UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA DE ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Escalonamento no Linux: Uma Experiência com Abordagem Hierárquica

Júlio César Moriguti

Prof. Dr. Luis Fernando Friedrich
Orientador

Thadeu Botteri Corso
Membro da banca

José Mazzucco Júnior
Membro da banca

Florianópolis, Julho de 2003

SUMÁRIO

1. Introdução 4

1.1 Motivações e Objetivos 4

1.2 Organização do Texto 4

2. Escalonamento de Processos.. 5

2.1 Classificação 6

2.2 Escalonamento de Tempo-Real 7

2.3 Abordagem Hierárquica..13

3. Escalonamento no Linux... 18

3.1 Política de Escalonamento 18

3.2 Preempção de Processos 19

3.3 O Algoritmo Escalonador 20

3.4 Estruturas de Dados Usadas pelo Escalonador 22

3.5 A Função schedule 25

3.6 Performance do algoritmo escalonador 30

3.7 Chamadas de sistema relacionadas ao escalonador 33

4. Abordagem Hierárquica no Linux – Uma Experiência de Implementação...35

4.1 Requisitos da Interface...16

4.2 Controle de Fluxo 35

4.3 Interface 35

4.4 Estruturas de Dados 36

4.5 Implementação do Escalonador do Linux..37

4.6 Resultados..37

5. Conclusão 40

6. Referências Bibliográficas 41

Resumo

Este trabalho pretende apresentar mecanismos que tornem um sistema operacional de propósito geral em um sistema ainda mais flexível, que permita o carregamento de políticas de escalonamento próprias do usuário. Desse modo, o custo e a dificuldade de se adicionar uma nova política num sistema operacional são reduzidos.

O objetivo é o desenvolvimento de um protótipo que, embora não seja totalmente funcional, permita a análise prática desse sistema proposto. A implementação dos trabalhos será feita no sistema operacional Linux.

O presente trabalho é uma monografia de conclusão de curso do programa de graduação da Computação da UFSC.

Abstract

This work intends to present mechanisms that turn an general operational system of general intention in a more flexible system, that allows user scheduling policies to be loaded into the system. In this way, the cost and the difficulty of adding a new politicy in an operational system are reduced.

The objective is the development of a prototype that, even so is not total functional, allows the practical analysis of this considered system. The implementation of the works will be made in the operational system Linux.

Capítulo 1

Introdução

1.1 Motivações e Objetivos

Os sistemas operacionais precisam atender aplicações cada dia mais variadas, sendo aplicações de tempo-real ou científicas. O escalonamento é um dos principais fatores que influenciam na eficiência dos sistemas operacionais em atender essas aplicações.

Este trabalho tem como objetivo geral desenvolver uma estrutura que torne o sistema mais flexível, permitindo que se adapte às necessidades do usuário. Neste sentido, o trabalho deverá:

- Pesquisar sobre técnicas de escalonamento de processos;
- Pesquisar sobre sistemas de tempo-real;
- Realizar um estudo sobre o escalonador do Linux;
- Implementar uma estrutura que possibilite o carregamento dinâmico de escalonadores;

1.2 Organização do Texto

Será feita uma rápida abordagem no capítulo 2 sobre os principais fundamentos de escalonamento, onde será apresentada uma classificação e alguns algoritmos. O capítulo 3 contém informações sobre o funcionamento do escalonador do Linux. No capítulo 4 estão os detalhes da implementação, com explicações sobre o que foi feito e com a análise de alguns resultados. No capítulo 6 encontram-se as conclusões do trabalho. No anexo A está um artigo referente a este trabalho, enquanto que no anexo B está o código fonte da implementação.

Capítulo 2

Escalonamento de Processos

Escalonamento refere-se à atribuição de processos concorrentes a um ou mais processadores, fato que é comum em sistemas operacionais multitarefa executando em computadores pessoais. Geralmente, esse tipo de escalonamento é feito pelo sistema operacional, e baseia-se no compartilhamento do processador através de técnicas de *time-sharing*, ou seja, o tempo de uso do processador é dividido em pequenas fatias de tempo (*time-slice*) que são utilizadas pelos diversos processos ordenadamente. Dessa maneira, permite-se uma utilização eficiente e justa do processador disponível. Para garantir a imparcialidade, o sistema operacional cuida da suspensão e ativação dos processos, através de uma técnica chamada de preempção.

Casavant e Kuhl (1988) definem formalmente o problema do escalonamento enxergando-o como um recurso para o gerenciamento de recursos. Dessa forma, pode-se dividir o problema do escalonamento em três principais componentes:

- Consumidores (processos dos usuários e do sistema)
- Recursos (processadores, memória, redes de comunicação, etc.)
- Algoritmo de escalonamento

Os consumidores utilizam recursos computacionais, e o algoritmo de escalonamento é responsável por distribuir os processos entre os processadores de acordo com as necessidades e a disponibilidade de recursos. Complementando essa definição, Shirazi e Hurson (1992) e Baumgartner e Wah (1991) afirmam que essa distribuição de processos é atrelada a um objetivo, baseado no qual todas as decisões de escalonamento são tomadas. Exemplos de possíveis objetivos são: diminuir o tempo de execução dos processos e balancear a carga do sistema.

O escalonamento de processos é implementado através de um algoritmo de escalonamento, que é organizado como um conjunto de componentes, permitindo uma visão modular do algoritmo. Geralmente são divididos em políticas, que implementam diferentes partes do processo de escalonamento, facilitando seu projeto e compreensão.

Quando se fala em escalonamento, processos são tradicionalmente classificados em "I/O-bound" ou "CPU-bound". O primeiro faz grande uso de dispositivos de I/O e gasta muito tempo esperando por operações de I/O terminarem; o segundo são aplicações numéricas que gastam muito tempo de CPU.

Uma classificação alternativa apresenta três classes de processos:

Processos interativos

Esses interagem constantemente com usuários, e assim gastam muito tempo esperando por operações do mouse e do teclado. Quando o dado é recebido, o processo deve acordar rapidamente, ou o sistema parecerá lento. Tipicamente, o atraso médio deve cair entre 50 e 150 ms. Típicos programas interativos são editores de texto, aplicações gráficas.

Processos batch

Esses não precisam interagir com o usuário, e portanto geralmente rodam em background. Já que esses processos não precisam interagir com o usuário, eles geralmente são penalizados pelo escalonador. Típicos programas batch são compiladores, bancos de dados.

Processos tempo-real

Esses processos não devem ser nunca bloqueados por um processo de prioridade menor, devem ter pequeno um tempo de resposta e, mais importante, esse tempo deve ter baixa variação. Típicos programas tempo-real são aplicações multimedia, controladores de vídeo, e programas que coletam dados de sensores físicos.

2.1 Classificação

O escalonamento pode ser estático ou dinâmico. No estático, o escalonamento é feito antes da execução do processo e é definido explicitamente no código fonte ou durante a compilação do programa a ser escalonado, não sendo modificado até o final de sua execução. O escalonamento dinâmico é efetuado em tempo de execução, e pode ser modificado em qualquer momento enquanto o processo estiver sendo executado.

Os algoritmos de escalonamento estático requerem que um grande número de informações sobre o processo a ser executado esteja disponível antes de sua execução (por exemplo, tempo de execução do processo, localização dos dados a serem utilizados, etc.), o

que raramente acontece, visto que a maioria dos problemas resolvidos computacionalmente não é determinística [BW 89]. Se a estimativa da necessidade de recursos computacionais não corresponder à realidade, um escalonamento estático fatalmente ocasionará a degradação do desempenho. Silva (1997) apresenta em seu trabalho uma discussão sobre alguns aspectos relacionados ao escalonamento estático. O escalonamento dinâmico possibilita uma maior flexibilidade através da utilização de informações sobre o estado do sistema computacional para auxiliar nas suas decisões, possibilitando um melhor desempenho. Também é possível modificar a distribuição dos processos (através da migração de processos), permitindo a adequação do sistema em caso de uma mudança brusca das cargas nos processadores. Uma desvantagem dessa abordagem é a sobrecarga imposta no sistema pelo próprio algoritmo de escalonamento e pela necessidade de possíveis migrações de processos.

A responsabilidade pelo escalonamento pode ser centralizada em um processador (não distribuído) ou dividida entre vários processadores (distribuído). Se distribuído, o escalonamento pode ser organizado de maneira cooperativa (quando as decisões de Escalonamento de Processos em Sistemas Distribuídos escalonamento em cada processador são tomadas levando em consideração uma visão global do sistema) ou de maneira não cooperativa, quando cada processador decide baseado apenas em informações locais.

2.2 Escalonamento de Tempo-Real

Em sistemas de tempo real que seguem a abordagem assíncrona os aspectos de implementação estão presentes mesmo na fase de projeto. Na implementação de restrições temporais, é de fundamental importância o conhecimento das propriedades temporais do suporte de tempo de execução usado e da escolha de uma abordagem de escalonamento de tempo real adequada à classe de problemas que o sistema deve tratar.

Em sistemas onde as noções de tempo e de concorrência são tratadas explicitamente, conceitos e técnicas de escalonamento formam o ponto central na previsibilidade do comportamento de sistemas de tempo real. Nos últimos anos, uma quantidade significativa de novos algoritmos e de abordagens foi introduzida na literatura tratando de escalonamento de tempo real. Infelizmente muitos desses trabalhos definem técnicas restritas e conseqüentemente de uso limitado em aplicações reais.

O termo *escalonamento* ("scheduling") identifica o procedimento de ordenar tarefas na fila de Pronto. Uma escala de execução ("schedule") é então uma ordenação ou lista que indica a ordem de ocupação do processador por um conjunto de tarefas disponíveis na fila de Pronto. O escalonador ("scheduler") é o componente do sistema responsável em tempo de execução pela gestão do processador.

Os algoritmos escalonadores são classificados quanto à forma de cálculo de ordenação das tarefas. Os algoritmos são ditos preemptivos ou não preemptivos quando em qualquer momento tarefas se executando podem ou não, respectivamente, ser interrompidas por outras mais prioritárias. Algoritmos de escalonamento são identificados como estáticos quando o cálculo da escala é feito tomando como base parâmetros atribuídos às tarefas do conjunto em tempo de projeto (parâmetros fixos). Os dinâmicos, ao contrário, são baseados em parâmetros que mudam em tempo de execução com a evolução do sistema.

Modelo de Tarefas

O conceito de tarefa é uma das abstrações básicas que fazem parte do que chamamos um problema de escalonamento. Tarefas ou processos formam as unidades de processamento seqüencial que concorrem sobre um ou mais recursos computacionais de um sistema. Uma simples aplicação de tempo real é constituída tipicamente de várias tarefas. Uma tarefa de tempo real, além da correção lógica ("correctness"), deve satisfazer seus prazos e restrições temporais ou seja, apresentar também uma correção temporal ("timeliness").

As restrições temporais, as relações de precedência e de exclusão usualmente impostas sobre tarefas são determinantes na definição de um modelo de tarefas que é parte integrante de um problema de escalonamento.

Restrições Temporais

Aplicações de tempo real são caracterizadas por restrições temporais que devem ser respeitadas para que se tenha o comportamento temporal ou necessário. Algumas dessas restrições são:

Tempo de execução

Expressa o pior tempo de execução das tarefas. Isso é importante num sistema distribuído quando o tempo de execução de uma tarefa pode ser diferente entre os nodos.

Deadline

Corresponde à capacidade de resposta requerida pelo sistema. Usualmente, existe apenas um deadline explícito para uma cadeia de tarefas (end-to-end deadline) e não apenas para cada tarefa individual. Uma abordagem comum é dividir esse end-to-end deadline em deadlines menores que são atribuídos a cada tarefa.

Tempo de liberação

Estão geralmente relacionados aos períodos dos processos onde restringem o tempo de início das tarefas. Muitos algoritmos escalonadores impoem tempos de liberação às tarefas como uma maneira de obter exclusão mútua entre tarefas que acessam o mesmo recurso.

Tempo de chegada

É o instante em que o escalonador toma conhecimento de uma ativação de uma tarefa. Em tarefas periódicas, o tempo de chegada coincide sempre com o início do períodо da ativação.

Tempo de início

Corresponde ao instante de início do processamento da tarefa em uma ativação.

Escalonamento de Tarefas Periódicas

Em muitas aplicações tempo real, atividades periódicas representam a maior parte da demanda computacional do sistema. Essas atividades periódicas tipicamente aparecem de aquisição de dados, laços de controle, monitoração de sistemas, que precisam ser executados em taxas específicas.

Essas atividades são ativadas numa seqüência infinita e em intervalos regulares chamados de Período. Essas características permitem que se obtenha garantias em tempo de projeto sobre a escalonabilidade de um conjunto de tarefas periódicas.

Escalonamento Taxa Monotônica

O termo *taxa monotônica* deriva do método de atribuir prioridades à um conjunto de processsos: atribuindo prioridades como uma função monotônica da taxa de um processo (periódico). Dada essa regra, a teoria fornece a seguinte inequação, que é condição suficiente para garantir que todos os processos completarão seu trabalho até o final dos seus períodos:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n}-1)$$

C_i and T_i representam o tempo de execução e o período respectivamente associados à uma tarefa periódica. Se o número de tarefas aumenta, o limite do escalonamento converge para *ln 2* (69%).

A prioridade (P) é calculada de acordo com o período das tarefas (T_p):

$$P = 1/T_p$$

Assim, a tarefa com o menor período tem a maior prioridade, seguida da tarefa com o segundo menor período, e assim por diante.

Infelizmente as suposições básicas dessa teoria contem alguns pontos fracos:

- Todas as tarefas são periódicas.
- Tarefas são independentes e não interagem.
- Deadline e período da tarefa são considerados sinônimos.
- O tempo de execução de cada tarefa é constante e não varia com o tempo.
- Todas as tarefas são importantes.

Tarefas aperiódicas são limitadas a rotinas de inicialização e de tratamento de erros. Essas considerações são muito restritivas para o uso desse modelo na prática. Entretanto, esse modelo tem algumas propriedades úteis, incluindo um simples "suficiente e não necessário" teste de escalonabilidade baseado na utilização dos processos (Liu, 1973); e um complexo "suficiente e necessário" teste de escalonabilidade (Lehoczky, 1989).

Escalonamento Earliest Deadline First (EDF)

A política de escalonamento do EDF corresponde a uma atribuição dinâmica de prioridades que define a ordenação das tarefas segundo os seus deadlines absolutos (d_i). A tarefa com maior prioridade é a que tem o deadline d_i mais próximo do tempo atual. A cada

chegada de tarefa a fila de prontos é reordenada, considerando a nova distribuição de prioridades. A cada ativação de uma nova tarefa T_i, um novo valor de deadline absoluto é determinado considerando o número de períodos que antecede a atual ativação (k): $d_{ik} = kP_i$. As principais características desse modelo são:

- As tarefas são periódicas e independentes.
- O deadline de cada tarefa coincide com o seu período (D_i=P_i).
- O tempo de execução de cada tarefa é conhecido e constante.

Escalonamento Deadline monotônico

A política do Deadline monotônico define uma atribuição estática de prioridades, baseada nos deadlines relativos das tarefas. Esse modelo assume deadlines (D_t) relativos menores ou iguais aos períodos das tarefas (P_t). Baseado nisso a seguinte relação é derivada:

$$C_i \leq D_i \leq T_i$$

onde *C* é o tempo de execução, *D* é o deadline e *T* é o período do processo *i*.

O cálculo das prioridades nesse modelo é parecido com o cálculo das prioridades do modelo de taxa monotônica. Prioridades atribuídas aos processos são inversamente proporcionais ao deadline. Assim, o processo com o menor deadline é atribuído à maior prioridade e o processo com o maior deadline é atribuído com a menor prioridade.

Nenhum teste de escalonabilidade foi dado por (Leung, 1982). Em [ABR91] é definido um teste suficiente e necessário para o DM, baseado no conceito de tempo de resposta de uma tarefa.

Tarefas dependentes: compartilhamento de recursos

Em um ambiente multitarefas o compartilhamento de recursos determina alguma forma de exclusão entre tarefas. Tarefas podem usar mecanismos de sincronismo como semáforos, monitores para implementar exclusão mútua.

O compartilhamento de recursos e as relações de exclusão decorrentes do mesmo, determinam bloqueios em tarefas mais prioritárias. Esses bloqueios são chamados de *inversões de prioridade*.

Relações de precedência e exclusão

Em aplicações de tempo real, muitas vezes, os processos não podem executar em ordem arbitrária. Algumas tarefas só podem iniciar a execução depois que uma outra tarefa terminar a sua execução. Há, portanto, uma certa relação de precedência entre essas tarefas.

Uma outra forma de relação entre as tarefas é a relação de exclusão. Uma tarefa T_i exclui T_j quando a execução de T_j que manipula o recurso compartilhado não pode executar porque T_i já ocupa o recurso.

Escalonamento de tarefas aperiódicas

Nem todas as tarefas tempo real são periódicas. Tarefas aperiódicas, também chamadas de servidores aperiódicos, respondem a chegadas de eventos aleatórios. Essas tarefas, na maioria das vezes, podem ter características não tempo reais, como por exemplo, tarefas que imprimem dados na tela ou que fazem a leitura do teclado.

Nem todas as tarefas tempo real são periódicas. Tarefas periódicas, também chamadas de servidores aperiódicos, respondem às chegadas aleatórias de eventos. Por exemplo, num sistema de freio, se o sistema demorar muito para responder ao pressionamento do pedal, um acidente pode acontecer.

Algumas tarefas aperiódicas podem ser convertidas em tarefas periódicas. Essa é basicamente a mesma transformação que converter um tratamento de interrupção em uma tarefa de polling.

Servidores de prioridade fixa

A estratégica básica para tratar processamento aperiódico é transformar esse processamento num processamento periódico. O conceito central introduzido para resolver esses problemas é a noção de um servidor aperiódico. Um servidor aperiódico é um processo que recebe uma carga de processamento e um período de recarga. Um servidor aperiódico trata requisições aleatórias na sua prioridade atribuída enquanto sua carga de processamento estiver disponível.

- Servidor de "Background"

  Esse servidor funciona atendendo as requisições aperiódicas quando a fila de prontos periódicos está vazia, ou seja, se as tarefas periódicas não estão sendo executadas ou pendentes, as tarefas aperiódicas podem usar o processador.

- "Polling server"

  A idéia básica desse servidor consiste na definição de uma tarefa periódica para atender as tarefas aperiódicas. Um espaço é aberto periódicamente na escala para a execução da carga aperiódica, através da tarefa servidora de "Polling". Essa tarefa possui um período e um tempo de execução e, portanto, tem a sua prioridade calculada como qualquer outra tarefa periódica do sistema.

- "Deferrable Server"

  Esse servidor também é baseado na criação de uma tarefa periódica que tem sua prioridade calculada como no "Polling Server". Mas ao contrário do "Polling Server", o DS conserva a sua capacidade mesmo quando não existir requisições durante a ativação da tarefa DS.

Outros escalonadores

Minimum-laxity-first (MLF)

O algoritmo minimum-laxity-first associa um *laxity* à cada processo do sistema, e então seleciona o processo com o menor *laxity* para ser executado. *Laxity* é definido como:

laxity = deadline – tempo atual – tempo de CPU preciso

Laxity é a medida da flexibilidade disponível para escalonar um processo. Um *laxity* de t_l significa que mesmo que um processo seja atrasado por t_l unidades de tempo, ainda cumprirá o deadline. Um *laxity* de zero significa que o processo precisa iniciar a execução ou correrá o risco de falhar no cumprimento do deadline.

A principal diferença entre MLP e EDF é que MLF leva em consideração o tempo de execução de um processo, e que EDF não considera.

2.3 Abordagem Hierárquica

Algoritmos escalonadores convencionais são projetados com um conjunto de desvantagens em mente. Aplicações executando sobre esses escalonadores são forçadas a lidar com essas desvantagens. Por isso é difícil escolher o escalonador certo para um sistema operacional. De fato, a abordagem hierárquica propõe que não deveria ser escolhido em princípio, possibilitando que diferentes políticas de escalonamento sejam

dinamicamente carregadas no kernel, respondendo às necessidades das aplicações, sendo arranjadas numa estrutura hierárquica.

O principal objetivo [JDR 01] é reduzir o custo de adicionar uma política de escalonamento específica a uma aplicação em um sistema operacional de propósito geral. O custo de adicionar uma política de escalonamento a um sistema operacional sem escalonadores carregados inclui o custo de obter e entender o código fonte do sistema operacional, e o custo de integrar o novo escalonador ao escalonador existente. É bem provável que o novo escalonador seja complexo e difícil de manter, assim como inflexível – sua política não pode ser facilmente substituída ou escrita de outra maneira.

Para alcançar os objetivos, alguns critérios devem ser seguidos [JDR 01]:

- Escalonadores são notificados de todos os eventos do sistema operacional que poderiam necessitar de uma decisão de escalonamento.
- Escalonadores devem ser capazes de agir apropriadamente depois de notificados da ocorrência de um evento.
- Escalonadores devem evitar o travamento do sistema, violando a integridade do SO.

A arquitetura hierárquica de escalonadores possui duas partes: a infra-estrutura de escalonadores, que é implementada no sistema operacional e fornece o framework para o escalonamento hierárquico, e a interface, que permite aos escalonadores carregados interagir com a estrutura. Esses escalonadores podem escalonar threads ou outros escalonadores; são dispostos numa hierarquia – o escalonamento começa no escalonador raiz e passa pelos nodos.

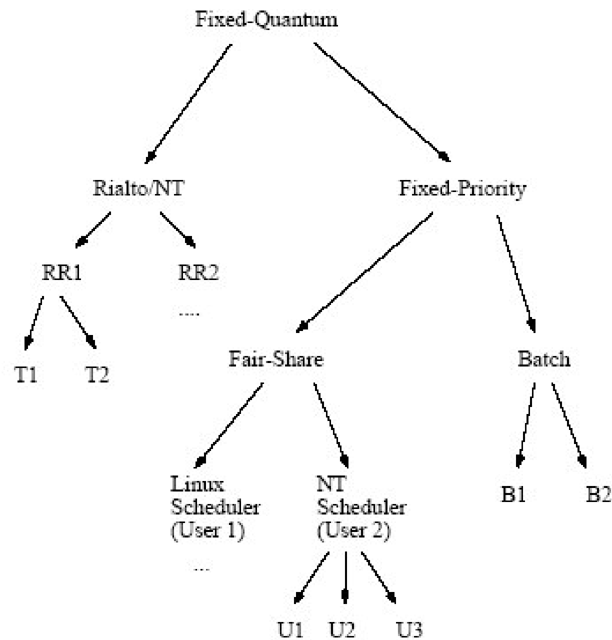

Fig 2.1: Exemplo de hierarquia de escalonadores

2.3.1 Infra-Estrutura Hierárquica de Escalonadores (HSI)

Escalonadores são bibliotecas de código que podem fazer parte do kernel ou carregadas no kernel em tempo de execução. Pelo menos um escalonador deve existir na estrutura para que o sistema operacional faça boot: escalonamento é necessário logo no processo de boot, num tempo em que seria difícil acessar o sistema de arquivos onde os escalonadores carregáveis são guardados. A estrutura mantém uma lista de todos os escalonadores, indexados pelo nome do escalonador. A implementação não deve ter dados estáticos, os dados estarão nas instâncias dos escalonadores.

As **instâncias dos escalonadores** são entidades ativas na hierarquia de escalonadores. Podem ser identificados pelo nome ou por um ponteiro, que referencia os dados que são alocados quando a instância é criada.

Processadores virtuais (VP) é o principal mecanismo que escalonadores usam para se comunicar. Cada processador virtual é compartilhado entre exatamente dois escalonadores, e representa o processador físico que o escalonador pai fornece ao escalonador filho. A separação entre a instância do escalonador e o processador virtual é importante no suporte a multiprocessadores. Processadores virtuais tem um estado –

executando, pronto ou esperando. Um processador virtual é mapeado em um processador físico apenas quando está executando.

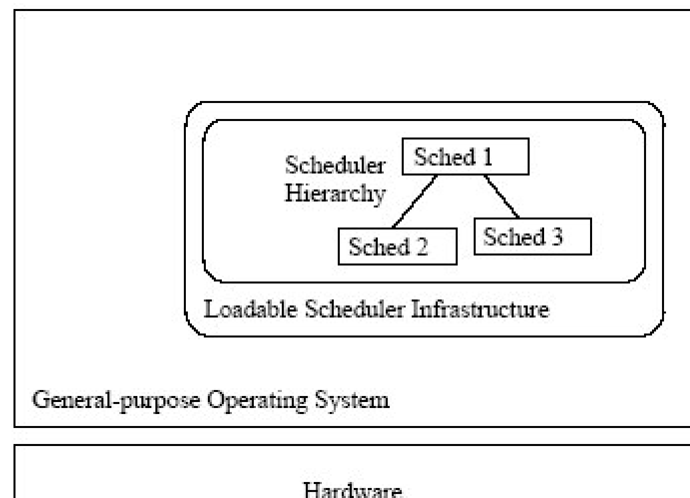

Fig 2.2: A infra-estrutura de escalonadores

Existem alguns arranjos hierárquicos de escalonadores que não faz sentido. Por exemplo, um escalonador tempo-real pode ser escalonado por um escalonador time-sharing; isso poderia ser feito simplesmente colocando a aplicação tempo-real para ser escalonada pelo escalonador time-sharing.

Existem dois principais aspectos no problema de decidir qual escalonador relaciona com qual escalonador. O primeiro aspecto é a necessidade de garantir que escalonadores podem cumprir os requisitos aos quais eles foram projetados, dado o tipo de escalonamento que eles recebem dos seus pais.

O segundo aspecto é o estabelecimento de uma igualdade semântica entre os escalonadores. Por exemplo, um escalonador EDF pode fornecer o "earliest deadline" ao escalonador pai; que seria compreensível se fosse outro escalonador EDF, mas não se fosse um escalonador baseado em prioridades.

2.3.2 Interface

Para suportar a implementação de escalonadores carregáveis, a estrutura de escalonadores deve:

- Manter a hierarquia de escalonadores – carregar e descarregar escalonadores, adicionar instâncias de escalonadores à hierarquia e remove-las.
- Pesquisar a hierarquia para tomar decisões de escalonamento

- Notificar escalonadores de eventos.
- Gerenciar tempo de processador de acordo com políticas especificadas pelo usuário.
- Suportar negociação entre aplicações e escalonadores.
- Prover serviços aos escalonadores, como mensagens, sincronização, timers...

2.3.3 Processadores Virtuais

Cada escalonador está relacionado com um ou mais escalonadores na hierarquia através de processadores virtuais. Numa relação, um escalonador é pai ou filho, dependendo se ele está acima ou abaixo de outro escalonador na hierarquia. Escalonadores relacionados notificam os outros de mudanças no estado dos processadores virtuais que eles compartilham.

2.3.4 Mandando Mensagens aos Escalonadores

Além de receber e enviar notificações sobre mudanças no estado dos processadores virtuais, um escalonador pode enviar uma mensagem ao seu pai. Mensagens não têm significados intrínsecos, escalonadores devem concordar em como interpretá-las.

Na prática, mensagens são mais usadas para alterar algum parâmetro de escalonamento num processador virtual.

Capítulo 3

Escalonamento do Linux

Como qualquer sistema multi-tarefa, Linux consegue o efeito de aparente execução simultânea de múltiplos processos trocando de um processo para outro em um período de tempo muito pequeno.

O algoritmo escalonador dos sistemas operacionais Unix deve cumprir vários objetivos: rápido tempo de resposta do processo, bom throughput de tarefas background, conciliação entre processos de alta e baixa prioridade, e assim por diante.

3.1 Política de Escalonamento

O conjunto de regras usadas para determinar quando e como selecionar novos processos para serem executados é chamado de política de escalonamento. O escalonamento no Linux é baseado na técnica time-sharing, vários processos rodam concorrentemente, o que significa que o tempo de CPU é dividido em pedaços, um para cada processo. Um único processador pode executar um único processo por vez. Se o processo atual não terminou quando o seu pedaço de tempo ou quantum termina, outro processo pode tomar seu lugar. Time-sharing conta com interrupções de timer e é transparente aos processos. Nenhum código adicional precisa ser inserido no programa para garantir o escalonamento.

A política de escalonamento também é baseada na classificação dos processos de acordo com suas prioridades. Cada processo é associado com um valor que mostra qual processo é mais apto a executar.

No Linux, a prioridade dos processos é dinâmica. O escalonador rastreia o que os processos estão fazendo e ajusta suas prioridades periodicamente; desse jeito, processos que foram negados o uso do CPU por um longo intervalo de tempo tem sua prioridade dinamicamente incrementada. De maneira correspondente, processos executando por um longo período de tempo são penalizados pela redução das suas prioridades.

3.2 Preempção de Processos

Os processos no Linux são preemptivos, se um processo entra no estado TASK_RUNNING, o kernel verifica se a sua prioridade dinâmica é maior que a prioridade do processo que está sendo executado. Se for, a execução do processo atual é interrompida e o escalonador é chamado para selecionar outro processo para executar (geralmente o processo que acabou de se tornar apto a executar). Um processo também pode ser interrompido quando seu quantum acaba. Quando isso acontece, o campo need_resched do processo atual é setado, assim o escalonador é chamado quando a interrupção do timer termina.

Por exemplo, vamos considerar um contexto no qual apenas dois programas, um editor de texto e um compilador, estão sendo executados. O editor é um programa interativo, logo tem uma prioridade dinâmica maior que a do compilador. Também, o editor é freqüentemente parado, já que a velocidade de entrada de dados do teclado é pequena. Entretanto, assim que o usuário pressiona uma tecla, acontece uma interrupção, e o kernel acorda o processo do editor. O kernel também determina que a prioridade dinâmica do editor é maior que a do processo atual (o compilador), e assim altera o campo need_resched do processo, forçando o escalonador a ser ativado quando o kernel termina de tratar a interrupção. O escalonador seleciona o editor e executa uma troca de processos. Como resultado, a execução do editor é retomada rapidamente e o caracter teclado pelo usuário aparece na tela. Quando o caracter é processado, o processo do editor se suspende para esperar por outro caracter do teclado, e o compilador pode continuar sua execução.

Um processo preemptado não é suspenso, já que ele continua no estado TASK_RUNNING, ele simplesmente não usa mais a CPU.

Alguns sistemas operacionais tempo-real permitem que qualquer processo (processos do sistema e do usuário) tenha qualquer prioridade. Outros, como Linux e Windows NT, têm prioridades diferentes para diferentes tipos de processos. Isso significa que o kernel do Linux não é preemptivo, ou seja, que um processo pode ser preemptado só quando está executando em User Mode. Kernels não preemptivos são muito mais fáceis de implementar, já que a maioria dos problemas de sincronização envolvendo estruturas de dados do kernel é facilmente evitada. Linux tem dois tipos de processos: user space e kernel

space. Processos rodando em user space podem mudar sua prioridade (através da chamada de sistema nice), mas todos eles podem ser preemptados por qualquer processo em kernel space. Kernel space tem três níveis de prioridades:

1. *Interrupções de Hardware*: O processo que trata uma interrupção de hardware tem a maior prioridade. O processo deve ser o mais curto possível, porque quando é executado todas as outras interrupções são desabilitadas.
2. *Interrupções de Software.* Esses processos tem a segunda maior prioridade: apenas interrupções de hardware pode preempta-los.
3. *Processos normais do kernel* rodam no nível mais baixo de prioridade do kernel. Porém eles preemptam todos os processos no espaço do usuário.

3.3 O Algoritmo Escalonador

O escalonador do Linux trabalha dividindo o tempo do CPU em épocas. Numa época, todos os processos têm um quantum específico cuja duração é calculada quando a época começa. Em geral, diferentes processos têm durações de quantum diferentes. O valor do quantum é a porção máxima de tempo atribuída ao processo naquela época. Quando um processo acaba o seu quantum, ele é preemptado e substituído por outro processo apto. Claro, um processo pode ser selecionado várias vezes na mesma época, enquanto o seu quantum não for esgotado. A época acaba quando todos os processos aptos esgotam seus quantums, nesse caso, o escalonador recalcula os quantums de todos os processos e uma nova época começa.

Cada processo tem um quantum base: é o valor do quantum atribuído pelo escalonador ao processo se o processo esgotou seu quantum na época anterior. O usuário pode mudar o quantum base do seu processo usando a chamada de sistema nice() e setpriority(). Um novo processo sempre herda o quantum base do seu pai.

A macro INIT_TASK altera o valor do quantum base do processo 0 para DEF_PRIORITY. Essa macro é definida a seguir:

```
#define DEF_PRIORITY (20*HZ/100)
```

Como HZ, que denota a freqüência das interrupções do timer, é alterado para 100 para IBM PCs, o valor de DEF_PRIORITY é de 20 ticks, cerca de 210 ms.

Usuários raramente mudam os quantum base dos seus processos, logo DEF_PRIORITY também representa o quantum base da maioria dos processos no sistema.

Para selecionar um processo para executar, o escalonador do Linux deve considerar a prioridade de cada processo. Existem dois tipos de prioridade:

Prioridade estática

Essa prioridade é atribuída pelo usuário aos processos tempo-real e varia de 1 à 99. Nunca é alterada pelo escalonador.

Prioridade dinâmica

Esse tipo se aplica apenas aos processos convencionais. É essencialmente o somatório do quantum base (que também é chamado de prioridade básica do processo) e do número de ticks do tempo de CPU restantes para o processo antes que seu quantum se esgote na época atual.

A prioridade de um processo tempo-real é sempre maior que a prioridade dinâmica de um processo convencional: o escalonador irá começar a executar processos convencionais apenas quando não houver mais processos tempo-real no estado de TASK_RUNNING.

Duração Do Quantum

A duração do quantum é crítica para a performance do sistema: não deve ser muito longa nem muito curta.

Se a duração do quantum for muito curta, o atraso do sistema causado pela troca de processos será muito alto. Por exemplo, se a troca de um processo requer 10 ms, e se o quantum também é 10 ms, então pelo menos 50% dos ciclos do processador serão dedicados à troca de processos.

Se a duração do quantum for muito longa, processos deixarão de ter o efeito de serem executados concorrentemente. Por exemplo, se o quantum for de cinco segundos, cada processo apto é executado por cinco segundos, mas então ele para por um longo tempo.

Acredita-se que durações de quantum muito longas também acabam degradando o tempo de resposta dos programas interativos. Geralmente isso é falso. Processos interativos

tem uma prioridade relativamente alta, logo eles rapidamente preemptam os processos batch, não importando a duração do quantum.

Em alguns casos, um quantum muito longo degrada a capacidade de resposta do sistema. Por exemplo, se dois usuários concorrentemente entram dois comandos nos seus respectivos prompts de shell, um comando é CPU-bound, enquanto o outro é uma aplicação interativa. Os dois shells iniciam um novo processo e passa a execução do comando do usuário para esse processo, e esses dois novos processos têm a mesma prioridade inicialmente. Agora, se o escalonador seleciona o processo CPU-bound para executar, o outro processo irá esperar por um quantum inteiro antes de começar a sua execução. Então, se essa duração for muito longa, o sistema poderá parecer sem resposta para o usuário que o iniciou.

A escolha da duração do quantum é sempre um compromisso. A regra adotada pelo Linux é: escolha a duração mais longa possível, enquanto o tempo de resposta do sistema for aceitável.

3.4 Estruturas de Dados Usadas pelo Escalonador

A lista *process_list* reúne todos os descritores de processos, enquanto a lista *runqueue_list* reúne os descritores de todos os processos aptos. Em ambos os casos, o descritor do processo init_task está no início da lista.

Cada descritor de processos inclui vários campos relacionados com escalonamento:

need_resched

Um flag usado por ret_from_intr para decidir quando chamar a função schedule.

state

São cinco os estados possíveis de um processo. Esses estados são os seguintes:

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
```

TASK_RUNNING – identifica um processo que está sendo executado, ou esperando pelo processador.

TASK_INTERRUPTIBLE – identifica um processo que está suspenso até que alguma condição seja verdadeira. Uma interrupção, um recurso do sistema que o processo está esperando, o envio de um sinal são exemplos de condições que podem acordar um processo.

TASK_UNINTERRUPTIBLE – identifica um processo que está suspenso como no estado TASK_INTERRUPTIBLE, exceto que nesse caso a envio de um sinal não acorda o processo.

TASK_STOPPED identifica um processo cuja execução foi interrompida. O processo entra nesse estado sob o envio de um sinal SIGSTOP, SIGTSTP, SIGTTIN ou SIGTTOU.

TASK_ZOMBIE. A execução do processo terminou, mas o processo pai ainda não executou a chamada de sistema wait() para retornar informações sobre o processo morto. Antes de chamar wait(), o processo não pode discartar as informações contidas no descritor do processo morto porque o pai pode precisar.

policy

A classe de escalonamento. Os valores permitidos são:

SCHED_FIFO

Um processo tempo-real First-In, First-Out. Quando o escalonador libera o CPU para um processo, ele deixa o descritor do processo na sua posição atual da lista runqueue. Se nenhum outro processo tempo-real está apto, o processo continuará usando o CPU, até que outro processo tempo-real com a mesma prioridade se torne apto a executar.

SCHED_RR

Um processo tempo-real Round Robin. Quando o escalonador libera o CPU para um processo, ele coloca o descritor do processo no fim da lista runqueue. Essa política garante a divisão justa do tempo de

CPU para todos os processos tempo-real SCHED_RR que tenham a mesma prioridade.

SCHED_OTHER

Um processo convencional. O campo policy ainda representa um flag binário SCHED_YIELD. Esse flag é alterado quando um processo invoca a chamada de sistema sched_yield. O escalonador coloca o descritor do processo no final da lista runqueue.

rt_priority

A prioridade estática dos processos tempo-real. Processos convencionais não usam esse campo.

priority

O quantum base (ou prioridade base) dos processos.

counter

O número de ticks restantes do processo antes do quantum acabar. Quando uma nova época começar, esse campo contém a duração do quantum do processo.

Quando um novo processo é criado, do_fork altera o campo counter de ambos os processos da seguinte maneira:

```
current->counter >>= 1;
p->counter = current->counter;
```

Em outras palavras, o número de ticks restantes ao pai é dividido em duas partes, uma para o pai e outra para o filho. Isso é feito para prevenir que usuários peguem um tempo de CPU ilimitado.

Percebe-se que os campos priority e counter têm características diferentes para os vários tipos de processos. Para processos convencionais, são usados para implementar time-sharing e calcular a prioridade dinâmica do processo. Para processos tempo-real SCHED_RR, são usados apenas para implementar time-sharing. Finalmente, para processos tempo-real SCHED_FIFO, não são usados, porque o algoritmo escalonador considera a duração do quantum ilimitada.

3.5 A Função schedule

A função schedule implementa o escalonador. O seu objetivo é achar um processo na lista runqueue e então liberar o CPU para ele. É chamada, diretamente ou de maneira lazy, por várias rotinas do kernel.

Invocação direta

O escalonador é chamado diretamente quando o processo atual deve ser bloqueado imediatamente porque o recurso de que necessita não está disponível. Nesse caso, a rotina do kernel que quer bloquear o processo procede da seguinte maneira:

1. Insere o atual na fila wait apropriada
2. Muda o estado do processo atual para TASK_INTERRUPTIBLE ou para TASK_UNINTERRUPTIBLE
3. Chama schedule
4. Checa se o recurso está disponível; se não, volta para o passo 2
5. Uma vez que o recurso esteja disponível, remove o processo atual da fila wait

Como pode se ver, a rotina do kernel verifica repetidamente se o recurso necessário está disponível; se não, o processo libera o processador para outro processo chamando schedule. Depois, quando o escalonador de novo libera o processador para o processo, a disponibilidade do recurso é verificada de novo.

O escalonador também é diretamente chamado por muitos drivers de dispositivos que executam longas tarefas iterativas. A cada ciclo da iteração, o driver verifica o valor do campo need_resched e, se necessário, chama schedule() para voluntariamente ceder a CPU.

Invocação lazy

O escalonador pode também ser chamado de uma maneira lazy setando o campo need_resched do processo atual. Como o valor desse campo é sempre verificado antes de continuar a execução de um processo do usuário, schedule será chamada brevemente.

Invocação lazy do escalonador se faz nos seguintes casos:

- Quando o processo atual esgota o seu quantum; isso é feito pela função update_process_times.

- Quando um processo é acordado e a sua prioridade é maior do que a do processo atual; essa tarefa é executada pela função reschedule_idle, que é chamada pela função wake_up_process:

```
if (goodness(current, p) > goodness(current, current))
        current->need_resched = 1;
```

- Quando as chamadas de sistema sched_setscheduler ou sched_ yield são chamadas.

Ações executadas pela função schedule

Antes de realmente escalonar um processo, a função schedule começa executando as funções deixadas pelos outros fluxos de controle do kernel em várias filas. A função chama run_task_queue na fila de processos tq_scheduler. Linux coloca uma função na fila de tarefas quando precisa parar sua execução até a próxima invocação de schedule:

```
run_task_queue(&tq_scheduler);
```

A função então executa todas as bottom halves ativas. Essas geralmente estão presentes para executar tarefas requisitadas por drivers de dispositivos:

```
if (bh_active & bh_mask)
        do_bottom_half( );
```

Agora vem o escalonamento, e então uma potencial troca de processos. O ponteiro pro processo é guardado na variável local prev e o campo need_resched é alterado para 0. A principal tarefa da função é alterar a variável local next para que ela aponte para o descritor do processo selecionado a substituir prev.

Primeiro, uma checagem é feita para determinar se prev é um processo tempo-real Round Robin que esgotou seu quantum. Se for, schedule atribui um novo quantum a prev e o coloca no fim da lista runqueue:

```
if (!prev->counter && prev->policy == SCHED_RR) {
        prev->counter = prev->priority; move_last_runqueue(prev);
}
```

Então schedule() examina o estado de prev. Se ele tiver sinais não-bloqueantes pendentes e o seu estado for TASK_INTERRUPTIBLE, a função acorda o processo. Essa ação não é a mesma que libera o processador a prev; ela apenas da a prev uma chance de ser selecionado para executar:

```
if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
        prev->state = TASK_RUNNING;
```

Se prev não está no estado TASK_RUNNING, schedule foi chamada diretamente pelo próprio processo porque ele teve que esperar por algum recurso externo; logo, prev deve ser removido da lista runqueue:

```
if (prev->state != TASK_RUNNING)
        del_from_runqueue(prev);
```

A seguir, schedule deve selecionar um processo para ser executado no próximo quantum. Para esse fim, a função varre a lista *runqueue_list*. Ela começa pelo processo referenciado pelo campo next_run de init_task, que é o descritor do processo 0 (swapper). O objetivo é guardar em next o ponteiro para o descritor do processo de maior prioridade. Para fazer isso, next é inicializado com o primeiro processo apto a ser checado, e c é inicializada com o seu "goodness":

```
if (prev->state == TASK_RUNNING) {
        next = prev;
        if (prev->policy & SCHED_YIELD) {
                prev->policy &= ~SCHED_YIELD;
                c = 0;
        } else
                c = goodness(prev, prev);
} else {
        c = -1000;
        next = &init_task;
}
```

Se o flag SCHED_YIELD de prev->policy está setado, prev voluntariamente cedeu o CPU chamando a chamada de sistema sched_yield. Nesse caso, a função atribui um goodness igual zero.

Então schedule repetidamente chama a função goodness nos processos aptos para determinar o melhor candidato:

```
p = init_task.next_run;
while (p != &init_task) {
        weight = goodness(prev, p);
        if (weight > c) {
                c = weight;
                next = p;
        }
        p = p->next_run;
}
```

O laço while seleciona o primeiro processo na lista runqueue que tenha o maior weight. Se o processo anterior está apto, é escolhido com respeito aos outros processos aptos com o mesmo weight.

Se a lista runqueue está vazia (não existe processos aptos à não ser pelo *swapper*), o laço não inicia e next aponta para init_task. Além disso, se todos os processos na lista runqueue tem uma prioridade menor do que à prioridade de next ou igual, nenhuma troca de processos acontecerá e o processo antigo continuará a ser executado.

Uma checagem posterior precisa ser feita na saída do laço para determinar se c é 0. Isso acontece apenas quando todos os processos na lista runqueue esgotaram seus quantum, isto é, todos eles tem o campo counter em zero. Quando isso acontece, uma nova época começa, então schedule atribui a todos os processos existentes (não apenas os TASK_RUNNING) um quantum novo, cuja duração é a soma do valor da prioridade mais metade do valor de counter:

```
if (!c) {
        for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
}
```

Dessa maneira, processos parados ou suspensos têm suas prioridades dinâmicas periodicamente incrementadas. A razão para incrementar o valor de counter dos processos parados ou suspensos é para dar preferência aos processos I/O-bound. Entretanto, mesmo

depois de um número infinito de incrementos, o valor de counter não pode ser maior do que duas vezes o valor da prioridade.

Agora vem a parte final de schedule: se um processo diferente de prev foi selecionado, uma troca de processos acontece. Antes disso, entretanto, o campo context_swtch de kstat é incrementado de 1 para atualizar as estatísticas mantidas pelo kernel:

```
if (prev != next) {
        kstat.context_swtch++;
        switch_to(prev,next);
}
return;
```

A instrução return que sai de schedule não será executada imediatamente pelo processo next mas posteriormente pelo processo prev quando o escalonador selecioná-lo de novo para executar.

Avaliação dos processos aptos

A tarefa do algoritmo escalonador inclui identificar o melhor candidato entre todos os processos na lista runqueue. Isso é o que a função goodness faz. Ela recebe como parâmetros de entrada prev (o ponteiro para o descritor do processo executado anteriormente) e p (o ponteiro para o descritor do processo à avaliar). O valor inteiro de c retornado por goodness mede o "goodness" de p e tem os seguintes significados:

$$c = -1000$$

p nunca deve ser selecionado; esse valor é retornado quando a list runqueue contem apenas init_task.

$$c = 0$$

p esgotou seu quantum. A menos que p seja o primeiro processo na lista runqueue e todos os processos aptos também esgotaram seus quantum, ele não será selecionado para executar.

$$0 < c < 1000$$

p é um processo convencional que não esgotou seu quantum; quanto maior o valor de c maior o goodness.

$$c >= 1000$$

p é um processo tempo-real; quanto maior o valor de c maior o goodness.

A função goodness é equivalente à:

```
if (p->policy != SCHED_OTHER)
        return 1000 + p->rt_priority;
if (p->counter == 0)
        return 0;
if (p->mm == prev->mm)
        return p->counter + p->priority + 1;
return p->counter + p->priority;
```

Se o processo é tempo-real, seu goodness é alterado para pelo menos 1000. Se for um processo convencional que esgotou seu quantum, seu goodness é setado em 0; senão, é setado em p->counter + p->priority.

Um pequeno bônus é dado à p se ele compartilha o espaço de endereçamento com prev (o campo mm do descritor dos seus processos aponte para o mesmo descritor de memória). A razão para esse bônus é que se p executar logo depois de prev, alguns dos dados podem ainda estar na cache.

3.6 Performance do algoritmo escalonador

O algoritmo escalonador do Linux é relativamente fácil de entender. Por essa razão, muitos kernel hackers tentam fazer melhorias. Entretanto, o escalonador é um componente um tanto misterioso do kernel. Enquanto se pode alterar significantemente sua performance apenas modificando alguns principais parâmetros, geralmente não existe suporte teórico para justificar os resultados obtidos. Além disso, não se pode ter certeza que os resultados positivos (ou negativos) obtidos serão os mesmos quando o conjunto de processos mudar (tempo-real, interativos, I/O-bound, background, etc.). Mas para quase todas as estratégias de escalonamento, é possível achar um conjunto artificial de requisições que realça os pontos fracos do sistema.

Vamos tentar destacar alguns pontos fracos do escalonador do Linux. Algumas dessas limitações se tornam significantes num sistema com muitos usuários. Numa estação de trabalho que executa algumas centenas de processos ao mesmo tempo, o escalonador do

Linux é razoavelmente eficiente. Como Linux nasceu num 80386 e continua muito popular no mundo PC, consideramos o atual escalonador do Linux apropriado.

O algoritmo não escalona bem

Se o número de processos existentes é muito grande, é ineficiente recalcular todas as prioridades dinâmicas de uma vez.

Nos kernels tradicionais do Unix, a prioridade dinâmica era recalculada todo segundo, piorando ainda mais o problema. Linux tenta, por outro lado, diminuir o overhead do escalonador. Prioridades são recalculadas apenas quando todos os processos esgotaram seus quantums. Entretanto, quando o número de processos é grande, a fase de cálculo é mais cara mas é executada menos freqüentemente.

Essa aproximação tem a desvantagem de quando o número de processos é muito alto, processos I/O-bound são raramente acelerados, e então processos interativos tem um tempo de resposta maior.

Quantum muito grande

O tempo de resposta ao usuário depende muito da carga do sistema, que é o número médio de processos que estão aptos, e esperando por tempo de CPU.

O tempo de resposta do sistema também depende da duração média do quantum dos processos aptos. No Linux, o quantum pré-definido parece muito alto para máquinas high-end tendo um system load muito alto.

Preferência por processos I/O-bound

A preferência por processos I/O-bound é uma boa estratégia para garantir um tempo de resposta pequeno para programas interativos, mas não é perfeita. De fato, alguns programas batch com quase nenhuma interação com o usuário são I/O-bound. Por exemplo, considerando um banco de dados que deve ler muita informação do disco rígido ou uma aplicação de rede que deve coletar dados de um servidor remoto. Mesmo que esses tipos de processos não precisem de um tempo de resposta curto, eles são privilegiados pelo algoritmo escalonador.

Por outro lado, programas interativos que são também CPU-bound podem parecer lento para os usuários, já que o incremento da prioridade dinâmica causada pelas operações bloqueantes pode não compensar o decremento causado pelo uso da CPU.

Fraco Suporte a Aplicações de Tempo-Real

Linux não foi projetado para ser um sistema operacional tempo real. A principal razão é o comportamento não preemptivo do kernel do Linux. Essa característica não preemtiva do kernel se da pela não execução eventual de uma tarefa com prioridade maior mesmo se isso for desejado. Essa situação aparece quando uma tarefa de baixa prioridade está sendo executada em kernel mode.

Existem muitas tarefas ou processos executando no Linux em um dado momento de uma maneira *time-sharing*. Uma tarefa está sendo executada em *user mode* ou em *kernel mode*. Por exemplo, quando uma tarefa de usuário solicita algum recurso do sistema usando uma chamada de sistema, ela está executando em *kernel mode*. Nenhum processo pode ser interrompido no kernel, mesmo que uma tarefa de maior prioridade necessite do processador de uma maneira urgente. Além disso, quando um processo entra em *kernel mode*, ele desabilita as interrupções. Como resultado, interrupções de dispositivos externos podem ser perdidas durante o processamento de uma tarefa em *kernel mode*. Isso é altamente indesejável em sistemas onde o tempo é crítico, como em aquisições de dados, aplicações de rede de alto desempenho. Para fazer o kernel do Linux se comportar como um kernel tempo-real, muitas mudanças são necessárias. De fato, seria necessário reescrever quase completamente o kernel do Linux.

Latência do escalonador

Latência no Linux é o tempo entre o sinal de acordar que sinaliza que um evento ocorreu e o escalonador ter uma oportunidade de escalonar o processo que está esperando para acordar. Esses sinais podem vir de interrupções de hardware ou de um outro processo. Será considerado o escalonamento decorrente de uma interrupção de hardware, porém o escalonamento sinalizado por um outro processo é igual.

Tempo de resposta

Existem quatro fatores que compoem o tempo de resposta do kernel à um evento:

1. Latência de interrupção. É o tempo que decorre entre o sinal da interrupção sendo ativado e a rotina de tratamento da interrupção sendo executada.
2. Duração do tratamento da interrupção. É o tempo que a rotina de tratamento da interrupção gasta para tratar a interrupção. Essa rotina é a que altera o flag need_resched, significando que o escalonamento é necessário.
3. Latência do escalonador. É o tempo que decorre entre a finalização da rotina de tratamento da interrupção e a função de escalonamento sendo executada.
4. Duração do escalonamento. É o tempo que o escalonador leva para decidir qual processo deve ser executado e a troca de contexto.

3.7 Chamadas de sistema relacionadas ao escalonador

Programadores podem trocar os parâmetros de escalonamento usando chamadas de sistema descritas na seguinte tabela:

Função	**Descrição**
nice( )	Muda a prioridade de processos convencionais.
getpriority( )	Retorna a prioridade máxima de um grupo de processos convencionais.
setpriority( )	Seta a prioridade de um grupo de processos convencionais.
sched_getscheduler( )	Retorna a política de escalonamento de um processo.
sched_setscheduler( )	Seta o método de escalonamento e a prioridade de um processo.
sched_getparam( )	Retorna a prioridade de escalonamento de um processo.
sched_setparam( )	Seta a prioridade de um processo.
sched_yield( )	Cede voluntariamente o processador sem bloquear.

sched_get_ priority_min( )	Retorna o valor da prioridade mínima de um método de escalonamento.
sched_get_ priority_max( )	Retorna o valor da prioridade máxima de um método de escalonamento.
sched_rr_get_interval( )	Retorna o valor do quantum do método Round Robin.

Tabela 3.1: Chamadas de sistemas relacionadas ao escalonamento

Grande parte das chamadas de sistema da tabela acima se aplica aos processos tempo-real, permitindo que usuários desenvolvam aplicações tempo-real. Entretanto, Linux não suporta a maioria das aplicações tempo-real mais exigentes porque seu kernel é não-preemptivo.

Normalmente, usuários podem sempre diminuir a prioridade de seus processos, mas se quiserem alterar a prioridade de processos pertencentes à outros usuários ou se quiserem aumentar a prioridade de seus próprios processos, precisam ter privilégios de superusuário.

Capítulo 4

Abordagem Hierárquica no Linux – Uma Experiência de Implementação

A presente implementação da hierarquia é claramente um protótipo [GMC 98], com algumas limitações. Por exemplo, a infra-estrutura permite apenas o carregamento de dois escalonadores simultaneamente, além do escalonador do Linux.

Os escalonadores serão carregados para dentro do kernel para aumentar a eficiência e para ter acesso às estruturas de dados usadas pelo kernel. Não existe uma razão pela qual um escalonador em espaço de usuário não pudesse ser usado. Esses escalonadores poderiam ser mais facilmente depurados porque ferramentas mais sofisticadas estão disponíveis em espaço de usuário, mas o overhead da troca de contexto é muito alto.

4.1 Controle de Fluxo

Cada escalonador fornece uma *rotina de pesquisa* que a infra-estrutura chama enquanto faz uma decisão de escalonamento. Essa rotina retorna uma referência para o processo a ser executado, ou informa que não tem nada para executar. Para selecionar um processo, o sistema pesquisa os escalonadores começando da raiz da hierarquia. Uma vez selecionado um processo, o dispatcher restaura o contexto e libera o processador.

4.2 Interface

Os escalonadores precisam de uma maneira de informar o sistema de que está sendo carregado. Para isso foi adicionada a função:

reg_sched(função, nome)

O parâmetro *função* é o endereço da função a ser chamada quando a infra-estrutura precisa fazer uma decisão. Essa função deve retornar um processo apto a ser executado ou NULL, que indica que nenhum processo precisa ser escalonado. Nesse caso, outro escalonador pode ser consultado, ou retornar para o escalonador do Linux. O parâmetro *nome* representa o nome do escalonador e serve para que os escalonadores sejam identificados no sistema. Isso permite que processos indiquem qual o escalonador que eles usarão.

Para os escalonadores serem descarregados do sistema é usada a função:

unreg_sched(nome)

Desse modo o usuário pode retirar um escalonador da infra-estrutura assim que o mesmo não for mais necessário. O parâmetro *nome* indica qual o escalonador está sendo descarregado.

Como se trata de escalonadores tempo-real, alguns parâmetros, que não são necessários em escalonadores normais, precisam ser setados. Alguns desses parâmetros são: deadline, tempo de execução, tempo de início, etc, dependendo do design da infra-estrutura. Esses parâmetros podem ser alterados pelo seguinte função:

set_opt(par, valor)

O parâmetro *par* indica qual das variáveis do processo está sendo alterada. O parâmetro *valor* é o novo valor da variável.

4.3 Estruturas de Dados

4.3.1 Escalonadores

A estrutura de dados *tcc_scheduler* é usada para manter informações sobre os escalonadores carregados no sistema.

Em [JDR 01] é proposta uma estrutura em que os escalonadores são colocados em uma árvore. Cada nodo dessa árvore é um escalonador e seus ramos são escalonadores imediatamente inferiores, sendo escalonados por ele.

Para simplificar a implementação, essas estruturas são armazenadas numa lista com espaço para dois escalonadores, porém é bastante simples aumentar a capacidade dessa lista. Novos escalonadores são inseridos no final da mesma.

A estrutura é a seguinte:

```
struct tcc_scheduler {
	struct task_struct * (*tcc_sched_func)(int);
	char *tcc_scheduler_name;
};
```

- Tcc_sched_func – ponteiro para a função escalonadora. Retorna um ponteiro para um processo ou NULL se não há processo a ser escalonado. Possui um parâmetro do tipo *int* que é um ponteiro para a lista *run_queue*.
- Tcc_scheduler_name – nome do escalonador. Identifica o escalonador na estrutura.

4.3.2 Descritor de Processos

Novos campos foram adicionados ao descritor de processos para que pudesse suportar algumas características importantes dos processos tempo-real.

- tcc_execution_time – tempo de execução do processo
- current_tcc_execution_time – tempo de execução atual do processo.
- tcc_start_time – tempo de início do processo.
- next_tcc_start_time – próximo tempo de início do processo.
- tcc_deadline – deadline do processo.
- tcc_scheduler_name – nome do escalonador que o processo pertence.

4.4 Implementação no Escalonador do Linux

No escalonador nativo do linux foi adicionado o seguinte código:

```
for(i=0 ; i<=tcc_n_sched ; i++) {
        struct task_struct *tcc_ts;
        tcc_ts = tcc_schedulers[i].tcc_sched_func((int)&runqueue_head)
        if(tcc_ts != NULL) {
                c = 1000; next = tcc_ts;
                goto continue_schedule_tcc;
        }
}
```

Existe um laço em que cada escalonador é pesquisado e retorna o processo selecionado ou NULL caso nenhum processo esteja apto. O *goto continue_schedule_tcc* é usado para interromper qualquer pesquisa, tanto dos escalonadores carregados quanto do escalonador nativo do linux, quando um processo foi selecionado por um escalonador.

4.5 Resultados

Todos os testes foram feitos num Athlon 1,3 Ghz com um único processador. O computador de teste tinha 256 MB de memória principal e estava equipado com um disco IDE de 40 GB.

A duração dos eventos foi medida usando o contador do Pentium pela instrução *rdtsc*. Essa instrução retorna um valor 64-bit contendo o número de ciclos do processador desde que foi ligado. Num processador de 1,3 Ghz o contador tem uma resolução de 0,8 ns.

Um dos principais critérios para avaliação de suporte de escalonadores carregáveis é o desempenho de aplicações que não usam a nova funcionalidade. Essas aplicações não devem ter o desempenho prejudicado.

Um valor que os escalonadores carregados podem alterar é o tempo de troca de contexto. Esse é o tempo entre a troca de um processo que está sendo executado pelo processador por um outro processo escolhido por qualquer um dos escalonadores.

Escalonador	Tempo (ciclos)	Tempo (µs)
Original do Linux	1330,27	1,02
TCC (sem escalonador carregado)	1370,979	1,05
TCC (com escalonador carregado)	1669,874	1,28
TCC (processo escalonado)	119,7041	0,092

Tabela 4.1: Latência do escalonador

O algoritmo escalonador do Linux demorou 1330,27 ciclos em média para avaliar cada um dos processos e selecionar um deles para ser executado. O escalonador modificado nessa implementação levou 1370,979 ciclos em média para executar o mesmo processo, quando não tinha escalonador carregado. Com o escalonador de teste do anexo C carregado, o mesmo escalonador levou 1669,874 ciclos para realizar a mesma tarefa. Esse acréscimo de tempo é decorrente da pesquisa que o escalonador carregado faz na lista *run_list* procurando por um processo a ser escalonado. O último tempo é o tempo em que o escalonador de teste seleciona um processo para ser executado. Esse é o menor dos tempos, já que esse escalonador é bem simples e não necessita de muitos cálculos.

O novo escalonador não teve um desempenho tão inferior ao do escalonador original, o que mostra que o sistema não foi prejudicado com a implementação da nova funcionalidade.

Outro critério que auxilia a avaliação de desempenho da infra-estrutura de escalonadores é a prova de que um escalonador de teste pode funcionar quando carregado nessa estrutura. Para esse teste foi usado o mesmo escalonador do anexo C que foi

carregado e usado por dois processos. Esse escalonador é muito simples para uso prático, porém é suficiente para se observar o sistema em funcionamento.

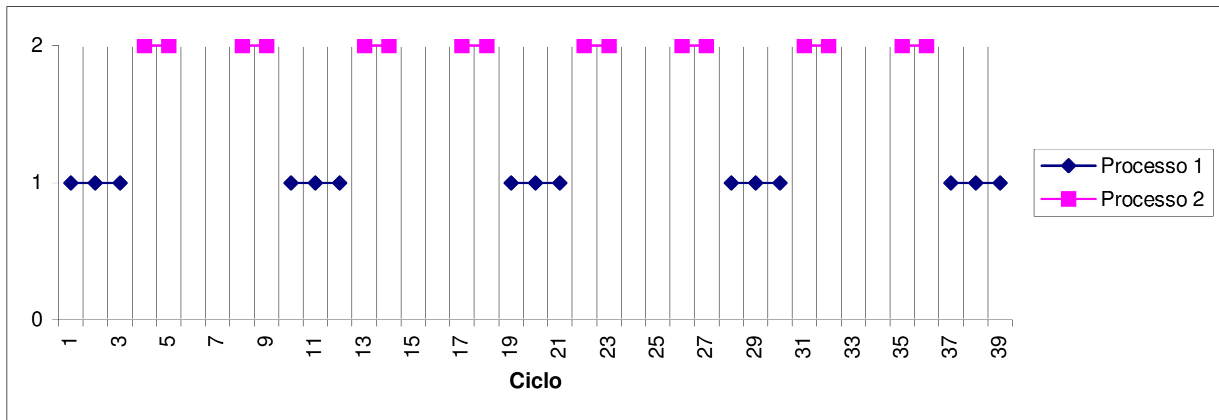

Gráfico 4.1: Execução de dois processos

O gráfico 4.1 mostra a execução de dois processos: Processo 1 e Processo 2. O Processo 1 tem período de 10 ciclos e tempo de execução de 3 ciclos, enquanto que o Processo 2 tem período de 4 ciclos e tempo de execução de 2 ciclos. Ao Processo 2 foi atribuída prioridade menor que a do Processo 1, e por isso ele é atrasado nos ciclos 6, 13, 22, e assim por diante.

Essa execução depende muito do escalonador carregado e dos parâmetros de cada processo.

Capítulo 5

Considerações Finais

Foram encontradas várias dificuldades durante a implementação da estrutura hierárquica de escalnadores. Essas dificuldades foram ocasionadas pela ainda imaturidade no estudo sobre escalonamento dinâmico e também pelo entendimento da estrutura do kernel do Linux.

Foi observado que a principal aplicação dessa estrutura hierárquica seria na área de sistemas de tempo-real, que tem características de escalonamento bem específicas e rígidas. Porém, para tais escalonadores serem carregados e funcionarem como foram projetados para funcionar, seria preciso algum suporte extra, como, por exemplo, funções para reprogramar o escalonador.

No protótipo não foram implementados os mecanismos de comunicação entre escalonadores, uma vez que não teriam impacto direto na perfomance do sistema. Esses mecanismos de comunicação seriam necessários no sistema com a hierarquia completa, que não foi o objetivo do trabalho.

Ainda, o desempenho dos escalonadores depende de aspectos próprios do sistema operacional. Escalonadores de tempo-real estão sujeitos a restrições que prejudicam a eficiência, por exemplo, o fato do kernel do Linux não ser preemptivo.

Foram alcançados os objetivos de aprender sobre sistemas operacionais, escalonamento, domínio da linguagem C, entre outros.

Para uma melhor análise da estrutura criada, acredita-se que seria interessante a implementação completa da hierarquia, com alguns escalonadores reais funcionando com aplicações reais. Algumas dessas aplicações podem precisar de alterações no código para funcionar de forma apropriada com as novas funcionalidades.

Referências Bibliográficas

[DPB 00] BOVET, Daniel P.;CESATI, Marco. **Understanding the Linux Kernel.** O'Reilly, 2000.

[GMC 98] CANDEA, George M.; JONES, Michael B. **Vassal: Loadable Scheduler Support for Multi-Policy Scheduling. 1998.**

[RTAI 03] DIAPM RTAI. **Realtime Application Interface.** www.rtai.org.

[JMF 01] FARINES, Jean Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva. **Sistemas de Tempo Real,** Escola de Computação, São Paulo, 2001.

[MBJ 03] JONES, Michael B.; REGEJR, John D. **CPU Reservations and Time Constraints: Implementation Experience on Windows NT.** 2003

[BW 89] BAUMGARTNER K.; WAH B. W. **GAMMON: A Load Balancing Strategy for a Local Computer System with a Multiaccess Network**, *IEEE Trans. on Computers*, vol. 38, no. 8, pp. 1098-1109, Agosto. 1989.

[DSN 03] NIKOLOPOULOS, Dimitrios S. **Scheduling in Linux.** 2003.

[JDR 01] REGEHR, John D. **Using Hierarchical Scheduling to Support Soft Real-Time Applications in General-Purpose Operating Systems.** Tese de doutorado. School of Engineering and Applied Science University of Virginia. 2001.

[SR 00] RHINE, Scott. **Loadable Scheduler Modules on Linux.** Disponível em <http://resourcemanagement.unixsolutions.hp.com/WaRM/prm_linux/docs/loadable_sched.html> . Acesso em: Junho, 2003.

[MAS 01] SOUZA, Márcio Augusto. **Análise de Técnicas de Monitoração e Métricas de Desempenho Para a Avaliação do Escalonamento de Processos.** Tese de doutorado. USP, 2001.

[PS 02] STAHLBERG, Patrick. **Linux Scheduling.** 2002.

[RTLINUX 03] RTLinux. www.fsmlabs.com.

Anexos

Anexo A – Artigo

Anexo B – Código Fonte

```
/*
 *	sched.c
 */
asmlinkage void schedule(void)
{
	struct schedule_data * sched_data;
	struct task_struct *prev, *next, *p;
	struct list_head *tmp;
	int this_cpu, c,i;

	unsigned long lat = tim();

	spin_lock_prefetch(&runqueue_lock);

	BUG_ON(!current->active_mm);
need_resched_back:
	prev = current;
	this_cpu = prev->processor;

	if (unlikely(in_interrupt())) {
		printk("Scheduling in interrupt\n");
		BUG();
	}

	release_kernel_lock(prev, this_cpu);

	/*
	 * 'sched_data' is protected by the fact that we can run
	 * only one process per CPU.
	 */
	sched_data = & aligned_data[this_cpu].schedule_data;

	spin_lock_irq(&runqueue_lock);

	/* 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);
		}

	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;
        /*
         * this is the scheduler proper:
         */
repeat_schedule:
        /*
         * Default process to select..
         */
        next = idle_task(this_cpu);
        c = -1000;

        tcc_ticks++;
        for(i=0 ; i<=tcc_n_sched ; i++) {
                struct task_struct *tcc_ts;
                tcc_ts = tcc_schedulers[i].tcc_sched_func((int)&runqueue_head);

                if(tcc_ts != NULL) {
                        c = 1000; next = tcc_ts;
                        goto continue_schedule_tcc;
                }
        }
        list_for_each(tmp, &runqueue_head) {
                p = list_entry(tmp, struct task_struct, run_list);
                if(p->policy!=SCHED_TCC && can_schedule(p,this_cpu)) {
                        int weight = goodness(p, this_cpu, prev->active_mm);
                        if (weight > c)
                                c = weight, next = p;
                }
        }
continue_schedule_tcc:
        /* 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;
	}

	/*
	 * from this point on nothing can prevent us from
	 * switching to the next task, save this fact in
	 * sched_data.
	 */
	sched_data->curr = next;
	task_set_cpu(next, this_cpu);
	spin_unlock_irq(&runqueue_lock);

	if (unlikely(prev == next)) {
		/* We won'  t go through the normal tail, so do this by hand */
		prev->policy &= ~SCHED_YIELD;
		goto same_process;
	}

#ifdef CONFIG_SMP
	/*
	 * maintain the per-process '  last schedule'   value.
	 * (this has to be recalculated even if we reschedule to
	 * the same process) Currently this is only used on SMP,
	 * and it'  s approximate, so we do not have to maintain
	 * it while holding the runqueue spinlock.
	 */
	sched_data->last_schedule = get_cycles();

#endif /* CONFIG_SMP */

	kstat.context_swtch++;
	prepare_to_switch();
	{
		struct mm_struct *mm = next->mm;
		struct mm_struct *oldmm = prev->active_mm;
		if (!mm) {
			BUG_ON(next->active_mm);
			next->active_mm = oldmm;
			atomic_inc(&oldmm->mm_count);
			enter_lazy_tlb(oldmm, next, this_cpu);
		} else {
			BUG_ON(next->active_mm != mm);
			switch_mm(oldmm, mm, next, this_cpu);
		}

		if (!prev->mm) {
			prev->active_mm = NULL;
```

```
                mmdrop(oldmm);
            }
        }
        /*
         * This just switches the register state and the
         * stack.
         */
        switch_to(prev, next, prev);
        __schedule_tail(prev);

same_process:
        reacquire_kernel_lock(current);
        if (current->need_resched)
            goto need_resched_back;
        tcc_latency = tim()-lat;
        return;
}
```

```
/*
 * tcc.c
 */
#include <linux/tcc.h>
#include <linux/sched.h>

int tcc_n_sched=-1;
struct tcc_scheduler tcc_schedulers[2];

int sys_tcc_reg_sched(int p,char *name) {
	if(tcc_n_sched>=1)
		return 1;
	tcc_n_sched++;
	tcc_schedulers[tcc_n_sched].tcc_sched_func = p;
	tcc_schedulers[tcc_n_sched].tcc_scheduler_name = (char*) kmalloc(sizeof(name));
	strcpy(tcc_schedulers[tcc_n_sched].tcc_scheduler_name,name);
	return 0;
}
int sys_tcc_unreg_sched(char *name) {
	if(tcc_n_sched<=-1)
		return 1;

	tcc_n_sched--;
	return 0;
}
int sys_tcc_set_opt(int opt,unsigned long v) {
	switch(opt) {
		case TCC_EXECUTION_TIME: // execution time
			current->tcc_execution_time = v;
			break;
		case TCC_START_TIME:
			current->tcc_start_time = v;
			break;
		case TCC_DEADLINE:
			current->tcc_deadline = v;
			break;
		deafult:
			return 1;
	}
	return 0;
}
unsigned long sys_tcc_get_ticks() {
	return tcc_ticks;
}
unsigned long sys_tcc_get_latency() {
	return tcc_latency;
}
```

```
/*
 * tcc.h
 */
#ifndef _TCC_TCC_H
#define _TCC_TCC_H

#include <linux/sched.h>

#define TCC_EXECUTION_TIME  0
#define TCC_START_TIME      1
#define TCC_DEADLINE        2

extern int sys_reg_sched(int,char *name);
extern int sys_unreg_sched(char *name);
extern int sys_tcc_set_opt(int opt,unsigned long v);
extern unsigned long sys_tcc_get_ticks();
extern unsigned long sys_tcc_get_latency();

extern int tcc_n_sched;

struct tcc_scheduler {
        struct task_struct * (*tcc_sched_func)(int);
        char *tcc_scheduler_name;
};
extern struct tcc_scheduler tcc_schedulers[2];

#endif
```

Anexo C – Escalonador Exemplo

```
struct task_struct *schedule() {
    struct list_head *tmp;
    struct task_struct *p;
    struct list_head *runqueue_head;

    list_for_each(tmp,runqueue_head) {
        p = list_entry(tmp,struct task_struct,run_list);
        if (p->policy == SCHED_TCC) {
            // se tem algum processo rodando esse é o escolhido
            if(p->current_tcc_execution_time > 0) {
                p->current_tcc_execution_time--;
                return p;
            }
            if(p->next_tcc_start_time <= tcc_get_ticks()) {
                p->next_tcc_start_time=tcc_get_ticks() + p->tcc_start_time;
                p->current_tcc_execution_time = p->tcc_execution_time-1;
                return p;
            }
        }
    }
    return NULL;
}
```