OS 第一次作业

> **4-1．用以下标志运行程序：./process-run.py -l 5:100,5:100。CPU 利用率（CPU 使用时间的百分比）应该是多少？为什么你知道这一点？利用 -c 标记查看你的答案是否正确。**

在创建进程时，`5:100`表示该进程有5个指令，且每个指令使用CPU的概率为100%。这里创建的两个进程都是如此。所以CPU利用率应该是100%

运行指令结果如下：

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-4-intro$ ./process-
run.py -l 5:100,5:100
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu
  cpu

Process 1
  cpu
  cpu
  cpu
  cpu
  cpu

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)
```

加上`-c`和`-p`后进行验证，运行结果如下：

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-4-intro$ ./process-
run.py -l 5:100,5:100 -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2        RUN:cpu         READY             1
  3        RUN:cpu         READY             1
  4        RUN:cpu         READY             1
  5        RUN:cpu         READY             1
```

```
  6           DONE       RUN:cpu             1
  7           DONE       RUN:cpu             1
  8           DONE       RUN:cpu             1
  9           DONE       RUN:cpu             1
 10           DONE       RUN:cpu             1

Stats: Total Time 10
Stats: CPU Busy 10 (100.00%)
Stats: IO Busy  0 (0.00%)
```

可以看到：10个时间周期中，CPU一直在运行，所以CPU利用率为100% 。说明答案正确。

> **4-2．现在用这些标志运行：./process-run.py -l 4:100,1:0。这些标志指定了一个包含 4 条指令的进程（都要使用 CPU），并且只是简单地发出 I/O 并等待它完成。完成这两个进程需要 多长时间？利用-c 检查你的答案是否正确。**

一共创建了2个进程。进程 0 包括 4 个指令，且每个指令使用CPU概率为100%；进程 1 包括 1 个指令，且每个指令使用CPU概率为0，意味着每个指令都会是IO指令。

对于IO指令，在README.md 文档中有 2 点注意事项：

- 进程执行 1 次IO指令分为 3 个阶段 `RUN:io` ，`BLOCKED` ，`RUN: io_end` 。其中前后两阶段要占用CPU各 1 个时钟周期，只用中间的阶段才会占用IOs 。
- `BLOCKED` 阶段的持续时间由参数 `-L IO_LENGTH` 决定，但本题并没有给出此参数。

下面给出我的猜测：进程0会的第一个指令时CPU指令，进程1的第一个指令第一时间也需要占用CPU，但是进程0的CPU使用概率为100%。所以一开始，进程0为 `RUNNING`，进程1为 `READY`，持续 4 个周期。进程0运行完所有指令，进入 `DONE` ，进程1进入 `RUN: io` ，持续 1 个周期 。进程0继续保持 `DONE` ，进程 1 进入 `BLOCK` ,此时没有进程需要CPU，由于没有给出持续时间，这里假设为 5 个周期 。最后进程 1 调用CPU进入 `RUN:io_done` ，持续 1 个周期 。共计 4 + 1 + 5 + 1 = 11个时钟周期。

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-4-intro$ ./process-
run.py -l 4:100,1:0
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu

Process 1
```

```
  io
  io_done

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)
```

加上 `-c` 和 `-p`后进行验证，运行结果如下：

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-4-intro$ ./process-
run.py -l 4:100,1:0 -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1        RUN:cpu         READY             1
  2        RUN:cpu         READY             1
  3        RUN:cpu         READY             1
  4        RUN:cpu         READY             1
  5           DONE        RUN:io             1
  6           DONE       BLOCKED                           1
  7           DONE       BLOCKED                           1
  8           DONE       BLOCKED                           1
  9           DONE       BLOCKED                           1
 10           DONE       BLOCKED                           1
 11*          DONE   RUN:io_done             1

Stats: Total Time 11
Stats: CPU Busy 6 (54.55%)
Stats: IO Busy  5 (45.45%)
```

我们可以发现猜测正确。

> **4-3．现在交换进程的顺序：./process-run.py -l 1:0,4:100。现在发生了什么？交换顺序是否重要？为什么？同样，用-c 看看你的答案是否正确。**

猜测：交换顺序后什么都没有改变，先调用CPU的一定是概率为100%的那一个。

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-4-intro$ ./process-
run.py -l 1:0,4:100
Produce a trace of what would happen when you run these processes:
Process 0
  io
  io_done
```

```
Process 1
  cpu
  cpu
  cpu
  cpu

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)
```

加上 `-c` 和 `-p`后进行验证，运行结果如下：

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-4-intro$ ./process-
run.py -l 1:0,4:100 -c -p
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        BLOCKED       RUN:cpu             1             1
  3        BLOCKED       RUN:cpu             1             1
  4        BLOCKED       RUN:cpu             1             1
  5        BLOCKED       RUN:cpu             1             1
  6        BLOCKED          DONE                           1
  7*   RUN:io_done          DONE             1

Stats: Total Time 7
Stats: CPU Busy 6 (85.71%)
Stats: IO Busy  5 (71.43%)
```

这里我们发现结果和猜想大相径庭。指令中进程的顺序代表执行顺序，而这个顺序至关重要。交换顺序后，一个进程在两个进程阻塞时占用CPU。执行总时长、CPU利用率、IO利用率都提高了很多。大大提高了OS的效率。

> **5-1．编写一个调用 fork()的程序。谁调用 fork()之前，让主进程访问一个变量（例如 x） 并将其值设置为某个值（例如 100）。子进程中的变量有什么值？当子进程和父进程都改变 x 的值谁，变量会发生什么？**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

```
    int demo = 0;
    demo = 1;
    int rc = fork();
    if (rc < 0) { //fork 失败
        printf("fork failed");
        exit(1);
    } else if (rc == 0) { //子进程
        printf("这里是子进程，  修改前demo = %d\n", demo);
        demo = getpid();
        printf("这里是子进程，  修改后demo = %d\n", demo);

    } else { //父进程
        printf("这里是父进程，  修改前demo = %d\n", demo);
        demo = getpid();
        printf("这里是父进程，  修改后demo = %d\n", demo);
    }

    return 0;
}
```

执行后结果为：

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-5-api$ ./q1
这里是父进程，  修改前demo = 1
这里是父进程，  修改后demo = 3539
这里是子进程，  修改前demo = 1
这里是子进程，  修改后demo = 3540
```

我们可以知道：子进程相当于把父进程中的变量完全复制一份，并从 `fork()` 处开始执行。在 `fork()` 之后父进程的变量 `demo` 也会复制一份存在子进程中，并且值与父进程的相同 。之后两个进程对各自存储的 `demo` 值进行更改，是互不影响的。

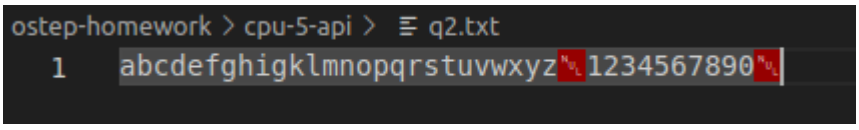
> **5-2．编写一个打开文件的程序（使用 open()系统调用），然后调用 fork()创建一个新进程。子进程和父进程都可以访问 open()返回的文件描述符吗？当它们并发（即同时）写入文件时， 会发生什么？**

执行下面的代码：

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <fcntl.h>


int main() {

    int fd = open("./q2.txt", O_WRONLY | O_CREAT | O_APPEND);

    if (fd == -1) {
        printf("open failed");
        exit(1);
    }

    int rc = fork();
    if (rc < 0) {
        perror("fork");
        exit(0);
    } else if (rc == 0) {  // 子进程
        char str1[]  = "1234567890";

        write(fd, str1, sizeof(str1));
    } else {  // 父进程
        char str2[]  = "abcdefghigklmnopqrstuvwxyz";

        write(fd, str2, sizeof(str2));

    }

    close(fd);
    return 0;
}
```

q2.txt文件被写入后：



可以发现，父进程先执行，子进程再执行，二者并不冲突。

> **5-4．编写一个调用 fork()的程序，然后调用某种形式的 exec()来运行程序/bin/ls。看看是 否可以尝试 exec()的所有变体，包括 execl()、execle()、execlp()、execv()、execvp()和 execvP()。为什么同样的基本调用会有这么多变种?**

查阅 `Linux manual page` 我们可以得知 `exec` 的各种变体的定义及描述。

```
       int execl(const char *pathname, const char *arg, ...
                       /* (char  *) NULL */);
       int execlp(const char *file, const char *arg, ...
                       /* (char  *) NULL */);
       int execle(const char *pathname, const char *arg, ...
                       /*, (char *) NULL, char *const envp[] */);
       int execv(const char *pathname, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
       int execvpe(const char *file, char *const argv[],
                       char *const envp[]);
```

- **l - execl(), execlp(), execle()**
  const char *arg* *和后续的省略号可以被认为是 arg0、arg1、...、argn。它们一起描述了一个或多个指向空终止字符串的指针的列表，**这些指针表示可用于执行的程序的参数列表。**按照惯例，**第一个参数应该指向与正在执行的文件关联的文件名。参数列表必须以空指针终止**，并且由于这些是可变参数函数，因此该指针必须强制转换 (char*) NULL。
- **v - execv(), execvp(), execvpe()**
  参数是指向以 null 结尾的字符串的指针数组，这些字符串表示新程序可用的参数列表。按照惯例，第一个参数应该指向与正在执行的文件关联的文件名。指针数组必须以空指针终止。
- **e - execle(), execvpe()**
  新进程映像的环境通过参数 envp 指定。 envp 参数是指向以 null 结尾的字符串的指针数组，并且必须以 null 指针结尾。
- **p - execlp(), execvp(), execvpe()**
  如果指定的文件名不包含斜杠 (/) 字符，这些函数将重复 shell 搜索可执行文件的操作。在 PATH 环境变量中指定的以冒号分隔的目录路径名列表中查找该文件。如果未定义此变量，则路径列表默认为包含 confstr(_CS_PATH) 返回的目录（通常返回值“/bin:/usr/bin”）的列表，也可能包含当前工作目录

这里我们尝试运行：

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>

int main()
{
        char * s = "/bin/ls";
        char * s1 = "ls";
        char * s2 = "/";
```

```
        char * str_vec[] = { s1, s2, NULL };
        for(int i = 0;i < 6; ++i) {
                int rc = fork();
                if (rc < 0) {
                        printf("fork failed");
                        exit(1);
                } else if (rc == 0) {
                        switch(i) {
                        case 0:
                                execl(s, s1, s2, NULL);
                                break;
                        case 1:
                                execle(s, s1, s2, NULL);
                                break;
                        case 2:
                                execlp(s, s, s2, NULL);
                                break;
                        case 3:
                                execv(s, str_vec);
                                break;
                        case 4:
                                execvp(s1, str_vec);
                                break;
                        case 5:
                                execvpe(s1, str_vec);
                                break;
                        default: break;
                        }
                } else {
                        wait(NULL);
                }
        }
        return 0;
}
```

得到结果：

```
dinghaitong@dinghaitong-VirtualBox:~/os_ws/ostep-homework/cpu-5-api$ ./q4
bin   cdrom  etc   lib    lib64   lost+found  mnt  proc  run   snap  swapfile  tmp
var
boot  dev    home  lib32  libx32  media       opt  root  sbin  srv   sys       usr
bin   cdrom  etc   lib    lib64   lost+found  mnt  proc  run   snap  swapfile  tmp
var
boot  dev    home  lib32  libx32  media       opt  root  sbin  srv   sys       usr
```

```
bin   cdrom  etc   lib    lib64   lost+found  mnt  proc  run   snap  swapfile  tmp
var
boot  dev    home  lib32  libx32  media       opt  root  sbin  srv   sys       usr
bin   cdrom  etc   lib    lib64   lost+found  mnt  proc  run   snap  swapfile  tmp
var
boot  dev    home  lib32  libx32  media       opt  root  sbin  srv   sys       usr
bin   cdrom  etc   lib    lib64   lost+found  mnt  proc  run   snap  swapfile  tmp
var
boot  dev    home  lib32  libx32  media       opt  root  sbin  srv   sys       usr
bin   cdrom  etc   lib    lib64   lost+found  mnt  proc  run   snap  swapfile  tmp
var
boot  dev    home  lib32  libx32  media       opt  root  sbin  srv   sys       usr
```

`exec`系统调用之所以会有多种变体（如`execl`、`execle`、`execlp`、`execv`、`execvp`和`execvP`等），是为了适应不同的编程需求和方便程序员使用。每种变种提供了不同的参数传递机制，主要区别在于如何指定命令路径、如何传递参数和环境变量。

> 7-1．使用 SJF 和 FIFO 调度程序运行长度为 200 的 3 个作业时，计算响应时间和周转 时间。

假设有三个任务ABC，三者的运行长度都是200，且同时到达。

- FIFO：
  - 周转时间 T1 = (200 + 400 + 600) / 3 = 400
  - 响应时间 T2 = (0 + 200 + 400) / 3 = 200
- SJF：
  - 周转时间 T1 = (200 + 400 + 600) / 3 = 400
  - 响应时间 T2 = (0 + 200 + 400) / 3 = 200

> 7-2．现在做同样的事情，但有不同长度的作业，即 100、200 和 300。

1. 这里假设三个任务ABC的运行长度分别是100、200、300，且**按照ABC的顺序执行。**
   - FIFO：
     - 周转时间 T1 = (100 + 300 + 600) / 3 = 333.3
     - 响应时间 T2 = (0 + 100 + 300) / 3 = 133.3
   - SJF：
     - 周转时间 T1 = (100 + 300 + 600) / 3 = 333.3
     - 响应时间 T2 = (0 + 100 + 300) / 3 = 133.3
2. 这里假设三个任务ABC的运行长度分别是100、200、300，且**按照CBA的顺序执行。**

- FIFO：
  - 周转时间 T1 = (300 + 500 + 600) / 3 = 466.6
  - 响应时间 T2 = (0 + 300 + 500) / 3 = 266.6
- SJF：
  - 周转时间 T1 = (100 + 300 + 600) / 3 = 333.3
  - 响应时间 T2 = (0 + 100 + 300) / 3 = 133.3

这里可以看出 SJF 策略性能优于 FIFO 策略。

> 7-3．现在做同样的事情，但采用 RR 调度程序，时间片为 1。

RR：

- 周转时间 T1 = (598 + 599 + 600) / 3 = 599
- 响应时间 T2 = (0 + 1 + 2) / 3 = 1

可以看到 RR 策略的响应时间极短，但是周转时间极长。这说明***性能和公平在调度系统中往往是矛盾的***。