操作系统实践报告

011950213 顾浩嘉

job6/sh3.c

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
- ```
  echo abc | wc -l >log
  pipe
    basic
      echo
      abc
    redirect
      basic
        wc
        -l
      >
      log

  redirect
  ```
  ```

```
pipe
  basic
    echo
    abc
  basic
    wc
    -l
>
log
```

解决思路

数据结构

定义一个名为tree的结构体，tree一共有5种类型，分别为token叶子，管道树，重定向树，后台树，基本树，任何一个复杂的命令都可以用这5种树来递归表示，任何树的叶子节点都应该为token树

```
enum {
    TREE_ASYNC,     // cmd &
    TREE_PIPE,      // cmdA | cmdB
    TREE_REDIRICT,  // cmd >output <input
    TREE_BASIC,     // cmd arg1 arg2
    TREE_TOKEN,     // leaf
};

typedef struct {
    int type;
    char *token;            // TREE_TOKEN
    vector_t child_vector;  // other tree
} tree_t;
```

在tree中

- type代表该树的类型，用于判断对应的处理函数
- token指向该树对应的命令字段
- child_vector指向以该节点为根的子树

一般来说

- 后台树的子树可以为管道树、重定向树、基本树
- 管道树的子树可以为重定向树、基本树
- 重定向树的子树可以为基本树、token叶子
- 基本树的子树一般为token叶子

基本函数

在main中

main调用read_and_execute，输出提示符，调用read_line，接着调用execute_line

```
void read_and_execute()
{
    char line[128];

    write(1, "# ", 2);
    read_line(line, sizeof(line));
    execute_line(line);
}
```

read_line消除输入指令字符串末尾的\n，统一格式

```
void read_line(char *line, int size)
{
    int count;

    count = read(0, line, size);
    if (count == 0)
        exit(EXIT_SUCCESS);
    assert(count > 0);
    if ((count > 0) && (line[count - 1] == '\n'))
        line[count - 1] = 0;
    else
        line[count] = 0;
}
```

execute_line首先生成一棵指令树，接着递归地去处理这一棵树

```
void execute_line(char *line)
{
    tree_t *tree;
    lex_init(line);
    tree = parse_tree();
    if (verbose)
        tree_dump(tree, 0);
    if (tree != NULL)
        tree_execute_wrapper(tree);
    lex_destroy();
}
```

在exec.c中

入口为tree_execute_wrapper，该函数首先调用tree_execute_builtin判断是否为内置指令，并且如果是，直接进行内置指令处理，无需创建子进程

接下来对于一般情况下的复杂指令树，先创建子进程，在子进程中调用tree_execute进行递归处理

对于最外层非后台树的指令，需要等待子进程退出

```
void tree_execute_wrapper(tree_t *this)
{
    if (tree_execute_builtin(this))
        return;
    int status;
    pid_t pid = fork();
```

```
    if (pid == 0) {
        tree_execute(this);
        exit(EXIT_FAILURE);
    }
    // cc a-large-file.c &
    if (this->type != TREE_ASYNC)
        wait(&status);
}
```

tree_execute_builtin直接使用系统调用，按sh1中的方式处理内置指令，并且返回布尔值

```
int tree_execute_builtin(tree_t *this)
{
    if(this->type!=TREE_BASIC)
        return 0;
    int argc=this->child_vector.count;
    tree_t *child0=tree_get_child(this,0);
    char *arg0=child0->token;

    if(strcmp(arg0,"exit")==0){
        exit(0);
        return 1;
    }
    if(strcmp(arg0,"pwd")==0){
        char p[128];
        getcwd(p,128);
        puts(p);
        return 1;
    }
    if(strcmp(arg0,"cd")==0){
        if(argc==1)
            return 1;
        tree_t *child1=tree_get_child(this,1);
        char *arg1=child1->token;
        int err=chdir(arg1);
        if (err<0)
            perror("cd");
        return 1;
    }
    return 0;
}
```

tree_execute是一般子树的处理入口，每访问一个新的树，都根据类型从该函数出发，进入对应的处理函数

```
void tree_execute(tree_t *this)
{
    switch (this->type) {
        case TREE_ASYNC:
            tree_execute_async(this);
            break;
        case TREE_PIPE:
            tree_execute_pipe(this);
            break;
```

```
        case TREE_REDIRICT:
            tree_execute_redirect(this);
            break;
        case TREE_BASIC:
              tree_execute_basic(this);
            break;
    }
}
```

对于后台树，直接调取子树并进入处理子树，将子树作为参数进入tree_execute

```
void tree_execute_async(tree_t *this)
{
    tree_t *body =tree_get_child(this,0);
    tree_execute(body);
}
```

对于基本树，其叶子节点必然是token叶子，处理函数将其所有叶子节点的token值复制进入argv数组

一个基本树对应一条基本指令，此时所有的输入输出重定向都应该完成，直接调用execvp对argv中的指令进行处理即可

```
#define MAX_ARGC 16
void tree_execute_basic(tree_t *this)
{
    int argc=0;
    char *argv[MAX_ARGC];

    int i;
    tree_t *child;
    vector_each(&this->child_vector,i,child)
        argv[argc++]=child->token;
    argv[argc]=NULL;
    execvp(argv[0],argv);
    perror("exec");
    exit(EXIT_FAILURE);
}
```

对于管道树，其叶子节点必然是两个子指令，左孩子指令的输出作为右孩子指令的输入

处理函数创建新进程与管道，在子进程中处理左子树，在父进程中处理右子树

在处理之前，将管道的写端连接到子进程，读端连接到父进程

```
void tree_execute_pipe(tree_t *this)
{
    int fd[2];
    pid_t pid;
    tree_t *left=tree_get_child(this,0);
    tree_t *right=tree_get_child(this,1);
    pipe(fd);
    pid=fork();
    if(pid==0){
        close(1);
        dup(fd[1]);
```

```
        close(fd[1]);
        close(fd[0]);
        tree_execute(left);
        exit(EXIT_FAILURE);
    }
    close(0);
    dup(fd[0]);
    close(fd[0]);
    close(fd[1]);
    tree_execute(right);
}
```

对于重定向树，从叶子节点中提取指令，定向符号与文件地址

通过对定向符号类型的判断，以对应方式打开文件，写入重定向的重定向文件标识符设置为1，读取重定向的重定向文件标识符设置为0

将重定向文件标识符定向到刚刚打开的文件的文件标识符

最后把子树中的指令树作为参数进入tree_execute

```
void tree_execute_redirect(tree_t *this)
{
    tree_t *body, *operator,*file;
    body=tree_get_child(this,0);
    operator=tree_get_child(this,1);
    file=tree_get_child(this,2);

    char *path;
    int fd;
    int r_fd;

    path=file->token;
    if(token_is(operator,"<")){
        fd=open(path,O_RDONLY);
        r_fd=0;
    }
    if(token_is(operator,">")){
        fd=creat(path,0666);
        r_fd=1;
    }
    if(token_is(operator,">>")){
        fd=open(path,O_APPEND|O_WRONLY);
        r_fd=1;
    }
    assert(fd>0);
    dup2(fd,r_fd);
    close(fd);
    tree_execute(body);
}
```

运行结果

```
#echo abc | wc -l >log
#cat log
1
```

job7/pi2.c

题目要求

使用N个线程根据莱布尼兹级数计算PI

- 与上一题类似，但本题更加通用化，能适应N个核心
- 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

解决思路

首先定义结构体data，用于传递线程参数，告诉每个线程计算的起点与终点

```
struct data{
    int start;
    int end;
};
```

定义结构体out，用于接收返回值

```
struct out{
    float num;
};
```

线程函数compute

从data中获取任务的范围，按照公式计算出该范围的值，为out分配空间，将结果储存在out中返回

```
void *compute(void *arg){
    struct out *result;
    result=malloc(sizeof(struct out));
    struct data *data;
    data=(struct data *)arg;
    for(int i= data->start;i< data->end;i++){
        if(i%2)
        result->num -= 1.0/(i*2+1);
        else
        result->num += 1.0/(i*2+1);
    }
    return (void *)result;
}
```

在main中

首先创建N个线程，第i个线程计算i到2(i+1)-1项，按该规律初始化线程参数，将参数传递进子线程

等待各个线程结束，当一个线程结束后，从返回的result指针指向的out结构体中取出计算值，累加

最后将结果乘以4输出

```
int main()
{
    pthread_t workers[CPU];
    struct data datas[CPU];

    for (int i = 0; i < CPU; i++) {
        struct data *data;
        data = &datas[i];
        data->start = i * MISSION;
        data->end = (i + 1) * MISSION;
        pthread_create(&workers[i], NULL, compute, data);
    }

    float sum = 0;
    for (int i = 0; i < CPU; i++) {
        struct out *result;
        pthread_join(workers[i], (void **)&result);
        sum += result->num;
    }

    printf("sum = %f\n", 4*sum);
    return 0;
}
```

运行结果

10000次计算，4个线程

```
➜  job7 git:(master)./pi2
sum = 3.141493
```

job8/pc.c

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
- 程序输出结果(实际输出结果是交织的)
- ```
  a
  b
  c
  ...
      a:A
      b:B
      c:C
      ...
          A
          B
          C
          ...
  ```

解决思路

本题使用条件变量，在while中需要用buffer_is_full()和buffer_is_empty()函数来获取buffer的状态，所以为了区分full与empty，4容量的buffer实际能够使用的容量只有3

本题producer与calculator互斥读写buffer1，calculator与consumer互斥读写buffer2，所以需要4个条件变量，2个互斥量

在main中

对6个变量进行初始化，创建3个线程分别调用消费者，计算者，生产者函数，输入参数为NULL

等待3个线程运行结束

```
    pthread_t consumer_tid,producer_tid,calc_tid;

    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);
    pthread_cond_init(&wait_empty_buffer1, NULL);
```
```

```
    pthread_cond_init(&wait_full_buffer1, NULL);
    pthread_cond_init(&wait_empty_buffer2, NULL);
    pthread_cond_init(&wait_full_buffer2, NULL);

    pthread_create(&consumer_tid, NULL, consume, NULL);
    pthread_create(&producer_tid, NULL, produce, NULL);
    pthread_create(&calc_tid, NULL, calculator, NULL);
    pthread_join(consumer_tid, NULL);
    pthread_join(producer_tid, NULL);
    pthread_join(calc_tid, NULL);
    return 0;
```

在生产者函数中，首先对临界区buffer1用mutex1加锁，判断buffer1是否满，满则等待

如果临界区不满，则按顺序使用put_item()加入字符，加入后发送信号wait_full_buffer1，解锁mutex1

```
void *produce(void *arg)
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        pthread_mutex_lock(&mutex1);
        while (buffer1_is_full())
            pthread_cond_wait(&wait_empty_buffer1, &mutex1);
        item = 'a' + i;
        put_item1(item);
        printf("%c\n", item);
        pthread_cond_signal(&wait_full_buffer1);
        pthread_mutex_unlock(&mutex1);
    }
    return NULL;
}
```

在消费者函数中，首先对临界区buffer2用mutex2加锁，判断buffer2是否空，空则等待

如果临界区不空，则按顺序使用get_item()获得字符，打印字符，完成后发送信号wait_empty_buffer2，解锁mutex2

```
void *consume(void *arg)
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        pthread_mutex_lock(&mutex2);
        while (buffer2_is_empty())
            pthread_cond_wait(&wait_full_buffer2, &mutex2);

        item = get_item2();
        printf("%c\n", item);

        pthread_cond_signal(&wait_empty_buffer2);
        pthread_mutex_unlock(&mutex2);
    }
```

```
    return NULL;
}
```

计算者函数是生产者消费者函数的结合体

首先对临界区buffer1用mutex1加锁，判断buffer2是否空，空则等待

如果临界区不空，则按顺序使用get_item()获得字符，处理字符，打印字符，完成后发送信号wait_empty_buffer1，解锁mutex1

接着对临界区buffer2用mutex2加锁，判断buffer2是否满，满则等待

如果临界区不满，则按顺序使用put_item()放入字符，完成后发送信号wait_full_buffer2，解锁mutex2

```
void *calculator(void *arg)
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        pthread_mutex_lock(&mutex1);
        while (buffer1_is_empty())
            pthread_cond_wait(&wait_full_buffer1, &mutex1);

        item = get_item1();
        printf("%c:%c\n", item,item-32);

        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_item2(item-32);

        pthread_cond_signal(&wait_full_buffer2);
        pthread_mutex_unlock(&mutex2);
    }
    return NULL;
}
```

运行结果

```
➜  job8 git:(master) ./pc
a
a:A
b
b:B
c
A
B
c:C
d
e
```

```
f
d:D
C
g
e:E
f:F
h
D
E
g:G
h:H
F
G
H
```

job9/pc.c

题目要求

使用信号量解决生产者、计算者、消费者问题
功能与 job8/pc.c 相同

解决思路

本题使用信号量，与使用条件变量不同，无需函数来得知buffer的容量利用情况，空和满可以结合信号量的初值确定，所以buffer的可用容量为4

2个buffer需要2个互斥信号量，空/满状态需要4个信号量

在main中

创建线程，初始化信号量，full信号量初值为0，empty初值为容量4，互斥量初值为1

```
int main()
{
    pthread_t consumer_tid,producer_tid,calc_tid;

    sem_init(&mutex1_sema, 1);
    sem_init(&mutex2_sema, 1);

    sem_init(&empty_buffer1_sema, CAPACITY );
    sem_init(&full_buffer1_sema, 0);

    sem_init(&empty_buffer2_sema, CAPACITY );
    sem_init(&full_buffer2_sema, 0);

    pthread_create(&consumer_tid, NULL, consume, NULL);
    pthread_create(&producer_tid, NULL, produce, NULL);
    pthread_create(&calc_tid, NULL, calculator, NULL);
    pthread_join(consumer_tid, NULL);
    pthread_join(producer_tid, NULL);
    pthread_join(calc_tid, NULL);
    return 0;
```

```
}
```

在生产者中

按照P(buffer1_empty)---P(mutex1)---操作---V(mutex1)---V(buffer1_full)的顺序执行操作，将字符依序加入buffer1

```
void *produce(void *arg)
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        sem_wait(&empty_buffer1_sema);
        sem_wait(&mutex1_sema);

        item = 'a' + i;
        put_item1(item);
        printf("%c\n", item);

        sem_signal(&mutex1_sema);
        sem_signal(&full_buffer1_sema);
    }
    return NULL;
}
```

在消费者中

按照P(buffer2_full)---P(mutex2)---操作---V(mutex2)---V(buffer2_empty)的顺序执行操作，在buffer2中取出字符

```
void *consume(void *arg)
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        sem_wait(&full_buffer2_sema);
        sem_wait(&mutex2_sema);

        item = get_item2();
        printf("%c\n", item);

        sem_signal(&mutex2_sema);
        sem_signal(&empty_buffer2_sema);
    }
    return NULL;
}
```

在计算者中

按照P(buffer1_full)---P(mutex1)---操作---V(mutex1)---V(buffer1_empty)的顺序执行操作，在buffer1中取出字符，并执行字符转换和输出工作

接着按照P(buffer2_empty)---P(mutex2)---操作---V(mutex2)---V(buffer2_full)的顺序执行操作，将处理后的字符依序加入buffer2

```
void *calculator(void *arg)
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        sem_wait(&full_buffer1_sema);
        sem_wait(&mutex1_sema);

        item = get_item1();
        printf("%c:%c\n", item,item-32);

        sem_signal(&mutex1_sema);
        sem_signal(&empty_buffer1_sema);

        sem_wait(&empty_buffer2_sema);
        sem_wait(&mutex2_sema);

        put_item2(item-32);

        sem_signal(&mutex2_sema);
        sem_signal(&full_buffer2_sema);
    }
    return NULL;
}
```

运行结果

```
➜  job9 git:(master) ./pc2
a
a:A
b
b:B
c
c:C
d
d:D
e
e:E
f
g
h
A
B
C
D
f:F
g:G
h:H
E
F
```

```
G
H
```

job10/pfind

题目要求

- 功能与 sfind 相同
  - 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
- 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

解决思路

首先按照生产者消费者模型建立信号量

```
void sema_init(sema_t *sema, int value)
{
    sema->value = value;
    pthread_mutex_init(&sema->mutex, NULL);
    pthread_cond_init(&sema->cond, NULL);
}

void sema_wait(sema_t *sema)
{
    pthread_mutex_lock(&sema->mutex);
    while (sema->value <= 0)
        pthread_cond_wait(&sema->cond, &sema->mutex);
    sema->value--;
    pthread_mutex_unlock(&sema->mutex);
}

void sema_signal(sema_t *sema)
{
    pthread_mutex_lock(&sema->mutex);
    ++sema->value;
    pthread_cond_signal(&sema->cond);
    pthread_mutex_unlock(&sema->mutex);
```

```
}
```

使用预定义的结构体构建队列，作为任务的缓冲区，建立头指针和尾指针，用于数据读写

```
struct task {
    int is_end;
    char path[128];
    char string[128];
};
struct task q[CAPACITY];
int h=0,t=-1;
```

在main中

首先完成对输入参数的匹配，如果查找的已经是文件，则直接调用find_file，不再使用多线程

```
if(S_ISREG(info.st_mode))
    {
        find_file(path,string);  //deal direct file
        return 0;
    }
```

在本题中，生产者与消费者互斥读写，消费者之间互斥读，需要3个信号量，进行初始化

该处full信号量初值为0，empty信号量初值为容量，互斥信号量初值为1

```
sema_init(&mutex_s, 1);   //mutex
sema_init(&empty, CAPACITY);
sema_init(&full, 0);
```

初始化线程，主线程创建2个子线程，调用worker_entry函数，无须参数

```
pthread_create(&consumer1_tid, NULL, worker_entry, NULL);
pthread_create(&consumer2_tid, NULL, worker_entry, NULL);
```

主线程运行find_dir函数，运行结束后创建子线程个数个无效任务，等待子线程结束

```
find_dir(path,string);   //main thread

for(int i=0;i<2;i++) producespecial();

pthread_join(consumer1_tid, NULL);
pthread_join(consumer2_tid, NULL);
return 0;
```

在函数find_dir中

与sfind类似，首先打开该目录，忽略.与..目录，当查找到一项时，如果是目录，则修改path，递归调用find_dir，如果是文件，则创建有效任务，按照P(empty)----P(mutex)----操作----V(mutex)----V(full)的顺序执行生产者模型

```
if (entry->d_type == DT_REG)
        {
            sema_wait(&empty);
            sema_wait(&mutex_s);
            t++;
            t=t%CAPACITY;
            q[t].is_end=0;
            strcpy(q[t].path,bak);
            strcpy(q[t].string,target);
            sema_signal(&mutex_s);
            sema_signal(&full);
            }
        }
```

在worker_entry中

按照P(full)----P(mutex)----操作----V(mutex)----V(empty)的顺序执行消费者模型，当获取到一项任务时，判断任务是否有效，如果无效则break，有效则提取path与str，进行find_file操作

需要注意的是，如果经历break，信号量的锁没有释放，所以要在while循环外部释放信号量

```
void *worker_entry(void *arg)
{
    while (1)
    {
        sema_wait(&full);
        sema_wait(&mutex_s);

        struct task task;
        task=q[h%CAPACITY];
        h++;
        if(task.is_end)
            break;
        find_file(task.path,task.string);

        sema_signal(&mutex_s);
        sema_signal(&empty);
    }
    sema_signal(&mutex_s);     //for break
    sema_signal(&empty);
}
```

运行结果

为了测试缓存利用情况，buffer的空间被设置成极端情况的1，程序正常

```
➜  job10 git:(master) ./pfind . strtarget
./test/testfile2: file2 strtarget
./testfile1: file1 strtarget
```