Escalonador Diferente

Projeto Fase 02

Cassius Puodzius, Lucas Baraças, Marcelo Risse

2° Quadrimentre 2013
2 de Julho de 2013

Conteúdo

1 Introdução

No projeto anterior, a equipe alterou o init e criou o processo pai e filho que imprimiam mensagens em loops infinitos, porém as impressões eram feitas em blocos, durante um tempo que o processo pai ficava imprimindo mensagens e um tempo que o processo filho ficava imprimindo suas mensagens. O printf faz uso da system call write; porém quando o write é chamado o processo não é chaveado. O objetivo dessa fase, então, é alterar o kernel do linux 2.6 de forma que o escalonador troque de processo na chamada da system call write.

2 Escalonamento do Linux

O escalonamento é utilizado para possibilitar a execução de diversos processos em uma (ou mais) CPU simultaneamente. Para tanto, o escalonador utiliza o compartilhamento da CPU no tempo, sendo que cada processo dispõe de uma certa fração desse tempo (*quantum*) para a sua execução. Ao término do processo, ao final de seu *quantum*ou após o tratamento de uma interrupção (este último caso se aplica ao Linux em especial), o escalonador é chamado, a fim de determinar qual o próximo deverá ser executado. Essa escolha é baseada no tipo da fila de execução a ser escalonada e nas prioridades dos processos nessas fila, sendo que processos de tempo-real possuem maior prioridade sobre os demais, bem como os processos de maior prioridades são escalonados primeiro.

2.1 Prioridades de Processos

As prioridadade em processos no Linux podem ser de dois tipo:

Estática : Uma prioridade é atribuída diretamente ao processo.

Dinâmica : A prioridade é calculada a partir de uma estimativa de tempo de execução do processe no próximo *quantum*e da parcela de tempo que o processo executou no seu *quantum*passado (maior contribuição no cálculo de prioridade).

O processos amarrados à CPU (*CPU-Bound*) tendem a ter prioridades mais altas do que processos amarrados a Entrada-Saída (*IO-Bound*), justamente por passarem uma parcela maior do tempo em execução nos seus *quantum*s. A prioridade dos processos dependem também do tipo do processo, sendo que, de fato, processos de tempo-real e os demais processos possuem filas de execução diferentes (*struct cfs_rq* e *struct rt_rq* respectivamente).

2.2 Política de Escalonamento

Define a maneira com a qual os processos serão escalonados, dependendo do estado do sistema e das características dos processos. Processos podem ser preeptivos ou não preemptivos, definindo se o processo permitirá ser interrompido durante a execução ou não. Exemplos de políticas de escalonamento podem ser FIFO, ou seja, uma fila de

execução que escolhe para executar sempre o processo que esta há mais tempo na fila, ou Round Robin, que funciona por meio de um token que é passado de processo em processo, permitindo-o utilizar a CPU (uma vez que os processos são interrompidos ao expirar o tempo de uso do token, essa política é um exemplo de política que não pode ser utilizada com processos não-preemptivos).

3 Alteração do Kernel

O gerenciador do escalonamento está implementado em sched.c, aonde, após nos certificarmos que o write invoca o escalonador utilizando o GDB com o QEMU, começamos a inspecionar o código do kernel.

```
/*
 * schedule() is the main scheduler function.
 */
asmlinkage void __sched schedule(void)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq *rq;
        int cpu;

need_resched:
        preempt_disable();
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
        rcu_sched_qs(cpu);
        prev = rq->curr;
        switch_count = &prev->nivcsw;

        release_kernel_lock(prev);
need_resched_nonpreemptible:

        schedule_debug(prev);

        if (sched_feat(HRTICK))
                hrtick_clear(rq);

        raw_spin_lock_irq(&rq->lock);
        update_rq_clock(rq);
        clear_tsk_need_resched(prev);

                if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
                if (unlikely(signal_pending_state(prev->state, prev)))
                        prev->state = TASK_RUNNING;
                else
                        deactivate_task(rq, prev, 1);
                switch_count = &prev->nvcsw;
        }

        pre_schedule(rq, prev);

        if (unlikely(!rq->nr_running))
                idle_balance(cpu, rq);

        put_prev_task(rq, prev);
        next = pick_next_task(rq);
```

```
        if (likely(prev != next)) {
                sched_info_switch(prev, next);
                perf_event_task_sched_out(prev, next);

                rq->nr_switches++;
                rq->curr = next;
                ++*switch_count;

                context_switch(rq, prev, next); /* unlocks the rq */
```

No final do trecho do sched.c aqui colocado, temos que o escalonador vai escolher o processo next para rodar, mediante a chamada "*next = pick_next_task(rq);*".

```
/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
        const struct sched_class *class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in
         * the fair class we can call that function directly:
         */
        if (likely(rq->nr_running == rq->cfs.nr_running)) {
                p = fair_sched_class.pick_next_task(rq);
                if (likely(p))
                        return p;
        }

        class = sched_class_highest;
        for ( ; ; ) {
                p = class->pick_next_task(rq);
                if (p)
                        return p;
                /*
                 * Will never be NULL as the idle class always
                 * returns a non-NULL p:
                 */
                class = class->next;
        }
}
```

Neste trecho observamos que na maioria das vezes haveremos apenas processos fair, e que nesse caso, o próximo processo é escolhido a partir da fila de processos fair (i.e. cfs_rq). Podemos observar o que foi dito acima a partir de *if (likely(rq-¿nri_running == rq-¿cfs.nr_running))*, onde likely é uma macro para auxiliar o compilador a dispor essa instrução na memória.

Em sched_fair.c está implementado o baixo-nível em termos do escalonador. Nesse código é tratada a manipulação do processos nas suas filas de execução, bem como todo o gerenciamento (a baixo-nível) dos processos. Em especial, procuramos como funciona a função pick_next_task_fair.c, para que possamos alterar o Kernel.

```
static struct task_struct *pick_next_task_fair(struct rq *rq)
{
        struct task_struct *p;
        struct cfs_rq *cfs_rq = &rq->cfs;
```

```
        struct sched_entity *se;

        if (!cfs_rq->nr_running)
                return NULL;

        se = pick_next_entity(cfs_rq);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);

        p = task_of(se);
        hrtick_start_fair(rq, p);

        return p;
}
```

Vemos que o próximo processo (aqui sendo encapsulado por um entity) é obtido na seguinte chamada: *se = pick_next_entity(cfs_rq);*. Analisando-se, finalmente, esta função, vemos que o processo mais à esquerda de um árvore rubro-negram será escolhido como o próximo processo. Para tentarmos, então, cumprir nosso objetivo, alteramos o kernel para analisar se o processo mais à esquerda é o que está rodando no momento, caso seja, o kernel tira esse processo da fila de execução, escolhe o novo processo mais à esquerda e depois reinclui o processo retirado anteriormente na fila. O diff desse código pode ser visto a seguir:

```
 static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
 {
-       struct rb_node *left = cfs_rq->rb_leftmost;
+       struct rb_node *left;
+       struct sched_entity *curr;
+
+       curr = rb_entry(cfs_rq->rb_leftmost, struct sched_entity, run_node);
+
+       if(&curr->run_node == cfs_rq->rb_leftmost) {
+               __dequeue_entity(cfs_rq, curr);
+               curr->load.weight = 0;
+               __enqueue_entity(cfs_rq, curr);
+       }
+
+       left = cfs_rq->rb_leftmost;

        if (!left)
                return NULL;
-
+
        return rb_entry(left, struct sched_entity, run_node);
 }
```

3.1 Resultado Final

Anteriormente à mudança tinhamos a saída da Fig. 1, enquanto que após a nossa alteração do escalonador tivemos a saia da Fig. 2.

Vemos que após a alteração a intercalação entre as impressões do pai e do filho ficaram muito mais intercaladas, no entanto ainda não está exatamente um para um, o que acreditamos ocorrer devido ao cálculo dinâmico de prioridades do Linux, que pode alterar de maneira não desejada a prioridade dos processos pai ou filho, uma vez que estamos

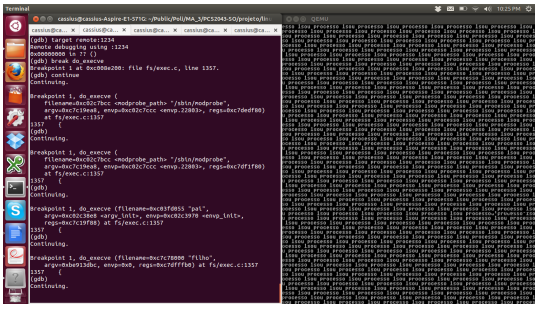

Figura 1: Saída sem a alteração do Kernel

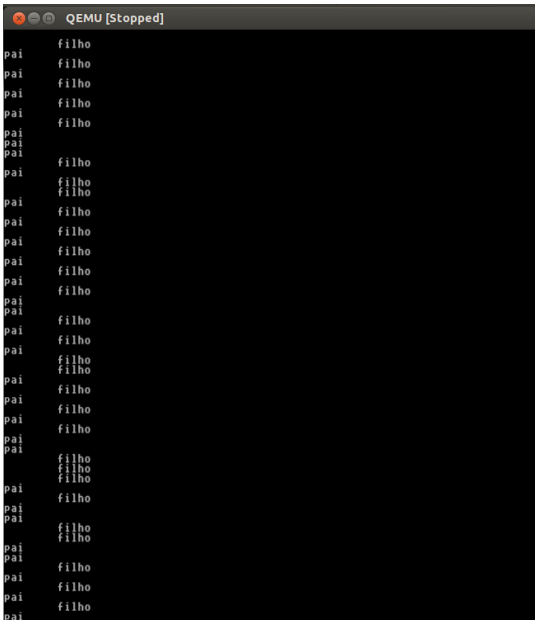

Figura 2: Saída com a alteração do Kernel

mudando manualmente a prioridade de apenas um deles, e também devido à influência de outras syscalls/interrupts que vão acabar chaveando, provavelemente, o processo que está sendo executado.