

5. 进程运行轨迹的跟踪与统计

5.1. 实验目的

掌握 Linux 下的多进程编程技术；

通过对进程运行轨迹的跟踪来形象化进程的概念；

在进程运行轨迹跟踪的基础上进行相应的数据统计，从而能对进程调度算法进行实际的量化评价，更进一步加深对调度和调度算法的理解，获得能在实际操作系统上对调度算法进行实验数据对比的直接经验。

5.2. 实验内容

进程从创建（Linux 下调用 `fork()`）到结束的整个过程就是进程的生命期，进程在其生命期中的运行轨迹实际上就表现为进程状态的多次切换，如进程创建以后会成为就绪态；当该进程被调度以后会切换到运行态；在运行的过程中如果启动了一个文件读写操作，操作系统会将该进程切换到阻塞态（等待态）从而让出 CPU；当文件读写完毕以后，操作系统会在将其切换成就绪态，等待进程调度算法来调度该进程执行……

本次实验包括如下内容：

1. 基于模板 `process.c` 编写多进程的样本程序，实现如下功能：

所有子进程都并行运行，每个子进程的实际运行时间一般不超过 30 秒；

父进程向标准输出打印所有子进程的 id，并在所有子进程都退出后才退出；

在 Linux 0.11 上实现进程运行轨迹的跟踪。基本任务是在内核中维护一个日志文件 `/var/process.log`，把从操作系统启动到系统关机过程中所有进程的运行轨迹都记录在这一 log 文件中。

2. 在修改过的 0.11 上运行样本程序，通过分析 log 文件，统计该程序建立的所有进程的等待时间、完成时间（周转时间）和运行时间，然后计算平均等待时间，平均完成时间和吞吐量。可以自己编写统计程序，也可以使用 python 脚本程序 `stat_log.py` 进行统计。
3. 修改 0.11 进程调度的时间片，然后再运行同样的样本程序，统计同样的时间数据，和原有的情况对比，体会不同时间片带来的差异。

实现过程：

首先，对 `process.c` 进行编写，根据指导书可知，`process` 涉及到对 `fork` 和 `wait` 的调用，并且需要实现如下功能：

所有子进程都并行运行，每个子进程的实际运行时间一般不超过 30 秒；

父进程向标准输出打印所有子进程的 id，并在所有子进程都退出后才退出；

在 Linux 0.11 上实现进程运行轨迹的跟踪。基本任务是在内核中维护一个日志文件 `/var/process.log`，把从操作系统启动到系统关机过程中所有进程的运行轨迹都记录在这一 log 文件中。

CV 样本模板并进行修改后得到如下结果：

```
#include <stdio.h>      // 包含标准输入输出库，用于 printf 函数
#include <stdlib.h>      // 包含标准库，用于 exit 函数
#include <unistd.h>      // 包含 Unix 标准函数定义，用于 fork, getpid, getppid, sleep 等函数
#include <time.h>        // 包含时间处理函数，用于 times 函数
#include <sys/times.h>    // 包含进程时间信息结构体定义，用于 times 函数
#include <sys/wait.h>     // 包含进程等待函数定义，用于 wait 函数

#define HZ 100          // 定义 HZ 为 100，表示系统时钟频率，用于计算时间
/*
```

```

* 此函数按照参数占用 CPU 和 I/O 时间
* last: 函数实际占用 CPU 和 I/O 的总时间, 不含在就绪队列中的时间, >=0 是必须的
* cpu_time: 一次连续占用 CPU 的时间, >=0 是必须的
* io_time: 一次 I/O 消耗的时间, >=0 是必须的
* 如果 last > cpu_time + io_time, 则往复多次占用 CPU 和 I/O
* 所有时间的单位为秒
*/
void cpuio_bound(int last, int cpu_time, int io_time);
int main(int argc, char * argv[])
{
    int i;          // 循环计数器
    pid_t pid[8];    // 用于存储 fork 返回的进程 ID
    // 循环 8 次, 创建 8 个子进程
    for(i = 0; i < 8; i++){
        pid[i] = fork(); // 创建一个子进程, 返回值存储在 pid 变量中
        if(pid[i] == 0)    // 如果 pid 为 0, 表示当前是子进程
        {
            cpuio_bound(10, i, 8-i); // 调用 cpuio_bound 函数模拟 CPU 和 I/O 负载
            return 0; // 子进程执行完毕后退出
        }
        else if(pid[i] < 0)
        {
            printf("fork error!");
            exit(1);
        }
        else
        {
            // 打印子进程 ID 和父进程 ID, 在父进程中打印比在子进程打印快 ( )
            printf("I'm a parent process, id = [%d], childid = [%d]\n", getpid(), pid[i]);
        }
    }
    // 父进程等待所有子进程结束
    for(i = 0; i < 8; i++)
        wait(NULL); // 等待一个子进程结束, 参数为 NULL 表示不关心子进程的退出状态
    return 0; // 父进程结束
}

// cpuio_bound 函数用于模拟 CPU 和 I/O 负载
void cpuio_bound(int last, int cpu_time, int io_time)
{
    struct tms start_time, current_time; // 定义两个 tms 结构体, 用于存储时间信息
    clock_t utime, stime; // 定义两个 clock_t 变量, 用于存储用户态和内核态时间
    int sleep_time;       // 定义一个变量, 用于存储 I/O 等待时间
    // 循环直到 last 减到 0
    while (last > 0)
    {

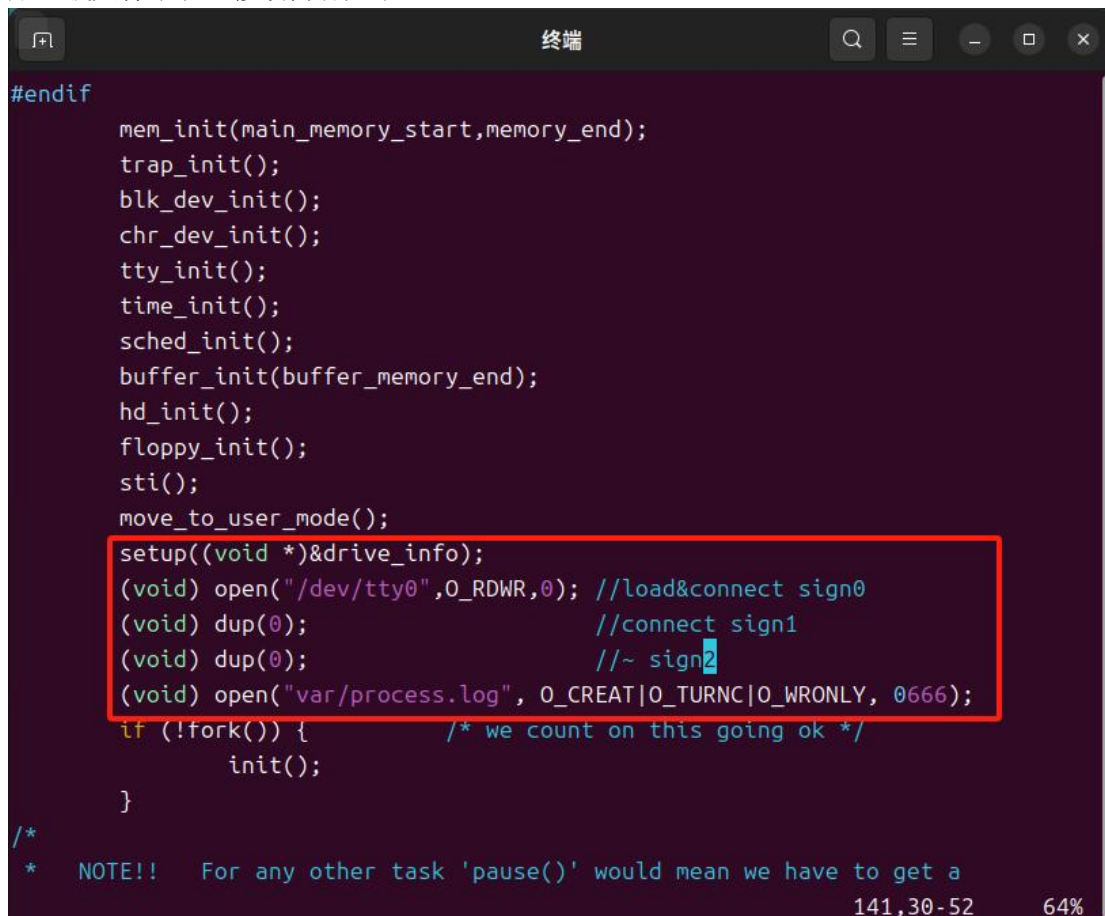
```

```

times(&start_time); // 获取当前时间信息, 存储在 start_time 中
do{
    times(&current_time); // 再次获取当前时间信息, 存储在 current_time 中
    utime = current_time.tms_utime - start_time.tms_utime; // 计算用户态时间差
    stime = current_time.tms_stime - start_time.tms_stime; // 计算内核态时间差
} while (((utime + stime) / HZ) < cpu_time); // 如果 CPU 时间不足 cpu_time, 则继续循环
last -= cpu_time; // 减去已经使用的 CPU 时间
if (last <= 0 ) // 如果 last 小于等于 0, 则跳出循环
    break;
sleep_time=0; // 初始化 I/O 等待时间
// 循环等待 I/O 时间
while (sleep_time < io_time){
    sleep(1); // 休眠 1 秒
    sleep_time++; // I/O 等待时间加 1
}
last -= sleep_time; // 减去已经使用的 I/O 时间
}
}

```

之后, 为了得到完整的 log 文件, 我们需要做到让虚拟机启动的时候就开始日志记录, 也就是在内核启动的时候就打开 log 文件, 而内核的入口在 `/linux-0.11/init/main.c` 文件中, 所以, 我们需要依据实验指导书, 对 `move_to_user_mode` 后进行修改, 让其第一时间就开始加载文件系统, 修改内容如下:



```

#endif
mem_init(main_memory_start, memory_end);
trap_init();
blk_dev_init();
chr_dev_init();
tty_init();
time_init();
sched_init();
buffer_init(buffer_memory_end);
hd_init();
floppy_init();
sti();
move_to_user_mode();
setup((void *)&drive_info);
(void) open("/dev/tty0", O_RDWR, 0); //load&connect sign0
(void) dup(0);                      //connect sign1
(void) dup(0);                      //~ sign2
(void) open("var/process.log", O_CREAT|O_TRUNC|O_WRONLY, 0666);
if (!fork()) {                      /* we count on this going ok */
    init();
}
/*
* NOTE!! For any other task 'pause()' would mean we have to get a
141,30-52 64%

```

#注意 这里 TRUNC 写错,make 时已修改正确#

添加代码的含义是：关联 `stdin` `stdout` 和 `stderr` 三个文件描述符后，进行 `log` 文件的创建和写入，每次打开 `log` 的时候都会清空复写，但是新版实验环境开机之后与 `stdin` 有关的内容会出现 `bug`，个人换回了 20 以前的旧实验环境解决该问题。

之后，要完成 `log` 文件的书写过程，显然实验指导书里面已经给出了详细代码，只需将其放入 `../kernel/printk.c` 中来实现 `printk()` 的调用功能（重写一个文件覆盖即可）详细内容与指导书上的相同，故略：

```
#include <linux/sched.h>
#include <sys/stat.h>

static char logbuf[1024];
int fprintk(int fd, const char *fmt, ...)
{
    va_list args;
    int count;
    struct file * file;
    struct m_inode * inode;

    va_start(args, fmt);
    count=vsprintf(logbuf, fmt, args);
    va_end(args);

    if (fd < 3) /* 如果输出到stdout或stderr, 直接调用sys_write即可 */
    {
```

接下来，要完成实现跟踪进程运行轨迹的功能，根据实验指导书的介绍，我们需要找到所有发生进程状态切换的代码部分，对其修改一个向 `log` 打印变化内容的功能，经过搜索，发现需要修该 `kernel` 下的 `fork.c`、`sched.c`、`exit.c` 三部分：

`fork.c`:

在 `fork.c` 中可以发现滴答数 `Jiffies` 被赋予到 `p->start_time` 中，后续 `p->state = TASK_RUNNING` 时，设置进程状态为就绪。所有就绪进程的状态都是 `TASK_RUNNING(0)`，被全局变量 `current` 指向的是正在运行的进程，所以需要对这部分进行修改，内容如下：

```
p->cutime = p->cstime = 0;
p->start_time = jiffies;
fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies);
p->tss.back_link = 0;
p->tss.esp0 = PAGE_SIZE + (long) p;

set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
p->state = TASK_RUNNING; /* do this last, just in case
fprintk(3, "%ld\t%c\t%ld\n", last_pid, 'J', jiffies);
return last_pid;
```

`sched.c`:

Linux 0.11 支持四种进程状态的转移，再 `sched.c` 中都能找到：就绪到运行、运行到就绪、运行到睡眠和睡眠到就绪，此外还有新建和退出两种情况。其中：就绪与运行间的状态转移是通过 `schedule()`（它亦是调度算法所在）完成的；运行到睡眠依靠的是 `sleep_on()` 和 `interruptible_sleep_on()`，还有进程主动睡觉的系

统调用 `sys_pause()` 和 `sys_waitpid()`（此函数在 `exit.c` 中定义）；

睡眠到就绪的转移依靠的是 `wake_up()`。

因此，只要在这些函数的适当位置（即状态 `state` 更新处）插入适当的处理语句（`fprintk()` 语句）就能完成进程运行轨迹的全面跟踪了。

具体修改内容如下：

`schedule()`:

```
{
    int i,next,c;
    struct task_struct ** p;

    /* check alarm, wake up any interruptible tasks that have got a signal */

    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) {
            if ((*p)->alarm && (*p)->alarm < jiffies) {
                (*p)->signal |= (1<<(SIGALRM-1));
                (*p)->alarm = 0;
            }
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                (*p)->state==TASK_INTERRUPTIBLE)
            {
                (*p)->state=TASK_RUNNING;
                fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
            }
        }

    /* this is the scheduler proper: */

    while (1) {
        -- 插入 --
    }
}
```

122,5-26 26%

```
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                    (*p)->priority;
    }
    if (task[next]->pid != current->pid)
    {
        if (current->state == TASK_RUNNING)
            fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);
            fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);
    }
    switch_to(next);
}

int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    -- 插入 --
}
```

149,3-9 33%

`sleep_on()`:


```

void sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp = *p;
    *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
    schedule();
    if (tmp)
    {
        tmp->state=0;
        fprintf(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
    }
}

```

interruptible_sleep_on():

```

void interruptible_sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp=*p;
    *p=current;
repeat: current->state = TASK_INTERRUPTIBLE;
    fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
    schedule();
    if (*p && *p != current) {
        (**p).state=0;
        fprintf(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
        goto repeat;
    }
    *p=NULL;
    if (tmp)
    {
        tmp->state=0;
        fprintf(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
    }
}

```

sys_pause():

```
int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    if (current->pid != 0)
        fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
    schedule();
    return 0;
}
```

wake_up():

```
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        if ((*p).state == TASK_UNINTERRUPTIBLE)
            fprintf(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
        (*p).state=0;
        *p=NULL;
    }
}
```

然后，需要将其中调用的 waitpid 等部分进行修改，这些均存放在 exit.c 中，所以将与其关于 state 部分的设置一并修改成需要的结果，具体操作如下：

sys_waitpid():

```
if (options & WNOHANG)
    return 0;
current->state=TASK_INTERRUPTIBLE;
fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
schedule();
if (!(current->signal &= ~(1<<(SIGCHLD-1))))
    goto repeat;
else
```

do_exit():

```
if (current->leader)
    kill_session();
current->state = TASK_ZOMBIE;
fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies);
current->exit_code = code;
tell_father(current->father);
schedule();
return (-1);    /* just to suppress warnings */
```

然后，检查无误后，在 0.11 目录下 make all:

```

Root device is (3, 1)
Boot sector 512 bytes.
Setup is 312 bytes.
System is 126721 bytes.
rm system.tmp
rm tools/kernel -f
sync
boen@boen-VMware-Virtual-Platform:~/oslab/linux-0.11$

```

接下来，就到了管理 log 文件的环节，首先，需要把 Linux-0.11 挂载到我的 Ubuntu 上，来完成交换文件的目的，将 process.c（不要带注释）拷贝到/hdc/usr/root 中，之后卸载 hdc

```

boen@boen-VMware-Virtual-Platform:~/oslab$ sudo ./mount-hdc
[sudo] boen 的密码:
boen@boen-VMware-Virtual-Platform:~/oslab$ ./dbg-bochs
umount hdc first

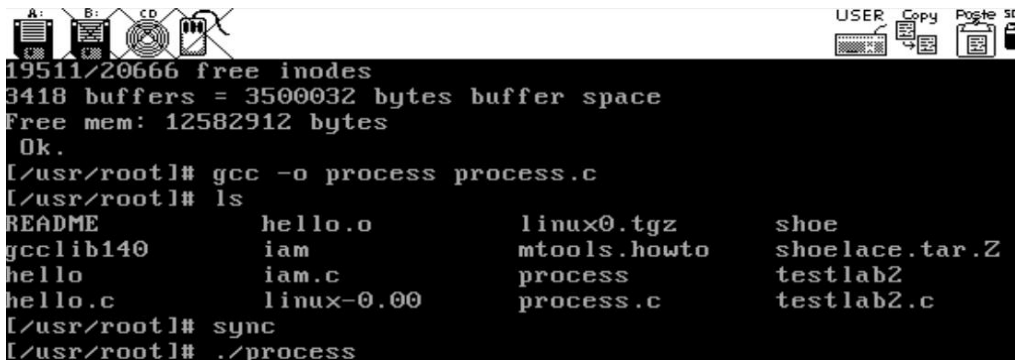
```

```

boen@boen-VMware-Virtual-Platform:~/oslab$ sudo umount hdc
umount: hdc: not mounted.
boen@boen-VMware-Virtual-Platform:~/oslab$ ./dbg-bochs
=====

```

然后，在 bochs 里对 process.c 进行编译，之后输入./process，再启动挂载程序，将 hdc/var/process.log 文件复制粘贴到 oslab 目录下，观察 log 文件：



```

19511/20666 free inodes
3418 buffers = 3500032 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]# gcc -o process process.c
[/usr/root]# ls
README          hello.o          linux0.tgz       shoe
gcclib140       iam              mtools.howto    shoelace.tar.Z
hello           iam.c            process          testlab2
hello.c         linux-0.00       process.c        testlab2.c
[/usr/root]# sync
[/usr/root]# ./process

```

观察 log 文件（红线部分）可以发现 process 进程也被记录在内：

534	3	W	144125
535	0	R	144125
536	4	R	144131
537	4	W	144131
538	0	R	144132
539	4	R	145732
540	16	N	145733
541	16	J	145734
542	4	W	145734
543	16	R	145735
544	17	N	145736
545	17	J	145737
546	18	N	145738
547	18	J	145739
548	19	N	145739
549	19	J	145740
550	20	N	145740
551	20	J	145741
552	21	N	145742
553	21	J	145743
554	22	N	145743
555	22	J	145744
556	23	N	145744
557	23	J	145745
558	24	N	145746
559	24	J	145746
560	16	W	145747

之后，来到数据统计的部分，首先要更改 python 的版本为 2（2.7.18 即可），然后在 `oslab/file` 目录下输入 `python2 stat_log.py ../process.log -g |more`，得到统计结果，以及 `process` 运行后的进程统计结果（红框部分）：

Process	Turnaround	Waiting	CPU Burst	I/O Burst
0	375179	78	6	0
1	26	0	2	23
2	24	0	23	0
3	378154	6	262	377885
4	210314	14	81	210218
5	4	0	3	0
6	325	1	16	307
7	69	0	68	0
8	117	0	116	0
9	29	0	28	0
10	96	0	95	0
11	5	1	3	0
12	9	1	7	0
13	5	0	4	0
14	7	1	5	0

15	4	1	3	0
16	5628	1	14	5612
17	1828	115	0	1712
18	2747	1058	201	1487
19	2981	1951	400	629
20	3936	2790	600	545
21	4534	3298	800	435
22	5110	3787	1000	322
23	5457	4046	1200	210
24	5614	4104	1405	105

最后,是对时间片的修改,根据调度函数 `schedule()` (定义在 `kernel/sched.c` 中),其能够综合考虑进程优先级并能动态反馈调整时间片,其基本流程如下:

1. 是选取 `counter` 值最大的就绪进程进行调度。其中运行态进程(即 `current`)的 `counter` 数值会随着时钟中断而不断减 1。

2. 而当没有 `counter` 值大于 0 的就绪进程时,要对所有的进程做

`(*p)->counter = ((*p)->counter >> 1) + (*p)->priority`

其效果是对所有的进程(包括阻塞态进程)都进行 `counter` 的衰减,并再累 `priority` 值。这样一个进程在阻塞队列中停留的时间越长,其优先级越大,被分配的时间片也就会越大。

而进程的 `counter` 是在 `fork()` 中设定的,具体而言,时间片 `p->counter` 在进程创建之初就被初始化为 `p->priority`,只与优先级 `priority` 有关

`*p = *current;` //用来复制父进程的 PCB 数据信息,包括 `priority` 和 `counter`

`p->counter = p->priority;` //初始化 `counter`

每个进程的 `priority` 都是继承自父亲进程的,并且假定不会调用 `nice` 系统调用,因此时间片的初值就是进程 0 的 `priority`,即宏 `INIT_TASK` (定义在 `include/linux/sched.h` 中)

`#define INIT_TASK \`

`{ 0,15,15, }//分别对应 state;counter;和 priority;.....`

那么,只需要修改 `INIT_TASK` 的 `priority` 值即可完成各进程初始化时间片的修改。

将每次所获得的 log 放到 `oslab/file` 目录下,并执行以下语句:

`python2 stat_log.py process.log 16 17 18 19 20 21 22 23 24 -g`

分别修改了 `priority` 为 5 15 25,结果如下:

```

6 17 18 19 20 21 22 23 24 -g
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
    16          7222         1         14         7206
    17          7218        5617        1600          0
    18          7026        5520        1400         105
    19          6663        5252        1200         210
    20          6075        4759        1000         315
    21          5226        4005         800         420
    22          4397        3270         600         526
    23          3299        2269         400         629
    24          2698        1027         200        1470
Average:    5536.00    3524.44
Throughout: 0.12/s

```

```

g 16 17 18 19 20 21 22 23 24 -g
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
    16          5628         1         14         5612
    17          1828        115          0        1712
    18          2747       1058         201        1487
    19          2981       1951         400         629
    20          3936       2790         600         545
    21          4534       3298         800         435
    22          5110       3787        1000         322
    23          5457       4046        1200         210
    24          5614       4104        1405         105
Average:    4203.89    2350.00
Throughout: 0.16/s

```

```

6 17 18 19 20 21 22 23 24 -g
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
    16          7222         1         14         7206
    17          7217        5617        1600          0
    18          7016        5490        1400         125
    19          6651        5222        1200         229
    20          6118        4742        1000         375
    21          5313        4059         800         453
    22          4300        3076         600         623
    23          3327        2198         400         728
    24          3123        1178         200        1744
Average:    5587.44    3509.22
Throughout: 0.12/s

```

5.3. 实验报告

完成实验后，在实验报告中回答如下问题：

5.3.1 结合自己的体会，谈谈从程序设计者的角度看，单进程编程和多进程编程最大的区别是什么？

从程序设计者的角度来看,单进程编程与多进程编程的最大区别在于资源管理和控制的复杂性。单进程编程中,程序在一个单一的执行流中运行,资源管理和任务调度相对简单,因为所有的操作都在同一个上下文中进行,无需考虑进程间通信和同步问题。然而,在多进程编程中,每个进程都有自己的内存空间和执行上下文,这增加了程序设计的复杂性。设计者需要处理进程间的通信(如管道、消息队列)、数据同步(如互斥锁、信号量)以及资源竞争问题,以确保程序的并发性和稳定性。此外,多进程编程可以提高程序的响应性和吞吐量,但同时也带来了更高的资源消耗和设计难度。

5.3.2 你是如何修改时间片的? 仅针对样本程序建立的进程,在修改时间片前后, log 文件的统计结果(不包括 Graphic)都是什么样? 结合你的修改分析一下为什么会这样变化,或者为什么没变化?

在 Linux0.11 内,修改时间片分别为 5、15 (default)、25, log 文件的统计结果(不包括 Graphic)吞吐率 `throughout` 分别为 0.12、0.16、0.12。

观察到随时间片的增大 `n`, `throughout` 先升高后降低,当时间片较小,进程调度主要由时间片超时引起,而随着时间片的增大,因中断或睡眠导致的调度次数增加,时间片超时引起的调度次数减少。因此,为了提高内核效率,需要合理设置时间片,避免过大或过小,以平衡因超时和中断睡眠导致的调度次数。

5.3.1. 评分标准

`process.c`, 50%

日志文件建立成功, 5%

能向日志文件输出信息, 5%

5 种状态都能输出, 10% (每种 2%)

调度算法修改, 10%

实验报告, 20%