MC514 - Sistemas Operacionais

Entendendo e Modificando o Escalonador do Linux 2.4

Grupo:

Gustavo Sverzut Barbieri	008849
Ivens Prates Telles Alves	008908
Tiago Moreira de Melo	010011

1 Introdução

Como outros sistemas operacionais interativos conhecidos, o Linux simula o efeito de execução simultânea de múltiplos processos trocando a execução de um processo para outro em um curtíssimo intervalo de tempo.

É nesse contexto que se insere seu escalonador de processos — responsável direto pela escolha de que processo será executado e quando — e seu esquema de agendamento. Este é o objeto de estudo desse trabalho.

2 Agendamento

Para projetar um algoritmo de agendamento, é necessário ter alguma idéia do que um bom algoritmo deveria fazer. Dentre os principais objetivos, é importante destacar:

- Alta taxa de término de processos em "background";
- Garantir que todos os processos tenham a chance de serem executados;
- Conciliar as necessidades de processos de baixa e alta prioridade;
- Rapidez de resposta ao usuário.

A partir do exposto acima, vemos que esses objetivos são conflitantes, o que torna o projeto do algoritmo de agendamento especialmente difícil. O conjunto de regras usadas para determinar quando e como um processo será selecionado para execução é chamado de **Política de Agendamento**.

O algoritmo de agendamento do Linux é "preemptivo", isto é, ele pode parar a execução de um processo para executar outro. Isto é feito selecionando um processo e o executando por um intervalo máximo de tempo. Se o processo ainda estiver executando ao fim deste período, ele é suspenso e o algoritmo seleciona outro processo para ser executado. Isto também pode

ocorrer caso um processo de prioridade de tempo-real que estava preso agora está pronto para rodar.

O intervalo de tempo supracitado é chamado *quantum*. É válido lembrar que um processador só pode executar um único processo em um dado instante. Os *quanta* de todos os processos constituem uma época.

A duração do *quantum* é um fator crítico para a performance do sistema. Ela não deveria ser muito longa ou muito curta. Se a duração do *quantum* for muito curta, o atraso provocado pela troca constante de processos se torna excessivamente grande.

Suponhamos, por exemplo, que uma troca de processos leva 10 ms. Se a duração do quantum é também 10 ms, então pelo menos 50% do uso do processador será desperdiçado com troca de processos. Se, por outro lado, a duração do *quantum* for muito longa, o efeito de processamento concorrente irá desaparecer.

O quantum adotado pelo algoritmo de agendamento do Linux é de aproximadamente, 20 ciclos, o que parece ser um valor adequado às restrições expostas, correspondendo aceitavelmente tanto aos sistemas que usam mais entrada e saída como, por exemplo, servidores e ao mais interativos como os *desktops*.

A política de agendamento também é baseada na classificação de processos de acordo com sua prioridade, isto é, cada processo é associado a um valor que denota o quão apropriado a ser executado ele é.

No Linux, essa prioridade é dinâmica, isto é, o escalonador acompanha o que os processos estão fazendo e ajusta suas prioridades periodicamente, baseando-se no restante de *quantum* do processo ao fim de sua época. Por exemplo, um processo que ainda possuia uma quantidade de ciclos disponíveis ficou preso ("`TASK_INTERRUPTIBLE`") à espera de uma entrada. Ele é então posto para dormir e até o fim da época ele continua a dormir. No próximo recálculo de *quantum* dos processos, este processo vai ganhar uma certa quantidade de ciclos a mais. Com isso o escalonador privilegia os processos ditos "*IO-Bound*" ou interativos e então melhora a performance geral do sistema, pois processos deste tipo tendem a rodar apenas por um determinado período de tempo e depois eles automaticamente entram em estado de espera.

Mas até a versão estável atual (linux-2.4.20) o kernel do Linux não é totalmente preemptivo, pois as tarefas que rodam em modo kernel somente serão interrompidas quando ocorrer uma interrupção, falha de página (*page-fault*) ou chamada ao escalonador (`schedule()`), mesmo que uma tarefa de tempo-real esteja pronta para rodar. Existem *patches* para torná-lo preemptivo ajudando assim em sistemas onde tarefas de tempo real são comuns como sistemas multimeios para edição de vídeo e som.

O escalonador do kernel em desenvolvimento (linux-2.5.70) é bem diferente do atual. Este é preemptivo e seu escalonador não é mais **O(n)** e sim **O(1)**, com uma lista de processo para cada CPU, assim em sistemas com vários processadores e dezenas de milhares de processos não se gasta tempo com o recálculo dos *quanta* na troca de época. Porém nosso estudo restringe-se à versão estável, para mais informações vide: `http://www.linuxgazette.com/issue89/vinayak2.html`

2.1 Agendamento de tarefas de tempo real

O processo de escolha do próximo processo a executar é primeiro verificar se existe um processo categorizado como "*Real-Time*" (categorias `SCHED_RR` e `SCHED_FIFO`), se possuir e este possuir prioridade maior do que a do processo em execução (processos *real-time* tem sempre preferência a processos comuns – `SCHED_OTHER`) o processo em execução é interrompido ("*preempted*") e este processo é posto a rodar. Ao acabar, o processo é colocado ao fim da fila de execução e o processo que rodava anteriormente é recolocado em execução.

2.2 Agendamento de tarefas comuns

Dentro os vários processos comuns a política é diferente das dos processos de tempo-real. Desta vez coloca-se para rodar o processo com maior "fator bom" ("*goodness*") no CPU atual. Este fator é proporcional à prioridade do processo, ao número de ciclos restantes e ainda leva em conta se vale a pena mudar o processo de CPU e se vale a pena mudar de processo.

Com isso economiza-se com troca de contexto, preenchimento do *cache* do processador e também o custoso processo de migração de CPU.

3 Algoritmo de agendamento do Linux 2.4

Faremos agora uma breve descrição de alguns conceitos importantes para a compreensão do algoritmo de agendamento do kernel do Linux. Posteriormente, descreveremos como alterar a política de agendamento deste algoritmo, de forma a favorecer um usuário específico.

Alguns conceitos importantes a saber antes de começar a olhar o algoritmo são:

- Um processo pode estar em um dos estados:
  - `TASK_RUNNING`: o processo está rodando;

  - `TASK_INTERRUPTIBLE`: o processo está interrompido a espera de algum sinal;
  - `SCHED_RR` indica um processo de tempo real do tipo "round-robin". `SCHED_FIFO` indica um processo de tempo real do tipo *First In, First Out*. `SCHED_OTHER` indica um processo comum.

- Um processo pode não estar rodando por que acabou seu tempo, por que quis deixar de rodar explicitamente (`SCHED_YIELD`), por que está preso à espera de um sinal ou por que foi parado para que se executasse algo mais importante ("*preempted*").

3.1 A função `goodness()`

A função `goodness()` tem como objetivo retornar "quão bom" é um processo para um determinado processador. Esta informação será usada mais tarde para escolher qual processo deverá ser colocado em execução.

Caso o processo não tenha mais tempo de execução ou ele tem porém resolveu deixar o processador por vontade própria, ele não deverá ser escolhido. Porém os dois casos diferem em um ponto: caso ele não tenha mais tempo, retorne o valor `0` para indicar que ele quer um recálculo de *quantum*, ou seja, a época já acabou para ele; caso ele deixou o processador por vontade própria, a época ainda não acabou para ele e portanto não precisa recalcular os *quanta*. Veja o código:

```
/*
 * select the current process after every other
 * runnable process, but before the idle thread.
 * Also, dont trigger a counter recalculation.
 */
weight = -1;
if (p->policy & SCHED_YIELD)
        goto out;

/*
 * Non-RT process - normal case first.
 */
if (p->policy == SCHED_OTHER) {
        /*
         * Give the process a first-approximation goodness value
         * according to the number of clock-ticks it has left.
         *
```

```
                 * Don't do any other calculations if the time slice is
                 * over..
                 */
                weight = p->counter;
                if (!weight)
                        goto out;
```

Ela leva em conta o fato da migração de um processo entre processadores é custosa e por isso ela dá um prêmio para o processo se ele já está a rodar neste processador. Isto é feito na parte:

```
#ifdef CONFIG_SMP
                /* Give a largish advantage to the same processor...   */
                /* (this is equivalent to penalizing other processors) */
                if (p->processor == this_cpu)
                        weight += PROC_CHANGE_PENALTY;
#endif
```

Ela também leva em conta o fato de ser vantajoso não trocar de processo em execução pois se ela não o fizer, economizará a custosa troca de contexto e também reaproveitará o *cache* do processador e muito provavelmente não terá que mudar outras coisas, como a paginação da memória. Isto está codificado como:

```
                /* .. and a slight advantage to the current MM */
                if (p->mm == this_mm || !p->mm)
                        weight += 1;
                weight += 20 - p->nice;
                goto out;
```

E para terminar, caso um processo seja do tipo de tempo real, dê uma vantagem enorme para ele:

```
        /*
         * Realtime process, select the first one on the
         * runqueue (taking priorities within processes
         * into account).
         */
        weight = 1000 + p->rt_priority;
```

Lembrando que a função `goodness()` serve somente para dar uma base de "quão bom" é o processo e esta informação será usada mais tarde para determinar *quem roda* e não *o quanto roda*, veremos que se a modificarmos não ganharemos tempo a mais para o usuário e sim que ele rode primeiro. Este não é nosso objetivo, portanto não modificaremos a `goodness()`.

3.2 A função `schedule()`

Esta função é o escalonador propriamente dito. Ela começa com algumas conferências para saber se está tudo certo e depois ela prossegue com as partes abaixo listadas:

3.2.1 Procedimento para agendamento de tarefas de tempo-real

Esta parte começa com o seguinte código:

```
/* move an exhausted RR process to be last.. */
if (unlikely(prev->policy == SCHED_RR))
        if (!prev->counter) {
                prev->counter = NICE_TO_TICKS(prev->nice);
                move_last_runqueue(prev);
        }
```

O qual vai conferir se a tarefa em execução é de tempo-real e do tipo "round-robin". Caso isto for verdadeiro e esta tarefa não possuir mais tempo de execução, ela tem o seu *quantum* preenchido e volta para o final da fila de execução.

Neste trabalho teremos que privilegiar um usuário dando mais tempo de execução para este. Para atingir este objetivo, teríamos que modificar logo após a linha com `prev->counter = NICE_TO_TICKS(prev->nice);` verificando se este usuário[1] é o usuário que queremos modificar. O código para privilegiar o usuário `mc514_uidmc514_uid` seria:

```
/* move an exhausted RR process to be last.. */
if (unlikely(prev->policy == SCHED_RR))
        if (!prev->counter) {
                prev->counter = NICE_TO_TICKS(prev->nice)

                if (unlikely(mc514_uid == prev->uid))
```

[1]para sabermos qual usuário é dono do processo é só pegar o campo `uid` da estrutura `task_struct`

```
                        prev->counter += 20; /* Privilegia o usuario
                                              * com 20 ticks
                                              */

                        move_last_runqueue(prev);
                }
```

Note que o *quantum* de tarefas de tempo-real são quantidades fixas, as quais dependem do valor do `nice` do processo. O valor do `nice` pode ser alterado como o comando de mesmo nome[2] ou pela chamada de sistema, também, de mesmo nome.

3.2.2 Recolocar tarefas com sinais pendentes a rodar

Em seguida ele confere se o processo que estava rodando estava interrompido e se estiver, verifica se existe sinais pendentes para ele, caso afirmativo, coloque o processo como ativo (`TASK_RUNNING`), caso o processo esteja no estado "rodando" (`TASK_RUNNING`) não faça nada e caso não seja nenhum destes dois estados, retire-o da lista de processos a rodar.

```
switch (prev->state) {
        case TASK_INTERRUPTIBLE:
                if (signal_pending(prev)) {
                        prev->state = TASK_RUNNING;
                        break;
                }
        default:
                del_from_runqueue(prev);
        case TASK_RUNNING:;
}
prev->need_resched = 0;
```

[2]com o comando **nice** qualquer usuário pode alterar o valor do nice de um processo, porém existem restrições diversas para usuários comuns como, por exemplo, eles não podem aumentar a prioridade de um processo e eles só podem mudar as prioridades de seus próprios processos. Já o usuário é o único que pode criar tarefas de tempo-real e pode, ainda, alterar o valor do `nice` de qualquer processo.

3.2.3 Escolhendo um processo para rodar

O Linux 2.4[3] escolhe o próximo processo a rodar percorrendo uma lista de processos que podem ser executados.

A seleção é trivial: primeiro pegamos o processo base[4] do CPU atual (`idle_task(this_cpu)`). Depois percorremos a lista e verificamos se o processo pode ser colocado neste CPU. Caso isto seja verdade, prossiga verificando o "quão bom" é o processo para este CPU (função `goodness()`3.1) e conferimos se este processo é melhor que o previamente selecionado. No final teremos selecionado o melhor processo para rodar neste CPU.

```
/*
 * Default process to select..
 */
next = idle_task(this_cpu);
c = -1000;
list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
                int weight = goodness(p, this_cpu, prev->active_mm);
                if (weight > c)
                        c = weight, next = p;
        }
}
```

3.3 Recalculando os *quanta* dos processos

Caso todos os processos ativos retornem `0` ou `-1` na função `goodness()`, o valor de `weight` será `0` também e isto indica que os *quanta* dos processos precisam ser recalculados.

Vê-se, pelo código abaixo, que primeiramente liberamos as interrupções com `spin_unlock_irq(&runqueue_lock)`, isto é feito pois o processo de recálculo pode ser muito demorado e não permitir interrupções durante este tempo pode levar a uma situação ruim. Então liberamos as interrupções porém pegamos a trava de acesso à lista de tarefas `read_lock(&tasklist_lock)` e somente então começaremos a percorrer a lista de processos (`for_each_task(p)`) para atribuir novos *quanta* aos processos.

[3]Note que isto torna o algoritmo **O(n)** e para um sistema com vários processos isto pode ser um impecílio, por isso o Linux 2.5 adotou um método melhor de se escolher o próximo processo a rodar

[4]É um processo que sempre está rodando neste CPU, ele não pode ser terminado ou mudado de processador

Verificamos que na atribuição de um novo *quanta* levamos em conta se o processo não rodou todo seu tempo na época anterior (`p->counter = (p->counter >> 1)`...). Isto é feito para beneficiar processos que são iterativos e processos que doam seu tempo facilmente (usando `SCHED_YIELD`). Esta técnica de dar maior tempo a processos deste tipo é boa pois eles frequentemente não esgotarão seu *quantum*, economizando algumas trocas de contexto e o sistema vai parecer melhor para este tipos de tarefas sem prejudicar as tarefas mais consumidoras de CPU.

Note bem que se quisermos modificar o escalonador para privilegiar um usuário, poderíamos antes de liberar as interrupções criar uma variável local[5] para guardar este usuário e dentro do laço do `for_each_task(p)` verificamos se o dono do processo `p` é o mesmo que o escolhido.

```
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
                p->counter = (p->counter >> 1) \
                        + NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
}
```

Com as mudanças para privilegiar o usuário guardado em `mc514_uid`, o código ficará:

```
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
        struct task_struct *p;
        uid_t mc514_uid_local = mc514_uid;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p) {
```

[5]Usaremos uma variável local para evitarmos que durante o laço occorra uma mudança de usuário privilegiado. Se isto acontecesse, ficaríamos com dois usuários privilegiados durante uma época.

```
                        p->counter = (p->counter >> 1) \
                                + NICE_TO_TICKS(p->nice);

                        if (unlikely(mc514_uid_local == p->uid))
                            p->counter += 20; /* Privilegiar
                                               * o usuário com
                                               * 20 ticks
                                               */
                }
                read_unlock(&tasklist_lock);
                spin_lock_irq(&runqueue_lock);
                goto repeat_schedule;
        }
```

O restante do escalonador é pouco importante para nosso estudo.

4 Considerações Finais e Detalhes de Implementação

Para mudarmos o escalonador do Linux 2.4 para que um usuário seja privilegiado, deveremos alterar a função `schedule()` (no arquivo sched.c) nos pontos indicados acima.

Para lembrarmos o código do usuário (*user identifier* ou `uid`) iremos armazenar este em uma variável global, a qual chamaremos aqui de `mc514_uid`. Para que seja possível mudar o usuário privilegiado a partir do *user space*, deveremos criar uma chamada de sistema, a qual chamaremos de `sys_privilegia( uid_t uid )`, e esta será o caminho de entrada para o *kernel space*, onde alterará a variável citada.

Se quisermos que o processo do usuário já comece privilegiado teremos que modificar, também, o arquivo fork.c na função `do_fork()` acrescentando abaixo de

```
p->counter = (current->counter + 1) >> 1;
current->counter >>= 1;
if (!current->counter)
        current->need_resched = 1;
```

o código:

```
if ( unlikely( mc514_uid == p->uid ) )
  p->counter += 20; /* privilegiar o usuario com 20 ticks */
```