## 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。

## 实验内容

* 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 Linux0.11 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。
* 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（在 /home/teacher/ 目录下） ——进行统计。
* 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。

## 实验报告

|  |
| --- |
| * 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？ * 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？ |
| 可以充分利用CPU和外设资源 |
| 修改sched.c 中INIT\_TASK 的初始值，时间片越大吞吐率越低，平均等待时间越小，因为较大的的时间片导致进程切换较慢，使得CPU和外设的利用率低，同时还需要更长的时间等待切换 |

## 实验步骤

1. 编写样本程序 process.c

|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <time.h>  #include <sys/times.h>  #define HZ 100  void cpuio\_bound(int last, int cpu\_time, int io\_time);  int main(int argc, char \* argv[])  {  pid\_t p\_tab[16];  int i;  for(i=0;i<8;i++)  {  if(!(p\_tab[i]=fork()))  {  cpuio\_bound(16,16-2\*i,2\*i);  return 0;  }  }  wait(&i);  for(i=0;i<8;i++)  printf("%d ",p\_tab[i]);  return 0;  }  /\*  \* 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)  {  struct tms start\_time, current\_time;  clock\_t utime, stime;  int sleep\_time;  while (last > 0)  {  /\* CPU Burst \*/  times(&start\_time);  /\* 其实只有t.tms\_utime才是真正的CPU时间。但我们是在模拟一个  \* 只在用户状态运行的CPU大户，就像“for(;;);”。所以把t.tms\_stime  \* 加上很合理。\*/  do  {  times(&current\_time);  utime = current\_time.tms\_utime - start\_time.tms\_utime;  stime = current\_time.tms\_stime - start\_time.tms\_stime;  } while ( ( (utime + stime) / HZ ) < cpu\_time );  last -= cpu\_time;  if (last <= 0 )  break;  /\* IO Burst \*/  /\* 用sleep(1)模拟1秒钟的I/O操作 \*/  sleep\_time=0;  while (sleep\_time < io\_time)  {  sleep(1);  sleep\_time++;  }  last -= sleep\_time;  }  } |

1. 修改main.c以打开log文件

|  |
| --- |
| 注释原有部分 |
| 2020-05-04 22-21-39 的屏幕截图 |
| 前移 |
| 2020-05-04 22-22-20 的屏幕截图 |

1. 编写printk以写入log

|  |
| --- |
| /\*  \* linux/kernel/printk.c  \*  \* (C) 1991 Linus Torvalds  \*/  /\*  \* When in kernel-mode, we cannot use printf, as fs is liable to  \* point to 'interesting' things. Make a printf with fs-saving, and  \* all is well.  \*/  #include <stdarg.h>  #include <stddef.h>  #include <linux/kernel.h>  static char buf[1024];  extern int vsprintf(char \* buf, const char \* fmt, va\_list args);  #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);  /\* 如果输出到stdout或stderr，直接调用sys\_write即可 \*/  if (fd < 3)  {  \_\_asm\_\_("push %%fs\n\t"  "push %%ds\n\t"  "pop %%fs\n\t"  "pushl %0\n\t"  /\* 注意对于Windows环境来说，是\_logbuf,下同 \*/  "pushl $logbuf\n\t"  "pushl %1\n\t"  /\* 注意对于Windows环境来说，是\_sys\_write,下同 \*/  "call sys\_write\n\t"  "addl $8,%%esp\n\t"  "popl %0\n\t"  "pop %%fs"  ::"r" (count),"r" (fd):"ax","cx","dx");  }  else  /\* 假定>=3的描述符都与文件关联。事实上，还存在很多其它情况，这里并没有考虑。\*/  {  /\* 从进程0的文件描述符表中得到文件句柄 \*/  if (!(file=task[0]->filp[fd]))  return 0;  inode=file->f\_inode;  \_\_asm\_\_("push %%fs\n\t"  "push %%ds\n\t"  "pop %%fs\n\t"  "pushl %0\n\t"  "pushl $logbuf\n\t"  "pushl %1\n\t"  "pushl %2\n\t"  "call file\_write\n\t"  "addl $12,%%esp\n\t"  "popl %0\n\t"  "pop %%fs"  ::"r" (count),"r" (file),"r" (inode):"ax","cx","dx");  }  return count;  }  int printk(const char \*fmt, ...)  {  va\_list args;  int i;  va\_start(args, fmt);  i=vsprintf(buf,fmt,args);  va\_end(args);  \_\_asm\_\_("push %%fs\n\t"  "push %%ds\n\t"  "pop %%fs\n\t"  "pushl %0\n\t"  "pushl $buf\n\t"  "pushl $0\n\t"  "call tty\_write\n\t"  "addl $8,%%esp\n\t"  "popl %0\n\t"  "pop %%fs"  ::"r" (i):"ax","cx","dx");  return i;  } |

1. 寻找状态切换点 exit.c

|  |
| --- |
| 2020-05-04 23-07-02 的屏幕截图  2020-05-04 23-08-17 的屏幕截图 |

1. 寻找状态切换点 fork.c

|  |
| --- |
| 2020-05-04 23-09-09 的屏幕截图  2020-05-04 23-09-19 的屏幕截图 |

1. 寻找状态切换点sched.c

（状态变换较多，只需搜索state，因为状态变了，state一定变化）

|  |
| --- |
| 2020-05-04 23-29-17 的屏幕截图2020-05-04 23-24-48 的屏幕截图2020-05-04 23-38-30 的屏幕截图2020-05-04 23-25-22 的屏幕截图2020-05-04 23-25-41 的屏幕截图2020-05-04 23-25-55 的屏幕截图2020-05-04 23-26-15 的屏幕截图2020-05-04 23-26-31 的屏幕截图2020-05-04 23-26-58 的屏幕截图 |

1. 测试结果

|  |
| --- |
| 2020-05-05 10-34-04 的屏幕截图  2020-05-05 10-35-44 的屏幕截图 |

1. 五种状态都有，且能建立日志文件和输出信息

|  |
| --- |
|  |

1. 更改调度方式

|  |
| --- |
| 2020-05-05 10-36-21 的屏幕截图  2020-05-05 10-39-34 的屏幕截图 |

1. 结果时间片分别为 1 15 150

|  |
| --- |
| 运行截图（编译后直接运行都一样） |
| 2020-05-05 09-14-23 的屏幕截图 |
| 2020-05-05 10-53-45 的屏幕截图 |