卡通人物

描述已自动生成

**操作系统**

**课程实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 实验名称 | 进程运行轨迹的跟踪与统计 | | |
| 姓名 | 常添 | 院系 | 计算学部 |
| 班级 | 2203102 | 学号 | 2022111699 |
| 任课教师 | 郑铁然 | 指导教师 | 郑铁然 |
| 实验地点 | G709 | 实验时间 | 2024.12.15 |

****

**实验5：进程运行轨迹的跟踪与统计**

**1. 实验目的**

* 掌握 Linux 下的多进程编程技术，通过 fork() 创建和管理多个子进程。
* 深入理解进程状态的生命周期，包括就绪态、运行态、阻塞态等状态的切换。
* 实现对进程运行轨迹的跟踪和记录，通过分析日志文件 /var/process.log，统计进程的相关运行数据（如等待时间、完成时间和运行时间）。
* 修改时间片轮转调度算法，量化分析时间片大小对进程调度性能的影响。
* 提高对调度算法原理的理解，积累调试和分析实际操作系统性能的经验。

**2. 实验环境**

* **主机环境：** 
  + 操作系统：Ubuntu 或其他 Linux 发行版。
  + 虚拟机：Bochs 模拟器（建议使用版本 2.4.\*，避免缓冲区问题）。
  + 编程语言：C语言，用于编写样本程序和修改 Linux 0.11 源代码。
  + Python：用于统计分析日志文件（stat\_log.py）。
* **实验环境：** 
  + Linux 0.11 内核源码。
  + Bochs 虚拟机下运行 Linux 0.11。
  + 日志文件存储路径：/var/process.log。

**3. 实现功能**

1. **多进程样本程序：**
   * 编写一个样本程序 process.c，利用 fork() 创建多个并行运行的子进程。
   * 每个子进程模拟不同的 CPU 和 I/O 任务模式，通过调用 cpuio\_bound() 函数进行实际运行。
2. **日志文件 /var/process.log 的生成和记录：**
   * 在 Linux 0.11 内核中修改代码，实现对所有进程状态切换的跟踪。
   * 每当发生状态切换（新建、新增就绪队列、运行、阻塞、退出），在日志文件中记录如下格式： pid 状态 滴答时间
3. **进程调度算法修改：**
   * 修改时间片轮转调度算法，调整进程时间片分配策略，观察其对进程调度性能的影响。
4. **数据统计与分析：**
   * 利用日志文件，通过 Python 脚本 stat\_log.py 计算以下指标：
     + 平均周转时间（从创建到退出的时间）。
     + 平均等待时间（进程在就绪队列中的累计时间）。
     + 系统吞吐率（单位时间完成的进程数）。
   * 比较不同时间片设置下的统计结果，并进行性能分析。

**4. 代码修改流程：**

**4.1 样本程序 process.c 的编写**

1. 创建一个样本程序，利用 fork() 创建若干子进程。
2. 每个子进程调用 cpuio\_bound() 模拟运行。
3. 父进程等待所有子进程退出后结束。

process.c 的完整代码：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <sys/times.h>

#define HZ 100

void cpuio\_bound(int total\_time, int cpu\_time, int io\_time);

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

pid\_t pid1, pid2;

int exited\_pid;

printf("Parent Process PID = [%d]\n", getpid());

pid1 = fork();

if (pid1 == 0) {

/\* First child process \*/

printf("[%d] is running now.\n", getpid());

cpuio\_bound(10, 1, 0);

exit(0);

}

pid2 = fork();

if (pid2 == 0) {

/\* Second child process \*/

printf("[%d] is running now.\n", getpid());

cpuio\_bound(10, 1, 0);

exit(0);

}

/\* Wait for child processes to exit \*/

exited\_pid = wait(NULL);

printf("[%d] has exited.\n", exited\_pid);

exited\_pid = wait(NULL);

printf("[%d] has exited.\n", exited\_pid);

printf("This is Changtian's OS lab.\n");

return 0;

}

/\*

Simulates CPU-bound and I/O-bound operations.

@param total\_time Total time (in seconds) for the operation.

@param cpu\_time Continuous CPU usage time (in seconds).

@param io\_time I/O operation time (in seconds).

If total\_time > cpu\_time + io\_time, the CPU and I/O operations are repeated.

\*/

void cpuio\_bound(int total\_time, int cpu\_time, int io\_time) {

struct tms start\_time, current\_time;

clock\_t user\_time, system\_time;

int sleep\_time;

while (total\_time > 0) {

/\* Simulate CPU burst \*/

times(&start\_time);

do {

times(&current\_time);

user\_time = current\_time.tms\_utime - start\_time.tms\_utime;

system\_time = current\_time.tms\_stime - start\_time.tms\_stime;

} while (((user\_time + system\_time) / HZ) < cpu\_time);

total\_time -= cpu\_time;

if (total\_time <= 0)

break;

/\* Simulate I/O burst \*/

sleep\_time = 0;

while (sleep\_time < io\_time) {

sleep(1);

sleep\_time++;

}

total\_time -= sleep\_time;

}

}

**4.2 定义日志记录函数**

在 kernel/printk.c 中新增 fprintk() 函数，用于内核态记录日志。

代码实例：

#include <stdarg.h>

#include <linux/sched.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) {

// 输出到stdout或stderr

\_\_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"

"call sys\_write\n\t"

"addl $8,%%esp\n\t"

"popl %0\n\t"

"pop %%fs"

::"r" (count), "r" (fd): "ax", "cx", "dx");

} else {

file = task[1]->filp[fd]; // 获取文件描述符

if (!file || !file->f\_inode->i\_dev) 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;

}

**4.3 记录进程状态切换**

在以下函数中添加 fprintk() 调用，记录进程状态：

1. 新建（N）和就绪（J）：copy\_process()（kernel/fork.c）。
2. 运行态（R）和就绪态（J）：schedule()（kernel/sched.c）。
3. 阻塞态（W）：sleep\_on() 和 interruptible\_sleep\_on()（kernel/sched.c）。
4. 退出（E）：do\_exit()（kernel/exit.c）。

// 在 copy\_process() 中

fprintk(log\_fd, "%d\t%c\t%ld\n", current->pid, 'N', jiffies);

fprintk(log\_fd, "%d\t%c\t%ld\n", current->pid, 'J', jiffies);

// 在 schedule() 中记录状态切换

if (current->state == TASK\_RUNNING) {

fprintk(log\_fd, "%d\t%c\t%ld\n", current->pid, 'R', jiffies);

}

// 在 do\_exit() 中记录退出

fprintk(log\_fd, "%d\t%c\t%ld\n", current->pid, 'E', jiffies);

**4.4 调整调度算法**

修改 kernel/sched.c 中的调度逻辑，调整时间片大小。

代码修改点：

1. 在 copy\_process() 中：
2. p->counter = p->priority \* 2; // 增加初始时间片
3. 在 schedule() 中：
4. (\*p)->counter = ((\*p)->counter >> 1) + (\*p)->priority + 5; // 增加重分配时间片

**4.5 修改 kernel/fork.c**

fork.c 负责创建新进程，是记录新建态 (N) 和进入就绪态 (J) 的关键位置。

**4.5.1 记录新建态（N）**

在 copy\_process() 函数中找到以下代码：

p->pid = last\_pid;

在其后添加：

fprintk(log\_fd, "%d\t%c\t%ld\n", p->pid, 'N', jiffies);

**4.5.2 记录进入就绪态（J）**

在 copy\_process() 函数末尾，设置子进程状态为 TASK\_RUNNING 的地方：

p->state = TASK\_RUNNING;

在其后添加：

fprintk(log\_fd, "%d\t%c\t%ld\n", p->pid, 'J', jiffies);

**4.6 修改 kernel/exit.c**

exit.c 负责进程退出和父进程的等待，是记录退出态（E）的地方。

**4.6.1 记录退出态（E）**

在 do\_exit() 函数中找到通知父进程子进程退出的地方：

current->state = TASK\_ZOMBIE;

在其前面添加：

fprintk(log\_fd, "%d\t%c\t%ld\n", current->pid, 'E', jiffies);

**4.6.2 父进程等待子进程退出**

在 sys\_waitpid() 中找到以下代码：

p->state = TASK\_RUNNING;

在其后添加：

fprintk(log\_fd, "%d\t%c\t%ld\n", p->pid, 'J', jiffies);

**4.7 修改 init/main.c**

main.c 是系统启动的入口点，负责初始化日志文件。

**4.7.1 打开日志文件**

找到 main() 函数中的 move\_to\_user\_mode();，在其后添加：

int log\_fd;

log\_fd = open("/var/process.log", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

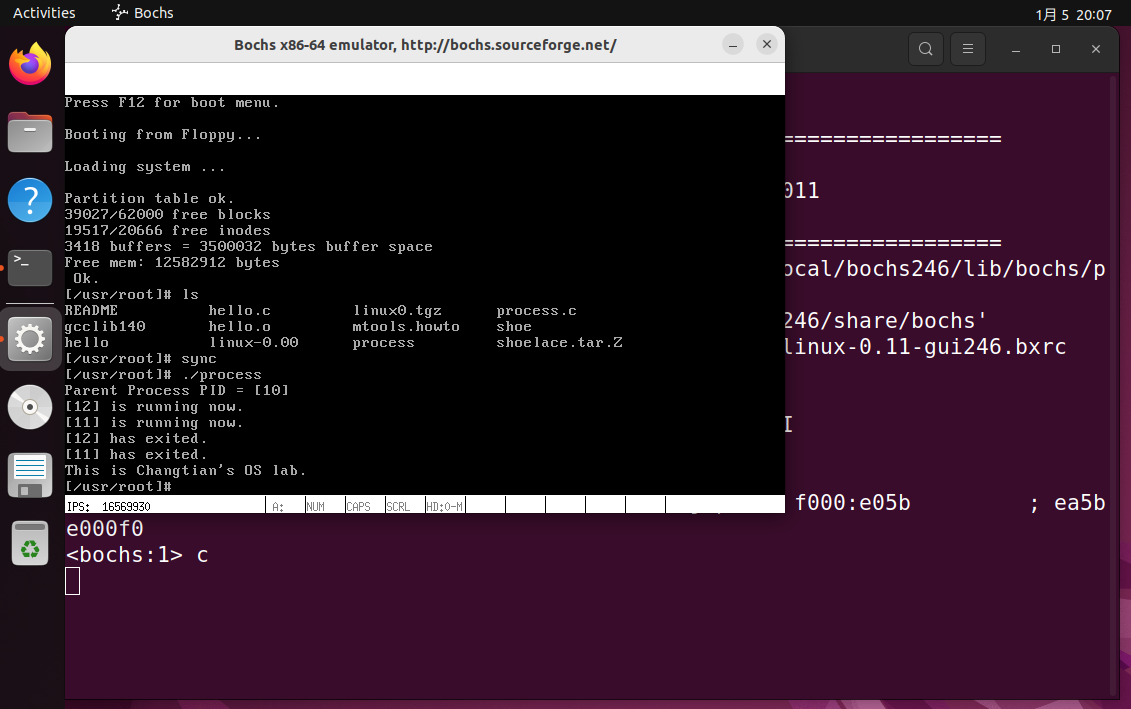
if (log\_fd < 0) {

printk("Failed to open log file\n");

}

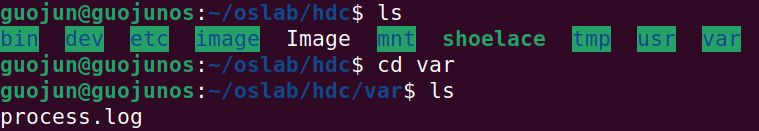
**5. 实验结果**

在bochs上的运行结果：

****

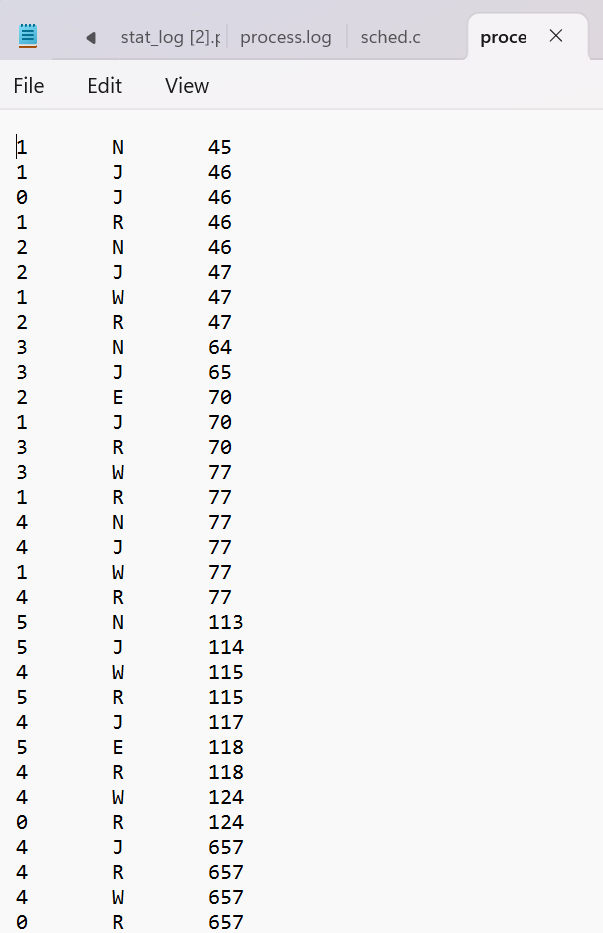
**查看"/var/process.log"的情况：**

在var/目录下创建process.log成功：

****

**图形用户界面, 应用程序

描述已自动生成**

****

可以看到5种状态 N: 新建 (New) J: 进入就绪队列 (Join Ready Queue)

R: 运行 (Running) W: 阻塞 (Waiting) E: 退出 (Exit)

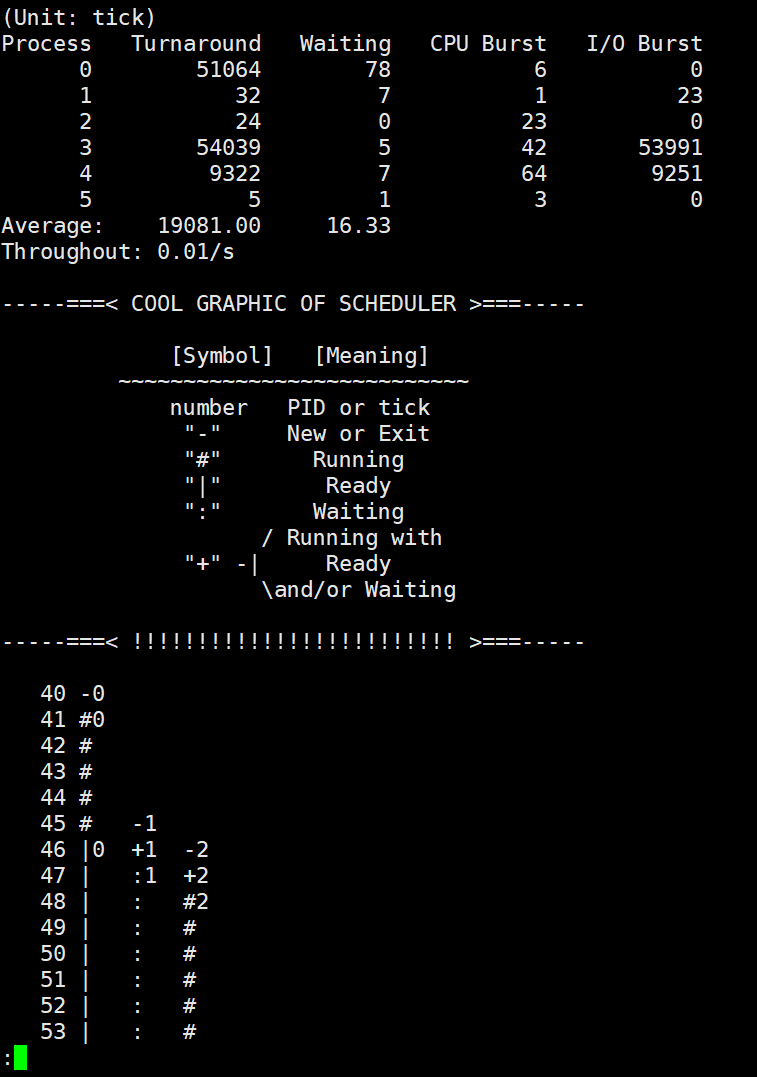
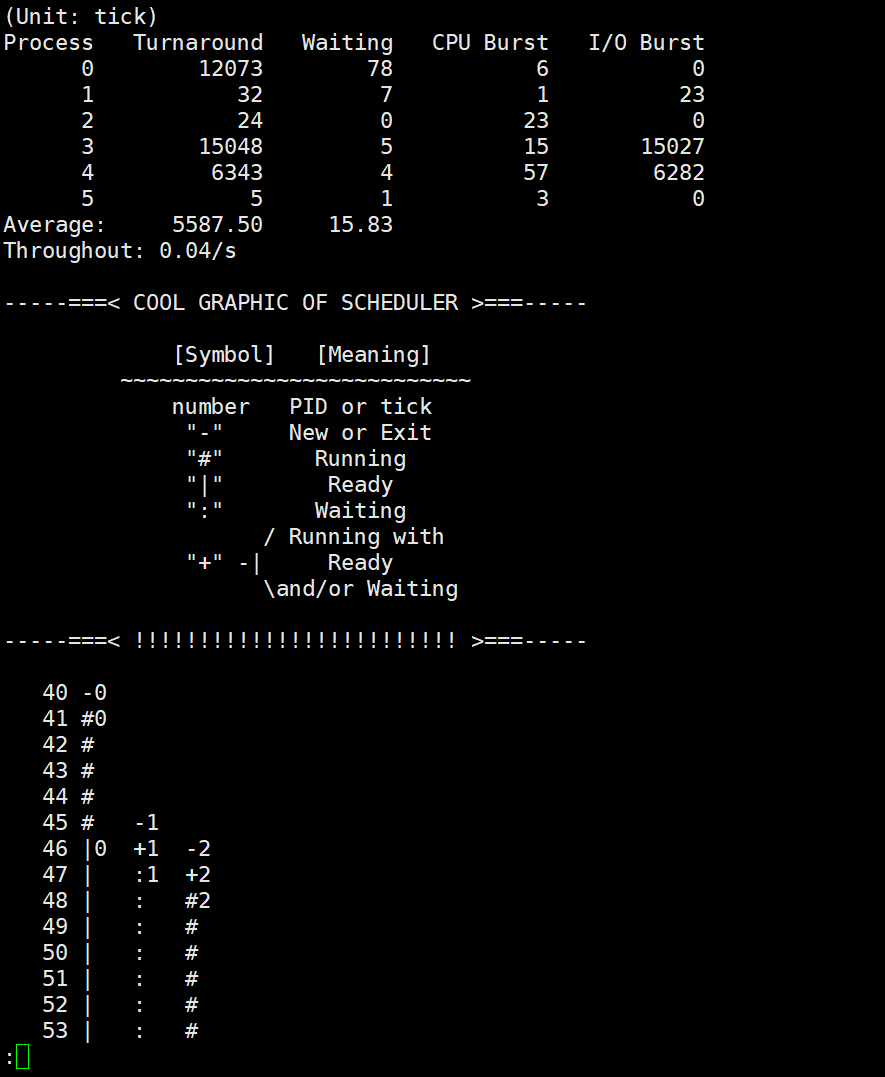
这5种状态在日志文件中都有输出。

**6. 问题回答**

**6.1 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看，单进程编程和多进程编程最大的区别在于并发性和资源管理：

1. 并发性：
   * 单进程编程是串行的，任务按顺序执行，简单易理解，但无法充分利用多核 CPU 的性能。
   * 多进程编程可以同时执行多个任务，提高了程序的并发性和性能，尤其在多核处理器上，但需要处理复杂的同步与通信问题。
2. 资源管理：
   * 单进程只需管理自身的资源，逻辑较为简单。
   * 多进程需要处理资源共享与竞争（如文件、内存、网络连接），需要设计合适的同步机制（如信号量、共享内存），以避免资源冲突或死锁。

**6.2 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？**

**左图为修改前的日志分析结果，右图为修改后的日志分析结果，下面是详细分析:**

**修改时间片的方式**

在 Linux 0.11 的调度算法中，时间片的分配逻辑位于 kernel/sched.c 文件中。默认情况下，每次重新分配时间片时，counter 的值通过以下公式计算：

(\*p)->counter = ((\*p)->counter >> 1) + (\*p)->priority;

* (\*p)->counter >> 1：当前时间片值右移一位，相当于除以 2（减少旧时间片的权重）。
* (\*p)->priority：进程优先级，用于生成新的时间片。

**修改内容：** 将时间片分配公式中的右移一位改为右移两位，增加时间片的长度：

(\*p)->counter = ((\*p)->counter >> 2) + (\*p)->priority;

* (\*p)->counter >> 2：当前时间片值右移两位，相当于除以 4（更大幅度降低旧时间片的影响）。

此修改使得进程可以获得更长的连续运行时间，从而减少上下文切换的频率。

**修改前（第一张图）**

* **指标**：
  + **平均周转时间**: 19081.00 ticks
  + **平均等待时间**: 16.33 ticks
  + **吞吐率**: 0.01/s
* **观察**：
  + 周转时间非常高，表明任务完成耗时较长。
  + 吞吐率较低，说明单位时间内完成的任务数很少。
  + I/O 密集型任务（如 PID 3 和 4）的 I/O 时间占据了主要部分，导致进程切换较频繁。

**修改后（第二张图）**

* **指标**：
  + **平均周转时间**: 5587.50 ticks
  + **平均等待时间**: 15.83 ticks
  + **吞吐率**: 0.04/s
* **观察**：
  + 平均周转时间大幅降低，表明进程完成的效率显著提升。
  + 吞吐率提高了 4 倍，表明单位时间内完成的任务数增加。
  + I/O 密集型任务的运行时间减少，表明 I/O 等待的阻塞时间有所优化。