v1.5.2 CFS调度系统及其他修改

李荣 2024-04-17

CFS调度系统

引入了CFS调度系统，大部分代码是由杨晓峰师兄基于实验班的实验完成的。CFS调度系统引入了实时进程和正常进程的概念，针对不同类型的进程使用不同的调度方法。

CFS调度系统维护了两个队列：实时进程的就绪队列 `rt_rq` 和正常进程的就绪队列 `rq` 。队列使用双链表实现。目前系统中仅有 `hd_service` 和 `task_tty` 两个进程为实时进程，其他均为正常进程。当实时进程的状态为READY，会被加入到实时进程的就绪队列 `rt_rq` 中，实时就绪队列采用静态优先级+FIFO的算法调度，即当实时进程就绪队列不为空时，选用静态优先级最高的实时进程作为下一执行进程，如果几个进程优先级相同，则先执行先入队的实时进程。正常就绪队列使用CFS调度算法，每个进程在真实时间片的基础上增加了一个虚拟运行时间的概念 `vruntime` ，调度器使用 `vruntime` 来统计进程运行的累计时间， `vruntime` 的计算公式如下：

$$vruntime+ = 实际运行时间 \times 1024/进程权重 \quad (1)$$

进程权重由nice决定nice取值范围从-20~19，值越大权重越小：

```
157 //added by zq      xiaof yang, 2周前 • cfs调度
158 int nice_to_weight[40] = {
159  /* -20 */     88761,     71755,     56483,     46273,     36291,
160  /* -15 */     29154,     23254,     18705,     14949,     11916,
161  /* -10 */      9548,      7620,      6100,      4904,      3906,
162  /*  -5 */      3121,      2501,      1991,      1586,      1277,
163  /*   0 */      1024,       820,       655,       526,       423,
164  /*   5 */       335,       272,       215,       172,       137,
165  /*  10 */       110,        87,        70,        56,        45,
166  /*  15 */        36,        29,        23,        18,        15
167 };
```

新创建的正常进程， `vruntime` 为0；fork得到的子进程继承父进程的 `vruntime` ；从sleep状态唤醒的进程，其 `vruntime` 会被修正为当前队列中所有进程最小的 `vruntime` ，这样被唤醒的进程会立即抢占cpu资源但又不至于饿死其他进程。进程进入sleep状态或者退出前，都会被移出队列（ `out_rq()` ），进程被唤醒或者创建，都会被加入队列（ `in_rq()` ），总之，队列中只存在就绪状态的进程。

目前的调度策略为：一轮时间被分为10份时间片，前9/10的时间若实时队列不为空，则从实时队列中选择一个进程执行，若实时队列为空，则从正常队列中选择一个进程执行。后1/10的时间里，调度器会跳过实时队列直接从正常队列中选择一个进程执行。在每次进入调度器时，若当前进程为正常进程，在切换到下一个进程前，会更新其 `vruntime` ，并将当前进程从队列中取出来再根据 `vruntime` 重新插入队列中。若实时队列、正常队列均为空，调度器会将idle进程（pid = 2）加入队列，然后执行idle进程，idle进程永远都是ready状态。

新增4个与cfs调度系统相关的系统调用：

```
void nice(int val);
void set_rt(int turn_rt);
void rt_prio(int prio);
void get_proc_msg(proc_msg* msg);
```

nice系统调用的内容就是在用户态改变nice值，通过改变nice值规定正常进程的优先级。实现过程中，首先要对传入的的不合理的nice值进行收缩到-20到19的区间，同时对该进程的nice进行重新赋值，并根据nice_to_weight[]数组对进程权重weight更新。

set_rt系统调用的作用是实现用户态设置实时进程和正常进程的种类，首先判断是否执行转换，将现在执行的进程从队列中出队，将p_proc_current进程的种类转换成传入的参数，重新入队。

rt_prio系统调用用于改变实时进程的优先级。

get_proc_msg系统调用获取进程相关信息，在测试时调用。

将task_tty进程改为信号量唤醒

原本的 `task_tty` 进程采用轮询的方法来检查缓冲区是否有数据，若键盘缓冲区或鼠标缓冲区存在数据，则将其移入到tty的缓冲区，在将其输出到显存上。在引入cfs调度系统后，因为 `task_tty` 被分为实时进程，在前9/10的时间里会不断轮询，严重挤压了正常进程的cpu资源使用，导致用户进程的响应时间非常长。

为了解决这个问题，我将 `task_tty` 改为信号量唤醒。在鼠标和键盘的中断处理函数中，若存在数据输入，则使用信号量唤醒 `task_tty` ，唤醒后进程会被加入实时队列。 `task_tty` 在处理完所有输入数据后，会应为信号量进入sleep状态，被移出实时队列。

增加idle进程

在将 `task_tty` 进程改为信号量唤醒后，系统会出现类似死锁的状态：系统中所有进程都处于sleep状态，无法响应任何中断，从而无法唤醒任何一个进程。

假设系统中存在三个进程，用户进程init，服务进程task_tty和hd_service。进程task_tty改为信号量唤醒后，只有当键盘中断发生时才会被唤醒，平时都是sleep状态；hd_service平时也是sleep状态，只有当其他进程发送读写请求时，才会被唤醒。hd_service被唤醒后，会从队列中取出读写任务，**关中断**，向硬件发送请求，**进入sleep状态**，等待被中断唤醒；用户进程init在调用相关读写系统调用时也会进入sleep状态。

在修改前，进程task_tty不会陷入sleep状态，当init和hd_service进入sleep状态时，task_tty可以接收sata中断，然后唤醒hd_service，hd_service唤醒init进程，因此不会产生问题。

在修改后，进程task_tty和hd_service先陷入sleep状态，当init调用系统调用（磁盘读写有关）后，init进入sleep状态，唤醒hd_service。hd_service被唤醒后，取出硬盘读写任务，**关中断**，将任务发送给磁盘，将自身设置为sleep后，系统调用yield()，yield()本身也会关中断（汇编 cli），然后进入schedule()函数。在schedule()中因为是关中断，**所以无法接收中断**，所以也无法唤醒hd_service（也就是他本身，schedule是个函数不是进程），但因为**目前所有进程都是sleep状态，schedule找不到一个就绪的进程，会一直死循环。**

我添加了一个idle进程，pid=2，权限为ring0，idle进程会不断将自身的pcb从就绪队列中取出来，然后执行hlt指令。这样的话，当就绪队列不为空时，idle进程不会占用时间片；当就绪队列为空时，才将idle进程放入队列中，避免发生无法响应中断的情况。

```
PUBLIC void idle()
{
    while(1){
        out_rq(p_proc_current);
        asm volatile("sti; hlt": : :"memory");
    }
}
```

其他修改

寄存器宏定义

```
#define k_cs ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_ds ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_es ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_fs ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_ss ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_gs (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK
```

上述寄存器明明是ring1权限（task），却命名为k_xx；修改后如下图所示。

```
#define k_cs ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_KRNL
#define k_ds ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_KRNL
#define k_es ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_KRNL
#define k_fs ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_KRNL
#define k_ss ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_KRNL
#define k_gs (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_KRNL

#define task_cs ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define task_ds ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define task_es ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define task_fs ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define task_ss ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define task_gs (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK
```

子进程使用父进程3G~4G的页表

在fork时，子进程会使用父进程3G~4G范围的页表。具体修改为添加了函数 `init_user_page_pte()` ，函数 `kern_fork()` 会调用该函数来创建子进程的页表目录和页表。

适配gcc-10、gcc-11

修改了部分代码，使minios适配gcc-10、gcc-11。