

# Project 2

## Task 1

### 实验要求

1. 创建十个CPU-bound程序，并将他们绑定在同一个CPU核心上，修改这些进程(线程)的优先级使得其中5个进程占用大约70%的CPU资源，另外5个进程使用剩下的30%。在同一组中的进程应该具有相同的优先级，使用 top 或 htop 命令验证实验结果。
2. 在相同的CPU核心上，再创建一个实时进程，验证当这个进程在运行时，会抢占其他十个进程

### 实验过程

1. 创建十个CPU-bound程序，并将他们绑定在同一个CPU核心上。

十个程序的创建可以用fork()分别生成十个子进程，让进程运行在指定的CPU上，即修改进程的CPU亲和力，有两种做法：

- a. 通过taskset命令修改

```
taskset -c 0 ./test # -c参数代表要绑定的cpu核，./test代表要运行的程序
```

- b. sched\_setaffinity

`sched_setaffinity` 函数通过 `cpu_set_t` 结构体数据类型的掩码(mask)指定 cpu，掩码的操作可以通过一些宏定义实现，比如 `CPU_SET` 等。操作哪一个线程则通过参数一 `pid` 指定，如果 `pid==0`，那么为当前正在调用 `sched_setaffinity` 函数的线程指定 cpu。

```
cpu_set_t mask;  
CPU_ZERO(&mask);  
CPU_SET(n, &mask);  
sched_setaffinity(0, sizeof(cpu_set_t), &mask);
```

经测试后者优先级大于前者，本实验中采用后者即 `sched_setaffinity`。

2. 修改这些进程(线程)的优先级使得其中5个进程占用大约70%的CPU资源，另外5个进程使用剩下的30%。

CFS分配CPU使用比时，这个比例会受到nice值的影响，nice值低比重就高，nice高比重就低，定量关系为：

$$vruntime = time \times \frac{NICE_0\_LOAD}{weight}$$

nice值与weight的映射关系如下：

```
const int sched_prio_to_weight[40] = {
    /* -20 */    88761,    71755,    56483,    46273,    36291,
    /* -15 */    29154,    23254,    18705,    14949,    11916,
    /* -10 */    9548,    7620,    6100,    4904,    3906,
    /* -5 */    3121,    2501,    1991,    1586,    1277,
    /* 0 */    1024,    820,    655,    526,    423,
    /* 5 */    335,    272,    215,    172,    137,
    /* 10 */    110,    87,    70,    56,    45,
    /* 15 */    36,    29,    23,    18,    15,
};
```

CPU资源之比为7:3，则优先级前者更高，weight值越大，nice值可分别设置为3和7，weight比值为526:215，且子进程之间的优先级相同。

3. 验证当实时进程在运行时，会抢占其他十个进程。

linux的两种实时进程调度算法：

a. SCHED\_FIFO实时调度策略，先到先服务。一旦占用cpu则一直运行。一直运行直到有更高优先级任务到达或自己放弃。

b. SCHED\_RR实时调度策略，时间片轮转。当进程的时间片用完，系统将重新分配时间片，并置于就绪队列尾。放在队列尾保证了所有具有相同优先级的RR任务的调度公平。

通过以下两个函数来获得线程可以设置的最高和最低优先级

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

通过sched\_setscheduler设置调度器

```
pid_t pid = getpid();
struct sched_param param;
param.sched_priority = sched_get_priority_max(SCHED_FIFO);
sched_setscheduler(pid, SCHED_RR, &param); // SCHED_RR
pthread_setschedparam(pthread_self(), SCHED_FIFO, &param); // SCHED_FIFO
```

## 实验步骤

1、创建十个CPU-bound程序

```
nice -n 7 ./test1
nice -n 3 ./test1
```

```
pjq@ubuntu: ~  
  
0[|||||]100.0% Tasks: 119, 252 thr, 212 kthr; 4 running  
1[|||||]0.0% Load average: 9.91 5.21 2.18  
2[|||||]1.3% Uptime: 22:50:48  
3[|||||]0.0%  
Mem[|||||]758M/1.90G  
Swp[|||||]912M/923M  
  
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command  
50977 root 23 3 2496 84 0 R 14.4 0.0 0:27.59 ./test1  
50978 root 23 3 2496 84 0 R 14.4 0.0 0:27.58 ./test1  
50980 root 23 3 2496 84 0 R 14.4 0.0 0:27.59 ./test1  
50979 root 23 3 2496 84 0 R 13.8 0.0 0:27.58 ./test1  
50981 root 23 3 2496 84 0 R 13.8 0.0 0:27.58 ./test1  
50969 root 27 7 2496 80 0 R 5.9 0.0 0:14.25 ./test1  
50970 root 27 7 2496 80 0 R 5.9 0.0 0:14.25 ./test1  
50971 root 27 7 2496 80 0 R 5.9 0.0 0:14.25 ./test1  
50972 root 27 7 2496 80 0 R 5.9 0.0 0:14.25 ./test1  
50973 root 27 7 2496 80 0 R 5.9 0.0 0:14.25 ./test1  
50146 pjq 20 0 6532 2016 864 R 1.3 0.1 0:53.61 /snap/htop/3359/usr/local/bin/htop  
1685 pjq 20 0 4206M 119M 21616 S 0.7 6.1 4:20.22 /usr/bin/gnome-shell  
1854 pjq 20 0 620M 9980 5460 S 0.7 0.5 1:39.95 /usr/bin/vmtoolsd -n vmusr --blockFd 3  
1 root 20 0 164M 5964 3436 S 0.0 0.3 0:15.57 /sbin/init auto noprompt  
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit
```

## 2、创建一个实时进程

```
sudo ./test3 # root权限
```

```
pjq@ubuntu: ~  
  
0[|||||]100.0% Tasks: 121, 252 thr, 211 kthr; 4 running  
1[|||||]13.8% Load average: 11.69 7.57 3.48  
2[|||||]2.7% Uptime: 22:53:11  
3[|||||]9.7%  
Mem[|||||]760M/1.90G  
Swp[|||||]912M/923M  
  
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command  
51069 root RT 0 2364 420 356 R 98.0 0.0 1:20.91 ./test3  
1532 pjq 20 0 317M 33636 7136 S 11.3 1.7 4:03.01 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth  
50001 pjq 20 0 803M 25316 14160 S 8.0 1.3 0:11.00 /usr/libexec/gnome-terminal-server  
1685 pjq 20 0 4206M 116M 21244 S 6.0 6.0 4:21.46 /usr/bin/gnome-shell  
1566 pjq 20 0 317M 33636 7136 R 2.0 1.7 0:39.56 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth  
50146 pjq 20 0 6532 1800 648 R 2.0 0.1 0:55.58 /snap/htop/3359/usr/local/bin/htop  
50971 root 27 7 2496 80 0 R 0.7 0.0 0:17.84 ./test1  
50973 root 27 7 2496 80 0 R 0.7 0.0 0:17.84 ./test1  
50977 root 23 3 2496 84 0 R 0.7 0.0 0:36.36 ./test1  
50978 root 23 3 2496 84 0 R 0.7 0.0 0:36.35 ./test1  
50980 root 23 3 2496 84 0 R 0.7 0.0 0:36.35 ./test1  
50981 root 23 3 2496 84 0 R 0.7 0.0 0:36.36 ./test1  
1 root 20 0 164M 5736 3216 S 0.0 0.3 0:15.57 /sbin/init auto noprompt  
373 root 19 -1 60108 5956 5248 S 0.0 0.3 0:07.64 /lib/systemd/systemd-journald  
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9Kill F10Quit
```

## Task2

### 实验要求

修改Linux源代码，为每个进程添加调度次数的记录

具体要求：

- 1、在 task\_struct 结构体中添加数据成员变量 int ctx，用于记录进程的调度次数
- 2、在进程对应的 /proc/ 目录下添加只读文件 ctx
- 3、当读取 /proc//ctx 时，输出进程当前的调度次数

## 实验过程

### 1、ctx声明

每个进程在内核中都有一个进程控制块(PCB)来维护进程相关的信息,Linux内核的进程控制块是task\_struct结构体。

```
struct task_struct *task;
```

它包含着该进程的信息，ctx在此声明。

```
struct task_struct {
    .....
    randomized_struct_fields_start

    int          ctx; // add here
    void         *stack;
    refcount_t    usage;
```

### 2、ctx初始化

在进程创建时，初始化ctx。有关进程创建的函数在kernel/fork.c，在使用fork/vfork/clone时系统调用底层都将调用fork.c中的kernel\_clone，（5.10.x以前的版本为\_do\_fork）

```
pid_t kernel_clone(struct kernel_clone_args *args)
{
    u64 clone_flags = args->flags;
    struct completion vfork;
    struct pid *pid;
    struct task_struct *p;
    .....
    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    .....
```

在copy\_process函数中查看task\_struct \*p的创建过程。

```
static __latent_entropy struct task_struct *copy_process(
    struct pid *pid,
    int trace,
    int node,
    struct kernel_clone_args *args)
{
    int pidfd = -1, retval;
    struct task_struct *p;
    .....
    p = dup_task_struct(current, node);
```

dup\_task\_struct中的tsk即为创建好的task\_struct

```
static struct task_struct *dup_task_struct(struct task_struct *orig, int node)
{
    struct task_struct *tsk;
    tsk = alloc_task_struct_node(node);
    .....
    tsk->stack = stack;
    tsk->ctx = 0; // add here
}
```

### 3、ctx更新

在调用该进程时对ctx值进行更新。有关进程调度的函数在kernel/sched/core.c中定义，\_\_schedule()是调度器的主函数，主要实现了两个功能，一个是选择下一个要运行的进程，另一个是进程上下文切换context\_switch。在切换到该进程时对ctx进行加一操作。

```
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf)
{
    prepare_task_switch(rq, prev, next);
    arch_start_context_switch(prev);
    .....
    switch_to(prev, next, prev);
    next->ctx++; // add here
    barrier();

    return finish_task_switch(prev);
}
```

### 4、目录创建

proc中各个进程目录文件的创建定义在fs/proc/base.c中，

```
static const struct pid_entry tgid_base_stuff[] = {
    DIR("task",      S_IRUGO|S_IXUGO, proc_task_inode_operations,
        proc_task_operations),
    .....
    REG("environ",   S_IRUSR, proc_environ_operations),
    REG("auxv",       S_IRUSR, proc_auxv_operations),
    ONE("status",     S_IRUGO, proc_pid_status),
    ONE("personality", S_IRUSR, proc_pid_personality),
    ONE("limits",     S_IRUGO, proc_pid_limits),
    .....
#ifdef CONFIG_HAVE_ARCH_TRACEHOOK
    ONE("syscall",    S_IRUSR, proc_pid_syscall),
#endif
    REG("cmdline",    S_IRUGO, proc_pid_cmdline_ops),
    ONE("stat",       S_IRUGO, proc_tgid_stat),
    ONE("statm",      S_IRUGO, proc_pid_statm),
    REG("maps",       S_IRUGO, proc_pid_maps_operations),
    ONE("ctx",        S_IRUGO, proc_pid_ctx) // add here
}
```

在fs/proc/base.c中定义函数proc\_pid\_ctx

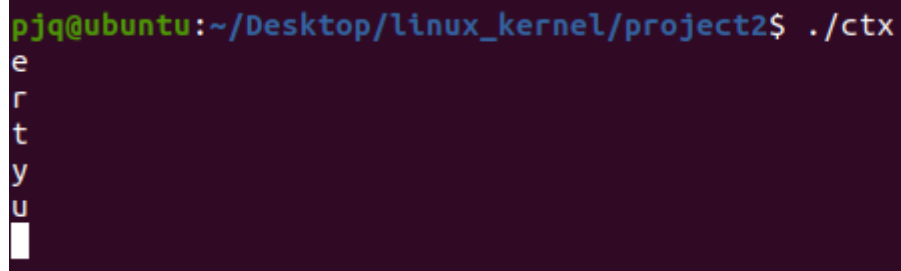
```
static int proc_pid_ctx(struct seq_file *m, struct pid_namespace *ns,
struct pid *pid, struct task_struct *task)
{
    seq_printf(m, "ctx: %d\n", task->ctx);
    return 0;
}
```

## 实验步骤

### 1、编译linux内核

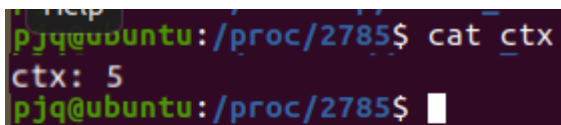
```
make menuconfig
make -j4
make modules_install
make install
```

### 2、运行实例程序



```
pjq@ubuntu:~/Desktop/linux_kernel/project2$ ./ctx
e
r
t
y
u
█
```

### 3、验证



```
pjq@ubuntu:~/Desktop/linux_kernel/project2$ cat /proc/2785/ctx
ctx: 5
pjq@ubuntu:~/Desktop/linux_kernel/project2$
```