Uma Arquitetura Baseada em Eventos para Desenvolvimento de Políticas de Escalonamento de Processos

Luciano Porto Barreto

Laboratório de Sistemas Distribuídos - LaSiD
Departamento de Ciência da Computação - DCC
Universidade Federal da Bahia - UFBA
Av. Adhemar de Barros, S/N, Prédio do CPD, Campus de Ondina
Salvador-BA, CEP 40.170-110

lportoba@ufba.br

Resumo. *Sistemas baseados em eventos fornecem abstrações específicas que facilitam o desenvolvimento e a manutenção de políticas em diversos domínios. Embora as políticas de escalonamento de processos sejam geralmente descritas a partir de eventos do sistema (e.g., bloqueio e desbloqueio de processos), a implementação de escalonadores reais não explora tal modelo. Este artigo apresenta uma arquitetura baseada em eventos voltada ao desenvolvimento de escalonadores de processos e sua implementação no núcleo do Linux.*

Abstract. *Event-based systems provide domain-specific abstractions that help the development and maintenance of policies in several domains. Although process scheduling policies are generally modelled in terms of system events (e.g., process blocking and unblocking), their implementation does not rely on such a model. This paper presents an event-based architecture for the development of process scheduling policies and its implementation in the Linux kernel.*

1 Introdução

O desenvolvimento de um escalonador de processos é uma tarefa laboriosa e geralmente restrita a especialistas em sistemas operacionais, pois é necessário um conhecimento preciso da estrutura e do funcionamento do núcleo. Por razões de desempenho, o desenvolvimento de um escalonador requer a escrita de código otimizado e de baixo nível, geralmente em C ou Assembly. Mesmo nos sistemas baseados na arquitetura de micro-núcleo, a exemplo de Chorus [14] e QNX [7] e nos sistemas operacionais ditos *extensíveis* como SPIN [4], Vino [15] e Exokernel [9], é difícil modificar ou integrar uma nova política de escalonamento. De fato, a validação de grande parte dos trabalhos relacionados ao escalonamento é freqüentemente efetuada através da simulação das políticas ao invés da análise de sua implementação em sistemas reais.

Os escalonadores de processos são geralmente descritos de modo informal e incompleto na literatura. Tradicionalmente, seu funcionamento é apresentado como um autômato de estados finitos na forma de um grafo orientado, no qual os arcos representam o conjunto de eventos do sistema operacional que disparam a transição entre cada estado. A noção de estado é empregada para reagrupar processos com mesmo status de execução (*e.g.*, pronto, bloqueado, em execução); geralmente implementada através de filas. Entretanto, as implementações de escalonadores não refletem este modelo de eventos. Por exemplo, as filas de processos são por vezes utilizadas para armazenar processos em diferentes estados de execução para reduzir o custo associado à gestão dessas filas. O escalonador Linux, por exemplo, utiliza a fila de processos prontos também para armazenar processos que excederam seu *quantum* de execução e o processo corrente. Essas práticas de implementação acarretam diferenças importantes entre a documentação existente e os escalonadores reais empregados pelos sistemas operacionais.

Sistemas baseados em eventos têm sido intensamente explorados para a concepção de ambientes de desenvolvimento e sistemas de execução dedicados [4, 11]. O grau de abstração fornecido por eventos torna a programação mais clara e intuitiva, o que facilita a compreensão dos programas e sua respectiva depuração e manutenção. Apesar dessas vantagens e da estreita correlação entre eventos e escalonamento de processos, sistemas baseados em eventos foram timidamente utilizados nesse contexto.

Esse contexto motivou a concepção de Bossa [2] - um framework baseado em eventos voltado ao desenvolvimento de políticas de escalonamento. Este framework é composto de uma linguagem de programação dedicada que fornece abstrações de alto nível para a construção de escalonadores (*e.g.*, filas de processos, critério de escalonamento, atributos de processos), e uma arquitetura de eventos que permite a integração de programas escritos nessa linguagem diretamente no núcleo do sistema operacional. O framework Bossa foi utilizado na implementação de políticas de escalonamento tradicionais como *Rate Monotonic*, *Earliest Deadline First* [12] e Linux; e variantes baseadas na noção de progresso empregadas no contexto de aplicações multimídia [1]. A eficiência dos escalonadores implementados em Bossa foi comprovada através de programas de benchmark específicos ao escalonamento de processos e pela análise da execução de aplicações reais. Até o presente momento, desconhecemos outras abordagens concretas da aplicação de eventos para a construção de escalonadores.

Neste artigo descrevemos a arquitetura de eventos de Bossa e sua implementação no núcleo do sistema operacional Linux, sem apresentar aspectos da linguagem Bossa. A seção 2 descreve os componentes básicos do modelo de eventos utilizado em Bossa e detalhes de seu desenvolvimento e integração no Linux são discutidos na seção 3. A seção 4 descreve brevemente trabalhos correlatos na área e a seção 5 conclui o artigo apresentando perspectivas futuras.

2 Escalonamento baseado em eventos

A arquitetura de Bossa é composta por três componentes : uma versão modificada do núcleo do sistema operacional, uma política de escalonamento e um suporte de execução; descritos a seguir.

- **Núcleo Bossa.** O núcleo utilizado por Bossa possui características adicionais para facilitar a integração de políticas de escalonamento. Mais especificamente, esse núcleo é enriquecido com notificações de evento associadas ao escalonamento de processos e mecanismos que permitem configurar e instalar uma nova política de escalonamento.

- **Política de escalonamento.** O comportamento do escalonador é definido por uma política de escalonamento. Esta política define as ações que são efetuadas pelo escalonador quando da ocorrência de eventos do núcleo. Uma política de escalonamento Bossa é implementada a partir de um conjunto de abstrações específicas tais como as filas de espera e variáveis para armazenagem de processos, e tratadores de evento.

- **Suporte de execução.** Bossa define um suporte de execução que serve de interface entre o núcleo e a política de escalonamento. Esse suporte de execução separa a política de escalonamento das especificidades do núcleo (*e.g.*, controladores de dispositivos, chamadas de sistema). Além disso, ele é responsável pela utilização de mecanismos de baixo nível específicos ao sistema, a exemplo da comutação de processos.

A figura 1 apresenta o funcionamento de Bossa ilustrando o fluxo de informações entre o núcleo, a política de escalonamento e o suporte de execução. Os pontos de escalonamento no núcleo são redefinidos como notificações de evento enviadas ao suporte de execução Bossa (1). Na recepção de um evento de escalonamento, o suporte de execução repassa o evento à política de escalonamento (2), que por sua vez invoca o tratador de evento correspondente e retorna um valor indicando o novo estado do escalonador (3). O suporte de execução utiliza a informação de estado de um escalonador para verificar se ele deve efetuar uma troca de processo. Se o estado

do escalonador indicar que o escalonador deve continuar a execução do processo corrente, não há ação específica a ser realizada pelo suporte de execução. Se o escalonador indica que o processo corrente deve ser suspenso e que um processo pronto está disponível, o suporte de execução invoca novamente a política de escalonamento para eleger um novo processo (4,5). Por fim, se o estado do escalonador indica que o processo corrente deve ser suspenso e que não há processos prontos, o suporte de execução despacha o processo *idle* do núcleo após ter interrompido o processo corrente.

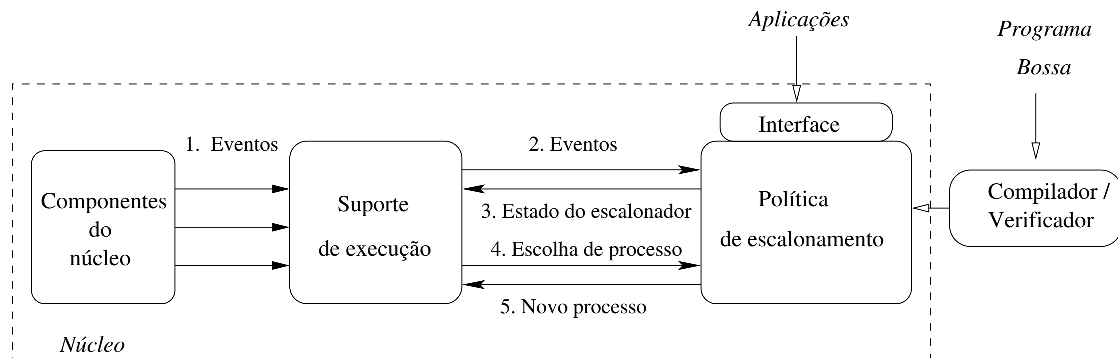

Figura 1: Visão global do framework Bossa.

Nas próximas seções, detalharemos a implementação do núcleo Bossa, de uma política de escalonamento e do suporte de execução. Inicialmente, apresentaremos uma visão global de seu funcionamento e, em seguida, descreveremos sua implementação no Linux.

2.1 O núcleo Bossa

Os componentes do núcleo Bossa efetuam notificações de eventos referentes ao escalonamento de processos para o suporte de execução. Para tanto, a arquitetura Bossa disponibiliza eventos específicos para sinalizar a criação (`process.new`) e o término (`process.end`) de processos, tiques de relógio (`system.clocktick`), escolha de um novo processo (`bossa.schedule`), bloqueio (`block`) e desbloqueio (`unblock`) de processos.

Os eventos de Bossa visam fornecer uma representação de alto nível dos mecanismos do núcleo onde esses eventos são gerados para facilitar o desenvolvimento de escalonadores. Por exemplo, a criação de processos no Linux pode ser realizada pelas chamadas de sistema `fork()` e `clone()`. Em Bossa, a funcionalidade de criação de processos corresponde ao evento `process.new.fork`. Da mesma forma, os diversos mecanismos que provocam o bloqueio de um processo correspondem ao evento `block`. Assim, o programador de uma política que deseja especificar um tratamento para esses eventos não precisa conhecer os detalhes de implementação do núcleo.

Embora a noção de eventos mascare a diversidade dos mecanismos do núcleo associados ao escalonamento, certas políticas precisam associar um comportamento a eventos específicos. Por exemplo, quando do bloqueio de um processo, a política de escalonamento do Windows NT recalcula a prioridade do processo segundo o tipo de recurso por ele solicitado. Em Bossa, isto requer que o desenvolvedor possa associar tratamentos específicos para cada um desses eventos de bloqueio. Para lidar com essas situações, organizamos alguns eventos de Bossa em uma hierarquia (denotada pelo símbolo `*`). Dessa forma, o desenvolvedor pode tratar o caso genérico de bloqueio de processo (`block.*`), bem como lidar com uma situação particular, por exemplo, assim que o processo é colocado em espera por um recurso de rede (`block.net.*`).

Notificações de evento

Uma notificação de evento (`e`) gerada pelo núcleo Bossa corresponde a um tipo de dado que contém informações sobre a classe do evento (*e.g.*, bloqueio, desbloqueio), sobre o processo gerador do evento (chamado de processo *gerador*) e sobre o processo para o qual o evento é destinado

(chamado processo *destino*). Neste artigo, utilizamos a notação `e.target` e `e.source` para referenciar os processos gerador e destino do evento `e`, respectivamente.

É importante notar que os valores de `e.target` e `e.source` variam em função do tipo do evento. Consideremos dois exemplos. Na notificação da passagem de um tique de relógio (`system.clocktick`), `e.target` é o processo corrente, para o qual o evento é destinado, ao passo que a variável `e.source` é nula pois tal evento não é gerado por um processo. No evento de criação de processo (`process.new.fork`), o valor de `e.source` contém a identificação do processo pai, gerador desse evento, e o valor de `e.target` contém a identificação do novo processo.

2.2 Política de escalonamento

Uma política de escalonamento Bossa especifica o comportamento do escalonador quando da recepção de um evento. A figura 2 apresenta o pseudo-código de um escalonador Bossa. Esse escalonador define inicialmente os atributos associados a cada processo (*e.g.*, prioridade, deadline), as filas de espera e variáveis utilizadas para armazená-los. A parte principal do escalonador encontra-se no código dos tratadores de evento (agrupados na função `handle_event`). Para tratar um evento, o escalonador deve identificar o evento recebido para, em seguida, disparar a execução do tratador correspondente. Como ilustra a figura 2, todos os eventos retornam o novo estado do escalonador (discutido a seguir).

```
// Declarações : atributos, filas de espera
int handle_event(event) {
// Código dos tratadores de evento e cálculo do estado do escalonador
 switch (event) {
  case system.clocktick : {...}
  case unblock.*        : {...}
  case block.*          : {...}
  ...
 }
 return scheduler_state;
}
// funções auxiliares
```

Figura 2: Pseudo-código de um escalonador Bossa.

Estado de um escalonador

Um escalonador pode ser considerado como um processo virtual que divide o tempo do processador entre seus processos. Assim como um processo convencional, um escalonador Bossa possui um *estado* : *Scheduler_Running*, *Scheduler_Blocked* ou *Scheduler_Ready*. Um escalonador está no estado *Scheduler_Running* quando um de seus processos está em execução. Um escalonador está no estado *Scheduler_Blocked* quando nenhum processo está apto para ser executado. Enfim, um escalonador está no estado *Scheduler_Ready* assim que um dos processos está pronto para ser executado.

Tratadores de evento

Um tratador de evento define as ações realizadas pelo escalonador quando da recepção de um evento proveniente do suporte de execução. Por exemplo, em uma política de escalonamento de tempo compartilhado, o tratador do evento `system.clocktick` deve normalmente verificar se o processo em execução excedeu seu *quantum* para saber se é necessário interrompê-lo. Da mesma forma, o tratador de um evento de bloqueio de processo deve colocar o processo corrente em uma fila de processos bloqueados. Ao fim de cada tratador de evento, a política deve calcular o novo estado do escalonador para reenviá-lo ao suporte de execução.

2.3 Suporte de execução

O suporte de execução serve de intermediário entre o núcleo e a política de escalonamento. Ele também é responsável pela mudança de contexto entre processos e utiliza os mecanismos do

núcleo destinados a este propósito. O suporte de execução mantém certas informações tais como o processo corrente, o último processo a ser executado, e o escalonador instalado no sistema.

É importante frisar que o suporte de execução desconhece os atributos dos processos ou as filas de espera utilizadas pela política de escalonamento. De fato, ele tem acesso somente aos atributos de sistema associados ao processo (*e.g.*, pid, gid, criador do processo). Mesmo ignorando os detalhes de implementação da política de escalonamento, o suporte de execução pode descobrir se esta política deve escolher um novo processo através da análise do valor do estado do escalonador. Esta independência entre o suporte de execução e a política de escalonamento permite que o escalonador seja facilmente substituível ou testado fora do núcleo.

3 Implementação no Linux

Na seção anterior apresentamos o modelo conceitual de uma arquitetura para o desenvolvimento de escalonadores de processo modulares e baseados em eventos. Nesta seção, apresentamos a implementação dessa arquitetura no núcleo do Linux (versões 2.2.16 e 2.4.18) através da descrição das modificações efetuadas nesse núcleo, a implementação do suporte de execução e trechos ilustrativos de uma política de escalonamento. Além disso, mostramos os problemas e soluções empregadas para compatibilizar o modelo de eventos de Bossa com as especificidades de funcionamento do núcleo do Linux.

3.1 O núcleo Bossa/Linux

A introdução das notificações de eventos demandou a modificação de diversas partes do núcleo Linux tais como os controladores de dispositivos, os sistemas de arquivos, a camada de rede, as chamadas de sistema e alguns processos do núcleo (*i.e.*, os *daemons*). A título de ilustração, a tabela 1 apresenta o número de notificações de eventos introduzidas em cada diretório da raiz da distribuição do núcleo na implementação da versão 2.4.18 do núcleo Bossa/Linux.

Diretório	Sub-sistemas	Eventos	Arquivos
`drivers`	Controladores de dispositivos	126	33
`net`	Rede	44	16
`fs`	Sistema de arquivos	54	19
`kernel`	Sincronização, interrupções, *timers*, etc.	24	8
`mm`	Gerenciamento de memória	24	7
`arch`	*diversos*	14	4
`include`	*diversos*	11	6
`lib`	*diversos*	9	2
`ipc`	Comunicação entre processos	6	2
`init`	Inicialização do sistema	1	1

Tabela 1: Número de notificações de evento por diretório do núcleo Bossa/Linux.

Um evento Bossa é inserido em cada local do núcleo onde um componente do sistema efetua uma chamada ao escalonador. No Linux, esse ponto de escalonamento caracteriza-se por uma chamada à função `schedule()` ou de modo indireto através de uma chamada à função `schedule_timeout()`. Tais chamadas são geralmente seguidas ou precedidas de uma mudança no estado do processo. No núcleo do Linux, a mudança de estado de um processo consiste em modificar o campo `state` da estrutura `task_struct` associado ao processo. De fato, a mudança de estado pode ser realizada de diversas maneiras segundo a vontade do programador e pode ainda variar de uma versão do núcleo a outra.

Para introduzir as notificações de evento, foi preciso identificar todas as partes do núcleo onde havia uma chamada direta ou indireta à função `schedule()`. Em seguida, para caracterizar a natureza do evento (*e.g.*, bloqueio, desbloqueio), foi necessário levar em conta o contexto da chamada ao escalonador. Para tanto, analisamos se havia mudança no estado do processo (do ponto de vista do Linux) antes ou após a chamada à função `schedule()`. No Linux, o estado de

um processo pode assumir os valores TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE e TASK_STOPPED indicando que o processo corrente deve ser bloqueado. Assim, uma chamada à função schedule() precedida de tais mudanças de estado caracteriza um evento de bloqueio em Bossa. A figura 3 ilustra esta situação apresentando a notificação do evento de bloqueio SIGNAL_PFTRACED (realizada pela macro BOSSA_BLOCK). Esse evento é produzido assim que um sinal ptrace é enviado para interromper a execução de um processo.

```
int do_signal(struct pt_regs *regs, sigset_t *oldset) { // arch/i386/kernel/signal.c
  ...
  current->state = TASK_STOPPED;
  notify_parent(current, SIGCHLD);
  BOSSA_BLOCK(SIGNAL_PFTRACED,current); // notificação de evento
  schedule();
}
```

Figura 3: Notificação de um evento de bloqueio quando do envio do sinal ptrace.

Assim que o evento deve ser enviado ao escalonador, o núcleo utiliza a função bossa_notify_event, apresentada na figura 4. Os argumentos desta função são o nome do evento, o processo destino e gerador do evento. Esta função agrupa as informações associadas ao evento e chama o ponto de entrada do escalonador (função handle_event) para disparar a execução do tratador de evento correspondente.

```
inline int bossa_notify_event // kernel/bossa.c
 (int event_type, struct bossa_struct * target, struct bossa_struct * source){
  e.type =  event_type; e.source = source; e.target = target;
  ...
    // send event to the scheduler
  root_s_state = root_scheduler.ops-> handle_event(&e,x,e.target->next_list.next);
}
```

Figura 4: Notificação de evento Bossa em Linux.

3.2 Suporte de execução

O suporte de execução foi concebido para permitir a integração de escalonadores ao núcleo de maneira simples e modular. Para isso, substituímos o código da função schedule() do Linux, tornando-a o ponto de entrada do suporte de execução Bossa. O suporte de execução mantém ainda variáveis que armazenam o processo corrente (running), o último processo a ser executado (old_running), e o escalonador instalado no núcleo (root_scheduler).

A figura 5 apresenta um trecho da parte principal do suporte de execução Bossa. Inicialmente, utilizamos um *lock* do núcleo (linha 3) que bloqueia as interrupções para poder manipular as variáveis do suporte de execução e invocar a política de escalonamento de modo seguro (*i.e.*, sem a interferência de outros componentes do núcleo). Em seguida, notificamos o evento ao escalonador (linha 5) se a variável running->deferred não estiver vazia e o processo corrente não for o processo *idle* (rts_idle_process). A variável running->deferred contém a identificação do último evento notificado pelo núcleo. Esta variável pode estar vazia caso a notificação do evento armazenada nesta variável tenha sido anulada por outro evento antes da chamada ao suporte de execução. Por exemplo, um evento de bloqueio seguido do evento desbloqueio correspondente não gera alguma notificação ao escalonador.

O suporte de execução verifica em seguida o estado do escalonador conforme apresentado na seção 2.2. Se o escalonador encontra-se no estado *Scheduler_Ready* (linha 9), um novo processo deve ser escolhido. Para isso, o suporte de execução invoca a função do_schedule do escalonador (linha 10), que implementa o tratador do evento bossa.schedule. Um escalonador no estado *Scheduler_Blocked* indica que o mesmo está inativo e, em conseqüência, o suporte de execução despacha o processo *idle* sem a intervenção da política de escalonamento (linha 13). Em seguida, o suporte de execução verifica se o processo escolhido é diferente do processo corrente. Neste caso, o suporte de execução chama a função switch_to() (linha 20) do núcleo Linux

```
 (1) asmlinkage void schedule(void) { // kernel/sched.c
  ...
 (2) need_resched_back:
  ...
 (3)   spin_lock_irq(&bossa_scheduler_lock); // get global kernel lock
 (4)   if (running != &rts_idle_process && running->deferred) {
 (5)     bossa_notify_event(running->deferred,running,NULL); // notify event
 (6)     running->deferred = NULL;
 (7)   }
 (8)   old_running = running;
 (9)   if (root_s_state == SCHEDULER_READY) { // a new process must be elected
(10)     root_s_state = root_scheduler.ops-> do_schedule();
(11)   }
(12)   if (root_s_state == SCHEDULER_BLOCKED)
(13)     running = &rts_idle_process; // dispatch idle process
(14)   spin_unlock_irq(&bossa_scheduler_lock); // release kernel lock
(15)   if (running == old_running) // no process change
(16)     goto outbossa;
(17)   ... // prepare for context switch
(18)   prev = old_running->attr;
(19)   next = running->attr;
(20)   switch_to(prev, next, prev); // context switch
(21) outbossa:
(22)   if (root_s_state == SCHEDULER_READY) {// do it again if scheduler state has changed
(23)      goto need_resched_back;
(24)   }
(25)   return;
}
```

Figura 5: Trecho do suporte de execução Bossa em Linux.

que efetua a mudança de contexto. Se o processo escolhido pelo escalonador é aquele que se encontra em execução, nenhuma mudança de contexto é realizada. Por fim, verificamos se houve um mudança de estado do escalonador para o estado *Scheduler_Ready* (linha 22). Tal mudança de estado pode se produzir caso as interrupções, que podem gerar eventos, estiverem habilitadas. Neste caso, retorna-se ao início do tratamento do suporte de execução.

3.3 Política de escalonamento

A implementação de uma política de escalonamento consiste na definição de atributos de processo, filas de espera e variáveis, código dos tratadores de evento (incluindo o cálculo do estado do escalonador), e funções auxiliares. É importante frisar que os trechos de código descritos nessa seção são automaticamente gerados pelo compilador Bossa (figura 1). Portanto, o programador não deve se preocupar com detalhes de implementação de baixo nível (ver [3] para informações adicionais sobre a linguagem Bossa).

Atributos de processo

O método mais utilizado para adicionar atributos ao contexto de execução do núcleo Linux consiste em incluir novos campos da política diretamente na estrutura `task_struct`. Em Bossa, os atributos de processo são introduzidos na estrutura `bossa_struct`. O campo `attr` dessa estrutura permite à política o acesso aos atributos de processo do Linux (definidos na estrutura `task_struct`). Adicionamos ainda no final da estrutura `task_struct` o atributo `bossa` (de tipo `bossa_struct`) que permite ao suporte de execução o acesso aos campos definidos por uma política Bossa.

Filas de espera e variáveis

Em Bossa, as filas de espera são implementadas como listas encadeadas contendo estruturas de tipo `bossa_struct`. Para facilitar a gestão das filas de espera e variáveis, Bossa fornece funções específicas para alocar e liberar blocos de memória que utilizam essas estruturas e funções para mover um processo de um estado para outro. Para efetuar uma mudança de estado, definimos um

conjunto de funções que recebem como parâmetros a fila de espera ou a variável de origem e a fila de espera ou a variável de destino. Por exemplo, a função `move_proc_queue` move o conteúdo de uma variável para uma fila de espera ao passo que a função `move_queue_proc` realiza o tratamento inverso.

Tratadores de evento

Esta seção descreve a implementação dos tratadores de evento em Bossa. A figura 6 apresenta o ponto de entrada de um escalonador Bossa (função `handle_event`) e o tratamento efetuado pelo evento de bloqueio. O tratador de evento `block.*` move o processo corrente (`running_0`) para a fila de espera de processos bloqueados (`blocked`) e, em seguida, calcula o novo estado do escalonador[1]. A função `empty_queue` retorna o valor *verdadeiro* caso a fila de espera esteja vazia e *falso* no caso contrário.

```
int handle_event(struct event_struct *e,...) {
  switch (event_mask0(e->type)) {
    case EVENT_BLOCK: {
      move_proc_queue(running_0, &blocked);
      if (empty_queue(&ready)) { // no ready process
        return SCHEDULER_BLOCKED; }
      else { return SCHEDULER_READY; }
    }
    ...
}
```

Figura 6: Ponto de entrada de um escalonador Bossa.

A figura 7 apresenta a função `do_schedule` que implementa o tratador de evento `bossa.schedule`. Em primeiro lugar, a função `select()` (específica à política de escalonamento) é invocada para escolher um novo processo consultando a fila de processos prontos `ready`. O processo escolhido é então colocado na variável local `running_0` e na variável do núcleo `running` (importada pelo escalonador), que é utilizada pelo suporte de execução para efetuar uma mudança de contexto.

```
int do_schedule (void) {
    struct bossa_struct * new = select(&ready); // select a new process
    move_queue_proc(new, running_0); // update Bossa running process
    running = running_0; // update run-time system running process
  }
  return SCHEDULER_RUNNING; // return scheduler state, always running in this case
}
```

Figura 7: Seleção e despacho de um processo em Bossa.

3.4 Adaptação da implementação do modelo de eventos para o Linux

Os testes de execução das políticas de escalonamento Bossa mostraram que algumas das hipóteses relativas aos eventos estavam incorretas. Por exemplo, no tratador de evento de bloqueio (`block`), o processo a ser bloqueado (`e.target`) estava por vezes na fila de processos prontos e não na variável que armazena o processo corrente (`running`). Isto provocava erros de execução assim que o tratador tentava colocar esse processo na fila de processos bloqueados. Tais erros de execução eram decorrentes da interferência entre as execuções dos tratadores de evento (*i.e.*, uma seqüência de tratamentos de eventos antes da intervenção do suporte de execução). No evento de bloqueio apresentado, isto ocorre, por exemplo, se a execução precedente do tratador de tique de relógio (`system.clocktick`) colocar o processo corrente (`e.target`) na fila de processos prontos quando do término de seu *quantum*.

Esse tipo de interferência entre a execução dos tratadores de evento pode ocorrer porque alguns eventos de Bossa (*e.g.*, tique de relógio e o desbloqueio de um processo) são disparados

[1] Note que esse tratador de evento nunca deve retornar o estado *Scheduler_Running*.

a partir da execução dos *bottom halves* do núcleo. No Linux, como em diversos sistemas operacionais, o tratamento de interrupções de hardware é efetuado inicialmente por um *top half*, que é executado imediatamente e, em seguida, por um *bottom half*, cuja execução é postergada. Após certo tempo, o núcleo dispara a execução de todos os *bottom halves* que foram postergados. Assim, um escalonador Bossa pode tratar uma seqüência de eventos gerados pelos *bottom halves* antes que o estado do escalonador seja considerado para atualizar o valor do processo em execução utilizado por Bossa. Para eliminar esses erros na implementação de um escalonador, foi necessário introduzir código adicional na política para tratar essas situações. Esse código adicional verifica o estado atual do processo antes de movê-lo para uma fila de espera ou uma variável. A utilização da linguagem Bossa dispensa a escrita desse código, já que esse tratamento é efetuado automaticamente pelo compilador.

4 Trabalhos correlatos

Escalonadores de processos são geralmente implementados diretamente no núcleo do sistema operacional. Existem algumas propostas voltadas à implementação de infraestruturas de escalonamento, mas estas não exploram a expressividade do modelo de eventos associado ao escalonamento de processos. Ford e Susarla [6] propuseram uma plataforma voltada ao desenvolvimento de escalonadores hierárquicos nos quais um proocesso se comporta como um escalonador através da doação de seu tempo de execução a outros processos. Vassal [5] é outro exemplo de infraestrutura que permite a carga dinâmica de escalonadores no núcleo do Windows NT. Regehr *et al.* [13] propuseram um modelo baseado em uma biblioteca específica e um protocolo que define a execução de um escalonador. O programador utiliza esses dois componentes para escrever uma política de escalonamento.

Algumas propostas utilizam a abstração de eventos para o desenvolvimento de sistemas operacionais. SPIN [4] permite a especialização de componentes do núcleo através da implementação de tratadores de eventos escritos em Modula-3. Entretanto, em SPIN não é possível utilizar eventos para substituir o escalonador do núcleo. TinyOS [8] é um sistema operacional voltado para sistemas embarcados que utiliza eventos para a comunicação entre tarefas e os associa a equipamentos tais como sensores e transdutores. A linguagem HIPEC [11] permite a implementação de políticas de substituição de página através da associação de tratadores a eventos específicos, como falta de página. Essas propostas demonstram a viabilidade da utilização de eventos em sistemas operacionais, mas não abordam sua aplicação na concepção de escalonadores de processos.

5 Conclusões

Neste artigo apresentamos uma arquitetura baseada em eventos para a implementação de escalonadores de processos e sua integração no núcleo do Linux. Essa arquitetura permite a escrita de políticas de escalonamento modulares e independentes dos mecanismos do núcleo a partir de eventos que capturam conceitos do domínio. Este modelo balisou a definição das abstrações fornecidas pela linguagem Bossa, cujo compilador automatiza a geração do código C referente às políticas de escalonamento e proporciona a verificação de escalonadores [10].

A independência entre uma política Bossa e o núcleo aprimora a portabilidade dos escalonadores e viabiliza sua execução e teste fora do núcleo (*e.g.*, através de simulações). A noção de estado, associado a um escalonador, propicia a construção de infraestruturas hierárquicas de escalonamento, nas quais escalonadores com características específicas podem ser empregados de modo a prover diferentes garantias às aplicações. Além disso, toda funcionalidade de um escalonador concentra-se em um único lugar, o que facilita a manutenção e compreensão do funcionamento da política de escalonamento.

A validação da genericidade do modelo de eventos requer a integração da arquitetura Bossa em outros sistemas operacionais. Estudos preliminares foram feitos em BSD e atualmente existem trabalhos voltados ao Windows NT e variantes do Linux dedicadas aos sistemas embarcados.

Disponibilidade

A distribuição do núcleo Bossa/Linux, o compilador/verificador e exemplos de políticas de escalonamento encontram-se publicamente disponíveis em `http://www.emn.fr/x-info/bossa`

Referências

[1] L. P. Barreto. Atendendo aos requisitos de QoS de aplicações multimídia através da especialização de serviços do sistema operacional. In *IX Simpósio Brasileiro de Sistemas Multimídia e Web (Webmídia'2003)*, pages 21–34, November 2003.

[2] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming OS schedulers with domain-specific languages and aspects: New approaches for OS kernel engineering. In *Proceedings of the 1st Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–6, April 2002.

[3] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *Proceedings of the 10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, March 2002.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 267–284, December 1997.

[5] G. M. Candea and M. B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Berkeley, CA, August 1998.

[6] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 91–105, Berkeley, CA, USA, October 1996.

[7] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, USA, April 1992.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'2000)*, pages 93–104, October 2000.

[9] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 52–65, October 1997.

[10] J. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW'2002)*, pages 54–61, Saint-Emillion, France, September 2002.

[11] C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: High performance external virtual memory caching. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 153–164, Berkeley, CA, USA, November 1994.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[13] J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.

[14] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, April 1992.

[15] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 213–227, Berkeley, CA, USA, October 1996.