《操作系统实验》

实验报告

学　部：　　电子信息与电气工程学部

专　业：　　计算机科学与技术

班　级：　　电计1701班

小组成员：　　应承轩（201785071）

实验1 进程管理

一、子实验1

进程父子关系：

核心代码：

```
pid_t t1 = fork();
if (t1 == 0){
    // inside son process
    pid_t t2 = fork();
    if (t2 == 0){
        std::cout << "子2" << std::endl; cout_pid();
    } else {
        std::cout << "子1" << std::endl; cout_pid(); cout_son_pid(t2);
    }
} else {
    std::cout << "父" << std::endl; cout_pid(); cout_son_pid(t1);
}
```

实验结果：

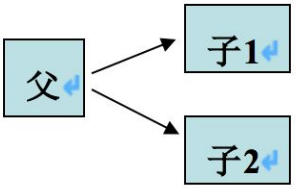
```
/Users/yingchengxuan/CLionProjects/course/cmake-build-debug/course
Experiment1
父
my pid:52741
my son's pid:52742
子1
my pid:52742
my son's pid:52743
子2
my pid:52743
```

讨论与分析：

这个结果看起来是比较合理的，他们的pid是递增的。Macos系统的pid是自增分配的，可以揭示父子进程创建的先后顺序。

二、子实验2

进程父子关系：



核心代码：

```
pid_t t1 = fork();
if (t1 == 0){
    std::cout << "子1" << std::endl; cout_pid();
} else {
    pid_t t2 = fork();
    if (t2 == 0){
        std::cout << "子2" << std::endl; cout_pid();
    } else {
        std::cout << "父" << std::endl; cout_pid(); cout_son_pid(t1); cout_son_pid(t2);
    }
}
```

实验结果：

```
/Users/yingchengxuan/CLionProjects/course/cmake-build-debug/course
Experiment1
子1
my pid:53072
父
my pid:53069
my son's pid:53072
my son's pid:53073
子2
my pid:53073
```

讨论与分析：同子实验1

这个结果看起来是比较合理的，他们的pid是递增的。Macos系统的pid是自增分配的，可以揭示父子进程创建的先后顺序。

三、子实验3

实验内容：

编写一个命令处理程序，能处理max(m,n), min(m,n)和 average(m,n,l)这几个命令。（使用exec函数族）

核心代码：

调用者

```
char *arglist[] = {"./course", "average", "111", "222", "333", NULL};
char *env[] = {NULL};
execve( file: arglist[0], arglist, env);
```

被调用者

```
if (!strcmp(argv[1], "max")) {
    int t1, t2, t;
    sscanf(argv[2], "%d", &t1);
    sscanf(argv[3], "%d", &t2);
    t = t1 > t2 ? t1 : t2;
    std::cout << "max(" << t1 << "," << t2 << ")=" << t << std::endl;
} else if (!strcmp(argv[1], "min")) {
    int t1, t2, t;
    sscanf(argv[2], "%d", &t1);
    sscanf(argv[3], "%d", &t2);
    t = t1 < t2 ? t1 : t2;
    std::cout << "min(" << t1 << "," << t2 << ")=" << t << std::endl;
} else if (!strcmp(argv[1], "average")) {
    float t1, t2, t3, t;
    sscanf(argv[2], "%f", &t1);
    sscanf(argv[3], "%f", &t2);
    sscanf(argv[4], "%f", &t3);
    t = (t1 + t2 + t3) / 3.0;
    std::cout << "average(" << t1 << "," << t2 << "," << t3 << ")=" << t << std::endl;
} else {
    std::cout << argv[1] << "not valid params" << std::endl;
}
```

实验结果：

```
/Users/yingchengxuan/CLionProjects/course/cmake-build-debug/course
average(111,222,333)=222

/Users/yingchengxuan/CLionProjects/course/cmake-build-debug/course
max(111,222)=222

/Users/yingchengxuan/CLionProjects/course/cmake-build-debug/course
min(111,222)=111
```

我实现的特点：

我把这些内容在一个程序中实现的，通过判断argc的个数，如果argc=1说明是一个执行不带参数，则调用exe部分，否则解析对应的参数。

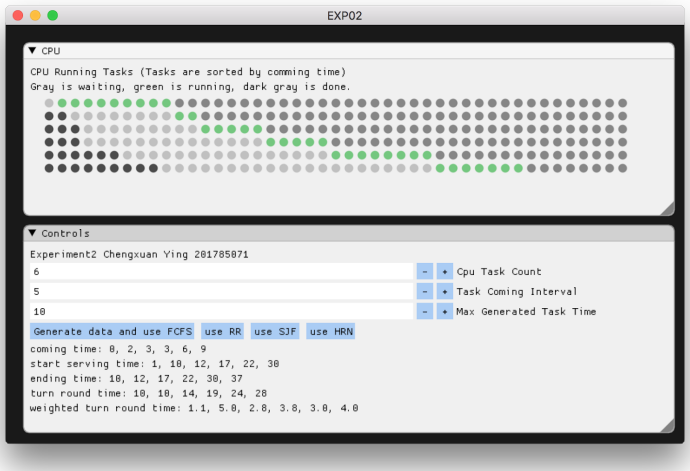
完整代码：

https://github.com/chengsyuan/OS_Homework/tree/master/HW1

实验2 处理器调度

实验内容：
模拟单处理器调度算法、可视化展示过程、计算各种指标衡量调度算法的性能。

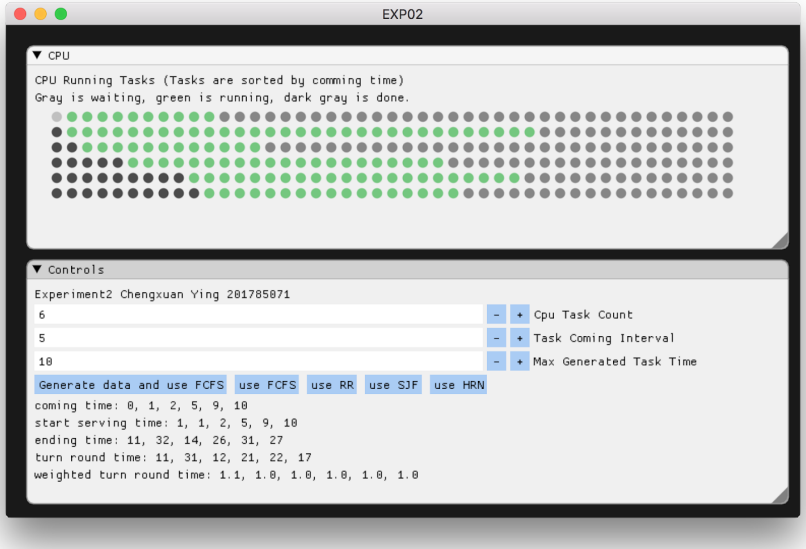
实验结果：
主界面（FCFS）：



带有颜色的圆点，代表任务的执行情况
（灰色代表等待，绿色代表执行中，深灰色代表结束）。

用户可以点击各种算法，查看具体的模拟结果。
具体使用过程请查看*演示视频*。

RR界面：



核心代码：

FCFS实现：

```
def FCFS_step(self):
    if self.is_finished():
        return

    activated_tasks = self.get_active_tasks()
    assert len(activated_tasks) <= 1 # FCFS's condition

    if len(activated_tasks) == 1:
        # get ongoing task
        activated_task = activated_tasks[0]
        self.tasks[activated_task]['running_time'] += 1

        # check if done
        if self.tasks[activated_task]['running_time'] >= self.tasks[activated_task]['cpu_time']:
            self.tasks[activated_task]['status'] = 'done'
            self.tasks[activated_task]['end_time'] = self.time
            self.activate_first_come_task()
    else:
        self.activate_first_come_task()

    self.time += 1
    return
```

RR实现：

```
def RR_step(self):
    if self.is_finished():
        return

    self.activate_all_come_task()
    activated_tasks = self.get_active_tasks()

    if len(activated_tasks) >= 1:
        # get ongoing task
        for activated_task in activated_tasks:
            # CPU sharing
            self.tasks[activated_task]['running_time'] += 1.0 / len(activated_tasks)

            # check if done
            if self.tasks[activated_task]['running_time'] >= self.tasks[activated_task]['cpu_time']:
                self.tasks[activated_task]['status'] = 'done'
                self.tasks[activated_task]['end_time'] = self.time

    self.time += 1
    return
```

由于SJF和HRN方式的实现，仅仅在FCFS的基础上更改了排序的策略。为了让篇幅简约，在这里不展示详细代码。请自行查看完整代码。

实验结果：

使用FCFS、SJF和HRN方式，对CPU任务进行调度，周转时间均优于使用RR方式的实现。

完整代码：

https://github.com/chengsyuan/OS_Homework/tree/master/HW2

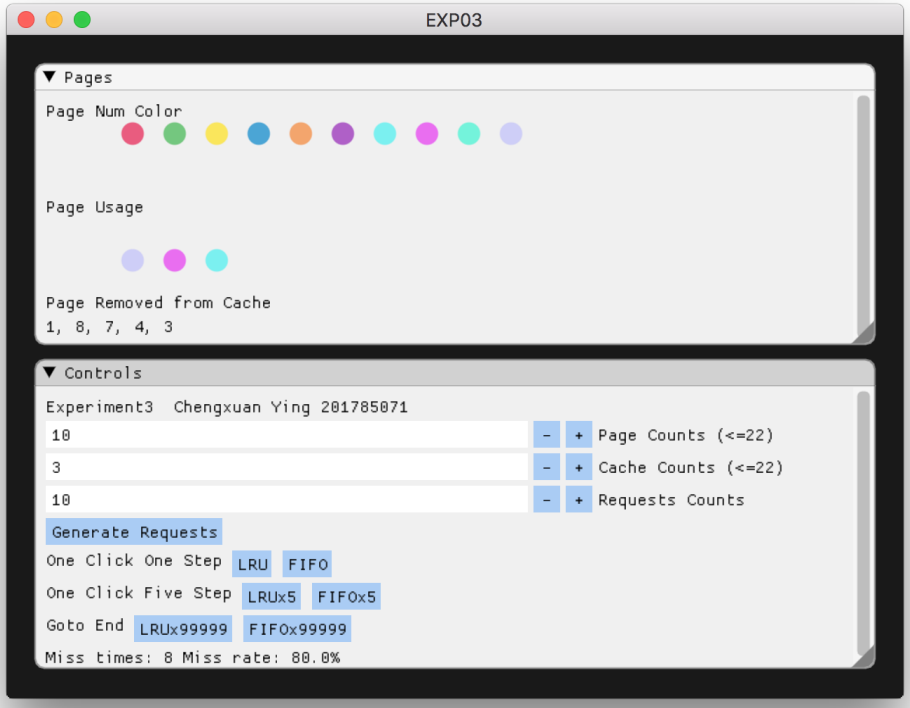
演示视频（30秒）：

https://github.com/chengsyuan/OS_Homework/blob/master/HW2/DEMO.mp4

实验3 存储管理上机作业

实验内容：
模拟页面置换算法、可视化展示过程、计算缺页率

实验结果：
主界面：



带有颜色的圆点，代表在内存中的页和磁盘中的页的对应情况。

随着用户点击对应算法的STEP，模拟器会模拟下一个时间节点的Page读入情况。
由于是动态的界面，具体使用过程请查看*演示视频*。

关键代码：

生成页面访问序列、以及类的定义：

```
class Page_Managment():
    def __init__(self, max_page=10, max_cached_page=3):
        self.max_page = max_page
        self.max_cached_page = max_cached_page
    def genReqs(self, req_cnt=10):
        self.reqs = np.random.randint(self.max_page, size=(req_cnt))
    def resetCache(self):
        self.cache = []
        self.idx = 0
        self.miss_times = 0
```

LRU实现：

```
def LRU_step(self):
    if self.idx >= len(self.reqs): # oob
        return

    # match
    for idx, i in enumerate(self.cache):
        if i['page_num'] == self.reqs[self.idx]:
            self.cache[idx]['idx'] = self.idx # update LRU
            self.idx += 1
            return

    # drop LRU
    if len(self.cache) >= self.max_cached_page:
        self.cache = sorted(self.cache, key=lambda x: x['idx'])
        self.cache = self.cache[1:]

    self.cache.append({
        'idx': self.idx,
        'page_num': self.reqs[self.idx]
     })
    self.idx += 1
    self.miss_times += 1
```

FIFO实现：

```
def FIFO_step(self):
    if self.idx >= len(self.reqs): # oob
        return

    # match
    for idx, i in enumerate(self.cache):
        if i['page_num'] == self.reqs[self.idx]:
            self.idx += 1
            return

    # drop LRU
    if len(self.cache) >= self.max_cached_page:
        self.cache = sorted(self.cache, key=lambda x: x['idx'])
        self.cache = self.cache[1:]

    self.cache.append({
        'idx': self.idx,
        'page_num': self.reqs[self.idx]
     })
    self.idx += 1
    self.miss_times += 1
```

实验结果：

在我们的实验中，使用了均匀采样生成待访问序列，然后对FIFO和LRU进行了模拟。由于待访问页号比较随机，在我们的实验中FIFO和LRU的效果区别不大。在我们之前体系结构的实验中，由于页是顺序访问的，LRU的效果会较好。

我觉得这个是我未来可以研究的一个点。

完整代码：

https://github.com/chengsyuan/OS_Homework/tree/master/HW3

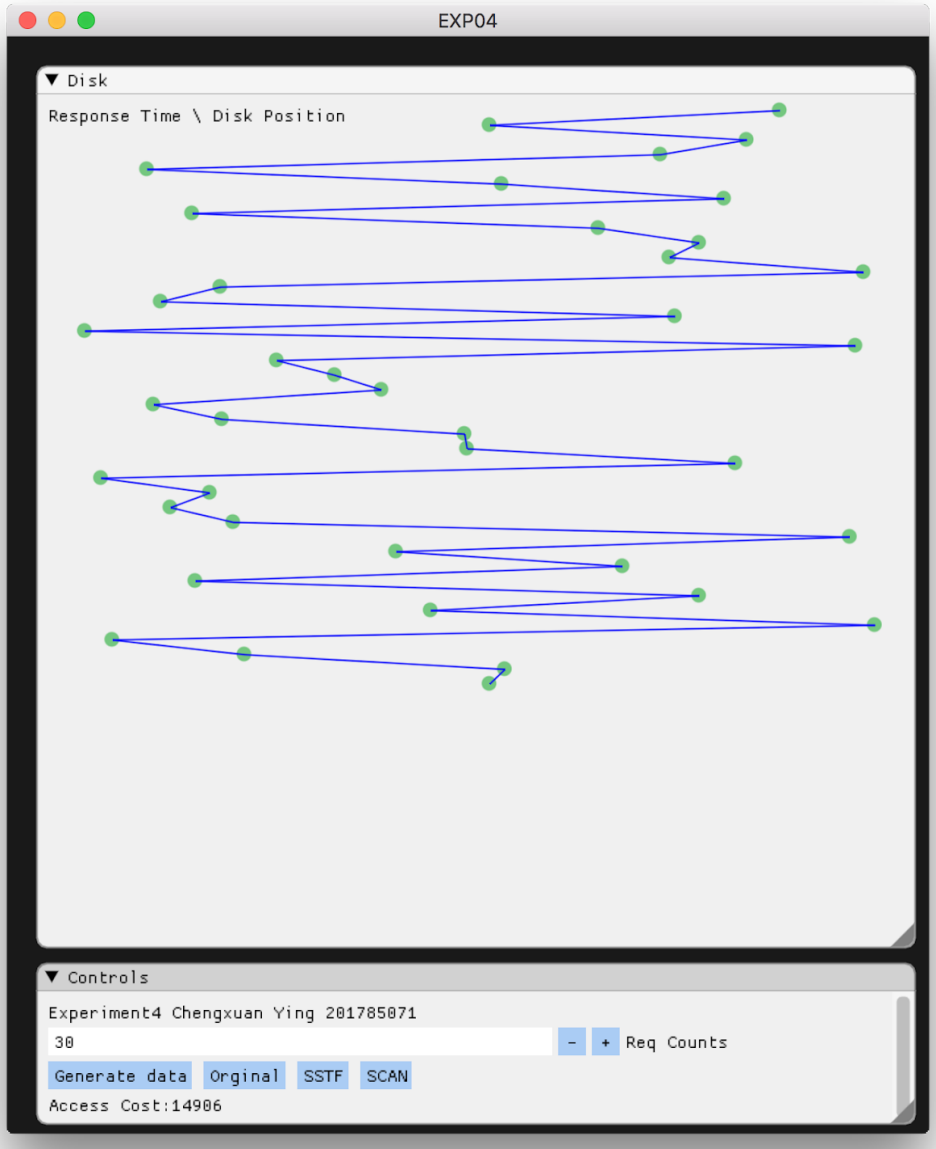
演示视频（1分钟）：

https://github.com/chengsyuan/OS_Homework/blob/master/HW3/DEMO.mp4

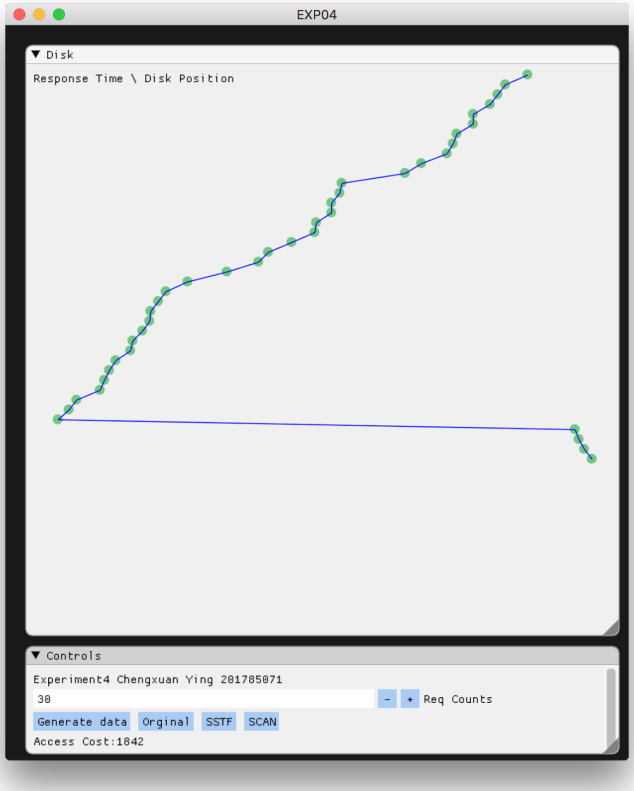
实验4 磁盘移臂调度算法实验

实验内容：
实现SSTF和SCAN算法，实现其可视化

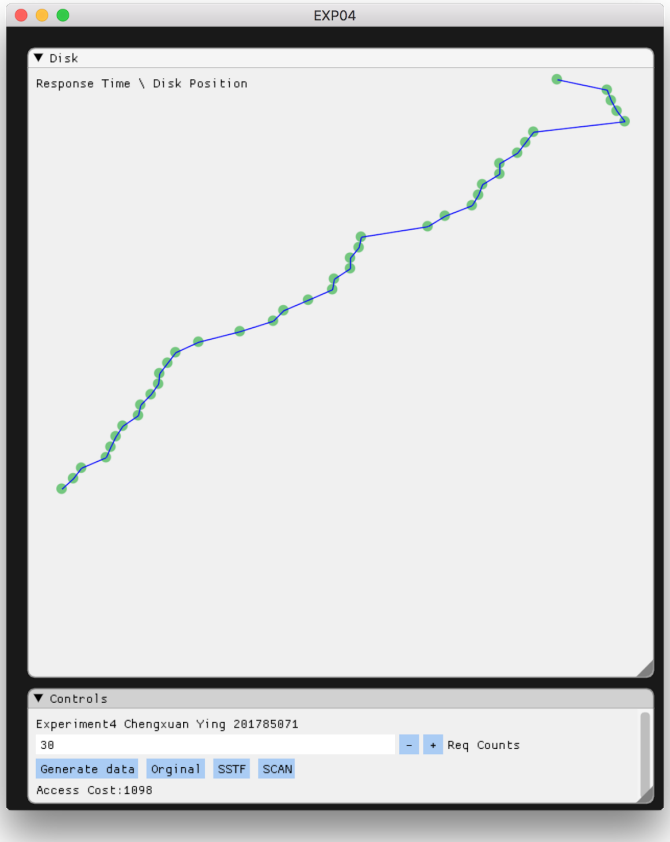
实验结果：
主界面：原始访问顺序



SSTF模式界面：


EXP04
Disk
Response Time \ Disk Position
Controls
Experiment4 Chengxuan Ying 201785071
30
Req Counts
Generate data
Orginal
SSTF
SCAN
Access Cost:1842

SCAN模式界面：


EXP04
Disk
Response Time \ Disk Position
Controls
Experiment4 Chengxuan Ying 201785071
30
Req Counts
Generate data
Orginal
SSTF
SCAN
Access Cost:1098

实验结果：

时间\方式	无调度	SSFT	SCAN
总时间	14986	1842	1098

核心代码：

生成访问序列：

```
def genReqs(tot=1000, req_cnt=10):
    return np.random.randint(tot, size=(req_cnt))
```

SSFT：

```
def SSTF(v, reqs):
    if len(reqs) == 1:
        return np.hstack([v, reqs])

    dist = abs(reqs - v)
    idx_next = np.argmin(dist)
    v_next = reqs[idx_next]
    reqs_next = np.hstack([reqs[:idx_next], reqs[idx_next+1:]])

    return np.hstack([v, SSTF(v_next, reqs_next)])
```

SCAN：

```
def SCAN(v, reqs, flag=1): #1up 0down
    reqs = np.array(sorted(reqs)).astype(np.int)
    if len(reqs) == 1:
        return np.hstack([v, reqs])

    dist = (reqs - v)
    if flag==1:
        dist[dist < 0] = 999999
    else:
        dist[dist > 0] = 999999
    dist = abs(dist)

    idx_next = np.argmin(dist)
    v_next = reqs[idx_next]
    reqs_next = np.hstack([reqs[:idx_next], reqs[idx_next+1:]])

    if flag==1 and idx_next==len(dist)-1:
        flag = 0
    elif flag==0 and idx_next==0:
        flag = 1

    return np.hstack([v, SCAN(v_next, reqs_next, flag)])
```

实验结果：

在我们的实验中，使用了均匀采样生成待访问序列，然后对三种方式进行了模拟，发现SCAN的表现由于SSFT，且优于无调度访问。

完整代码：

https://github.com/chengsyuan/OS_Homework/tree/master/HW4

演示视频（30秒）：

https://github.com/chengsyuan/OS_Homework/blob/master/HW4/DEMO.mp4

实验5 文件管理作业

实验内容：

使用**空闲表**管理磁盘空闲块，完成四个子任务：

（1） 随机生成2k-10k的文件50个，文件名为1.txt、2.txt、……、50.txt，按照上述算法存储到模拟磁盘中。

（2） 删除奇数.txt（1.txt、3.txt、……、49.txt）文件

（3） 新创建5个文件（A.txt、B.txt、C.txt、D.txt、E.txt），大小为：7k、5k、2k、9k、3.5k，按照与（1）相同的算法存储到模拟磁盘中。

（4） 给出文件A.txt、B.txt、C.txt、D.txt、E.txt的盘块存储状态和所有空闲区块的状态。

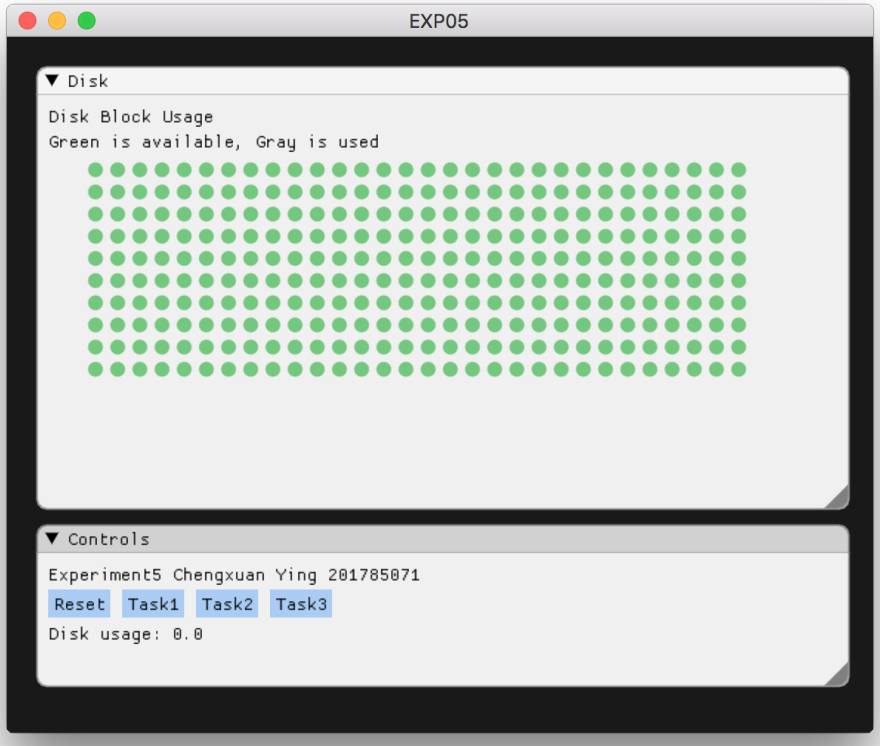
我的实验和题目的区别：

考虑到程序GUI的美观，我把涉及到个数的任务，对个数进行适当的减少，以适合可视化展示。

原来：生成500个文件、现在：生成500*0.6=300个文件。

实验结果：

主界面：

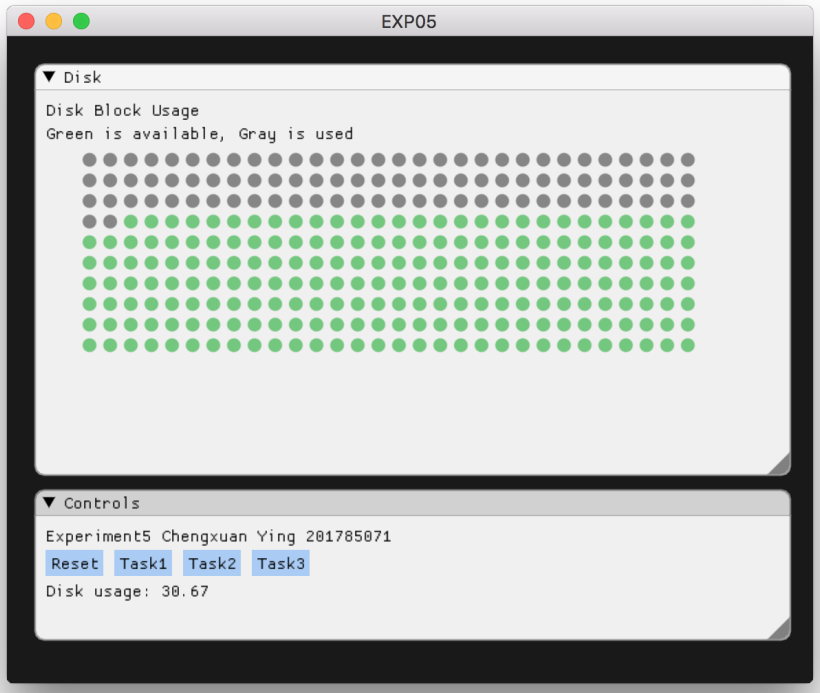


带有颜色的圆点，代表在磁盘块使用的情况。绿色代表可用，灰色代表占用，其他彩色代表被A.txt-E.txt占用的情况。

具体使用过程请查看*演示视频*。

实验结果：

Task1:



Task2:

Task3:

核心代码：

Task1:

```
def task1(self):
    for i in range(30):
        filename = str(i+1) + '.txt'
        filesize = np.random.randint(2*1024, 10*1024)
        self.addfile(filename, filesize)
```

Task2:

```
def task2(self):
    for i in range(30):
        if (i + 1) % 2 == 1:
            filename = str(i+1) + '.txt'
            self.delfile(filename)
```

Task3:

```
def task3(self):
    self.addfile('A.txt', 7*1024)
    self.addfile('B.txt', 5*1024)
    self.addfile('C.txt', 2*1024)
    self.addfile('D.txt', 9*1024)
    self.addfile('E.txt', 3.5*1024)
```

实验结论：

本次是纯模拟性的实验，且没有对照实验，因此本次试验没有结论。

完整代码：

https://github.com/chengsyuan/OS_Homework/tree/master/HW5

演示视频（30秒）：

https://github.com/chengsyuan/OS_Homework/blob/master/HW5/DEMO.mp4