OSTEP Notes

1 目录

2 C2 操作系统介绍

code（下附）需到ubuntu中跑，其中common.h文件不包含在gcc库中，需要自己手动输入命令参数`-I .h文件路径`（例如我的：`-I ~/myfiles/OSTEP/head_files/`)或者配置环境变量，才可以执行

输入命令：

```
hcs@ubuntu:~/myfiles/OSTEP$ gcc -o cpu cpu.c -Wall -I
~/myfiles/OSTEP/head_files/
hcs@ubuntu:~/myfiles/OSTEP$ ./cpu "A"
A
A
A
A
^C
hcs@ubuntu:~/myfiles/OSTEP$
```

接下来运行同一个程序的不同实例时出现如下错误：

```
hcs@ubuntu:~/myfiles/OSTEP$ ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
bash: syntax error near unexpected token `;'
```

尝试把分号去掉，使用&同时运行多个程序，结果程序执行根本停不下来（这是为什么？？？），勿试

`./cpu A & ./cpu B & ./cpu C & ./cpu D &`

```
hcs@ubuntu:~/myfiles/OSTEP$ ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 3341
[2] 3342
[3] 3343
[4] 3344
hcs@ubuntu:~/myfiles/OSTEP$ A
B
C
D
A
A
^C
hcs@ubuntu:~/myfiles/OSTEP$ B
^C
hcs@ubuntu:~/myfiles/OSTEP$ C
^C
hcs@ubuntu:~/myfiles/OSTEP$ D
^C
hcs@ubuntu:~/myfiles/OSTEP$ A
B
^C
hcs@ubuntu:~/myfiles/OSTEP$ ^C
hcs@ubuntu:~/myfiles/OSTEP$ ^C
hcs@ubuntu:~/myfiles/OSTEP$ C
^C
hcs@ubuntu:~/myfiles/OSTEP$ D
^C
hcs@ubuntu:~/myfiles/OSTEP$ A
B
```

ctrl+c也无法停止，程序不断执行，强制只好关闭

配置环境自动添加头文件路径：

`C_INCLUDE_PATH=/home/hcs/myfiles/OSTEP/head_files:/MyLib`

`export C_INCLUDE_PATH`

附上参考网址

或者可以直接将`common.h`文件直接复制到到`usr/include/`目录下，则不用输入命令且不用配置环境，该目录为gcc库函数目录（网上方法，没有亲测）

2.1 code

2.1.1 cpu.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int main(int argc, char *argv[])
{
//  argc = 2;
    if(argc != 2)
    {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while(1)
    {
        Spin(1);
        printf("%s\n",str);
    }

    return 0;
}
```

2.1.2 common.h

```c
#ifndef __common_h__
#define __common_h__

#include <sys/time.h>
#include <sys/stat.h>
#include <assert.h>

double GetTime() {
    struct timeval t;
    int rc = gettimeofday(&t, NULL);
    assert(rc == 0);
    return (double) t.tv_sec + (double) t.tv_usec/1e6;
}

void Spin(int howlong) {
    double t = GetTime();
    while ((GetTime() - t) < (double) howlong)
    ; // do nothing in loop
}
```

```
20
21 #endif // __common_h__
```

3 第4章 抽象：进程

操作系统的最基本抽象：进程。它很简单地被视为一个正在运行的程序。

P26 作业 补充：模拟作业的process-run.py程序下载：张慕晖的博客、github（已经翻译成中文）

跑该程序的详细过程查看README文件.（I/O默认花费5个时间片，也可自行定义，如`-L 6`，花费6个时间片）

问题：

3.1 问题1：

1、用以下参数运行程序：`./process-run.py -l 5:100,5:100`。CPU利用率（CPU处于使用状态的时间比例）是多少？为什么？用参数`-c`和`-p`验证你的回答的正确性。

应该是100%，因为进程0会运行5个时间片，然后进程1接着马上运行5个时间片，进程没有发起I/O请求，CPU也没有空闲时间；`./process-run.py -l 5:100,5:100 -c`

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -l 5:100,5:100 -c
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
```

3.2 问题2：

2、用以下参数运行程序：`./process-run.py -l 4:100,1:0`。这些参数给定了两个进程，其中一个包含4条CPU指令，另一个只发出一个I/O请求并等待请求结束。两个进程都结束执行需要多长时间？用参数`-c`和`-p`验证你的回答的正确性。

CPU：4（4条指令，4个时间片）；
IO发出请求：1个时间片；
I/O请求执行完毕：5个时间片；
共4+1+5=10个时间片

（结果参考了运行结果，因为我不知道所谓的“I/O请求花费5个时间片”算不算发出请求的指令的时间……当然从情理上和实验上来说都是不算的）：这个是某个博主的想法，我认为不对，当然做到第4题回头思考，觉得不对劲，我认为是“I/O请求花费5个时间片”是算在默认的5个时间片内的，最后一个时间片10*，个人思考认为是独立的一个时间片，用来确认I/O是否已经DONE的。

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./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    WAITING                    1
  7        DONE    WAITING                    1
  8        DONE    WAITING                    1
  9        DONE    WAITING                    1
 10*       DONE       DONE

Stats: Total Time 10
Stats: CPU Busy 5 (50.00%)
Stats: IO Busy  4 (40.00%)
```

3.3 问题3：

3、现在切换进程的顺序：`./process-run.py -l 1:0,4:100`。切换顺序对于结束执行的时间有影响吗？继续用参数`-c`和`-p`验证你的回答的正确性。

表 4.2　　跟踪进程状态：CPU 和 I/O

时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	Process0 发起 I/O
4	阻塞	运行	Process0 被阻塞
5	阻塞	运行	所以 Process1 运行
6	阻塞	运行	
7	就绪	运行	I/O 完成
8	就绪	运行	Process1 现在完成
9	运行	—	
10	运行	—	Process0 现在完成

P23，从表4.2中找答案，可以看出当PID0发起I/O时，PID0被阻塞，CPU切换到PID1开始执行，PID0在PID1运行的过程中，由于进程1还未结束，而I/O已经结束，只能进入就绪状态，等待进程1的完成，才能开始执行进程0.

因此，此题，进程0发起I/O请求，花费1个时间片，由于进程0发起I/O请求，所以，切换到进程1利用CPU，当I/O完成时（花费5个时间片），进程2也已完成，共花费6个时间片.

交换进程后，总时间会减少，cpu和IO的利用率都会提高；交换顺序很重要，增加了并行性，可以增加各种设备的利用率.

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -l 1:0,4:100 -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING    RUN:cpu          1          1
  3     WAITING    RUN:cpu          1          1
  4     WAITING    RUN:cpu          1          1
  5     WAITING    RUN:cpu          1          1
  6*       DONE       DONE

Stats: Total Time 6
Stats: CPU Busy 5 (83.33%)
Stats: IO Busy  4 (66.67%)
```

3.4 问题4：

4、下面我们探索一下其他的参数。参数`-S`指定了进程发出I/O请求时系统的反应策略。当该参数的值为`SWITCH_ON_END`时，系统不会在当前进程发出I/O请求时切换到另一个进程，而是等待进程结束之后再切换。如果你用以下参数运行程序（`-l 1:0,4:100 -c -S SWITCH_ON_END`），会发生什么?

当前进程发起I/O请求，花费1个时间片，由于没有切换到另外一个进程，I/O将执行5个时间片。I/O执行完成后，切换到另外一个进程，4个指令，花费4个时间片完成

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -l 1:0,4:100 -c -S
SWITCH_ON_END -p
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING      READY                    1
  3     WAITING      READY                    1
  4     WAITING      READY                    1
  5     WAITING      READY                    1
  6*       DONE    RUN:cpu          1
  7        DONE    RUN:cpu          1
  8        DONE    RUN:cpu          1
  9        DONE    RUN:cpu          1

Stats: Total Time 9
Stats: CPU Busy 5 (55.56%)
Stats: IO Busy  4 (44.44%)
```

3.5 问题5：

5、现在把`-S`参数的值置为SWITCH_ON_IO，此时只要进程发出I/O请求，就会切换到别的进程。（参数为`-l 1:0,4:100 -c -S SWITCH_ON_IO`）。那么，会发生什么呢？用参数`-c`和`-p`验证你的回答的正确性。

这一问其实就是第3问。(`-S SWITCH_ON_IO`似乎是默认值)

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -l 1:0,4:100 -c -S SWITCH_ON_IO -p
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING    RUN:cpu          1          1
  3     WAITING    RUN:cpu          1          1
  4     WAITING    RUN:cpu          1          1
  5     WAITING    RUN:cpu          1          1
  6*       DONE       DONE

Stats: Total Time 6
Stats: CPU Busy 5 (83.33%)
Stats: IO Busy  4 (66.67%)
```

3.6 问题6：

6、I/O请求结束时系统的执行策略也很重要。如果将参数`-I`的值置为`IO_RUN_LATER`，则**I/O请求完成时，发出请求的进程不会立刻开始执行，当前运行中的进程会继续运行。**如果使用以下参数组合，会发生什么？（`./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p`）

在做到这个问题之前，我一直在纠结I/O请求到底算不算在默认的5个时间片内，经过第二题和第四题的比较，我认为是算在内的，所以还是继续按照这个分析来进行计算。

PID0发出I/O请求并执行（1个时间片+4个时间片），PID0进入阻塞状，PID1、2、3依次执行（5+5+5=15个时间片），PID1、2、3运行完成后，切换回PID0，发出I/O请求并执行（1+4=5个时间片*2==>共10个时间片），最后确认I/O是否完成，DONE（单独的1个时间片）,其中PID0的第一次读写和PID1并行，需减去4个时间片：

(1+4) + 5x3 + (1+4)x2 + 1 - 4 = 27个时间片

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p
Time     PID: 0     PID: 1     PID: 2     PID: 3        CPU        IOs
  1      RUN:io      READY      READY      READY          1
  2     WAITING    RUN:cpu      READY      READY          1          1
  3     WAITING    RUN:cpu      READY      READY          1          1
  4     WAITING    RUN:cpu      READY      READY          1          1
  5     WAITING    RUN:cpu      READY      READY          1          1
  6*      READY    RUN:cpu      READY      READY          1
  7       READY       DONE    RUN:cpu      READY          1
  8       READY       DONE    RUN:cpu      READY          1
  9       READY       DONE    RUN:cpu      READY          1
 10       READY       DONE    RUN:cpu      READY          1
 11       READY       DONE    RUN:cpu      READY          1
 12       READY       DONE       DONE    RUN:cpu          1
 13       READY       DONE       DONE    RUN:cpu          1
 14       READY       DONE       DONE    RUN:cpu          1
 15       READY       DONE       DONE    RUN:cpu          1
 16       READY       DONE       DONE    RUN:cpu          1
 17      RUN:io       DONE       DONE       DONE          1
 18     WAITING       DONE       DONE       DONE                    1
 19     WAITING       DONE       DONE       DONE                    1
 20     WAITING       DONE       DONE       DONE                    1
 21     WAITING       DONE       DONE       DONE                    1
 22*     RUN:io       DONE       DONE       DONE          1
```

```
 23     WAITING       DONE       DONE       DONE                    1
 24     WAITING       DONE       DONE       DONE                    1
 25     WAITING       DONE       DONE       DONE                    1
 26     WAITING       DONE       DONE       DONE                    1
 27*       DONE       DONE       DONE       DONE

Stats: Total Time 27
Stats: CPU Busy 18 (66.67%)
Stats: IO Busy  12 (44.44%)
```

3.7 问题7：

7、将参数`-I`的值换成`IO_RUN_IMMEDIATE`，重新执行上述命令，此时，**当I/O请求完成时，发出请求的进程会立刻抢占CPU**。现在程序的运行结果有何不同？为什么让刚刚执行完I/O的进程立刻开始运行可能是个好主意?

就像上一题所分析的那样，如果把I/O请求分散开来，则可以提高设备的利用率。在设计调度策略的时候应当考虑到进程的I/O密集程度，这个思路见于多级反馈队列调度算法中——如果进程用完了时间片则下移一个队列；否则，如果在时间片结束之前就发出了I/O请求，则保留在当前队列中。此时花费的总时间就是3条I/O指令加上15条CPU指令的时间。

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -l
3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_IMMEDIATE -c -p
Time     PID: 0     PID: 1     PID: 2     PID: 3        CPU        IOs
  1      RUN:io      READY      READY      READY          1
  2     WAITING    RUN:cpu      READY      READY          1          1
  3     WAITING    RUN:cpu      READY      READY          1          1
  4     WAITING    RUN:cpu      READY      READY          1          1
  5     WAITING    RUN:cpu      READY      READY          1          1
  6*     RUN:io      READY      READY      READY          1
  7     WAITING    RUN:cpu      READY      READY          1          1
  8     WAITING       DONE    RUN:cpu      READY          1          1
  9     WAITING       DONE    RUN:cpu      READY          1          1
 10     WAITING       DONE    RUN:cpu      READY          1          1
 11*     RUN:io       DONE      READY      READY          1
 12     WAITING       DONE    RUN:cpu      READY          1          1
 13     WAITING       DONE    RUN:cpu      READY          1          1
 14     WAITING       DONE       DONE    RUN:cpu          1          1
 15     WAITING       DONE       DONE    RUN:cpu          1          1
 16*       DONE       DONE       DONE    RUN:cpu          1
 17        DONE       DONE       DONE    RUN:cpu          1
 18        DONE       DONE       DONE    RUN:cpu          1

Stats: Total Time 18
Stats: CPU Busy 18 (100.00%)
Stats: IO Busy  12 (66.67%)
```

3.8 问题8：

8、用下列随机生成的参数组合运行程序，比如`-s 1 -l 3:50,3:50`，`-s 2 -l 3:50,3:50`和`-s 3 -l 3:50,3:50`。你能否预测程序运行结果？将`-I`参数的值分别置为`IO_RUN_IMMEDIATE`和`IO_RUN_LATER`时，运行结果有何区别？将`-S`参数的值分别置为`SWITCH_ON_IO`和`SWITCH_ON_END`时，运行结果有何区别？8.

8.1 输入`./process-run.py -s 1 -l 3:50,3:50`

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 1 -l 3:50,3:50
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  io
  io

Process 1
  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)
```

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 1 -l 3:50,3:50 -c
-p
Time     PID: 0     PID: 1        CPU        IOs
  1     RUN:cpu      READY          1
  2      RUN:io      READY          1
  3     WAITING    RUN:cpu          1          1
  4     WAITING    RUN:cpu          1          1
  5     WAITING    RUN:cpu          1          1
  6     WAITING       DONE                    1
  7*     RUN:io       DONE          1
  8     WAITING       DONE                    1
  9     WAITING       DONE                    1
 10     WAITING       DONE                    1
 11     WAITING       DONE                    1
 12*       DONE       DONE

Stats: Total Time 12
Stats: CPU Busy 6 (50.00%)
Stats: IO Busy  8 (66.67%)
```

输入附加参数`-I IO_RUN_LATER`时，结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 1 -l 3:50,3:50 -I
IO_RUN_LATER -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1     RUN:cpu      READY          1
  2      RUN:io      READY          1
  3     WAITING    RUN:cpu          1          1
  4     WAITING    RUN:cpu          1          1
  5     WAITING    RUN:cpu          1          1
  6     WAITING       DONE                    1
  7*     RUN:io       DONE          1
  8     WAITING       DONE                    1
  9     WAITING       DONE                    1
 10     WAITING       DONE                    1
 11     WAITING       DONE                    1
 12*       DONE       DONE
```

```
15
16 Stats: Total Time 12
17 Stats: CPU Busy 6 (50.00%)
18 Stats: IO Busy  8 (66.67%)
```

但是，为什么会是这样呢？可能是因为进程数太少了？？？

输入参数`-I IO_RUN_IMMEDIATE`或者`-S SWITCH_ON_IO`时，运行结果和`./process-run.py -s 1 -l 3:50,3:50 -c -p`运行结果相同

`-I IO_RUN_IMMEDIATE`时结果相同，因为Process0发出的I/O请求在Process1完全执行结束之后才完成：

`-S SWITCH_ON_IO`时结果也相同，因为这个是默认值。

`-S SWITCH_ON_END`时运行时间变长，如下：

```
1  hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 1 -l 3:50,3:50 -S
   SWITCH_ON_END -c -p
2  Time     PID: 0     PID: 1        CPU        IOs
3    1     RUN:cpu      READY          1
4    2      RUN:io      READY          1
5    3     WAITING      READY                    1
6    4     WAITING      READY                    1
7    5     WAITING      READY                    1
8    6     WAITING      READY                    1
9    7*     RUN:io      READY          1
10   8     WAITING      READY                    1
11   9     WAITING      READY                    1
12  10     WAITING      READY                    1
13  11     WAITING      READY                    1
14  12*       DONE    RUN:cpu          1
15  13        DONE    RUN:cpu          1
16  14        DONE    RUN:cpu          1
17
18 Stats: Total Time 14
19 Stats: CPU Busy 6 (42.86%)
20 Stats: IO Busy  8 (57.14%)
```

时间变长

8.2、使用参数`-s 2 -l 3:50,3:50`，输出如下：

```
1  hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 2 -l 3:50,3:50
2  Produce a trace of what would happen when you run these processes:
3  Process 0
4    io
5    io
6    cpu
7
8  Process 1
9    cpu
10   io
11   io
```

```
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)
```

`-I IO_RUN_LATER`时，输出如下，出现了两个进程同时I/O的情况（不过此时我们认为I/O是可以并行的，不存在等待I/O设备的问题）：

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 2 -l 3:50,3:50 -I
IO_RUN_LATER -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING    RUN:cpu          1          1
  3     WAITING     RUN:io          1          1
  4     WAITING    WAITING                     2
  5     WAITING    WAITING                     2
  6*     RUN:io    WAITING          1          1
  7     WAITING    WAITING                     2
  8*    WAITING     RUN:io          1          1
  9     WAITING    WAITING                     2
 10     WAITING    WAITING                     2
 11*    RUN:cpu    WAITING          1          1
 12        DONE    WAITING                     1
 13*       DONE       DONE

Stats: Total Time 13
Stats: CPU Busy 6 (46.15%)
Stats: IO Busy  11 (84.62%)
```

`-I IO_RUN_IMMEDIATE`时，输出完全相同（因为I/O请求很多，所以当前I/O结束之后，CPU处于空闲状态，可以直接开始执行下一条指令）。

`-S SWITCH_ON_IO`时输出完全相同（因为这是默认值……）。

`-S SWITCH_ON_END`时运行时间大大增加了：

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 2 -l 3:50,3:50 -S
SWITCH_ON_END -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING      READY                     1
  3     WAITING      READY                     1
  4     WAITING      READY                     1
  5     WAITING      READY                     1
  6*     RUN:io      READY          1
  7     WAITING      READY                     1
  8     WAITING      READY                     1
  9     WAITING      READY                     1
 10     WAITING      READY                     1
 11*    RUN:cpu      READY          1
 12        DONE    RUN:cpu          1
 13        DONE     RUN:io          1
 14        DONE    WAITING                     1
 15        DONE    WAITING                     1
```

```
 16        DONE    WAITING                   1
 17        DONE    WAITING                   1
 18*       DONE     RUN:io          1
 19        DONE    WAITING                   1
 20        DONE    WAITING                   1
 21        DONE    WAITING                   1
 22        DONE    WAITING                   1
 23*       DONE       DONE

Stats: Total Time 23
Stats: CPU Busy 6 (26.09%)
Stats: IO Busy  16 (69.57%)
```

8.3 使用参数`-s 3 -l 3:50,3:50`，输出如下：

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 3 -l 3:50,3:50
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  io
  cpu

Process 1
  io
  io
  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)
```

`-I IO_RUN_LATER`时：

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 3 -l 3:50,3:50 -I
IO_RUN_LATER -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1     RUN:cpu      READY          1
  2      RUN:io      READY          1
  3     WAITING     RUN:io          1          1
  4     WAITING    WAITING                     2
  5     WAITING    WAITING                     2
  6     WAITING    WAITING                     2
  7*    RUN:cpu    WAITING          1          1
  8*       DONE     RUN:io          1
  9        DONE    WAITING                     1
 10        DONE    WAITING                     1
 11        DONE    WAITING                     1
 12        DONE    WAITING                     1
 13*       DONE    RUN:cpu          1

Stats: Total Time 13
Stats: CPU Busy 6 (46.15%)
Stats: IO Busy  9 (69.23%)
```

`-I IO_RUN_IMMEDIATE`时输出不变，因为I/O请求结束的时间又一次恰好和CPU被占用的时间错开了。

`-S SWITCH_ON_IO`时输出不变（因为这仍然是默认值）。

`-S SWITCH_ON_END`时运行时间仍然会增加。

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./python process-run.py -s 3 -l
3:50,3:50 -S SWITCH_ON_END -c -p
Time     PID: 0     PID: 1        CPU        IOs
  1     RUN:cpu      READY          1
  2      RUN:io      READY          1
  3     WAITING      READY                    1
  4     WAITING      READY                    1
  5     WAITING      READY                    1
  6     WAITING      READY                    1
  7*    RUN:cpu      READY          1
  8        DONE     RUN:io          1
  9        DONE    WAITING                   1
 10        DONE    WAITING                   1
 11        DONE    WAITING                   1
 12        DONE    WAITING                   1
 13*       DONE     RUN:io          1
 14        DONE    WAITING                   1
 15        DONE    WAITING                   1
 16        DONE    WAITING                   1
 17        DONE    WAITING                   1
 18*       DONE    RUN:cpu          1

Stats: Total Time 18
Stats: CPU Busy 6 (33.33%)
Stats: IO Busy  12 (66.67%)
```

3.9 问题9：（自己乱整的）

9、自己乱整的

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 1 -l
5:50,5:50,5:50,5:50 -I IO_RUN_IMMEDIATE -c -p
Time     PID: 0     PID: 1     PID: 2     PID: 3        CPU        IOs
  1     RUN:cpu      READY      READY      READY          1
  2      RUN:io      READY      READY      READY          1
  3     WAITING    RUN:cpu      READY      READY          1          1
  4     WAITING     RUN:io      READY      READY          1          1
  5     WAITING    WAITING     RUN:io      READY          1          2
  6     WAITING    WAITING    WAITING     RUN:io          1          3
  7*     RUN:io    WAITING    WAITING    WAITING          1          3
  8     WAITING    WAITING    WAITING    WAITING                     4
  9*    WAITING     RUN:io    WAITING    WAITING          1          3
 10*    WAITING    WAITING    RUN:cpu    WAITING          1          3
 11*    WAITING    WAITING      READY    RUN:cpu          1          2
 12*    RUN:cpu    WAITING      READY      READY          1          1
 13     RUN:cpu    WAITING      READY      READY          1          1
 14*       DONE    RUN:cpu      READY      READY          1
 15        DONE    RUN:cpu      READY      READY          1
```

```
 16        DONE       DONE     RUN:io      READY          1
 17        DONE       DONE    WAITING     RUN:io          1          1
 18        DONE       DONE    WAITING    WAITING                     2
 19        DONE       DONE    WAITING    WAITING                     2
 20        DONE       DONE    WAITING    WAITING                     2
 21*       DONE       DONE    RUN:cpu    WAITING         1           1
 22*       DONE       DONE      READY     RUN:io         1
 23        DONE       DONE    RUN:cpu    WAITING         1           1
 24        DONE       DONE       DONE    WAITING                     1
 25        DONE       DONE       DONE    WAITING                     1
 26        DONE       DONE       DONE    WAITING                     1
 27*       DONE       DONE       DONE    RUN:cpu         1

Stats: Total Time 27
Stats: CPU Busy 20 (74.07%)
Stats: IO Busy  20 (74.07%)
```

```
hcs@ubuntu:~/myfiles/OSTEP/homework$ ./process-run.py -s 1 -l
5:50,5:50,5:50,5:50 -I IO_RUN_LATER -c -p
Time     PID: 0     PID: 1     PID: 2     PID: 3        CPU        IOs
  1     RUN:cpu      READY      READY      READY          1
  2      RUN:io      READY      READY      READY          1
  3     WAITING    RUN:cpu      READY      READY          1          1
  4     WAITING     RUN:io      READY      READY          1          1
  5     WAITING    WAITING     RUN:io      READY          1          2
  6     WAITING    WAITING    WAITING     RUN:io          1          3
  7*     RUN:io    WAITING    WAITING    WAITING          1          3
  8     WAITING    WAITING    WAITING    WAITING                     4
  9*    WAITING     RUN:io    WAITING    WAITING          1          3
 10*    WAITING    WAITING    RUN:cpu    WAITING          1          3
 11*    WAITING    WAITING     RUN:io      READY          1          2
 12*      READY    WAITING    WAITING    RUN:cpu          1          2
 13       READY    WAITING    WAITING     RUN:io          1          2
 14*    RUN:cpu      READY    WAITING    WAITING          1          2
 15     RUN:cpu      READY    WAITING    WAITING          1          2
 16*       DONE    RUN:cpu      READY    WAITING          1          1
 17        DONE    RUN:cpu      READY    WAITING          1          1
 18*       DONE       DONE    RUN:cpu      READY          1
 19        DONE       DONE    RUN:cpu      READY          1
 20        DONE       DONE       DONE     RUN:io          1
 21        DONE       DONE       DONE    WAITING                     1
 22        DONE       DONE       DONE    WAITING                     1
 23        DONE       DONE       DONE    WAITING                     1
 24        DONE       DONE       DONE    WAITING                     1
 25*       DONE       DONE       DONE    RUN:cpu         1

Stats: Total Time 25
Stats: CPU Busy 20 (80.00%)
Stats: IO Busy  19 (76.00%)
```

为什么随机化后，IO run later的反而时间更短？也不一定，好像如果是其他案例，结果相反。

4 第5章 插叙：进程API

4.1 p1.c 调用fork()

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

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n",(int)getpid());
    int rc = fork(); //me：获得子进程的PID
    if(rc < 0)  //fork failed; exit
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if(rc == 0)  //child (new process)
    {
        printf("hello, I am child (pid:%d)\n",(int)getpid());
    }
    else  //parent goes down this path (main)
    {
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
    }

    return 0;
}
```

4.2 p2.c 调用fork()和wait()

```c
//调用fork()和wait()
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, int *argv[])
{
    printf("hello world (pid:%d)\n",(int)getpid());
    int rc = fork();
    if(rc < 0)  //fork failed; exit
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if(rc == 0)  //child (new process)
    {
        printf("hello, I am child (pid:%d)\n",(int)getpid());
    }
    else  //parent goes down this path (main)
    {
        int wc = wait(NULL);
```

```
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc,
(int)getpid());
    }

    return 0;
}
```

运行结果：

```
hcs@ubuntu:~/myfiles/OSTEP/C5$ ./p2
hello world (pid:3034)
hello, I am child (pid:3035)
hello, I am parent of 3035 (wc:3035) (pid:3034)
```

4.3 p3.c 调用execvp()

`int execvp(const char *filename,const char *argv[])`

execvp: 这个函数如果正常运行是不会有返回的，有返回说明启动的程序出现异常。

- (1) 第一个参数是要运行的文件，会在环境变量PATH中查找file，并执行.
- (2) 第二个参数，是一个参数列表，如同在shell中调用程序一样，参数列表为0，1，2，3.....因此，

wensen.sh 作为第0个参数，需要重复一遍.

- (3) argv列表最后一个必须是 NULL.
- (4) 失败会返回 - 1， 成功无返回值，但是，失败会在当前进程运行，执行成功后，直接结束当前进程，可以在子进程中运行.

  ? ? ? execvp()无返回值是什么意思? ? ?

  ——

  execvp()参考链接

  ——

  wc是字符计算程序

```
//调用execvp()

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, int *argv[])
{
    printf("hello world (pid:%d)\n",(int)getpid());
    int rc = fork();
    if(rc < 0)  //fork failed; exit
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
```

```
    else if(rc == 0)  //child (new process)
    {
        printf("hello, I am child (pid:%d)\n",(int)getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");  //program: "wc" (word count) me:这个是系统程序吗，依据哪个路径去找呢？
        myargs[1] = strdup("p3.c");  //argument: file to count
        myargs[2] = NULL;  //mark end of array
        execvp(myargs[0],myargs);  //runs word count
        printf("this shouldn't print out\n");  //me: why???
    }
    else  //parent goes down this path (main)
    {
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc, (int)getpid());
    }

    return 0;
}
```

运行结果：

```
hcs@ubuntu:~/myfiles/OSTEP/C5$ ./p3
hello world (pid:3143)
hello, I am child (pid:3144)
 35 111 930 p3.c
hello, I am parent of 3144 (wc:3144) (pid:3143)
```

4.4 作业（编码）

4.4.1 1、hw5_1.c

hw5_1.c

写一个程序，调用`fork()`。在调用`fork()`之前，让主进程设置一个变量（如`x`）的值（如100）。子进程中这个变量的值是多少？当子进程和父进程都修改`x`的值时，会发生什么？

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

int main(int argc, int *argv[])
{
    int x = 100;
    printf("hello world, I am parent process. (PID:%d)", (int)getpid());
    int child_PID = fork();
    if(child_PID < 0)
    {
        printf("fork failed.\n");
    }
    else if(child_PID == 0)
    {
        printf("I am child(PID:%d), ", child_PID);
```

```
        printf("and x = %d\n", --x);
    }
    else
    {
        printf("I am parent(PID:%d) of child(PID:%d), ", (int)getpid(),
child_PID);
        printf("and x = %d\n", ++x);
    }

    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/C5$ ./hw5_1 > hw5_1.output | cat hw5_1.output -A
-b
     1  hello world, I am parent process. (PID:2944)$
     2  I am parent(PID:2944) of child(PID:2946), and x = 101$
     3  hello world, I am parent process. (PID:2944)$
     4  I am child(PID:0), and x = 99$
hcs@ubuntu:~/myfiles/OSTEP/C5$ ./hw5_1
hello world, I am parent process. (PID:2953)
I am parent(PID:2953) of child(PID:2954), and x = 101
hcs@ubuntu:~/myfiles/OSTEP/C5$ I am child(PID:0), and x = 99

hcs@ubuntu:~/myfiles/OSTEP/C5$
```

由结果可以看出：父进程和子进程对`x`的修改是互相独立的。这是因为`fork()`系统调用把内存空间复制了一份。

疑问？？？：

为什么输出文件打印出来是4行，多出的一行是`hw5_1.output`文件中的第3行，而直接输出到屏幕时，则只有3行**（理论上分析，我觉得3行是正确的）**。

另外，输出到文件时，子进程的PID比父进程大2；而输出到屏幕时，子进程的PID比父进程大1，为什么？？？

另外的一个问题，运行结果的第9行，正确的输出不应该是：

```
hcs@ubuntu:~/myfiles/OSTEP/C5$ ./hw5_1
hello world, I am parent process. (PID:2953)
I am parent(PID:2953) of child(PID:2954), and x = 101
I am child(PID:0), and x = 99
hcs@ubuntu:~/myfiles/OSTEP/C5$
```

才对吗？？？

4.4.2 2、hw02.c

hw02.c

写一个程序，通过`open()`系统调用打开一个文件，并调用`fork()`创建一个新的进程。子进程和父进程可以同时访问`open()`返回的文件描述符吗？如果它们同时写这个文件，会发生什么？

需要用到open()函数：

```
int open(const char*pathname,int flags);
int open(const char*pathname,int flags,mode_t mode);
参数说明：
1.pathname
  要打开或创建的目标文件
2.flags
  打开文件时，可以传入多个参数选项，用下面的
  一个或者多个常量进行“或”运算，构成falgs
  参数：
  O_RDONLY:    只读打开
  O_WRONLY:    只写打开
  O_RDWR:      读，写打开
这三个常量，必须制定一个且只能指定一个
  O_CREAT:     若文件不存在，则创建它，需要使
               用mode选项。来指明新文件的访问权限
  O_APPEND:    追加写，如果文件已经有内容，这次打开文件所
               写的数据附加到文件的末尾而不覆盖原来的内容
```

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

int main(int argc, char *argv[])
{
    int fd = open("./hw02.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
//O_TRUNC参数使每次打开文件总会清除文件内容
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child
        char str[] = "I am child\n";
        write(fd, str, strlen(str));
    } else { // parent goes down this path (main)
        char str[] = "I am parent\n";
        write(fd, str, strlen(str));
    }
    return 0;
}
```

相关系统函数解析：https://blog.csdn.net/dangzhangjing97/article/details/79631173

运行结果：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw02.c -o hw02
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw02
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ cat hw02.output -b
     1  I am parent
     2  I am child
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$
```

可以发现，子进程和父进程可以同时访问这个文件，且输出结果基本是正常的。

某博主：Linux实际上并不会保证并发的文件操作不出问题。不过，其实Linux的内部实现保证了`read()`和`write()`操作是串行执行的。详情可见：How do filesystems handle concurrent read/write?。

4.4.3 3、hw5_3.c

hw5_3.c

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

int main(int argc, char *argv[])
{
    int fd = open("./hw5_3.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
//O_TRUNC参数使每次打开文件总会清除文件内容
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child
        char str[] = "hello, I am child.\n";
        write(fd, str, strlen(str));
//        printf("%s\n",str);
    } else { // parent goes down this path (main)
        char str[] = "goodbye, I am parent.\n";
        write(fd, str, strlen(str));
//        printf("%s\n",str);
    }
    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw5_3.c -o hw5_3
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw5_3
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ cat hw5_3.output -b
     1  goodbye, I am parent.
     2  hello, I am child.
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$
```

由以上结果，似乎不用wait()是做不到先让子进程打印输出，然后父进程才 打印输出。

代码改进如下（只好加入wait()，**注意：hw03.c代码编译不通过**）：

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

```c
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int fd = open("./hw5_3.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
//O_TRUNC参数使每次打开文件总会清除文件内容
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child
        char str[] = "hello, I am child.\n";
        write(fd, str, strlen(str));
//        printf("%s\n",str);
    } else { // parent goes down this path (main)
        int wc = wait(NULL);
        char str[] = "goodbye, I am parent.\n";
        write(fd, str, strlen(str));
//        printf("%s\n",str);
    }
    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw5_3.c -o hw5_3
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw5_3
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ cat hw5_3.output -b
     1  hello, I am child.
     2  goodbye, I am parent.
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$
```

4.4.4 4、hw04.c

hw04.c

这一个程序看得稀里糊涂的，不是很懂。可能是让我们熟悉exec的各种形似调用?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < 6; i++) {
        switch (i) {
        case 0:
            printf("Parent: ready for execl(const char *path, const char *arg, ... /* (char  *) NULL */)\n");
            break;
        case 1:
```

```
            printf("Parent: ready for execlp(const char *file, const char *arg, ... /* (char  *) NULL */)\n");
            break;
        case 2:
            printf("Parent: ready for execle(const char *path, const char *arg, ... /*, (char *) NULL, char * const envp[] */)\n");
            break;
        case 3:
            printf("Parent: ready for execv(const char *path, char *const argv[])\n");
            break;
        case 4:
            printf("Parent: ready for execvp(const char *file, char *const argv[])\n");
            break;
        case 5:
            printf("Parent: ready for execvpe(const char *file, char *const argv[], char *const envp[])\n");
            break;
        default:
            printf("Should not come here\n");
            exit(1);
            break;
        }
        int rc = fork();
        if (rc < 0) { // fork failed; exit
            fprintf(stderr, "fork failed\n");
            exit(1);
        } else if (rc == 0) { // child
            char *argv[] = {"ls", "/", NULL};
            switch (i) {
            case 0:
                execl("/bin/ls", "ls", "/", NULL);
                exit(1);
                break;
            case 1:
                execlp("ls", "ls", "/", NULL);
                exit(1);
                break;
            case 2:
                execle("/bin/ls", "ls", "/", NULL, NULL);
                exit(1);
                break;
            case 3:
                execv("/bin/ls", argv);
                exit(1);
                break;
            case 4:
                execvp("ls", argv);
                exit(1);
                break;
            case 5:
                execvpe("ls", argv, NULL);
                exit(1);
                break;
            default:
                printf("Should not come here\n");
                exit(1);
```

```
            }
        } else { // parent goes down this path (main)
            wait(NULL);
        }
    }
    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw04.c -o hw04
hw04.c: In function ‘main’:
hw04.c:65:17: warning: implicit declaration of function ‘execvpe’ [-Wimplicit-function-declaration]
                execvpe("ls", argv, NULL);
                ^
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ls *04
hw04
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw04
Parent: ready for execl(const char *path, const char *arg, ... /* (char  *) NULL */)
bin   cdrom  etc   initrd.img      lib    lib64       media  opt   root  sbin  srv  tmp  var      vmlinuz.old
boot  dev    home  initrd.img.old  lib32  lost+found  mnt    proc  run   snap  sys  usr  vmlinuz
Parent: ready for execlp(const char *file, const char *arg, ... /* (char *) NULL */)
bin   cdrom  etc   initrd.img      lib    lib64       media  opt   root  sbin  srv  tmp  var      vmlinuz.old
boot  dev    home  initrd.img.old  lib32  lost+found  mnt    proc  run   snap  sys  usr  vmlinuz
Parent: ready for execle(const char *path, const char *arg, ... /*, (char *) NULL, char * const envp[] */)
bin   cdrom  etc   initrd.img      lib    lib64       media  opt   root  sbin  srv  tmp  var      vmlinuz.old
boot  dev    home  initrd.img.old  lib32  lost+found  mnt    proc  run   snap  sys  usr  vmlinuz
Parent: ready for execv(const char *path, char *const argv[])
bin   cdrom  etc   initrd.img      lib    lib64       media  opt   root  sbin  srv  tmp  var      vmlinuz.old
boot  dev    home  initrd.img.old  lib32  lost+found  mnt    proc  run   snap  sys  usr  vmlinuz
Parent: ready for execvp(const char *file, char *const argv[])
bin   cdrom  etc   initrd.img      lib    lib64       media  opt   root  sbin  srv  tmp  var      vmlinuz.old
boot  dev    home  initrd.img.old  lib32  lost+found  mnt    proc  run   snap  sys  usr  vmlinuz
Parent: ready for execvpe(const char *file, char *const argv[], char *const envp[])
bin   cdrom  etc   initrd.img      lib    lib64       media  opt   root  sbin  srv  tmp  var      vmlinuz.old
boot  dev    home  initrd.img.old  lib32  lost+found  mnt    proc  run   snap  sys  usr  vmlinuz
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$
```

我猜这些形式主要是为了满足用户不同的需求。实际上，`exec`后面的那些后缀的含义是这样的（参见了linux系统编程之进程（五）：exec系列函数（execl,execlp,execle,execv,execvp)使用）：

- `l`：参数以可变参数列表的形式给出，且以`NULL`结束（`execl()`，`execle()`，`execlp()`）
- 没有`l`：参数以`char *arg[]`形式给出，且`arg`最后一个元素必须为`NULL`（`execv()`，`execp()`，`execvpe()`）
- `p`：第一个参数不用输入完整路径，给出命令名即可，程序会在环境变量PATH当中查找命令（`execlp()`，`execp()`，`execvpe()`）
- 没有`p`：第一个参数需要输入完整路径（`execl()`，`execle()`，`execv()`）
- `e`：将环境变量传递给新进程（`execle()`，`execvpe()`）
- 没有`e`：不传递环境变量（`execl()`，`execlp()`，`execv()`，`execp()`）

4.4.5 5、hw5_5.c

hw5_5.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("I am parent (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("I am child (pid:%d)\n", (int) getpid());
        int wc = wait(NULL);
        printf("Child: wait() returns %d\n", wc);
    } else { // parent goes down this path (main)
        int wc = wait(NULL);
        printf("Parent: wait() returns %d\n", wc);
    }
    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw05.c -o hw05
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw05
I am parent (pid:16770)
I am child (pid:16771)
Child: wait() returns -1
Parent: wait() returns 16771
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$
```

分析运行结果，可以知道子进程调用wait()失败，返回-1，这是易于理解的，因为子进程没有自己的子进程，自然无法等待子进程完成，因而调用失败；父进程，调用wait()成功，返回的是子进程的PID。

但是，有意思的是，为什么父进程调用wait()成功了，没有表现出应有的礼貌，等待子进程执行完毕，而后执行呢？？？这很奇怪。

4.4.6 6、hw06.c

稍微修改一下上一题中的程序，改为使用`waitpid()`，而不是`wait()`。`waitpid()`何时是有用的?

hw06.c

hw06.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("I am parent (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child
        printf("I am child (pid:%d)\n", (int) getpid());
        int wc = waitpid(-1, NULL, 0);
        printf("Child: waitpid(-1) returns %d\n", wc);
    } else { // parent
        int wc = waitpid(rc, NULL, 0);
        printf("Parent: waitpid(%d) returns %d\n", rc, wc);
    }
    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw06.c -o hw06
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw06
I am parent (pid:16884)
I am child (pid:16885)
Child: waitpid(-1) returns -1
Parent: waitpid(16885) returns 16885
```

在需要等待某一个子进程执行完毕时，可以使用`waitpid()`。调用`waitpid(-1)`的效果与`wait()`基本是类似的。

4.4.7 7、hw07.c

hw07.c

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

```c
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("I am parent\n");
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        printf("I am child (before closing standard output)"); //mk1
        close(STDOUT_FILENO);
        printf("I am child (after closing standard output)");
    } else { // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw07.c -o hw07
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw07
I am parent
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$
```

疑问？？？不知道为什么“I am child (before closing standard output)”(mk1处)没有被打印出来。

4.4.8 8、hw08.c

hw08.c

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int main()
{
    int pipefd[2];
    char buf[80];
    pid_t pid;

    pipe(pipefd);

    int rc[2];
    rc[0] = fork();

    if (rc[0] < 0)  // fork failed
    {
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc[0] == 0)  // child 1
```

```
    {
        printf("Child 1 (pid=%d), writing to pipe.\n", (int) getpid());
        char s[] = "Hello world , this is transported by pipe.\n";
        write(pipefd[1], s, sizeof(s));
        close(pipefd[0]);
        close(pipefd[1]);
    }
    else  // parent
    {
        wait(NULL);
        rc[1] = fork();
        if (rc[1] < 0) {  // fork failed
            fprintf(stderr, "fork failed\n");
            exit(1);
        }
        else if (rc[1] == 0) {  // child 2
            printf("Child 2 (pid=%d), reading from pipe.\n", (int)
getpid());
            read(pipefd[0], buf, sizeof(buf));
            printf("%s\n", buf);
            close(pipefd[0]);
            close(pipefd[1]);
        }
        else {  // parent
            wait(NULL);
        }
    }
    return 0;
}
```

运行结果如下：

```
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ gcc hw08.c -o hw08
hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$ ./hw08
Child 1 (pid=17128), writing to pipe.
Child 2 (pid=17129), reading from pipe.
Hello world , this is transported by pipe.

hcs@ubuntu:~/myfiles/OSTEP/codes/C5/hw_codes$
```

阅读hw08.c可参考网上解析：[linux pipe使用小结]

5 第6章 机制：受限直接执行

操作系统必须以高性能的方式虚拟化 CPU，同时保持对系统的控制。

5.1 问题1：受限制的操作

表 6.1　　直接运行协议（无限制）

操作系统	程序
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argc/argv 设置程序栈	
清除寄存器 执行 call main() 方法	
	执行 main() 从 main 中执行 return
释放进程的内存将进程 从进程列表中清除	

表 6.1 展示了基本的直接执行协议（没有任何限制），使用正常的调用并返回跳转到程序的 main()，并在稍后回到内核。

表 6.2　　受限直接运行协议

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住系统调用处理程序的地址	
操作系统@运行（内核模式）	硬件	程序（应用模式）
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argv 设置程序栈 用寄存器/程序计数器填充内核栈 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到 main	
		运行 main …… 调用系统调用 陷入操作系统
	将寄存器保存到内核栈 转向内核模式 跳到陷阱处理程序	
处理陷阱 做系统调用的工作 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到陷阱之后的程序计数器	
		……从 main 返回 陷入（通过 exit()）
释放进程的内存将进程 从进程列表中清除		

LDE 协议（受限直接运行协议）有两个阶段。

- 第一个阶段（在系统引导时），内核初始化陷阱表，并且CPU 记住它的位置以供随后使用。内核通过特权指令来执行此操作（所有特权指令均以粗体突出显示）。
- 第二个阶段（运行进程时），在使用从陷阱返回指令开始执行进程之前，内核设置了一些内容（例如，在进程列表中分配一个节点，分配内存）。

- 这会将CPU 切换到用户模式并开始运行该进程。当进程希望发出系统调用时，它会重新陷入操作系统，
- 然后再次通过从陷阱返回，将控制权还给进程。
- 该进程然后完成它的工作，并从main()返回。这通常会返回到一些存根代码，它将正确退出该程序（例如，通过调用exit()系统调用，这将陷入OS 中）。此时，OS 清理干净，任务完成了。

5.2 问题2：在进程间切换

操作系统应该决定停止一个进程并开始另一个进程。但实际上，特别是，如果一个进程在 CPU 上运行，这就意味着操作系统没有运行。这是真正的问题——如果操作系统没有在 CPU 上运行，那么操作系统显然没有办法采取行动。于是有了，协作方式：等待系统调用。

5.2.1 协作方式：等待系统调用

在协作调度系统中，OS 通过等待系统调用，或某种非法操作发生，从而重新获得 CPU 的控制权。

但是这种方式是被动的。例如，如果某个进程（无论是恶意的还是充满缺陷的）进入无限循环，并且从不进行系统调用，会发生什么情况？那时操作系统能做什么？

5.2.2 非协作方式：操作系统进行控制

在协作方式中，当进程陷入无限循环时，唯一的办法就是使用古老的解决方案来解决计算机系统中的所有问题——重新启动计算机。

关键问题：如何在没有协作的情况下获得控制权
即使进程不协作，操作系统如何获得 CPU 的控制权？操作系统可以做什么来确保流氓进程不会占用机器？
（疑问？？？进程不协作指的是什么？某个进程进入无限循环属于进程不协作吗？）
提示：利用时钟中断重新获得控制权
即使进程以非协作的方式运行，添加时钟中断（timer interrupt）也让操作系统能够在 CPU 上重新运行。因此，该硬件功能对于帮助操作系统维持机器的控制权至关重要。

操作系统在启动时必须通知硬件哪些代码在发生时钟中断时运行。在启动过程中，操作系统也必须启动时钟，这当然是一项特权操作。一旦时钟开始运行，操作系统就感到安全了。注意，硬件在发生中断时有一定的责任，尤其是在中断发生时，要为正在运行的程序保存足够的状态，以便随后从陷阱返回指令能够正确恢复正在运行的程序。这一组操作与硬件在显式系统调用陷入内核时的行为非常相似，其中各种寄存器因此被保存（进入内核栈），因此从陷阱返回指令可以容易地恢复。

5.2.3 保存和恢复上下文

操作系统获得控制权之后，就需要有**调度程序（scheduler）**决定进程的切换。如果决定进行切换，OS 就会执行一些底层代码，即所谓的**上下文切换（context switch）**。

【上下文切换概念】：
操作系统要做的就是为当前正在执行的进程保存一些寄存器的值（例如，到它的内核栈），并为即将执行的进程恢复一些寄存器的值（从它的内核栈）。这样一来，操作系统就可以确保最后执行从陷阱返回指令时，不是返回到之前运行的进程，而是继续执行另一个进程。

表 6.3 展示了整个过程的时间线。在这个例子中，进程 A 正在运行，然后被中断时钟中断。硬件保存它的寄存器（在内核栈中），并进入内核（切换到内核模式）。在时钟中断处理程序中，操作系统决定从正在运行的进程 A切换到进程 B。此时，它调用 switch()例程，该例程仔细保存当前寄存器的值（保存到 A 的进程结构），恢复寄存器进程 B（从它的进程结构），然后切换上下文（switch context），具

体来说是通过改变栈指针来使用 B 的内核栈（而不是 A 的）。最后，操作系统从陷阱返回，恢复 B 的寄存器并开始运行它。

表 6.3　受限直接执行协议（时钟中断）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 系统调用处理程序 时钟处理程序	
启动中断时钟		
	启动时钟 每隔 *x* ms 中断 CPU	
操作系统@运行（内核模式）	硬件	程序（应用模式）
		进程 A……
	时钟中断 将寄存器（A）保存到内核栈（A） 转向内核模式 跳到陷阱处理程序	
处理陷阱 调用 switch()例程 将寄存器（A）保存到进程结构（A） 将进程结构（B）恢复到寄存器（B） 从陷阱返回（进入 B）		
	从内核栈（B）恢复寄存器（B） 转向用户模式 跳到 B 的程序计数器	
		进程 B……

在此协议中，有两种类型的寄存器保存/恢复。第一种是发生时钟中断的时候。在这种情况下，运行进程的用户寄存器由硬件隐式保存，使用该进程的内核栈。第二种是当操作系统决定从 A 切换到 B。在这种情况下，内核寄存器被软件（即 OS）明确地保存，但这次被存储在该进程的进程结构的内存中。后一个操作让系统从好像刚刚由 A 陷入内核，变成好像刚刚由 B 陷入内核。

xv6的上下文切换代码：

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
# Save old registers
movl 4(%esp), %eax # put old ptr into eax
popl 0(%eax) # save the old IP
movl %esp, 4(%eax) # and stack
movl %ebx, 8(%eax) # and other registers
movl %ecx, 12(%eax)
movl %edx, 16(%eax)
movl %esi, 20(%eax)
movl %edi, 24(%eax)
movl %ebp, 28(%eax)

# Load new registers
movl 4(%esp), %eax # put new ptr into eax
movl 28(%eax), %ebp # restore other registers
movl 24(%eax), %edi
movl 20(%eax), %esi
```

```
movl 16(%eax), %edx
movl 12(%eax), %ecx
movl 8(%eax), %ebx
movl 4(%eax), %esp # stack is switched here
pushl 0(%eax) # return addr put in place
ret # finally return into new ctxt
```

5.2.4 担心并发吗?

在系统调用期间发生时钟中断时会发生什么？”或“处理一个中断时发生另一个中断，会发生什么？这不会让内
核难以处理吗？”

操作系统可能简单地决定，在中断处理期间禁止中断（disable interrupt）。这样做可以确保在处理一个中断时，不会将其他中断交给 CPU。

5.2.5 小结

实现 CPU 虚拟化的关键底层机制，其统称为受限直接执行（limited direct execution）。基本思路很简单：就让你想运行的程序在 CPU 上运行，但首先确保设置好硬件，以便在没有操作系统帮助的情况下限制进程可以执行的操作。

【超级生动的例子】例如宝宝防护（baby proofing）房间的概念——锁好包含危险物品的柜子，并掩盖电源插座。当这些都准备妥当时，你可以让宝宝自由行动，确保房间最危险的方面受到限制。

OS 首先（在启动时）设置陷阱处理程序并启动时钟中断，然后仅在受限模式下运行进程，以此为CPU提供“宝宝防护”。这样做，操作系统能确信进程可以高效运行，只在执行特权操作，或者当它们独占 CPU 时间过长并因此需要切换时，才需要操作系统干预。

5.3 作业

6 第7章 进程调度：介绍