操作系统实践报告

- 姓名: 高远
- 学号: 161740229

目录

myecho.c

在`main`函数的定义中有两个参数`argc`, `argv`分别在程序运行时存储命令行的参数个数和参数字符串

```
int main(int argc, int *argv[]){}
```

比如shell中输入`./myecho a b c`时,

`argc  = 4`

`argv[0] = "./myecho"`

`argv[1] = "a"`

`argv[2] = "b"`

`argv[3] = "c"`

所以将`argv[1]`到`argv[argc - 1]`依次打印出来即可

mycat.c

用到三个函数

```
int open(const char *pathname, int flags,... /* mode_t mode*/);
ssize_t write(int fd, void *buf, size_t nbytes);
ssize_t read(int fd, void *buf, size_t nbytes);
```

用`open`打开文件返回文件描述符`fd`,再用`read()`函数从文件中读数据, 然后用`write()`向标准输出打印数据, 数据为每n个为一组,定义为`buffer`,不断循环知道文件结束, **标准输出的文件描述符为1**

mycp.c

类似于mycat.c, 用`open()`打开两个文件, 然后第一个文件调用`read()`第二个文件调用`write()`, 两个函数的文件描述符`fd`分别为打开文件返回的`fd`

mysys.c

`mysys`函数和`strtok`函数的定义

```
void mysys(char *command)
char *strtok(char *str, const char *delim)
```

`mysys`传入完整的命令, 将`command`复制一份后用`strtok`进行切割,

```
com = strtok(command," ");
argv[0] = com;
while(com) {
    com = strtok(NULL, " ");
    argv[i] = com;
    i++;
}//while
```

获得`argv`, 然后在子进程中用`execvp`执行`argv`

```
pid_t pid = fork();
if(pid == 0)
    execvp(c[0], c);
wait(NULL);
```

因为`exec`会将当前进程的地址空间全部清空所以必须创建子进程, 然后放在子进程中运行, 否则会影响程序后面的执行

sh3.c

首先定义一个指令的结构体如下

```
typedef struct command {
    int argc;
    char *argv[MAX_ARGC];
    char input[FILENAME_LEN];   //重定向输入
    char output[FILENAME_LEN];  //重定向输出
} shellcommand;
```

然后定义一个`commandcount`用于计数管道分割的所有指令的数量, 定义1个空的指令的结构体数组为全局变量

```
int commandcount;
shellcommand commandExternal[MAX_COMMAND];
```

调用以下五个函数完成指令的结构体数组

```
int  split(char str[], char delim[], char *target[MAX_ARGC])//根据delim分割str字符串,分割后存入target中
void command_dump(shellcommand c)//打印指令c的各项属性(argc, argv, input, output)
void init(shellcommand* c)//初始化指令c,将c中的各项属性设为空,在main中每次输入指令前调用
void parsepipecommand(char line[])//将一条指令先以|分割再以空格分割,放入全局的commandExternal中,并计算commandcount
void calculate(shellcommand * c)//包含在parsepipecommand中,获得指令中的重定向输入输出(input和output属性)和指令的argc属性
```

有关多进程的函数如下

```
void exec_simple(shellcommand c)
void exec_pipe(int childcount)
void mysys(char line[])
```

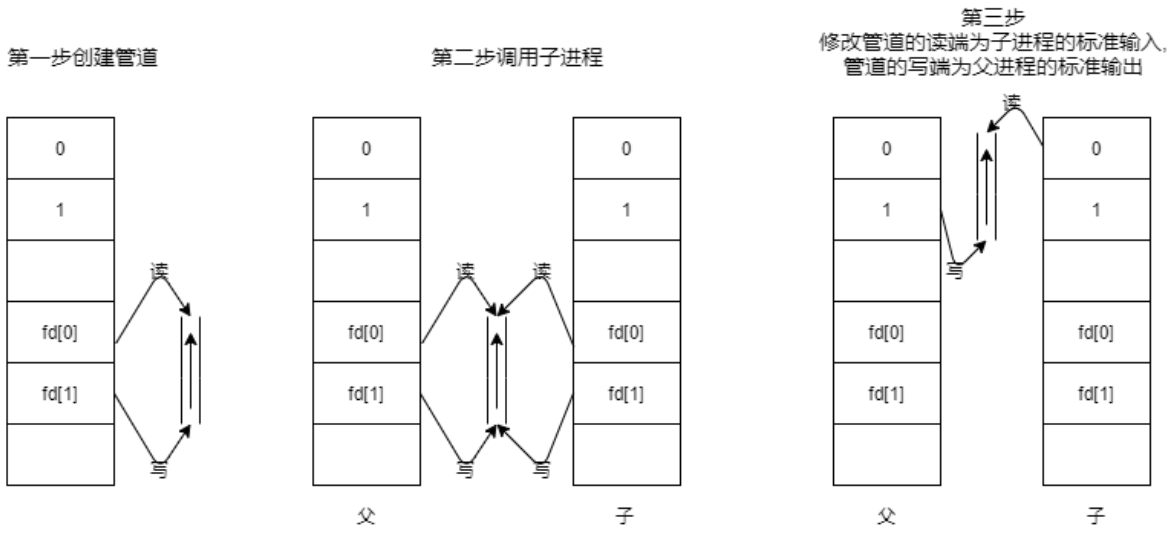
在主函数中调用死循环,每次循环先将全局定义的指令 `commandExternal` 置空, 再读取输入指令并将最后的回车去掉.调用 `mysys()`

```
for (;;) {
    int i;
    for (i = 0; i < MAX_COMMAND; i++)
        init(&commandExternal[i]);
    printf("$ ");
    char c[64];
    fgets(c, sizeof(c), stdin);
    for (i = 0; i < sizeof(c); i++) {
        if (c[i] == '\n')
            c[i] = c[i + 1];
    }
    mysys(c);
}
```

在 `mysys()` 中先用 `parsepipecommand()` 将指令切割, 然后判断第一条指令即 `commandExternal[0]` 是否为 `exit` 或 `cd` 指令, 若是则直接执行不调用新的进程, 否则 `fork()` 调用新的进程并调用函数 `exec_pipe()` 来执行指令

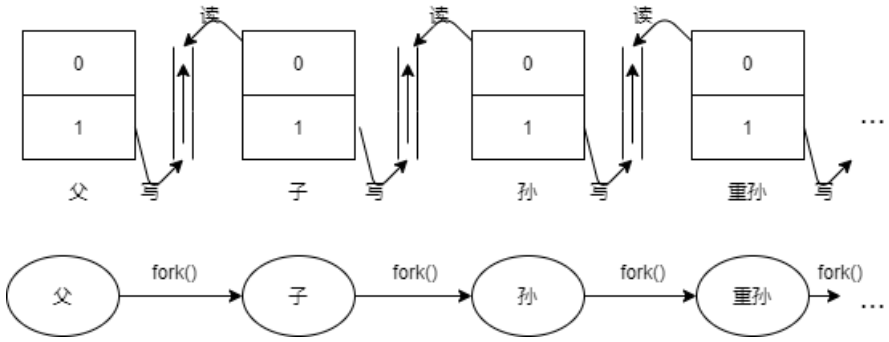
由于 `exec_pipe()` 函数中包括管道的实现, 所以下面介绍管道:

用管道实现进程的通信方法如下图



首先用两个文件描述符生成管道，然后调用子进程，由于文件描述符表是完全复刻一份的，所以有四个文件描述符连接一个管道，接着把父进程的标准输出用`dup2()`重定向到父进程的`fd[1]`，把子进程的标准输入`dup2()`重定向到子进程的`fd[0]`，最后关闭父子进程的`fd[0]`和`fd[1]`，进程间的管道通信就建立完成了

由上图易知多管道的实现形式如下



通过`exec_pipe()`函数的递归调用实现多个管道的连接

```
void exec_pipe(int childcount)
{
    if (childcount == 0)
        return;
    int fd_array[2];
    int pid;
    pipe(fd_array);
    pid = fork();
    if (pid == 0) {
        dup2(fd_array[0], 0);
        close(fd_array[0]);
        close(fd_array[1]);
        exec_simple(commandExternal[childcount - 1]);
    }
    dup2(fd_array[1], 1);
    close(fd_array[0]);
    close(fd_array[1]);
    exec_pipe(childcount - 1);
}                                   //exec_pipe
```

在该函数的内部子进程中执行指令，在外部父进程中递归迭代，从后往前逐个进程之间连接起来，这里后是指指令在全局变量指令数组中的位置.

如果没有管道, 说明只有一条非`exit`或`cd`的指令, 则在子进程中调用`exec_pipe()`实现该指令, 父进程下一次迭代遇到递归终止条件直接返回

由于全局变量会在子进程中继承, 而指令是存放在全局变量中的, 所以在每个进程中都会有一份指令的完整的复刻.

在`exec_simple()`中用`execvp()`实现一条指令, 首先查看该指令的`output`和`input`两个属性是否为空, 若均为空则直接调用`execvp()`否则对输入输出先进行重定向再执行指令

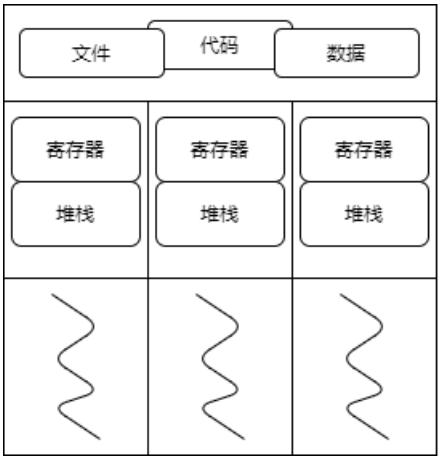
```
void exec_simple(shellcommand c)
{
    if (strcmp(c.output, "\0")) {
        int fd_out = open(c.output, O_CREAT | O_RDWR | O_TRUNC, 0666);
        dup2(fd_out, 1);
        close(fd_out);
    }
    if (strcmp(c.input, "\0")) {
        int fd_in = open(c.input, O_RDONLY);
        if (fd_in < 0) {
            printf("open error\n");
            exit(0);
        }
        dup2(fd_in, 0);
        close(fd_in);
    }
    execvp(c.argv[0], c.argv);
}
```

pi1.c

主线程使用的`master`和子线程所用的`worker`函数定义

```
void *master(void *arg)
void *worker(void *arg)
```

由于多线程中的全局变量是**共享的**, 即每个线程都可以访问相同的全局变量, 如下图

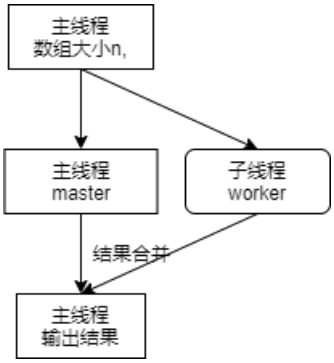


所以可以定义一个全局变量来保存线程计算的结果, 但同时也需要信号量来防止对内存的访问混乱

`main`中子进程和主进程分别执行`worker`和`master`，然后主进程等待子进程完成后两个进程的放在全局变量的结果相加

```
pthread_t tid;
pthread_create(&tid, NULL, &worker, num);
master(num);
pthread_join(tid, NULL);
```

执行过程如下图



pi2.c

定义一个结构体，存放数组和线程要操作的数组的索引，还有另一个结构体存储计算后的结果

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

之前的两个函数`worker`和`master`函数合并成一个`compute`函数，用于计算结果

```
void *compute(void *arg)
```

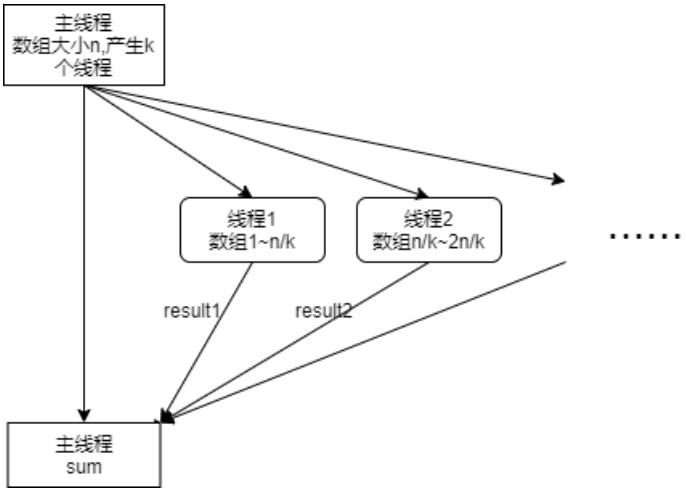
在`main()`每次迭代用一个结构体的指针指向当前线程要操作的结构体，结构体中的值初始化后送入线程中计算

```
for (i = 0; i < NR_CPU; i++) {
    param *_p;
    _p = &p[i];
    _p->array = array;
    _p->start = i * NR_CHILD;
    _p->end = (i + 1) * NR_CHILD;
    pthread_create(&worker[i], NULL, compute, _p);
}
```

然后用`pthread_join`等待线程结束并将结果导出到`result`结构体中，并将每次的`result`加到最后的`sum`中

```
for (i = 0; i < NR_CPU; i++) {
    result *res;
    pthread_join(worker[i], (void **) &res);
    sum += res->sum;
    free(res);
}
```

执行过程如下图



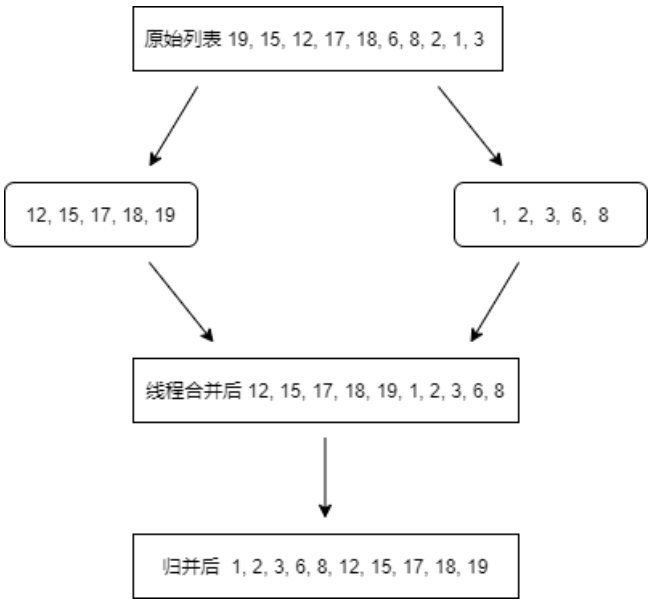
sort.c

仍然使用pi2.c的结构, 即使用结构体来访问数组的前半部分和后半部分,这次的`compute()`函数功能为选择排序.

- 第一个线程中对数组的前半部分选择排序
- 第二个线程中对数组的后半部分选择排序

然后主线程中等待两个线程执行结束后对数组前后两部分归并一次

流程图如下所示



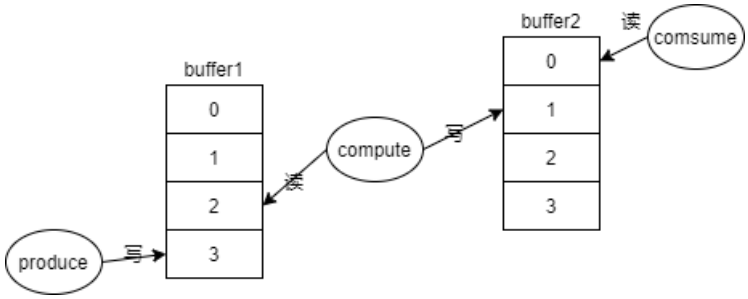
pc1.c

定义三个函数: 生产`produce`计算`compute`消费`consume`, 分别为一个线程所使用

```
void *produce(void *arg)
void *compute(void *arg)
void *consume(void *arg)
```

- `buffer1`和`buffer2`为两个宏定义的数组, 表示线程共享访问的区域
- `produce`线程和`compute`线程互斥访问共享内存`buffer1`
- `compute`线程和`consume`线程互斥访问共享内存`buffer2`
- 定义四个条件变量, 操作`buffer1`的为`wait_empty1`和`wait_full1`, 操作`buffer2`的为`wait_empty2`和`wait_full2`
- 在每次对`buffer`操作(读或者写)时用互斥量`mutex`加锁, 防止操作被别的线程中断
- 以`buffer1`的读写为例
  - 当`buffer1`满了时, `produce`不能继续生产, 要等待`compute`读一个数据, 待`compute`释放一个`wait_empty1`后继续生产
  - 当`buffer1`空了时, `compute`不能继续消耗, 要等待`produce`写一个数据, 待`produce`释放一个`wait_full1`后继续消耗
- `compute`线程分为两部分, 先等待`buffer1`不为空, 然后取数据计算送`buffer2`, 第二部分送数据到`buffer2`时等待`buffer2`中数据被取出, 即`buffer2`不满

三个线程读写操作流程图如下



`produce`线程执行的函数如下, `while`语句循环检查缓冲区`buffer1`是否为空

```
void *produce(void* arg)
{
    int i;
    ITEM_TYPE item;
    for ( i = 0; i < ITEM_COUNT; i++) {
        pthread_mutex_lock(&mutex);
        while (buffer1_is_full)
            pthread_cond_wait(&wait_empty1, &mutex);
        item = 'a' + i;
        printf("produce: in1 = %d\n", in1);
        put1(item);

        pthread_cond_signal(&wait_full1);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

最后: 为什么`pthread_cond_wait`需要互斥锁`mutex`作为参数?

即为什么是`pthread_cond_wait(cond, mutex)`而不是`pthread_cond_wait(cond)`

因为线程中, `pthread_cond_wait` 前还有一个 `mutex`, 若不将其释放, 会阻塞其他线程进入 `buffer` 临界区产生死锁, 所以应先释放 `mutex`, 再阻塞线程等待被唤醒, 被唤醒后再次调用 `mutex` 保护临界区

pc2.c

定义一个结构体实现信号量, 包括一个值 `value`,一个互斥锁 `mutex` 和一个条件变量 `cond`

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

- 信号量的值 `value` 只有在大于0时才继续线程, 否则将当前线程阻塞.
- 所以对应完成自定义信号量的的 `wait` 和 `signal` 函数, 这里的判断条件是 `value` 的值,为正则 `sema_wait` 可以继续进行, 为零或负则在函数中阻塞线程, 另外 `sema_signal` 则是释放互斥锁,在信号量中即将 `value` 自增1.
- 同样在信号量函数中也需要 `mutex` 锁防止线程被中断, 所以结构体中会有 `pthread_mutex_t` 类型的变量.

```
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);
}
```

所以在上一题的三个线程函数生产 `produce` 计算 `compute` 和消费 `consume` 中, 用前面定义的信号量和对应函数替换上一题中的条件变量即可.如下面 `produce` 为例

```
sema_wait(&emptybuffer1);
sema_wait(&sema_mutex1);
/*生产produce操作*/
sema_signal(&sema_mutex1);
sema_signal(&fullbuffer1);
```

注意: 仍然需要定义一个信号量 `sema_mutex` 来防止线程被中断, 只不过该 `mutex` 信号量的值 `value` 初始化为1.

`emptybuffer` 初始化要根据 `buffer` 容量设定, `fullbuffer` 初始化为0