linux_sir L18 2016-08-11 08:23 Linux 计算机专业课 146 0

实验代码

“操作系统原理与实践”实验报告

进程运行轨迹的跟踪与统计

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。

实验过程：

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；

```
#include    <stdio.h>
#include    <unistd.h>
#include    <time.h>
#include    <sys/times.h>

#define   HZ    100

void   cpuio_bound    int          int               int            ;

int   main   int        char
{
    int   sub_pid1,sub_pid2,sub_pid3;
    int   i  =  0;
    if  ((sub_pid1=fork())        <  0)
    {
        printf    ( "fork    error!!\n"      );

    }else    if  (sub_pid1      ==  0)
    {
    cpuio_bound(    10 , 0, 3);

    }else
    {
        printf    ( "new    sub_process:%ld       \n"  ,sub_pid1);
        waitpid(sub_pid1,        NULL, 0);
    }

    if  ((sub_pid2=fork())        <  0)
    {
        printf    ( "fork    error!!\n"      );

    }else    if  (sub_pid2      ==  0)
    {
    cpuio_bound(    10 , 3, 0);

    }else
    {
        printf    ( "new    sub_process:%ld       \n"  ,sub_pid2);
        waitpid(sub_pid2,        NULL, 0);
    }


    if  ((sub_pid3=fork())        <  0)
    {
        printf    ( "fork    error!!\n"      );

    }else    if  (sub_pid3      ==  0)
    {
        cpuio_bound(    10 , 3, 3);

    }else
    {
        printf    ( "new    sub_process:%ld       \n"  ,sub_pid3);
        waitpid(sub_pid3,        NULL, 0);
    }
    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          int               int
{
    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;
    }
}
```

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。

首先提供打印日志文件的功能

这里请注意在内核里打印日志，与用户空间打印日志使用的接口不同，不能再使用那些系统调用接口了。 在kernel/printk.c中添加fprintk，直接参照write的系统调用的实现部分，代码如下：

```
/*
 *   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    "sys/stat.h"
#include    "linux/sched.h"

#include    <linux/kernel.h>

static     char    buf[  1024  ];
static     char    logbuf[    1024  ];

extern    int    vsprintf     char             const    char                            ;

int    fprintk     int          const    char
{
    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)     /*  如果是输出到stdout 或stderr ,   则直接调用sys_write即可*/
    {
        __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     /*若>=3的描述符都与文件关联，事实上，还存在很多其它情况，这里不考虑o*/
    {

        if    (!(file=task[       0] - >filp[fd]))         /*  从进程0的文件描述表中取得文件句柄，所以，需要在main
数初始化时，进程0里创建日志的文件描述符*/
            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;
}
```

这里将创建文件描述符前移到了进程0那边，并让日志的文件描述符为3,并通过进程0的资源，方便被子进程直接继承使用。相应代码如下：

输入代码

```
                memory_end = 16*1024*1024;
        if (memory_end > 12*1024*1024)
                buffer_memory_end = 4*1024*1024;
        else if (memory_end > 6*1024*1024)
                buffer_memory_end = 2*1024*1024;
        else
                buffer_memory_end = 1*1024*1024;
        main_memory_start = buffer_memory_end;
#ifdef RAMDISK
        main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
#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();
/* add start */
/* 此处的几个文件描述符可被子进程直接继承*/
        setup((void *) &drive_info); //这句很关键，没有它，系统启动时找不到root inode
/*
* setup是一个系统调用，用于读取硬件分区表信息/并加载虚拟盘（若存在的话）并安装根文件系统设备
对应的系统调用实现是sys_setup()
所以，少这这句，启动时就会报 no root inode 错误信息了
**/
        (void) open("/dev/tty0",O_RDWR,0);
        (void) dup(0);
        (void) dup(0);
        (void)open("/var/process.log",O_CREAT|O_TRUNC|O_WRONLY,0666);
//      fprintk(3,"log start");
/* add end */
        if (!fork()) {          /* we count on this going ok */
                init();
        }
/*
 *   NOTE!!   For any other task 'pause()' would mean we have to get a
 * signal to awaken, but task0 is the sole exception (see 'schedule()')
 * as task 0 gets activated at every idle moment (when no other tasks
 * can run). For task0 'pause()' just means we go check if some other
 * task can run, and if not we return here.
 */
        for(;;) pause();
}

static int printf(const char *fmt, ...)
{
        va_list args;
        int i;
```

注意：这里这行代码

```
setup((    void    *)    &drive_info);        //这句很关键，没有它，系统启动时找不到root  inode
```

要一块带过来，没有它，系统就会报错，在加载根设备文件系统时，就会报找不到root inode

其次就是找到进程的各个时间点，并打印输出新的状态

进程创建 除进程0外，其它进程都是通过fork()创建新进程的，那么相应的代码在copy_process函数中（kernel/fork.c）：

```
int    copy_process(int        nr,long      ebp,long      edi,long      esi,long      gs,long      none,
        long   ebx,long     ecx,long     edx,
        long   fs,long      es,long      ds,
        long   eip,long      cs,long      eflags,long       esp,long      ss)
{
    struct      task_struct        *p;
    int    i;
    struct      file     *f;

    p  =  (struct      task_struct       *)   get_free_page();
    if    (!p)
        return    -  EAGAIN;
    task[nr]     =  p;
    *p  =  *current;            /*   NOTE!  this    doesn't     copy    the    supervisor      stack     */
    p - >state     =  TASK_UNINTERRUPTIBLE;
    p - >pid    =  last_pid;
    / *输出新创建进程的PID*/
    //printk("%ld\t%c\t%ld\n",p              - >pid,'N',jiffies);
    fprintk(     3,  "%ld\t%c\t%ld\n"       ,p - >pid    ,    'N'   ,jiffies);
    p - >father      =  current     - >pid;
    p - >counter      =  p - >priority;
    p - >signal      =  0;
    p - >alarm     =  0;
    p - >leader      =  0;             /*   process     leadership       doesn't     inherit      */
    p - >utime     =  p- >stime     =  0;
    p - >cutime     =  p - >cstime      =  0;
    p - >start_time         =  jiffies;
    p - >tss.back_link          =  0;
    p - >tss.esp0        =  PAGE_SIZE   +  (long)      p;
    p - >tss.ss0        =  0x10 ;
    p - >tss.eip        =  eip;
    p - >tss.eflags         =  eflags;
    p - >tss.eax        =  0;
    p - >tss.ecx        =  ecx;
    p - >tss.edx        =  edx;
    p - >tss.ebx        =  ebx;
    p - >tss.esp        =  esp;
    p - >tss.ebp        =  ebp;
    p - >tss.esi        =  esi;
    p - >tss.edi        =  edi;
    p - >tss.es        =  es   &  0xffff    ;
    p - >tss.cs        =  cs   &  0xffff    ;
    p - >tss.ss        =  ss   &  0xffff    ;
    p - >tss.ds        =  ds   &  0xffff    ;
    p - >tss.fs        =  fs   &  0xffff    ;
    p - >tss.gs        =  gs   &  0xffff    ;
    p - >tss.ldt        =  _LDT(nr);
    p - >tss.trace_bitmap          =  0x80000000    ;
    if    (last_task_used_math         ==  current)
        __asm__(  "clts     ;   fnsave     %0"::  "m"    (p - >tss.i387));
    if    (copy_mem(nr,p))        {
        task[nr]      =  NULL;
        free_page((long)         p);
        return    -  EAGAIN;
    }
    for    (i=  0;   i<NR_OPEN;i++)
        if    ((f=p   - >filp[i]))
            f - >f_count++;
    if    (current     - >pwd)
        current    - >pwd - >i_count++;
    if    (current     - >root)
        current    - >root  - >i_count++;
    if    (current     - >executable)
        current    - >executable      - >i_count++;
    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    */
    / *输出就绪状态的进程信息*/
    //printk("%ld\t%c\t%ld\n",p              - >pid,'J',jiffies);
    fprintk(     3,  "%ld\t%c\t%ld\n"       ,current     - >pid    ,   'J'    ,jiffies);
    return     last_pid;
}
```

注：打印是里面的两个fprintk语句。第一个是分配新的PID时，属于新建进程，即'N'状态，创建完之后，就变成就绪状态，即输出"J"

在fork()之后，状态的变换都在调度过程中，代码相应在kernel/sched.c

在schedule()函数中，将正在运行的进程转为阻塞，选择一个就绪状态的进程改为运行，

```
void    schedule(void)
{
    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)
            {
                if  ((*p)    - >state      !=    TASK_RUNNING)
                {
                    fprintk(     3,  "%ld\t%c\t%ld\n"       ,(*p)    - >pid,    'J'    ,jiffies);
                }
                (*p)   - >state=TASK_RUNNING;
            }
        }

/*    this    is    the    scheduler       proper:       */

    while     ( 1)   {
        c   =  - 1;
        next     =  0;
        i   =  NR_TASKS;
        p   =  &task[NR_TASKS];
        while     ( -- i)    {
            if    (!*  --  p)
                continue     ;
            if    ((*p)    - >state       ==  TASK_RUNNING && (*p)   - >counter      >  c)
                c   =  (*p)   - >counter,      next    =  i;
        }
        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)     //若next是当前任务时，实质不会发生switch_to，状态就不
变化
    {
        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"       ,current     - >pid,    'R'   ,jiffies);
    }
    switch_to(next);
}
```

在switch_to()中，将就绪态任务变为真正运行 状态，这里就在switch_to()前面进行标识。标识时，要注意判断新选中的进程和当前进程是否为同一个，若为同一个，则就会发生变化，若不为同一个进程，还要判断此进程的原状态是否为就绪态。

在sys_pause()函数中，进程会由运行态变为阻塞态（“W"）

```
int    sys_pause(void)
{
    if  (current      - >state      ! =   TASK_I NTERRUPTI BLE)     / / 用此条件来判断是否会发生状态变化
    {
        f pri nt k( 3,  "%ld\t%c\t%ld\n"       ,current     - >pid,    'W'  ,jiffies);
    }
    current     - >state      =  TASK_INTERRUPTIBLE;
    schedule();
    return     0;
}
```

在sleep_on()函数中，会将运行的进程改为阻塞态，并唤醒等待队列中第一个任务，变为就绪态

```
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;     //  临时保存当前等待队列的队头，以便后面将当前进程插入后，仍需要找到这个原来的第一个来
进行唤醒
    *p  =  current;
    if  (current      - >state      !=    TASK_UNINTERRUPTIBLE)
    {
        fprintk(     3,  "%ld\t%c\t%ld\n"       ,current     - >pid,    'W'  ,jiffies);
    }
    current     - >state      =  TASK_UNINTERRUPTIBLE;
    //printk      ( "%ld\t%c\t%ld\n"        ,current     - >pid,    'S'   ,jiffies);
    schedule();
    //  将原等待队列的第一个任务唤醒
    if    (tmp)
    {
        if  (tmp  - >state      !=   0)
        {
            fprintk(     3,  "%ld\t%c\t%ld\n"       ,tmp  - >pid,    'J'    ,jiffies);
        }
        tmp - >state     =0;

    }
}
```

在interruptible_sleep_on（）函数中，会将运行的进程改为可中断的阻塞状态，放入等待队列，并会唤醒原等待队列中的第一个

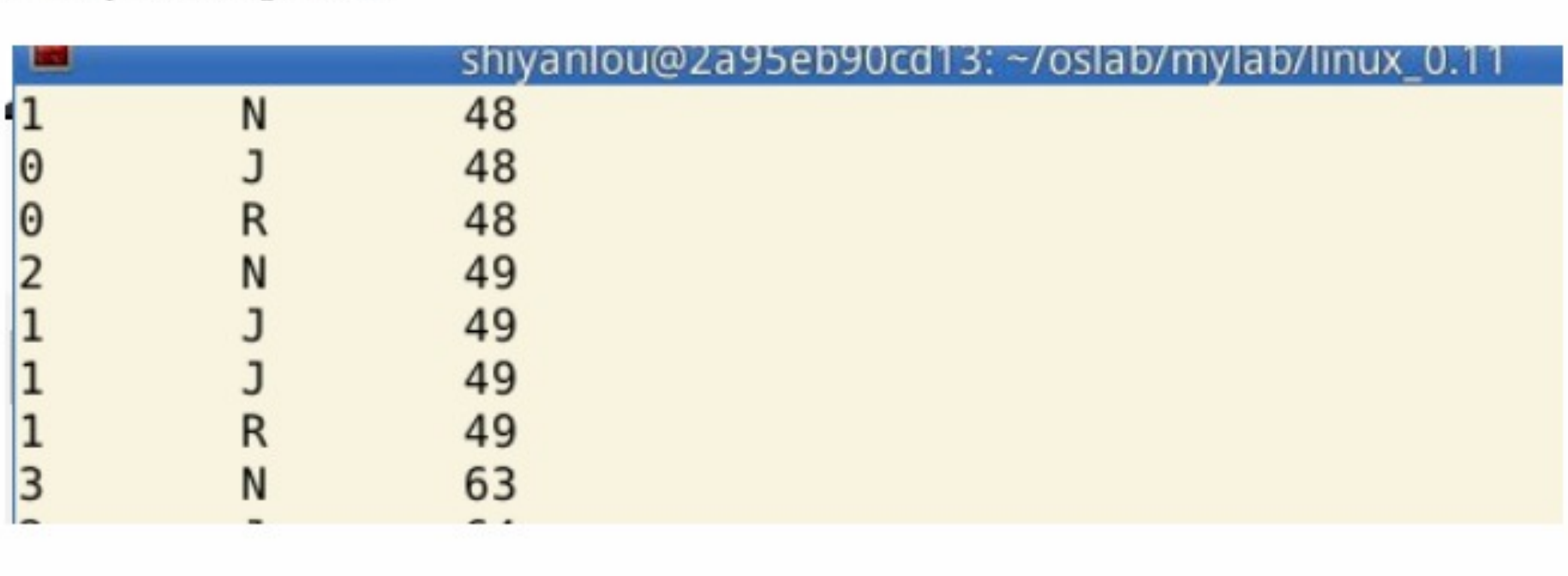
```
void    interruptible_sleep_on(struct                 task_struct         **p)
{
    struct      task_struct        *tmp;

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

这样，几个状态转换的关键点就都打上标记了。

运行程序

相应的process.log的内容：



```
1       N       48
0       J       48
0       R       48
2       N       49
1       J       49
1       J       49
1       R       49
3       N       63
```