操作系统实践报告

学号：161910126

姓名：赵安

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
pipe
  basic
    echo
    abc
  basic
```

```
    wc
    -l

cat main.c | grep int | wc -l
pipe
  pipe
    basic
      cat
      main.c
    basic
      grep
      int
  basic
    wc
    -l

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

redirect
  pipe
    basic
      echo
      abc
    basic
      wc
      -l
  >
  log

gcc big.c &
back
  basic
    gcc
    big.c

echo abc | wc -l &
back
  pipe
    basic
      echo
      abc
    basic
      wc
      -l

cat <input >output
redirect
  redirect
```

```
    basic
      cat
    <
    input
  >
  output
```

解决思路及关键代码

（1）解释 parse.h 中的 tree 的数据结构

```
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;

extern tree_t *tree_new(int type);
extern tree_t *tree_get_child(tree_t *this, int index);
extern void tree_append_child(tree_t *this, tree_t *child);
extern tree_t *parse_tree();
extern void tree_dump(tree_t *this, int level);
```

enum中定义了5种类型的tree：ASYNC、PIPE、REDIRICT、BASIC、TOKEN。树的子节点都应该为token树。

对于tree_t：

- type代表树的类型，与上述enum中的类型对应
- token指向该树对应的命令字段
- child_vector指向其子树

（2）解释 main.c

```
// 生产一棵指令树，通过递归调用的方式对这棵树进行参数读取分析
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();
}
// 计算读取到的指令字符串长度，进行错误处理和空指令处理
```

```
// 消除指令末尾的 \n 字符
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;
}
// 调用read_line函数为execute_line提供参数
// 输出提示符
void read_and_execute()
{
    char line[128];

    write(1, "# ", 2);
    read_line(line, sizeof(line));
    execute_line(line);
}
// 指令测试
void test()
{
    execute_line("cat /etc/passwd | sort | grep root >log");
}
// 对命令行参数进行处理
int main(int argc, char *argv[])
{
    if (argc == 2 && strcmp(argv[1], "-v") == 0)
        verbose = 1;
    while (1)
        read_and_execute();
    return 0;
}
```

(3) 解释exec.c 中的关键函数

- 首先调用tree_execute_wrapper函数，若为内置指令则直接进行内置指令处理
- 若不是内置指令，则创建子进程，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);
}
```

ree_execute_builtin直接使用系统调用，处理exit、pwd、cd基本指令，用于判断传入的树的token是否为内置命令。

```
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"))
    {
        exit(0);
        return 1;
    }

    if(!strcmp(arg0,"pwd"))
    {
        char p[256];
        getcwd(p,256);
        puts(p);
        return 1;
    }

    if(!strcmp(arg0,"cd"))
    {
        if(argc==1)
            return 1;
        tree_t *child1=tree_get_child(this,1);
        char *arg1=child1->token;
        int err=chdir(arg1);
        return 1;
    }
    return 0;
}
```

根据树的类型不同执行不同的树处理函数

```
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;
    }
}
```

对于重定向树，首先通过tree_get_child函数提取出指令部分、定向符号、文件地址

根据不同的定向符号执行不同的操作，对文件进行写入和读取的重定向

将子树作为参数调用tree_execute执行

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

    char *path = file->token;
    int fd1, fd2;
    if(token_is(op,"<"))
    {
        fd1 = open(path,O_RDONLY);
        fd2 = 0;
    }
    if(token_is(op,">"))
    {
        fd1 =creat(path,0666);
        fd2=1;
    }
    if(token_is(op,">>"))
    {
        fd1 = open(path,O_APPEND|O_WRONLY);
        fd2 =1 ;
    }
    assert(fd1 > 0);
    dup2(fd1,fd2);
    close(fd1);
    tree_execute(body);
}
```

对于基本树，将其叶子节点的token复制进argv数组，直接调用execvp执行命令。

```
#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);
    exit(EXIT_FAILURE);
}
```

对于管道树，其叶子节点是两个子指令，左孩子指令的输出作为右孩子指令的输入

创建子进程与管道，在子进程中处理左子树，在父进程中处理右子树

```
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async树，直接将其子树作为参数传入tree_execute

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

运行结果

```
(base) ekko@ubuntu:~$ echo abc | wc -l >log
(base) ekko@ubuntu:~$ 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

创建两个结构体用于线程运行的变量传递

```
struct param
{
    int start;
    int end;
};
struct result
{
    double sum;
};
```

将所需计算的项数平均分成n份，创建n个线程，将起始和结束参数通过线程传参的方式传递给各个线程执行的函数pi2()，pi2()函数最终返回每个线程计算得到的值，调用pthread_join接受各个线程的结果返回值，最后在main函数中将所有线程的执行结果相加进行计算求出pi

关键代码

```
    pthread_t workers[p_n];
    struct param params[p_n];
    int i;
    double PI = 0;

    for(i =0 ;i<p_n;i++)
    {
        struct param *param = &params[i];
        param->start = i * 100000 + 1;
        param->end = (i + 1) * 100000 ;
        pthread_create(&workers[i],NULL,pi2,param);
    }
```

```
    for(i =0 ; i <p_n; i++)
    {
        struct result *result;
        pthread_join(workers[i],(void **)&result);
        PI += result->sum;
        free(result);
    }
```

运行结果

```
ekko@ubuntu:~/Documents/OS/job7$ ./pi2
3.141592
```

---

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

首先创建两个buffer存放数据，两个in、out用于这两个buffer的读写控制，之后完善这个两个缓冲区的put、get函数。然后创建两个互斥变量mutex1、mutex2和四个条件变量wait_empty_buffer1、wait_full_buffer1、wait_empty_buffer2、wait_full_buffer2用于控制读写。

（1）生产者调用put1函数向buffer1中存放数据

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

（2）计算者调用get1函数和put2函数从buffer1中读取数据并进行计算之后将其存放至buffer2

首先对临界区buffer1用mutex1加锁，判断buffer2是否空，空则等待如果buffer不空，则使用get_item()读取数据并进行计算，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buffer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

（3）消费者从buffer2中取出数据

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。

在生产者、计算者、消费者三个函数中调用原子操作处理条件变量，实现读写互斥的操作。最后在main函数中将信号量和线程进程初始化和执行操作。

关键代码

```
// 消费者
void *consume(void *arg)
{
    int item;

    for (int i = 0; i < ITEM_COUNT; i++)
    {
        pthread_mutex_lock(&mutex2);
        while (buffer_is_empty2())
            pthread_cond_wait(&wait_full_buffer2, &mutex2);

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

        pthread_cond_signal(&wait_empty_buffer2);
        pthread_mutex_unlock(&mutex2);
    }
}
// 计算者
void *compute(void *arg)
{
    int item;

    for (int i = 0; i < ITEM_COUNT; i++)
    {
        pthread_mutex_lock(&mutex1);
        while (buffer_is_empty1())
            pthread_cond_wait(&wait_full_buffer1, &mutex1);

        item = get_item1();
        printf("  compute: %c",item);
```

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

        pthread_cond_signal(&wait_empty_buffer1);
        pthread_mutex_unlock(&mutex1);


        pthread_mutex_lock(&mutex2);
        while (buffer_is_full2())
            pthread_cond_wait(&wait_empty_buffer2, &mutex2);


        put_item2(item);

        pthread_cond_signal(&wait_full_buffer2);
        pthread_mutex_unlock(&mutex2);
    }
}
// 生产者
void *produce(void *arg)
{
    int item;

    for (int i = 0; i < ITEM_COUNT; i++)
    {
        pthread_mutex_lock(&mutex1);

        while (buffer_is_full1())
            pthread_cond_wait(&wait_empty_buffer1, &mutex1);

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

        pthread_cond_signal(&wait_full_buffer1);
        pthread_mutex_unlock(&mutex1);
    }
}
```

运行结果

```
ekko@ubuntu:~/Documents/OS/job8$ ./pc
produce: a
produce: b
produce: c
  compute: a:A
  compute: b:B
  compute: c:C
produce: d
produce: e
produce: f
    consume: A
    consume: B
    consume: C
  compute: d:D
  compute: e:E
```

```
  compute: f:F
    consume: D
    consume: E
    consume: F
produce: g
produce: h
  compute: g:G
  compute: h:H
    consume: G
    consume: 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

缓冲区的初始化及put、get操作与上一题类似，在此不多赘述。

首先完成对信号量结构体的创建与相关初始化、PV操作的实现：

```c
typedef struct {
    int value;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} sema_t;
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);
}
```

生产者：

若buffer1不为空，通过P操作将其锁住，等待其为空后向buffer1中写入数据，写完后调用V操作使得buffer1的full信号为1，使得计算者进程能够工作。

计算者：

若buffer1不为满，则通过P操作将其锁住，等待生产者进程完成。当生产者进程执行完毕后，向buffer1中读取数据并进行计算操作。然后通过V操作使得buffer1的empty信号为1，信号释放，重启生产者进程。

若buffer2不为空，则通过P操作将其锁住，等待消费者进程完成。当消费者进程执行完毕后，向buffer2中存入数据。然后通过V操作使得buffer2的empty信号为1，信号释放，重启消费者进程。

消费者：

若buffer2不为满，通过P操作将其锁住，等待其为满后将数据取出并打印出来，写完后调用V操作使得buffer2的full信号为1，使得计算者进程能够工作。

关键代码

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

    for (i = 0; i < ITEM_COUNT; i++)
    {
        sema_wait(&full2);
        sema_wait(&mutex_sema);

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

        sema_signal(&mutex_sema);
        sema_signal(&empty2);
    }
}
// 生产者
void *produce()
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        sema_wait(&empty1);
        sema_wait(&mutex_sema);

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

        sema_signal(&mutex_sema);
        sema_signal(&full1);
    }
}
// 计算者
void *compute(void * arg)
{
    for (int i = 0;i< ITEM_COUNT ; i++)
    {
        sema_wait(&full1);
```

```
        sema_wait(&empty2);
        sema_wait(&mutex_sema);

        int item = get_item1();
        printf("  compute: %c:",item);
        item = item - 32;
        printf("%c\n",item);
        put_item2(item);


        sema_signal(&mutex_sema);
        sema_signal(&full2);
        sema_signal(&empty1);
    }
}
```

运行结果

```
ekko@ubuntu:~/Documents/OS/job9$ ./pc
produce: a
produce: b
produce: c
produce: d
  compute: a:A
  compute: b:B
  compute: c:C
  compute: d:D
    consume: A
    consume: B
produce: e
produce: f
produce: g
produce: h
    consume: C
    consume: D
  compute: e:E
  compute: f:F
  compute: g:G
  compute: h:H
    consume: E
    consume: F
    consume: G
    consume: H
```

---

job10/pfind.c

题目要求

并行查找

- 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

首先完成对信号量结构体的创建与相关初始化、PV操作的实现，实现方式同上题，此处不再赘述。

然后创建任务结构体并初始化

```
struct task
{
    int is_end;
    char path[128];
    char string[128];
}mytask[CAPACITY];
```

定义两个指针指向任务的头部和尾部

```
int head = 0;
int tail = 0;
```

对于主线程：

若找到的是文件，且任务列表不为空，则调用P操作将任务队列锁住，等待子线程将任务执行完后，将查找到的当前文件写入任务列表，同时尾指针加一，然后调用V操作将full这个信号量加一，解锁子线程。

若找到的是文件目录，则进行递归查找。

上述执行完毕后，创建 WORER_NUMBER 个特殊任务， 把这些特殊任务加入到任务队列中，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。

对子线程：

若任务队列不为满，则等待父线程执行完毕。然后从任务列表中读取任务进行匹配字符串的操作，操作完毕后，调用V操作将信号量empty加一，解锁父线程。

关键代码

```
// 子线程执行
void *worker_entry(void *arg)
{
    while (1)
    {
        sema_wait(&full);
        sema_wait(&mutex);

        struct task task;
        task = mytask[head%CAPACITY];
        // printf("head:%d\n",head);
        head++;
        if ( task.is_end )
        {
            sema_signal(&mutex);
            sema_signal(&empty);
            break;
        }

        find_file(task.path, task.string);

        sema_signal(&mutex);
        sema_signal(&empty);
    }
}

// 主线程执行
void find_dir(char *path,char *target)
{
    char newpath[257];
    DIR *dir = opendir(path);
    // printf("100\n");

    if(dir == NULL)
        return;
    struct dirent *entry;

    // printf("105\n");
    while (entry = readdir(dir))
    {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
            continue;

        sprintf(newpath , "%s/%s",path,entry->d_name);
        if (entry->d_type == DT_DIR)
            find_dir(newpath,target);


        if (entry->d_type == DT_REG)
        {
            sema_wait(&empty);
            sema_wait(&mutex);
            // printf("newpath:%s\n",newpath);

            mytask[tail%CAPACITY].is_end = 0;
```

```
            strcpy(mytask[tail%CAPACITY].path,newpath);
            strcpy(mytask[tail%CAPACITY].string,target);
            // printf("tail:%d\n",tail);
            tail++;

            sema_signal(&mutex);
            sema_signal(&full);

        }
    }
    closedir(dir);
}
// main 函数
int main(int argc, char *argv[])
{
    char *path = argv[1];
    char *string = argv[2];

    struct stat info;
    stat(path, &info);


    if (!S_ISDIR(info.st_mode))
    {
        find_file(path, string);
        return 0;
    }

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

    pthread_t works_tid[WORKER_NUMBER];
    for(int i = 0; i < WORKER_NUMBER; i++)
        pthread_create(&works_tid[i],NULL,worker_entry,NULL);

    find_dir(path,string);

    for (int i = 0; i < WORKER_NUMBER ;i++)
    {
        sema_wait(&empty);
        sema_wait(&mutex);
        mytask[tail%CAPACITY].is_end = 1;
        tail++;
        sema_signal(&mutex);
        sema_signal(&full);
    }

    for(int i = 0; i < WORKER_NUMBER; i++)
        pthread_join(works_tid[i],NULL);
    return 0;
}
```

运行结果

```
ekko@ubuntu:~/Documents/OS/job10$ ./pfind test k
test/a.c: kk
test/a.c: kkas
test/b/x.c: bxk
test/b/z.c: bzk
test/b/y.c: byk
test/c.c: ck
test/c.c: cck
test/c.c: cccck
test/c.c: cccccck
```