# 虚拟内存作业思路分享

韩汶辰 2019.12.5

# 第一组

- 第一组要求没有page fault以及protection exception
- malloc一个给定长度的数组，首先访问一遍这个数组进行warm up，之后按照步长为page size进行读操作.
- （直接访问的话会产生**soft** page fault）
- 四组中，利用getrusage测出page fault的数量，里面会检测出soft page fault和hard page fault数量.
- 注：这一组的warm up使用步长1，并且数组足够大，来消除cache的影响。

```
arr = (char*) malloc(n * PAGE_SIZE);

for (int i = 0; i < n * PAGE_SIZE; i++){
    arr[i] = 1;
}

rusage st_usage;
getrusage(RUSAGE_SELF, &st_usage);

ll st_time = clock();
for (int i = 0; i < n; i++){
    sum += arr[i * PAGE_SIZE];
}
ll ed_time = clock();
free(arr);
```

# 第二组 & 第四组

- 第二组要求每次访问都触发page fault并访问regular file。
- 基本思想是通过mmap映射到这个文件，之后按照page size为步长访问这个文件。
- 实验发现如果按照顺序读文件的话并不会每次都产生page fault. 这里猜测是由于数据预取原则导致的。
- 所以新加了一组实验，改为写操作来产生page fault。

```
rusage st_usage;
getrusage(RUSAGE_SELF, &st_usage);

int fd = open("data.txt", O_RDONLY);
char *file_data = (char*) mmap(NULL, PAGE_SIZE * n,
PROT_READ | PROT_WRITE, MAP_FILE|MAP_PRIVATE, fd, 0);

ll st_time = clock();
for (int i = 0; i < n; i++){
    sum += file_data[i * PAGE_SIZE];
}
ll ed_time = clock();
```

# 第三组

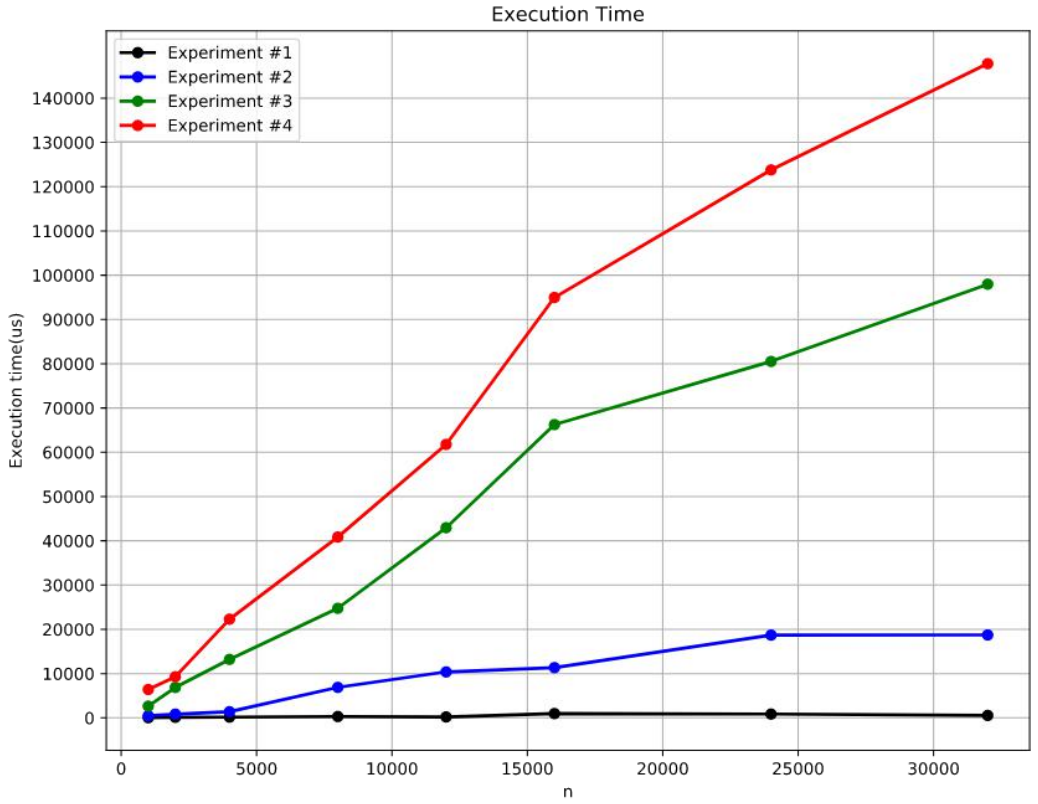
- 第三组要求每次产生protection exception并且访问anonymous file。
- 这里我的做法是先进行malloc，通过fork产生子进程，此时malloc的数据页是两个进程共享的，通过copy on write，就是父进程去写malloc的页就会产生protection exception。并且因为没有初始化，访问的是anonymous file。

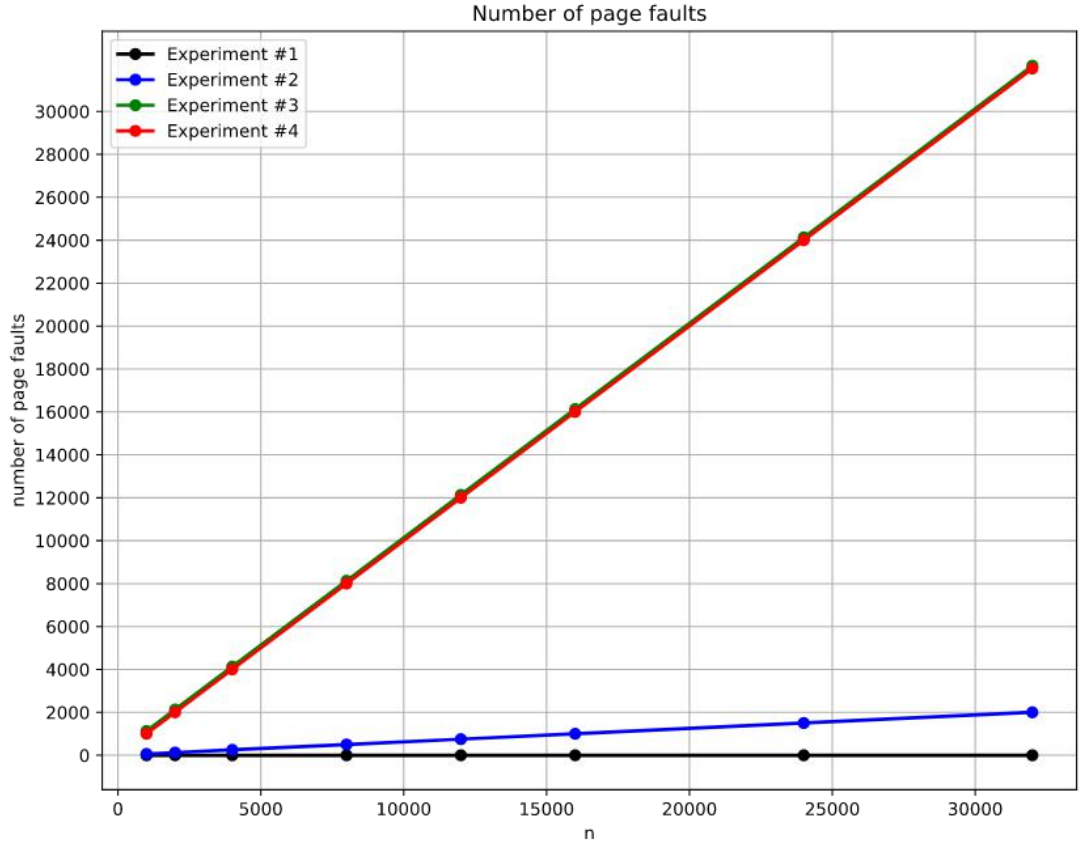
```
arr = (char*) malloc(n * PAGE_SIZE);

if ((pid = fork()) == 0){
    cont_flag = 0;
    while (!cont_flag){
        sigsuspend(&null_mask);
    }
    free(arr);
    exit(0);
}
ll st_time = clock();
for (int i = 0; i < n; i++){
    arr[i * PAGE_SIZE] = 1;
}
ll ed_time = clock();
free(arr);
```

# 我的预期

- 第一组应该是最快的，因为没有产生page fault。
- 第二组和第四组预期最慢，我们预期每次都会产生hard
- page fault，涉及磁盘io。
- 第三组只会产生soft page fault，速度应该介于中间。

# 实际结果（hard page fault数量为0）



# 反常现象1及解释

- 我们发现第2，4组的运行时间并没有像预期一样，有很慢的运行速度，数量级明显不一样。
- 使用iotop命令查看磁盘io，发现实验2(读操作)没有产生磁盘io。
- getrusage返回的硬缺页数量也为0。
- 重新启动第一次运行，会产生明显的磁盘io流量，之后就不会产生了。
- 这个应该是page cache机制导致的。
- page cache就是将磁盘读取的内容存储在内存中，并且只有内存不足时才会清除page cache；磁盘写操作直接写在内存，内核会周期性地将这些page更新到磁盘。

# 反常现象2及解释

- 第2组并没有每次都产生page fault，产生page fault数量大约是$\frac{1}{16}$.
- 可能是由于数据预取导致的吧，而改成写操作就会每次都产生page fault了。

# 其他现象

- 第一组相比于后几组要快很多，但后几组并未产生hard page fault。
- 运行时间与产生page fault数量正相关(观察第二组)。
- 可能是由于产生page fault，切换到内核来处理page fault需要花费时间导致的。