树中取token组成命令

    execvp(argv[0], argv);

}

4.运行结果

命令：cat <log | grep e | wc -l >logp &

结果：

back

pipe

pipe

redirict

basic

cat

<

log

basic

grep

e

redirict

basic

wc

-l

>

Logp

其中，log中内容为

Abc

Abce

输出后logp中内容为

1

**二、job7**

1. 题目要求

使用N个线程根据莱布尼兹级数计算PI

·莱布尼兹级数公式: 1 - 1/3 + 1/5 - 1/7 + 1/9 - ... = PI/4

·主线程创建N个辅助线程

·每个辅助线程计算一部分任务，并将结果返回

·主线程等待N个辅助线程运行结束，将所有辅助线程的结果累加

·本题要求 1: 使用线程参数，消除程序中的代码重复

·本题要求 2: 不能使用全局变量存储线程返回值

1. 实现思路

Caculate\_w()是子线程调用的函数，创建线程调用时传递一个参数n，表示计算莱布尼茨级数的初始值，接着进行计算，在线程结束时返回结果

1. 关键代码

typedef void \* *result\_t*;

typedef struct *result\_m*{

    double sum;

}*result*;

*result\_t* caculate\_w(*result\_t* *argv*)

{

    int i = (int)*argv*;

    int n = (int)*argv*;

    n += PLUS;

*result* \*rst = malloc(sizeof(*result*));

    double sum = 0;

    for(i; i < n; i += 2)

    {

        double b = ((i-1)/2)%2;

        sum += pow(-1,b)/i;

    }

    rst->sum = sum;

    //printf("%f:%f\n",rst->sum,sum);

    return rst;

}

int main()

{

*pthread\_t* work\_tid[N];

    double sum = 0;

    double PI;

    for(int i = 0; i < N; i++)

    {

        int start = PLUS\*i+1;

*result\_t* argv = (void \*)start;

        pthread\_create(&(work\_tid[i]), NULL, caculate\_w,argv);

        }

    for(int i = 0; i < N; i++)

    {

*result* \*result;

        pthread\_join(work\_tid[i],(void \*\*)&result);

        sum += result->sum;

        //printf("%f:%f\n",result->sum,sum);

        free(result);

    }

    PI = 4\*sum;

    printf("PI:%f\n",PI);

}

4.运行结果

PI:3.141526

**三、job8**

1. 题目要求

使用条件变量解决生产者、计算者、消费者问题

+ 系统中有3个线程：生产者、计算者、消费者

+ 系统中有2个容量为4的缓冲区：buffer1、buffer2

+ 生产者

- 生产'a'、'b'、'c'、‘d'、'e'、'f'、'g'、'h'八个字符

- 放入到buffer1

- 打印生产的字符

+ 计算者

- 从buffer1取出字符

- 将小写字符转换为大写字符，按照 input:OUTPUT 的格式打印

- 放入到buffer2

+ 消费者

- 从buffer2取出字符

- 打印取出的字符

1. 实现思路

使用条件变量的形式，创建producer（），caculater（），consumer（）三个线程，来实现对应功能。在对共享缓冲区修改时，首先要用一个互斥量mutex对该缓冲区上锁，防止内存出错。同时，为了实现线程间的同步，当生产（即放入字符）时，要等待缓冲区有空位的条件变量cond\_empty，结束后，要激活表示缓冲区有字符的条件变量cond\_full并开锁。当消费时，要等待表示缓冲区有字符的条件变量cond\_full，结束后，要激活表示缓冲区有空位的条件变量cond\_empty，并开锁。

1. 关键代码

Producer（），consumer（）与caclulater（）中的部分代码相似，只给出caculater（）

void \*caculater(void \**arg*)

{

    int i;

    int low\_iteam;

    int upper\_iteam;

    for(i = 0; i < COUNT; i++)

    {

        pthread\_mutex\_lock(&mutex1);

        while(buffer1\_is\_empty())

            pthread\_cond\_wait(&wait\_full\_buffer1,&mutex1);

        low\_iteam = get\_iteam1();

        upper\_iteam = low\_iteam + 'A' - 'a';

        printf("%c:%c\n",low\_iteam, upper\_iteam);

        pthread\_cond\_signal(&wait\_empty\_buffer1);

        pthread\_mutex\_unlock(&mutex1);

        pthread\_mutex\_lock(&mutex2);

        while(buffer2\_is\_full())

            pthread\_cond\_wait(&wait\_empty\_buffer2,&mutex2);

        put\_iteam2(upper\_iteam);

        pthread\_cond\_signal(&wait\_full\_buffer2);

        pthread\_mutex\_unlock(&mutex2);

    }

    return NULL;

}

4.运行结果

a

a:A

A

b

b:B

B

c

c:C

C

d

d:D

D

e

e:E

E

f

f:F

F

g

g:G

G

h

h:H

H

**四、job9**

1. 题目要求

使用信号量解决生产者、计算者、消费者问题

+ 系统中有3个线程：生产者、计算者、消费者

+ 系统中有2个容量为4的缓冲区：buffer1、buffer2

+ 生产者

- 生产'a'、'b'、'c'、‘d'、'e'、'f'、'g'、'h'八个字符

- 放入到buffer1

- 打印生产的字符

+ 计算者

- 从buffer1取出字符

- 将小写字符转换为大写字符，按照 input:OUTPUT 的格式打印

- 放入到buffer2

+ 消费者

- 从buffer2取出字符

- 打印取出的字符

1. 实现思路

构建sema\_t结构体作为信号量，用信号量实现内存保护和线程同步。生产时，等待empty\_smutex实现同步，使用smutex访问内存，结束后释放（互斥量的作用）smutex，激活full\_smutex实现同步。消费时，等待full\_smutex实现同步，使用smutex访问内存，结束后释放（互斥量的作用）smutex，激活empty\_smutex实现同步。

1. 关键代码

Producer（），consumer（）与caclulater（）中的部分代码相似，只给出caculater（）

void \*caculater(void \**arg*)

{

    int i;

    int low\_iteam;

    int upper\_iteam;

    for(i = 0; i < COUNT; i++)

    {

        sema\_wait(&full\_smutex1);

        sema\_wait(&smutex1);

        low\_iteam = get\_iteam1();

        upper\_iteam = low\_iteam + 'A' - 'a';

        printf("%c:%c\n",low\_iteam, upper\_iteam);

        sema\_signal(&smutex1);

        sema\_signal(&empty\_smutex1);

        //////////////

        sema\_wait(&empty\_smutex2);

        sema\_wait(&smutex2);

        put\_iteam2(upper\_iteam);

        sema\_signal(&smutex2);

        sema\_signal(&full\_smutex2);

    }

    return NULL;

}

4.运行结果

a

b

c

a:A

b:B

c:C

A

B

C

d

e

f

d:D

e:E

f:F

D

E

F

g

h

g:G

h:H

G

H

**五、job10**

1. 题目要求

并行查找 job10/pfind.c

+功能

+在文件或者目录中查找指定的字符串，并打印包含该字符串的行要求使用多线程完成

+主线程创建若干个子线程

+主线程负责遍历目录中的文件

+遍历到目录中的叶子节点时

+将叶子节点发送给子线程进行处理

+两者之间使用生产者消费者模型通信

+主线程生成数据

+子线程读取数据

+图示

+主线程创建 2 个子线程

+主线程遍历目录 test 下的所有文件

+把遍历的叶子节点 path 和目标字符串 string，作为任务发送到任务队列

+子线程不断的从任务队列中读取任务 path 和 string

在 path 中查找字符串 string

1. 实现思路

+创建一个任务队列;初始化时，任务队列为空。

+在主进程中，调用之前的find\_dir函数，对目录 path 进行递归遍历:

+遇见叶子节点时，把叶子节点的路径加入到任务队列中

+读取结束后创建 WORER\_NUMBER 个特殊任务，特殊任务的 is\_end 为真

子线程读取到特殊任务时，表示主线程已经完成递归遍历，不会再向任务队列中放置任务，此时，子线程可以退出

+把这些特殊任务加入到任务队列中

+等待所有的子线程结束

+用信号量实现进程同步

+采用改进后的生产者消费着模型，在add\_q，get\_q中使用信号量

1. 关键代码

int add\_q(*Task* \**node*)

{

    sema\_wait(&empty\_smutex);

    sema\_wait(&smutex);

    init\_node(&tq[in], *node*->path, *node*->string, *node*->is\_end);

    in = (in+1)%MAX;

    sema\_signal(&smutex);

    sema\_signal(&full\_smutex);

    return 0;

}

int get\_q(*Task* \**node*)

{

    sema\_wait(&full\_smutex);

    sema\_wait(&smutex);

    init\_node(*node*, tq[out].path, tq[out].string, tq[out].is\_end);

    out = (out+1)%MAX;

    sema\_signal(&smutex);

    sema\_signal(&empty\_smutex);

    return 1;

}

void\* worker\_entry(void\* *arg*)

{

    while (1) {

        struct *task* task;

        //从任务队列中获取一个任务 task;

        get\_q(&task);

        if (task.is\_end)

            break;

        //执行该任务;

        //puts(task.path);

        //puts(task.string);

        find\_file(task.path, task.string);

        //free(node);

    }

}

void find\_dir(char\* *path*, char\* *target*)

{

*DIR*\* dir = opendir(*path*);

    struct *dirent*\* entry;

    while ( entry = readdir(dir) ) {

        char cats[256] = {0};

        strcpy(cats, *path*);

        strcat(cats, "/");

        strcat(cats, entry->d\_name);

        if( strcmp(entry->d\_name, ".") == 0)

            continue;

        if( strcmp(entry->d\_name, "..") == 0)

            continue;

        if( entry->d\_type == DT\_DIR) {

            find\_dir(cats, *target*);

        }

        if( entry->d\_type == DT\_REG) {

*Task* node;

            init\_node(&node, cats, *target*, 0);

            add\_q(&node);

        }

    }

    closedir(dir);

}

4.运行结果

命令：./pfind test main

结果：test/hello/hello.c: int main()

test/world/wordld.c: int main()