

Operating System Labs Project 4

Project 4a: File Defragmentation

Part 1 概述:

本次实验主题为碎片整理，在一个文件系统中，其简易分布为:

boot -> superblock -> inode -> data -> swap

其中 **boot** 是启动区，对于本实验无需关心，**superblock** 记录了单位块的大小，**inode** 区，**data** 区与 **swap** 区的偏移量，最后还记录了 **inode** 与 **iblock** 的 **free** 链表的头部索引。

一个文件的大小往往远大于一个块，因此需要多个块来存储文件。初始时，磁盘空闲，可以较为理想的将文件连续存储在磁盘中，但随着空闲区域的减少，当磁盘不再有较大的连续区域时，便需要离散的存储文件的各个数据段。并且随着文件的删减与增加，地盘也会产生零头。文件的块索引分离越多，访问的时耗与开销都会增加，因此进行碎片整理是必要的。

Part2 data block 分类:

如果一个 **inode** 只包含 0 级目录，那么我们可以直接简单的从 0 开始排列所有文件块，但随着多级目录的加入，有两个需要解决的问题：

1. 如何通过多级目录访问文件块

2. 目录块本身也是数据块，也就是说 **data block** 既包含了文件块，也包含了 **inode** 的目录块，我们应该如何排列这两种数据块？

对于问题 1，递归是目录与目录树的常用解法，我们将在后文进行阐述。

对于问题 2，我们有两种方案：

我们首先假设存在 n 个 **data block**, $d[i]$ 表示第 i 个块。

(1) 视文件块与目录块为同一类型，混合顺序存储。那么可能 $d[0]$ - $d[100]$ 为文件块， $d[101]$ 为 1 级目录块， $d[102]$ 为 2 级目录块…… $d[120]$ 开始又为一段的文件块。

(2) 区分文件块与目录块，将索引块全部放在文件块之后存储。如果假设文件块一共有 m 个，那么 $d[0]$ - $d[m-1]$ 一定都是文件块， $d[m]$ 到 $d[n-1]$ 就一定都是目录块。

最终我们采用的是方案 2，因为碎片整理的目的就是尽可能的让文件块连续，因此我们选择方案 2 来优化排列。我们可以借助 **inode->size** 来获取文件块的总数量 **sum**，那么文件块的起始块为 0，目录块的起始块为 **sum**

Part3 算法流程:

- 1.创建好输出文件
- 2.打开输入输出流
- 3.复制 boot 启动块
- 4.读取超级块来获取 data,inode 的位置, 以及 block_size 的值
- 5.计算出文件块的总个数 sum
- 6.设置双指针 data_pointer,inode_pointer
- 7.枚举每一个 inode
- 8.通过递归得到当前 inode 全部的文件块索引与目录块索引
- 9.通过索引获得文件块的内容, 输出到新索引(data_pointer)
- 10.根据新索引重写目录块, 输出到新的索引 (inode_pointer)
- 11.将新的 inode 写入输出流
- 12.更新超级块的内容, 写入输出流
- 13.关闭输入输出流, 释放内存

Part4 定义函数与全局变量:

```
FILE *istream,*ostream;           //输入流与输出流
SuperBlock *superblock,*boot;     //文件的超级块,起始块
int inode_index;                   //inode 的起始地址(bytes)
int inode_count;                   //inode 的数量
int data_index;                    //data_block 的起始地址(bytes)
int block_size;                    //block 的大小
int block_count;                   //inode 对应文件占用块的数量
int inode_size = sizeof(inode);    //inode 结构所占字节数
char data[5000];                   //读取文件时存储数据的临时变量
inode *inode;                      //指向当前枚举到的 inode
int data_pointer = 0;              //指向数据块当前的块索引
int inode_pointer;                 //指向目录块当前的块索引

//以下为程序相关函数
void init();                       //初始化部分参数
int address(int number);           //输入一个块的索引, 返回其地址
int calculate_summary_of_blocks(); //计算出文件一共占用的块的个数

//从上至下分别为 0-3 级目录的 inode 读写函数
//四个函数的参数意义均为 read_address, write_address
void write_from_data_blocks(int,int);
void write_from_indirect_blocks(int,int);
void write_from_doubly_indirect_blocks(int,int);
void write_from_triply_indirect_blocks(int,int );
```

```

//以下为 debug 和 display 相关函数,您可以不关心该部分的函数
FILE*file;                                //需要打印的文件流
void write_usage();                        //命令行提示信息
void display(char* f1,char* f2);          //display 主函数
void print(char *f);                      //打印文件 f
void order(inode* node);                  //遍历一个 inode
void dfs(int val,int level);              //递归打印一个 inode

```

Part5 算法实现:

- 1.创建好输出文件,输出文件的命名为 <输入文件>+-defrag

```

//构造输出文件的文件名
char outFileName[100],extra[8] = "-defrag";
strcpy(outFileName,argv[1]);
int j = strlen(outFileName);
for(int i = 0;i < 7;i++)
    outFileName[j++] = extra[i];
outFileName[j] = '\0';

```

- 2.打开输入输出流

```

istream = fopen(argv[1],"r");
ostream = fopen(outFileName,"w+");
if(!istream || !ostream) //打开失败
{
    write(1,"open file faid!",strlen("open file faid!"));
    exit(1);
}

```

- 3.复制 boot 启动块

```

boot = (SuperBlock*)malloc(512);
fseek(istream,0,SEEK_SET);
fread(boot,512,1,istream);
fseek(ostream,0,SEEK_SET);
fwrite(boot,512,1,ostream);

```

- 4.读取超级块来获取 data,inode 的位置,以及 block_size 的值

```

//读入超级块
fseek(istream,512,SEEK_SET);
superblock = (SuperBlock*)malloc(512);
fread(superblock,512,1,istream);

//初始化参数
block_size = superblock->size;

```

```

data_index = (superblock->data_offset)*block_size+1024;
inode_index = (superblock->inode_offset)*block_size+1024;
inode_count = (data_index-inode_index)/inode_size;

```

5. 计算出文件块的总个数 sum

```

int calculate_summary_of_blocks()
{
    int sum = 0, inode_address = inode_index; //sum 为总数量
    for(int i=0; i<inode_count; i++, inode_address += inode_size)
    {
        //读入第 i 的 inode
        fseek(istream, inode_address, SEEK_SET);
        fread(inode, inode_size, 1, istream);
        if(inode->nlink <= 0) continue;

        //计算其文件占用块数量
        block_count = (inode->size)/block_size;
        if(inode->size % block_size != 0) block_count++;

        //添加到总和
        sum += block_count;
    }
    return sum;
}

```

6. 设置双指针 data_pointer, inode_pointer

```

int data_pointer = 0; //指向数据块当前的块索引
int inode_pointer; //指向目录块当前的块索引
inode_pointer = calculate_summary_of_blocks(); //计算出块的总数

```

7. 枚举每一个 inode

```

int inode_address = inode_index; //inode 的起始地址
//枚举全部的 inode
for(int i = 0; i < inode_count; i++, inode_address += inode_size)

```

8. 通过递归得到当前 inode 全部的文件块索引与目录块索引

9. 通过索引获得文件块的内容，输出到新索引(data_pointer)

10. 根据新索引重写目录块，输出到新的索引 (inode_pointer)

```

//读取出第 i 个 inode 的信息
fseek(istream, inode_address, SEEK_SET);
fread(inode, inode_size, 1, istream);

```

```

//如果空闲，下一个

```

```

if(inode->nlink <= 0) continue;

//计算出其对应的文件占用的块数量
block_count = (inode->size)/block_size;
if(inode->size % block_size != 0) block_count++;

//0 级目录处理
for(int v = 0;v < N_DBLOCKS && block_count > 0;v++)
{
    //构造读入地址与写出地址
    int read_address = address(inode->dblocks[v]);
    int write_address = address(data_pointer);

    inode->dblocks[v] = data_pointer;//重排数据块位置
    //写入新的位置
    write_from_data_blocks(read_address,write_address);
}

//1 级目录处理，结构同上
for(int v = 0;v < N_IBLOCKS && block_count > 0;v++)
{
    int read_address = address(inode->iblocks[v]);
    int write_address = address(inode_pointer);
    inode->iblocks[v] = inode_pointer;
    write_from_indirect_blocks(read_address,write_address);
}

//2 级目录处理，结构同上
if(block_count > 0)
{
    int read_address = address(inode->i2block);
    int write_address = address(inode_pointer);
    inode->i2block = inode_pointer;
    write_from_doubly_indirect_blocks(read_address,write_address);
}

//3 级目录处理，结构同上
if(block_count > 0)
{
    int read_address = address(inode->i3block);
    int write_address = address(inode_pointer);
    inode->i3block = inode_pointer;
    write_from_triply_indirect_blocks(read_address,write_address);
}

```

11. 将新的 inode 写入输出流

//将更新之后的 inode 写进输出流

```
fseek(ostream, inode_address, SEEK_SET);
```

```
fwrite(inode, sizeof(inode), 1, ostream);
```

12. 更新超级块的内容，写入输出流

//重设 free_iblock 的值

```
superblock->free_iblock = inode_pointer;
```

//将超级块写入输出流

```
fseek(ostream, 512, SEEK_SET);
```

```
fwrite(superblock, 512, 1, ostream);
```

13. 关闭输入输出流，释放内存

//关闭输入输出流

```
fclose(istream);
```

```
fclose(ostream);
```

//如果是 display 模式，调用 display 函数打印文件信息

```
if(argc == 3) display(argv[1], outFileNames);
```

//释放内存

```
free(inode);
```

```
free(superblock);
```

```
free(boot);
```

Part6 display mode:

为了检验实验结果，我们设计了两种 command:

第一种为实验要求的 ./defrag datafile, 程序会输出 datafile-defrag 文件

第二种为额外附加的 ./defrag datafile display, 若使用这种命令,程序会额外输出 datafile 在碎片整理前和碎片整理后的信息。

若您输入的 command 有误，那么会打印提示信息:

```
shirai@ubuntu:~/Desktop/Project/P4$ ./defrag
Sorry, you have invoked the program mistakenly.
Usage:
    ./defrag <fragmented disk file>           //common mode
    ./defrag <fragmented disk file> display  //display mode

For common mode, the program will output a file called <disk file -defrag>, which is generated after defragmentation.

For display mode, the program will additionally print the information about <disk file> and <disk file -defrag>.
```

调用 display mode 的效果:

```
shirai@ubuntu: ~/Desktop/Project/P4
shirai@ubuntu:~/Desktop/Project/P4$ ./defrag datafile display
-----The information of file before file defragmentation-----

Super Block Information:
block_size = 512
inode_offset = 0
data_offset = 4
swap_offset = 10243
free_inode = 14
free_iblock = 10133

inode 0:
Block numbers which store data:
1120 8393 1579 9539 7108 7883 4762 1980 1030 8610
4356 7885 142 2333 5896 6179 7319 6843 2855 4170
1506 7320 3863 3729 4364 4231 6288 7269 6627 7413
5428 8210 6717 2301 8167 4366 7068 5400 5444 4418
9761 3094 4564 1859 8995 508 9183 5604 3368 3118
7115 453 6985 610 4822 981 6902 1856 7612 4080
7289 5588 562 9594 3521 4933 6428 3439 98 1637
7861 9864 4736 2191 1488 3496 2703 437 9104 6076
3559 5985 6533 310 6599 1121 1296 3266 2982 8913
7350 36 4267 7917 9635 7792 2615 5828 997 2718
7470 8862 2347 1971 818 3840 5472 3526 4281 4341
9606 7845 91 5905 8160 6695 7030 9461 9965 10016
8139 7081 10057 2171 4764 9457 9968 7383 5050 730
10106 2285 9597 2219 4260 180 6064 9736 111 3843
3083 7960 3939 8992 5886 399 5788 5112 130 5570
3017 7215 5392 5192 1744 4614 4926 9132 9669 5660

-----The information of file after file defragmentation-----

Super Block Information:
block_size = 512
inode_offset = 0
data_offset = 4
swap_offset = 10243
free_inode = 14
free_iblock = 2136

inode 0:
Block numbers which store data:
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129
130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149
150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169
```


Part7 result:

我们根据 display 的信息记录下整理结果:

inode	文件块区间	目录块区间
0	[0,183]	[2114,2115]
1	[184,539]	[2116,2118]
2	[540,622]	2119
3	623	/
4	[624,645]	2120
5	[646,679]	2121
6	free	free
7	[680,684]	/
8	[685,714]	2122
9	free	free
10	[715,757]	2123
11	[758,797]	2124
12	798	/
13	[799,1248]	[2125,2128]
14	free	free
15	[1249,1251]	/
16	[1252,1259]	/
17	[1260,1263]	/
18	[1264,1753]	[2129,2132]
19	[1754,2113]	[2133,2135]
总	[0,2113]	[2114,2135]

观测结果可知 6,9,14 三个 inode 是 free 的, 而 data 一共有 2136 个 block, 我们成功的将其分隔成两块区域,[0,2113]为文件块, [2114,2135]为目录块

内存泄露测试:正常

```
shirai@ubuntu:~/Desktop/Project/P4$ valgrind --leak-check=full --show-reachable
=yes ./defrag datafile
==192889== Memcheck, a memory error detector
==192889== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==192889== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==192889== Command: ./defrag datafile
==192889==
==192889==
==192889== HEAP SUMMARY:
==192889==    in use at exit: 0 bytes in 0 blocks
==192889==   total heap usage: 7 allocs, 7 frees, 10,260 bytes allocated
==192889==
==192889== All heap blocks were freed -- no leaks are possible
==192889==
==192889== For lists of detected and suppressed errors, rerun with: -s
==192889== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```


Project 4b: xv6 Kernel Thread

Part 1 涉及文件:

1. xv6/include/syscall.h
2. xv6/kernel/defs.h
3. xv6/kernel/syscall.c
4. xv6/kernel/sysfunc.h
5. xv6/kernel/sysproc.c
6. xv6/kernel/sysproc.c
7. xv6/kernel/sysproc.c
8. xv6/user/ulib.c
9. xv6/user/user.h
10. xv6/user/usys.S

Part 2: 添加系统调用:

1. include/syscall.h 中添加
#define SYS_clone 22
#define SYS_join 23
2. kernel/defs.h 声明 clone 与 join 函数
int clone(void (*fcn)(void*), void *, void *);
int join(void **);
3. kernel/syscall.c 中添加
[SYS_clone] sys_clone,
[SYS_join] sys_join,
4. kernel/sysfunc.h 声明 sys_clone 与 sys_join 函数
int sys_clone(void);
int sys_join(void);
5. user/user.h 声明以下函数
int clone(void (*fcn)(void*), void *arg, void *stack);
int join(void **stack);

int thread_create(void(*start_routine)(void*), void *);
int thread_join();
6. user/usys.S 添加系统调用
SYSCALL(clone)
SYSCALL(join)

Part 3:实验原理

1.线程

在标准库 `pthread` 中，有函数

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void*(*start_routine)(void *arg), void *arg);
```

`pthread_t` 是用于唯一标识线程的数据类型，它由 `pthread_create()` 返回，并由应用程序在需要线程标识符的函数调用中使用，创建运行 `start_routine` 的线程，`arg` 作为唯一参数。如果 `pthread_create()` 成功完成，则线程将包含已创建线程的 ID。如果失败，则不会创建新线程，并且线程未定义。

`attr` 定义了线程的属性，主要用于实时编程。

第一个参数为指向线程标识符的指针。

第二个参数用来设置线程属性。

第三个参数是线程运行函数的起始地址。

最后一个参数是运行函数的参数。

线程的实现是在进程的基础上的，因此我们给 `proc.h` 中的 `proc` 结构体拓展属性 `stack` 便可以用来表示线程。

2. join

在线程管理中，我们需要一个类似于 `wait` 的函数，用于线程完成一个进程的內部。在标准库中，该函数是 `pthread_join`，它接受两个参数：
*要等待的线程的线程 ID，以及指向将接收完成线程的返回值的 `void` 变量的指针。如果您不关心线程返回值，则将 `NULL` 作为第二个参数传递。而在本次实验，我们需要撰写 `join` 和 `thread_join` 函数。

3.clone

线程是作为单独的进程实现的，但它们共享其虚拟内存空间和其他资源，但是，使用 `fork` 创建的子进程可以获取这些项的副本。标准库的 `clone` 系统调用是 `fork` 和 `pthread_create` 的通用形式，它允许调用者指定在调用进程和新创建的进程之间共享哪些资源。`clone()` 的主要用途是实现线程：在共享内存空间中并发运行的程序中的多个控制线程。

与 `fork()` 不同，这些调用允许子进程与调用进程共享其执行上下文的一部分，例如内存空间，文件描述符表和信号处理程序表。此过程与使用 `fork` 创建的过程不同；特别是，它与原始进程共享相同的地址空间和资源。

使用 `clone()` 创建子进程时，它将执行函数 `fn(arg)`。`fn` 参数是指向子进程在执行开始时调用的函数的指针。`arg` 参数传递给 `fn` 函数。`child_stack` 参数指定子进程使用的堆栈的位置。虽然属于同一进程组的克隆进程可以共享相同的内存空间，但它们不能共享相同的用户堆栈。因此，`clone()`

调用为每个进程创建单独的堆栈空间，进程和线程之间的区别是通过将不同的标志传递给 `clone()` 来实现的。

4.进程的 zombie 状态解释

进程的 zombie 状态——也称僵尸进程，是指的父进程已经退出,而该进程 dead 之后没有进程接受,就成为僵尸进程.(zombie)进程

在 `fork()/execve()`过程中，假设子进程结束时父进程仍存在，而父进程 `fork()`之前既没调用 `waitpid()`等待子进程结束，又没有显式忽略该信号，则子进程成为僵尸进程，无法正常结束，此时即使是 root 身份 kill-9 也不能杀死僵尸进程。补救办法是杀死僵尸进程的父进程(僵尸进程的父进程必然存在)，僵尸进程成为"孤儿进程"，过继给进程 `init`，`init` 会负责清理僵尸进程。

一个进程在调用 `exit` 命令结束自己的生命的时候，其实它并没有真正的被销毁，而是留下一个称为僵尸进程的数据结构(系统调用 `exit`，它的作用是使进程退出，但也仅仅限于将一个正常的进程变成一个僵尸进程，并不能将其完全销毁)。

僵尸进程是非常特殊的一种状态，它已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占有任何内存空间。但是如果父进程是一个循环，不会结束，那么子进程就会一直保持僵尸状态，这就是为什么系统中有时会有很多的僵尸进程。

在 part4 中我们可以看到，`join` 与 `wait`，`clone` 与 `fork` 有着高度的相似性，但又有一些差异，我们通过 `proc` 扩展的 `stack` 属性来进行区分和维护。

Part 4:函数实现:

1.kernel/sysproc.c 中

```
int sys_clone(void)
{
    void *fcn, *arg, *stack;

    if(argptr(0, (void *)&fcn, sizeof(void *)) < 0
        || argptr(1, (void *)&arg, sizeof(void *)) < 0
        || argptr(2, (void *)&stack, sizeof(void *)) < 0)
        return -1;

    if((uint)stack % PGSIZE != 0)
        return -1;
    if((proc->sz - (uint)stack) < PGSIZE)
        return -1;
```

```

        return clone(fcn, arg, stack);
    }

int sys_join(void)
{
    void **stack;
    if(argptr(0, (void *)&stack, sizeof(void *)) < 0)
        return -1;
    return join(stack);
}

```

2. user/ulib.c 中

```

int thread_create(void (*start_routine)(void *), void *arg)
{
    void *stack;
    stack = malloc(2 * PGSIZE);
    uint offset = (uint)stack % PGSIZE;
    if(offset != 0)
        stack = stack + (PGSIZE - offset);
    return clone(start_routine, arg, stack);
}

int thread_join()
{
    void *stack = malloc(sizeof(void*));
    int tid = join(&stack);
    free(stack);
    return tid;
}

```

3. kernel/proc.c 中

```

int join(void **stack)
{
    struct proc *p;
    int havekids, pid;
    acquire(&ptable.lock);
    for(;;){
        // Scan through table looking for zombie children.
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            // only consider child threads
            if(p->parent != proc || p->pgdir != proc->pgdir)
                continue;
            havekids = 1;

```

```

        if(p->state == ZOMBIE){
            // Found one.
            pid = p->pid;
            kfree(p->kstack);
            p->kstack = 0;
            p->state = UNUSED;
            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->killed = 0;
            *stack = (void *)p->stack;
            release(&ptable.lock);
            return pid;
        }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
        release(&ptable.lock);
        return -1;
    }

    // Wait for children to exit.
    sleep(proc, &ptable.lock); //DOC: wait-sleep
}

int clone(void (*fcn)(void *), void *arg, void *stack)
{
    int i, pid;
    struct proc *np;

    // Allocate process.
    if((np = allocproc()) == 0)
        return -1;

    // use the same address space
    np->pgdir = proc->pgdir;
    np->sz = proc->sz;
    np->parent = proc;
    *np->tf = *proc->tf;
    np->stack = (uint)stack;

    // Clear %eax so that fork returns 0 in the child.

```

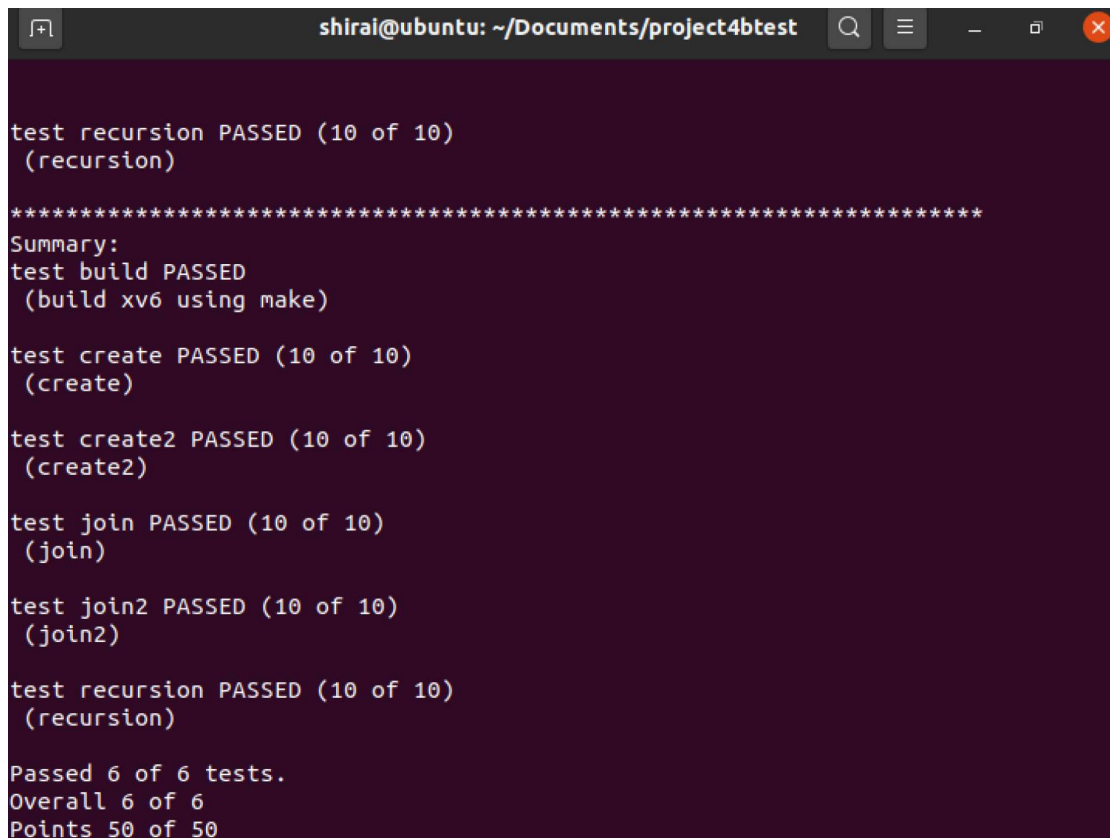
```

np->tf->eax = 0;
uint sp;
sp = (uint)stack + PGSIZE;
uint ustack[2];
ustack[0] = 0xffffffff;
ustack[1] = (uint)arg;
sp -= 2*sizeof(uint);
if(copyout(np->pgdir, sp, ustack, 2*sizeof(uint)) < 0)
    return -1;
np->tf->eip = (uint)fcn;
np->tf->esp = sp;
for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
        np->ofile[i] = filedup(proc->ofile[i]);
np->cwd = idup(proc->cwd);

pid = np->pid;
np->state = RUNNABLE;
safestrcpy(np->name, proc->name, sizeof(proc->name));
return pid;
}

```

Part 5: Test Result:



```

shirai@ubuntu: ~/Documents/project4btest

test recursion PASSED (10 of 10)
(recursion)

*****
Summary:
test build PASSED
(build xv6 using make)

test create PASSED (10 of 10)
(create)

test create2 PASSED (10 of 10)
(create2)

test join PASSED (10 of 10)
(join)

test join2 PASSED (10 of 10)
(join2)

test recursion PASSED (10 of 10)
(recursion)

Passed 6 of 6 tests.
Overall 6 of 6
Points 50 of 50

```