Linux kernel Project : Process Scheduler

Task1

1.1 预备知识：

- **普通进程CFS调度策略：**

  在Linux的较高版本内核中，取消了对普通进程的调度预先分配时间片的概念，而是虚拟运行时间（vruntime）的方式实现普通进程的优先级调度。CFS调度器的原则是尽量保证每一个任务具有相同的虚拟运行时间，其内部采用平衡二叉树——红黑树的数据结构实现。红黑树中的每一个节点采用vruntime做索引，其中最左侧的叶子节点就是vruntime最小的进程，也意味着其越需要CPU运行时间，所以下一次进行调度的时候，就选择该进程占用CPU。

  其中vruntime的计算公式如下：

  $$vruntime = time * \frac{NICE\_0\_LOAD}{weight}$$

  其中，$NICE_0_LOAD$表示nice为0的权重值。$weight$是一个和进程$NICE$值相关的权重，计算公式为: $weight = \frac{1024}{1.25^{NICE}}$当$NICE > 0$时，$weight < NICE\_0\_LOAD$，虚拟时钟比真实时钟跑的快，而当$NICE < 0$时，虚拟时钟比运行时钟跑的慢，进程可以更长时间占用CPU资源、优先级也较高。对上式进行化简后可以得到$vruntime$和$NICE$之间的关系：

  $$vruntime = time * \frac{NICE\_0\_LOAD * 1.25^{NICE}}{1024}$$

  而在本次实验中，为使得两组进程的CPU利用率之比为7：3，即两组进程实际的运行时间之比为7：3。在vruntime相同的条件下，经过计算后可以得到$\frac{7}{3} \approx 1.25^4$。所以，两组进程的优先级相差为。

- **实时进程抢占：**

  Linux 进程分为实时进程和普通进程两类。实时进程具有一定程度上的紧迫性，要求对外部事件做出非常快的响应；而普通进程则没有这种限制。所以，调度程序要区分对待这两种进程。

  实时进程优先级高于普通进程，如果当前 Linux 系统的执行队列中有实时进程时，调度器 会优先选择实时进程进行调度，也正因如此，实时进程会抢占普通进程。在Linux内核的具体实现上，总共有五个调度器类：stop_sched_class, dl_sched_class, rt_sched_class, fair_sched_class和idle_sched_class。它们的优先级由高到低，每当需要调度时，优先级高的调度器会被优先执行，直到没有可调度的进程，再切换到优先级更低的调度器类。

- **taskset 命令**

  ```
  taskset [options] mask command [arg]...
  taskset [options] -p [mask] pid
  ```

  taskset 命令用于设置或者获取一直指定的 PID 对于 CPU 核的运行依赖关系。也可以用 taskset 启动一个命令，直接设置它的 CPU 核的运行依赖关系。 CPU 核依赖关系是指，命令会被在指定的 CPU 核中运行，而不会再其他 CPU 核中运行的一种调度关系。需要说明的是，在正常情况下，为了系统性能的原因，调度器会尽可能的在一个 CPU 核中维持一个进程的执行。强制指定特殊的 CPU 核依赖关系对于特殊的应用是有意义的。

- **nice 命令**

```
nice [-n adjustment] [-adjustment] [--adjustment=adjustment] [--help] [--
version] [command [arg...]]
```

nice命令以更改过的优先序来执行程序，如果未指定程序，则会印出目前的排程优先序，内定的adjustment 为 10，范围为 -20（最高优先序）到 19（最低优先序）。

1.2 实现思路

为实现实验要求，我们需要先创建10个CPU密集型程序，在本次实验中，选择使用蒙特卡洛法估测π值的程序作为CPU-bound进程。根据上文的分析，为实现两个进程组占用CPU的比例为7：3，两个线程组之间的NICE值只差应该为4。因此，我们在命令行使用nice命令，将一组（5个）线程的NICE值设为4，另一组（5个）线程的NICE值设置为0。同时，使用taskset命令将这10个进程都绑定在CPU1上。将上述操作写成bash脚本，具体如下:

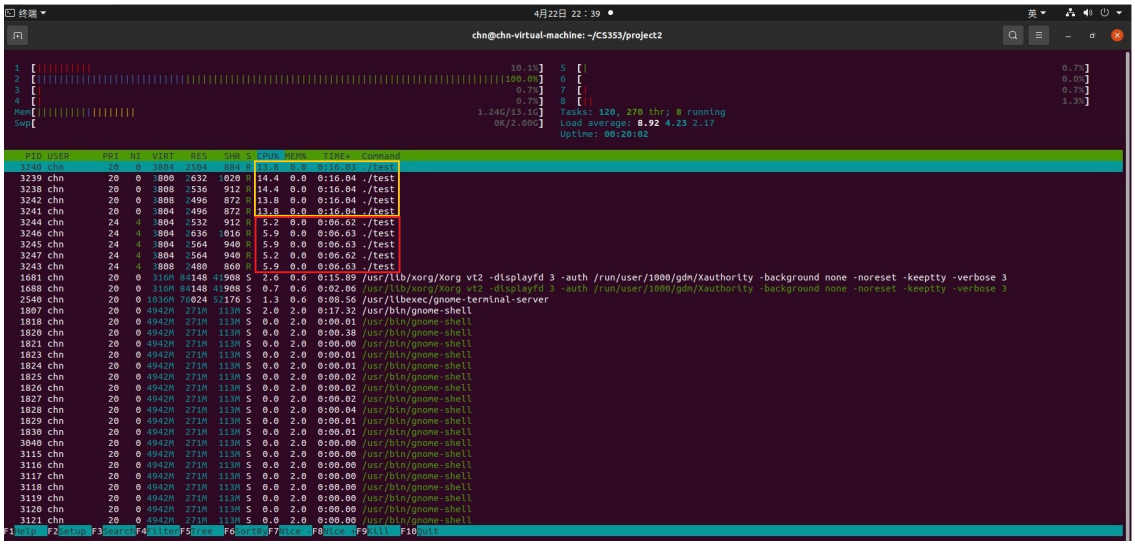
```
#!/bin/bash
if [ $? -eq 0 ]; then
        for i in {1..5}
        do
                nice -n 0 taskset -c 1 ./test &
        done
        for i in {6..10}
        do
                nice -n 4 taskset -c 1 ./test &
        done
fi
```

接着，使用root权限，创建一个实时进程。使用系统调用`sched_setscheduler()`设置进程的优先级和调度策略。之后在CPU1上运行该进程，并使用htop命令观察实时进程对普通进程的抢占效果。

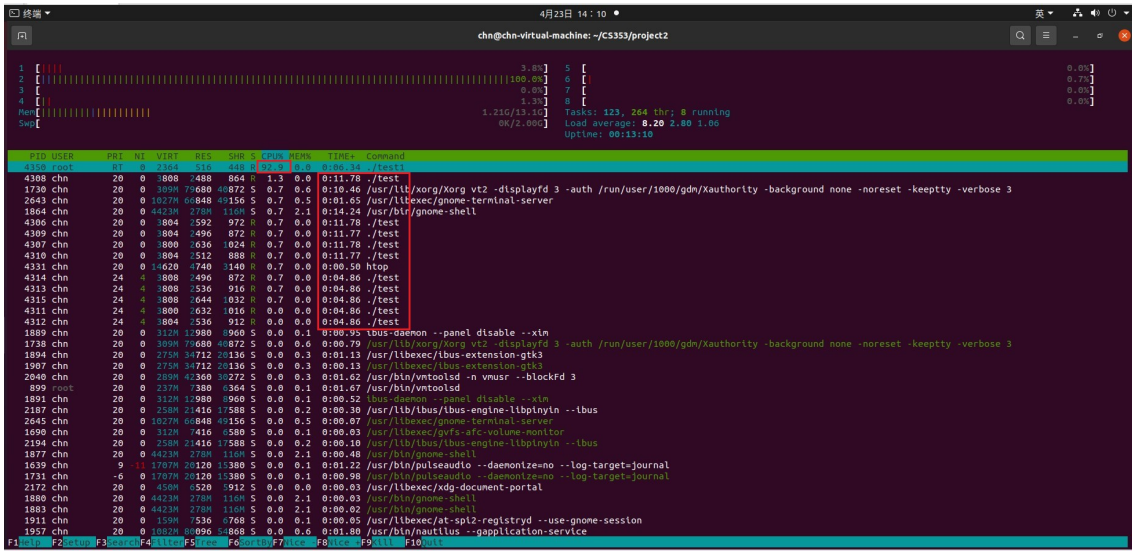
```
int sched_setscheduler(pid_t pid, int policy,const struct sched_param *param);

//sched_setscheduler()函数将pid所指定进程的调度策略和调度参数分别设置为param指向的
sched_param结构中指定的policy和参数。

//sched_param结构中的sched_priority成员的值可以为任何整数，该整数位于policy所指定调度策略
的优先级范围内(含边界值)
```

1.3 实验截图

1. 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%:



2. 实时进程在运行时，会抢占其他十个进程:



Task 2

2.1 预备知识

- **task_struct:**

  task_struct结构体是Linux下的进程控制块PCB。具体定义在内核源文件中的`/include/linux/sched.h`内。在Linux中，每一个LWP都对应着一个task_struct 进程控制块，PCB里包含着一个进程的所有信息。比如进程状态：state，进程的底层信息：thread_info，内存信息: mm，以及和进程调度相关的调度实体、调度类、优先级等等。

- **fork.c**

  在Linux中，除了0号进程以外，其余进程都是由父进程创建而来的，故所有进程共同组成了进程树。操作系统提供了`fork()` `clone()`两个接口来实现进程的创建。阅读源码后发现，其底层主要都是通过调用`_do_fork()`函数来进行进程的创建。源码的注释如下：

```
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 *
 * args->exit_signal is expected to be checked for sanity by the caller.
```

```
 */
long _do_fork(struct kernel_clone_args *args)
{
    ......
    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    ......
}
```

在`_do_fork()`的过程中，该函数调用了`copy_process()`来对父进程进行拷贝。其主要功能是创建一个旧进程的拷贝，并复制旧进程的寄存器信息以及运行环境。源码具体注释如下：

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

其中，调用`dup_task_struct()`来对父进程的task_struct结构体进行复制，返回一个指向当前进程task_struct的指针p。

- **core.c**

    进行进程调度的时候，需要调用`schedule()`这个函数，它负责从运行队列中找到一个进行，并将CPU分配给这个进程。调用方式具有直接调用和延迟调用两种方式。而真正实现进程调度的函数是`_schedule()`

    首先创建如下局部变量：

```
struct task_struct *prev, *next;//当前进程和一下个进程的进程结构体
unsigned long *switch_count;//进程切换次数
struct rq *rq;//就绪队列
int cpu;
```

    之后关闭内核抢占：

```
need_resched:
preempt_disable();//关闭内核抢占
cpu = smp_processor_id();
rq = cpu_rq(cpu);//与CPU相关的runqueue保存在rq中
rcu_note_context_switch(cpu);
prev = rq->curr;//将runqueue当前的值赋给prev
```

    选择相应的进程：

```
next = pick_next_task(rq, prev);//挑选一个优先级最高的任务排进队列
clear_tsk_need_resched(prev);//清除prev的TIF_NEED_RESCHED标志。
clear_preempt_need_resched();
```

最终完成进程调度。

- **base.c**

  在proc文件系统下，每个进程都有自己的目录`/proc/<PID>`，具体定义在`fs/proc/base.c`中。每个进程文件夹下所有文件的列表定义在`tgid_base_stuff[]`中，元素类型为`pid_entry`。在`base.c`文件中，定义了创建只读文件的宏`ONE`，以及其他针对文件或目录的操作。

```
#define ONE(NAME, MODE, show)                \
    NOD(NAME, (S_IFREG|(MODE)),          \
        NULL, &proc_single_file_operations, \
        { .proc_show = show } )
```

2.2 实现思路

- 在task_struct结构体定义中加入成员变量ctx：

```
#endif
    int             on_rq;
    int             ctx;

    int             prio;
    int             static_prio;
    int             normal_prio;
    unsigned int            rt_priority;

    const struct sched_class    *sched_class;
    struct sched_entity     se;
    struct sched_rt_entity      rt;
```

- 在`copy_process()`中实现对ctx变量的初始化，初始化发生在`dup_task_struct()`来对父进程的task_struct结构体进行复制之后，利用其返回值p来访问`ctx`变量。

```
    delayacct_tsk_init(p);  /* Must remain after dup_task_struct() */
    p->flags &= ~(PF_SUPERPRIV | PF_WQ_WORKER | PF_IDLE);
    p->flags |= PF_FORKNOEXEC;
    INIT_LIST_HEAD(&p->children);
    INIT_LIST_HEAD(&p->sibling);
    rcu_copy_process(p);
    p->vfork_done = NULL;
    spin_lock_init(&p->alloc_lock);

    init_sigpending(&p->pending);
    p->ctx = 0;
    p->utime = p->stime = p->gtime = 0;
#ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME
    p->utimescaled = p->stimescaled = 0;
#endif
```

- 在`_schedule()`函数中，每次选择完需要调度的进程`pick_next_task`之后，对进程变量`ctx`加1

```
next = pick_next_task(rq, prev, &rf);
clear_tsk_need_resched(prev);
clear_preempt_need_resched();
next->ctx++;
if (likely(prev != next)) {
    rq->nr_switches++;
    /*
     * RCU users of rcu_dereference(rq->curr) may not see
     * changes to task_struct made by pick_next_task().
     */
```

- 修改`fs/proc/base.c`，使得在每一个`/proc/<PID>`文件夹下新增一个只读文件`ctx`

```
    ONE("limits",       S_IRUGO,    proc_pid_limits),
    ONE("ctx",          S_IRUGO,proc_pid_ctx),
#ifdef CONFIG_SCHED_DEBUG
    REG("sched",        S_IRUGO|S_IWUSR, proc_pid_sched_operations),
```

之后，实现函数`proc_pid_ctx`，调用`seq_printf()`将`task_struct->ctx`打印在屏幕上。

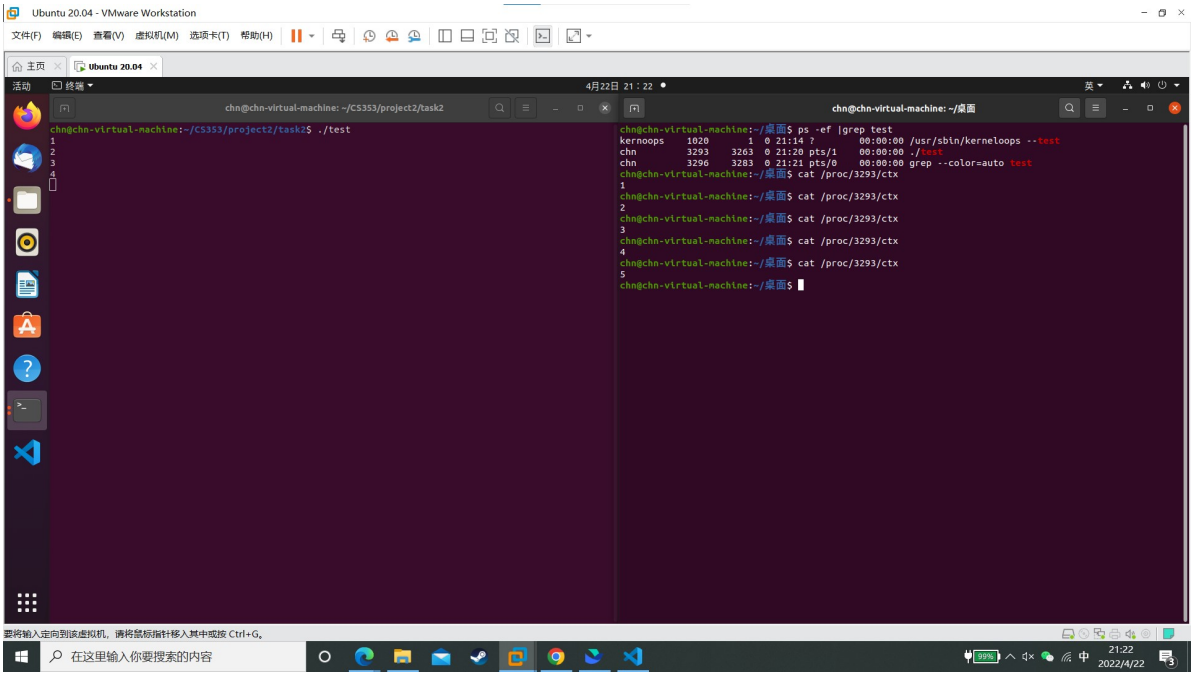
```
static int proc_pid_ctx(struct seq_file *m, struct pid_namespace * ns, struct pid *pid,struct task_struct *task){
    seq_printf(m,"%u\n",task->ctx);
    return 0;

}
```

- 在修改完内核之后，运用如下命令编译并安装内核

```
sudo make menuconfig
sudo make -j 8
sudo make modules_install
sudo make install
reboot
```

2.3 实验截图

在左侧终端运行test测试程序，在右侧终端读取`proc/<PID>/ctx`的内容，实验截图如下：



根据实验截图，我们可以观察到在调度过程中test进程下的ctx变量每调度一次就增加一。

实验心得

- 阅读源码的能力得到锻炼
- 对进程创建以及调度有了更深的理解，对`linux`内核源码的结构有了一定的理解
- 助教给的实验指导很是详尽，给我很大帮助
- 我在内核编译安装的过程中遇到了困难，在查阅资料和询问同学后，修改了.config文件，最终成功编译内核。

最后感谢陈全老师和各位助教的悉心解答。