南京航空航天大学 操作系统实践 实验报告

- 学院：计算机科学与技术/人工智能学院
- 学号：
- 姓名：MOYI HP

- 完成时间：2022/6/8
- 指导老师：

目录

说 明

```
1. 解析直接书写在代码相关处
2. 语句解析 格式为 /* 解析 */ 直接书写在语句后
3. 函数解析 格式如下 书写在函数前
/*
FUNC : <函数名>
     : <函数功能>
PARAM :
    @<参数名> : 参数解析
    ...
RETURN :返回值解析
*/
```

1. Job6 / sh3.c

1.1 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

1.2 解决思路

1. 为了实现 `基本命令`、`重定向命令`、`后台命令`、`管道命令`，原有的 描述命令的数据结构 已经不适合复杂的指令类型了，需要引入 `树形命令结构` 来描述命令
2. `命令/命令结点的类型` 分为
    - 基本 BASIC
    - 后台 BACK / ASYNC
    - 重定向 REDIRECT
    - 管道 PIPE
    - 叶结点 TOKEN
3. 结点的数据结构包括
    - 结点类型
    - TOKEN字符串
    - 子结点数组
4. 最后拆解一条复杂指令为现有四种指令的组合，使用 `递归方式` 基于四种指令实现复杂指令

1.3 关键代码

1.3.1 TREE数据结构

```
/* 命令类型 */
enum {
    TREE_ASYNC,     /* 异步模式 即 BACKEND 后台命令 */
    TREE_PIPE,      /* PIPE 管道命令 */
    TREE_REDIRICT,  /* REDIRICT 重定向命令 */
    TREE_BASIC,     /* BASIC 基本命令 */
    TREE_TOKEN,     /* TOKEN 非命令,表示该结点为叶子结点,描述具体命令或者参数,而非结构 */
};
```

```
typedef struct {
    int type;               /* 结点类型 */
    char *token;            /* 仅当类型为TREE_TOKEN时有效 token */
    vector_t child_vector;  /* 当类型为非TREE_TOKEN时有效 子结点数组 */
} tree_t;
```

1.3.2 main.c

```
/*
FUNC : read_and_execute
     : 读取一行指令 并 执行
 */
void read_and_execute()
{
    char line[128];

    write(1, getcwd(NULL, 128), strlen(getcwd(NULL, 128))); /* 打印提示符# */
    write(1, " - sh3 # ", 9);

    read_line(line, sizeof(line));                          /* 读取一行 */
    execute_line(line);                                     /* 执行一行 */
}
```

1.3.3 exec.c

```
/*
FUNC : tree_execute_redirect
     : 执行一条 重定向 指令, 重定向 输入/输出
*/
void tree_execute_redirect(tree_t *this)
{
    tree_t* body = tree_get_child(this, 0);                 /* body cmd */
    tree_t* operator = tree_get_child(this, 1);             /* op > | < */
    tree_t* file = tree_get_child(this, 2);                 /* red file */

    // ...

    path = file->token;
    if (token_is(operator,"<"))             /* 重定向输入,新开文件FD,使其为输入 */
        // ...
    else if (token_is(operator, ">"))       /* 重定向输出,新创文件FD,使其为输出 */
        // ...

    tree_execute(body);
}

#define MAX_ARGC 16
/*
FUNC : tree_execute_basic
     : 执行一条基本命令, 提取 ARGC ARGV[]
*/
void tree_execute_basic(tree_t *this)
{
   // ...
}
```

```c
/*
FUNC : tree_execute_pipe
     : 实现管道命令，将 左out -> 右in
other : 实现步骤为
    f0 - pipe -> f1
    out - pipe -> in
*/
void tree_execute_pipe(tree_t *this)
{
    int fd[2];
    pid_t pid = 0;
    tree_t *l = tree_get_child(this, 0);                /* cmd_l */
    tree_t* r = tree_get_child(this, 1);                /* cmd_r */

    pipe(fd);                                           /* f1 -> f0 */
    pid = fork();
    if (pid == 0)
    {
        close(1);                                       /* 1(std_out) close */
        dup(fd[1]);                                     /* out -> f0 */
        close(fd[0]);
        close(fd[1]);
        tree_execute(l);
        exit(EXIT_FAILURE);
    }

    close(0);                                           /* 0(std_int) close */
    dup(fd[0]);                                         /* f1 -> in */
    close(fd[0]);
    close(fd[1]);
    tree_execute(r);
}

/*
FUNC : tree_execute_builtin
     : 判断一条命令(树形结构)是否为 内置 命令，如果是执行它
*/
int tree_execute_builtin(tree_t *this)
{
   //  ...
}

/*
FUNC : tree_execute_async
     : 执行一条 异步/后台 命令
other :  ASYNC 命令 = 命令 + ASYNC修饰，后台部分由 主函数 决定是否等待完成，此处只需要执行命令 即可
*/
void tree_execute_async(tree_t *this)
{
    tree_t* body = tree_get_child(this, 0);
    tree_execute(body);
}

/*
FUNC : tree_execute
     :  根据 命令类型 ，  执行一条 拓展命令
```

```
*/
void tree_execute(tree_t *this)
{
    // ...
}

/*
FUNC : tree_execute_wrapper
     : 区分 内置 与 拓展 命令, 对于内置命令,调用函数直接执行,对于拓展命令,调用子进程执行
PARAM :
    @tree_t *this : 将要执行的指令(树形结构)
*/
void tree_execute_wrapper(tree_t *this)
{
    if (tree_execute_builtin(this))              /* 如果为 内置指令,直接执行 */
        return;

    int status;                                  /* 如果为 拓展指令,调用子进程执行 */
    pid_t pid = fork();
    if (pid == 0) {
        tree_execute(this);
        exit(EXIT_FAILURE);
    }

    // cc a-large-file.c &
    if (this->type != TREE_ASYNC)                /* 如果为 非后台命令 , 等待子进程执行
完毕 */
    {
        wait(&status);
    }
}
```

1.4 运行结果

```
./sh -v
sh3 # echo abc | wc -l
----- CMD TREE ----------------------------
- PIPE
  - BASIC
    echo
    abc
  - BASIC
    wc
    -l
-------------------------------------------
1

sh3 # cat main.c | grep int | wc -l
----- CMD TREE ----------------------------
- PIPE
  - BASIC
    cat
    main.c
  - PIPE
    - BASIC
      grep
      int
```

```
    - BASIC
      wc
      -l
----------------------------------------
6

sh3 # echo abc | wc -l >log
----- CMD TREE ---------------------------
- PIPE
  - BASIC
    echo
    abc
  - REDIRECT
    - BASIC
      wc
      -l
    >
    log
----------------------------------------

sh3 # cat log
----- CMD TREE ---------------------------
- BASIC
  cat
  log
----------------------------------------
1

sh3 # echo abc | wc -l &
----- CMD TREE ---------------------------
- ASYNC
  - PIPE
    - BASIC
      echo
      abc
    - BASIC
      wc
      -l
----------------------------------------
```

2. Job7 / pi2.c

2.1 题目要求

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

2.2 解决思路

- 假定总计算项数为`NR_TOTAL`，总CPU数目/子线程数目为`NR_CPU`,那么有每个子线程的任务量为`NR_CHILD = (NR_TOTAL / NR_CPU)`
- 为每个子线程函数（编号为 I , I 属于[1 , NR_CPU]）创建参数（其任务范围）为 [(i-1)*NR_CHILD +1 , i * NR_CHILD]
- 主线程循环创建NR_CPU个子线程，而后逐一等待其返回值并汇总，最终计算得出结果PI

2.3 关键代码

2.3.1 数据结构及常量定义

```
#define NR_TOTAL 50000                          /* 总计算项数 */
#define NR_CPU  2                               /* 总CPU数目 */
#define NR_CHILD (NR_TOTAL / NR_CPU)            /* 每个子进程的任务量 */
typedef struct Param            /* 每个线程的计算范围[start,end] */
{
    int start;
    int end;
} Param ;

typedef struct Result           /* 每个线程的返回值类型 */
{
    float val;
} Result ;
```

2.3.2 线程

```
    // STEP1 创建子线程 分配任务
    pthread_t workers[NR_CPU + 1];
                /* 根据NR_CPU创建对应数量的线程 i in 1..NR_CPU */
    Param params[NR_CPU + 1];
                /* 每个线程对应一个参数,规定计算范围 [(i-1)*NR_CHILD +1 , i *
NR_CHILD] */
    int i = 1;
    for (i = 1; i <= NR_CPU; i++)
    {
        Param* param = &params[i];
        param->start = (i - 1) * NR_CHILD + 1;
        param->end = i * NR_CHILD;
        pthread_create(&workers[i], NULL, calculate, param);
    }

    // STEP2 等待子线程执行 获取数据
    float total = 0.0;
    for (i = 1; i <= NR_CPU; i++)
    {
        Result* result = NULL;
        pthread_join(workers[i], (void**)&result);
        total += result->val;
        free(result);
    }

    // STEP3 汇总计算结果
    float PI = 4 * total;
```

```
    printf("PI = %.6f\n", PI);
```

2.4 运行结果

```
PI = 3.141577
```

---

3. Job8 / pc.c

3.1 题目要求

- 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

3.2 解决思路

- 首先由于有两个Buffer，所以需要重新定义 Buffer 数据结构
  - 成员变量包括： buf[CAPACITY],in,out
  - 操作包括：初始化、是否为空/满、读写数据
- 由于有三个参与者，两个Buffer
  - 需要 2 个互斥量
  - 需要 2 个满条件变量，2个空条件变量
- 生产者操作流程为
  - Lock Mutex1 -> Produce -> UnLock Mutex1
- 计算者操作流程为
  - Lock Mutex1 -> Calculate -> UnLock Mutex1 -> Lock Mutex2 -> Put res -> UnLock Mutex2
- 消费者操作流程为
  - Lock Mutex2 -> Consume -> UnLock Mutex2

3.3 关键代码

3.3.1 Buffer 数据结构

```
/* Buffer 数据结构定义 */
// 数据结构
#define CAPACITY 4
typedef struct Buffer
```

```
{
    int buf[CAPACITY];
    int in;
    int out;
} Buffer ;
// 1. 初始化
void buffer_init(Buffer* buffer);
// 2. 是否为空
int buffer_is_empty(Buffer* buffer);
// 3. 是否为满
int buffer_is_full(Buffer* buffer);
// 4. 读数据
int get_item(Buffer* buffer);
// 5. 写数据
void put_item(Buffer* buffer, int item);
```

3.3.2 生产者/计算者/消费者 线程函数

```
// 生产者线程函数 生产数据到 buffer1
void* produce(void* arg)
{
    int i,  item;
    for (i = 0; i < ITEM_COUNT; i++) {
        pthread_mutex_lock(&mutex1);                    /* LOCK MUTEX1 */
        while (buffer_is_full(&buffer1))                /* 等待 buffer1 不满 */
            pthread_cond_wait(&wait_empty_buffer1, &mutex1);

        item = 'a' + i;                                 /* 生产 */
        put_item(&buffer1, item);
        printf("produce item: %c\n", item);

        pthread_cond_signal(&wait_full_buffer1);        /* SIGNAL buffer1 满 */
        pthread_mutex_unlock(&mutex1);                  /* UNLOCK MUTEX1 */
    }
    return NULL;
}
// 计算者线程函数 从 buffer1 取数据 计算后 放到 buffer2
void* calculate(void* arg)
{
    int i, item;

    for (i = 0; i < ITEM_COUNT; i++) {
        // 取数据部分
        pthread_mutex_lock(&mutex1);                    /* LOCK MUTEX1 */
        while (buffer_is_empty(&buffer1))               /* 等待 buffer1 不空 */
            pthread_cond_wait(&wait_full_buffer1, &mutex1);
        item = get_item(&buffer1);                      /* 获取数据 */
        printf("    calculate get item: %c\n", item);
        pthread_cond_signal(&wait_empty_buffer1);       /* SIGNAL buffer1 空 */
        pthread_mutex_unlock(&mutex1);                  /* UNLOCK MUTEX1 */

        // 计算部分
        item = item + 'A' - 'a';

        // 放数据部分
        pthread_mutex_lock(&mutex2);                    /* LOCK MUTEX2 */
        while (buffer_is_full(&buffer2))                /* 等待buffer2不满 */
```

```
            pthread_cond_wait(&wait_empty_buffer2, &mutex2);
        put_item(&buffer2, item);                           /* 放数据 */
        printf("    calculate put item: %c\n", item);
        pthread_cond_signal(&wait_full_buffer2);            /* SIGNAL buffer2 有
数据 */
        pthread_mutex_unlock(&mutex2);                      /* UNLOCK MUTEX2 */
    }
    return NULL;
}

// 消费者线程函数
void* consume(void* arg)
{
    int i;
    int item;

    for (i = 0; i < ITEM_COUNT; i++) {
        pthread_mutex_lock(&mutex2);
        while (buffer_is_empty(&buffer2))
            pthread_cond_wait(&wait_full_buffer2, &mutex2);

        item = get_item(&buffer2);
        printf("        consume item: %c\n", item);

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

3.4 运行结果

```
produce item: a
produce item: b
produce item: c
    calculate get item: a
    calculate put item: A
        consume item: A
produce item: d
    calculate get item: b
    calculate put item: B
produce item: e
    calculate get item: c
        consume item: B
produce item: f
    calculate put item: C
    calculate get item: d
produce item: g
        consume item: C
    calculate put item: D
    calculate get item: e
        consume item: D
produce item: h
    calculate put item: E
    calculate get item: f
        consume item: E
```

```
    calculate put item: F
    calculate get item: g
    calculate put item: G
    calculate get item: h
    calculate put item: H
        consume item: F
        consume item: G
        consume item: H
```

---

4. Job9 / pc.c

4.1 题目要求

- 功能与 job8/pc.c 相同
- 使用信号量解决生产者、计算者、消费者问题

4.2 解决思路

核心在于对信号量的各个功能的实现

- `信号量sema` = `资源数量val` + `互斥量mutex` + `条件变量cond`
- `信号量sema操作` = `初始化init` + `等待wait` + `唤醒signal`
  - `初始化init` = `设定资源数量` *val = 1 时，sema 相当于一个 mutex* + `初始化 mutex cond`
  - `等待wait` = `查看是否有资源可以分配 val >= 1 ?` + `没有资源则等待条件` + `有则分配`
  - `唤醒signal` = `资源使用完毕归还系统 val++` + `唤醒在等待该资源的线程`

4.3 关键代码

4.3.1 SEMA 数据结构

```c
/* 信号量数据结构定义 */
// 数据结构
typedef struct Sema
{
    int value;                  /* 资源数量 */
    pthread_mutex_t mutex;      /* 互斥量 */
    pthread_cond_t cond;        /* 条件变量 */
} Sema ;
// 1. 初始化
void sema_init(Sema* sema, int value)
{
    sema->value = value;
    pthread_mutex_init(&sema->mutex, NULL);
    pthread_cond_init(&sema->cond, NULL);
}
// 2. 等待
void sema_wait(Sema* sema)
{
    pthread_mutex_lock(&sema->mutex);
    while ( sema->value <= 0)
    {
        pthread_cond_wait(&sema->cond, &sema->mutex);
    }
    sema->value--;
```

```
    pthread_mutex_unlock(&sema->mutex);
}
// 3. 唤醒
void sema_signal(Sema* sema)
{
    pthread_mutex_lock(&sema->mutex);
    sema->value++;
    pthread_cond_signal(&sema->cond);
    pthread_mutex_unlock(&sema->mutex);
}
```

4.3.2 SEMA 使用(例子：计算者)

```
// 计算者线程函数
// 从 buffer1 取数据 计算后 放到 buffer2
void* calculate(void* arg)
{
    int i, item;

    for (i = 0; i < ITEM_COUNT; i++) {
        // 取数据部分
        // 等待
        sema_wait(&full_buffer1_sema);              /* 等待buffer1中的一个满项 */
        sema_wait(&mutex_buffer1_sema);             /* LOCK buffer1 */
        // 取出数据
        item = get_item(&buffer1);
        printf("    calculate get item: %c\n", item);
        // 释放
        sema_signal(&mutex_buffer1_sema);           /* UNLOCK buffer1 */
        sema_signal(&empty_buffer1_sema);           /* 唤醒一个等待buffer1空项的线程
*/

        // 计算部分
        item = item + 'A' - 'a';

        // 放数据部分
        // 等待
        sema_wait(&empty_buffer2_sema);             /* 等待buffer2中的一个空项 */
        sema_wait(&mutex_buffer2_sema);             /* LOCK buffer2 */
        // 放入数据
        put_item(&buffer2, item);
        printf("    calculate put item: %c\n", item);
        // 释放
        sema_signal(&mutex_buffer2_sema);           /* UNLOCK buffer2 */
        sema_signal(&full_buffer2_sema);            /* 唤醒一个等待buffer2满项的线程
*/
    }
    return NULL;
}
```

4.4 运行结果

```
produce item: a
produce item: b
produce item: c
    calculate get item: a
```

```
    calculate put item: A
produce item: d
        consume item: A
    calculate get item: b
    calculate put item: B
        consume item: B
produce item: e
    calculate get item: c
    calculate put item: C
produce item: f
    calculate get item: d
        consume item: C
produce item: g
    calculate put item: D
    calculate get item: e
        consume item: D
produce item: h
    calculate put item: E
    calculate get item: f
        consume item: E
    calculate put item: F
    calculate get item: g
        consume item: F
    calculate put item: G
    calculate get item: h
        consume item: G
    calculate put item: H
        consume item: H
```

---

5. Job10 / pfind.c

5.1 题目要求

1. 功能要求

- 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
    - 在文件中查找字符串 打印文件名和该行
    - 对目录下的所有文件进行查找 找到包含字符串 main 的行打印文件名和该行

2. 实现要求

- 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

5.2 解决思路

- 判断`目标路径`为`普通文件`或者`目录`
  文件具有`stat`信息,stat中st_mode记录了该文件的类型(DIR/REG)
- 递归调用`find_dir/find_file`寻找目标字符串
- 任务队列中记录了`当前已经发现的任务`和`线程结束任务`,主线程调用`find_dir`作为生产者，向队列中`push`任务;子线程作为消费者，从队列中取出任务并执行

5.3 关键代码

5.3.1 任务队列数据结构

```
/* 任务与任务队列 */
/* 任务 */
typedef struct Task
{
    int is_end;                     /* 结束任务, ==1 子线程结束 */
    char path[128];                 /* 查找路径 */
    char string[128];               /* 目标字符串 */
} Task ;
/* 任务队列 */
#define QUEUE_SIZE 8                /* 最大任务量 8 - 1 = 7 */
typedef struct TaskQueue
{
    Task tasks[QUEUE_SIZE];
    int rear;
    int front;
} TaskQueue;
// 1. 初始化
void taskQueue_init(TaskQueue* tq);
// 2. 是否为空
int taskQueue_isEmpty(TaskQueue* tq);
// 3. 是否为满
int taskQueue_isFull(TaskQueue* tq);
// 4. 取出队列头
void taskQueue_pop(TaskQueue* tq, Task* task);
// 5. 向队列尾写入数据
void taskQueue_push(TaskQueue* tq, Task* task);
// 6. 显示队列内数据
void taskQueue_dump(TaskQueue* tq);
```

5.3.2 主线程 整体框架

```
if (S_ISDIR(info.st_mode))          /* 如果起始点文件为DIR,启用多线程处理 */
    {
        // 1. 创建一个任务队列;
        taskQueue_init(&tq);
        pthread_mutex_init(&mutex_tq, NULL);
        pthread_cond_init(&cond_empty_tq, NULL);
        pthread_cond_init(&cond_valid_tq, NULL);
        // 2. 创建 NR_WORKER 个子线程;
        pthread_t worker_tid[NR_WORKER];
        int i = 0;
        for (i = 0; i <= NR_WORKER - 1; i++)
        {
            pthread_create(&worker_tid[i], NULL, worker_entry, NULL);
        }
        // 3. 主线程工作 对目录 path 进行递归遍历 把叶子节点的路径加入到任务队列中
        find_dir(path, string);
        // 4. 创建 WORER_NUMBER 个特殊任务
        Task endtask;
        endtask.is_end = 1;
        endtask.path[0] = '\0';
        endtask.string[0] = '\0';
```

```
        for (i = 0; i <= NR_WORKER - 1; i++)
        {
            pthread_mutex_lock(&mutex_tq);
            while (taskQueue_isFull(&tq))
                pthread_cond_wait(&cond_empty_tq, &mutex_tq);

            taskQueue_push(&tq, &endtask);

            pthread_cond_signal(&cond_valid_tq);
            pthread_mutex_unlock(&mutex_tq);
        }
        // 5. 等待所有的子线程结束;
        for (i = 0; i <= NR_WORKER - 1; i++)
        {
            pthread_join(worker_tid[i], NULL);
        }
    }
```

5.3.3 主线程 调用FIND_DIR生产任务

```
if (entry->d_type == DT_REG)            /* 如果是文件 创建任务 */
        {
            char tmp_path[128];
            strcpy(tmp_path, path);

            Task task;
            task.is_end = 0;
            strcpy(task.path, strcat(strcat(path, "/"), entry->d_name));
            strcpy(task.string, target);

            pthread_mutex_lock(&mutex_tq);
            while (taskQueue_isFull(&tq))
                pthread_cond_wait(&cond_empty_tq, &mutex_tq);

            taskQueue_push(&tq, &task);

            pthread_cond_signal(&cond_valid_tq);
            pthread_mutex_unlock(&mutex_tq);

            strcpy(path, tmp_path);
        }
```

5.3.4 子线程 获取任务并执行

```
Task task;
    while (1)
    {
        // 从任务队列中获取一个任务 task;
        pthread_mutex_lock(&mutex_tq);              /* LOCK mutex_tq */
        while (taskQueue_isEmpty(&tq))
        {
            pthread_cond_wait(&cond_valid_tq, &mutex_tq);
        }
        taskQueue_pop(&tq, &task);
        pthread_cond_signal(&cond_empty_tq);
        pthread_mutex_unlock(&mutex_tq);
```

```
        // 执行该任务 task
        if (task.is_end)
            break;
        else
        {
            find_file(task.path, task.string);
        }
    }
```

5.4 运行结果

```
$ ./sfind test main
test/hello/hello.c: int main()
test/world/world.c: int main()
```