Obtendo Escalabilidade através de Programação Baseada em Eventos

Daniel de Angelis Cordeiro
danielc@ime.usp.br

Orientado por:
Alfredo Goldman e Dilma da Silva

22 de novembro de 2004

Servidores WEB

- É a classe de aplicativos mais estudada em artigos sobre escalabilidade:
  - além de ser interessante do ponto de vista teórico, possui importância econômica indiscutível;
  - atualmente é a classe de aplicativos que lida com o maior número de usuários simultâneos.
- O foco deste seminário serão servidores web que servem conteúdo estático. Entretanto, tudo que será apresentado pode ser utilizado em servidores com conteúdo dinâmico.

Etapas do processamento de uma requisição HTTP

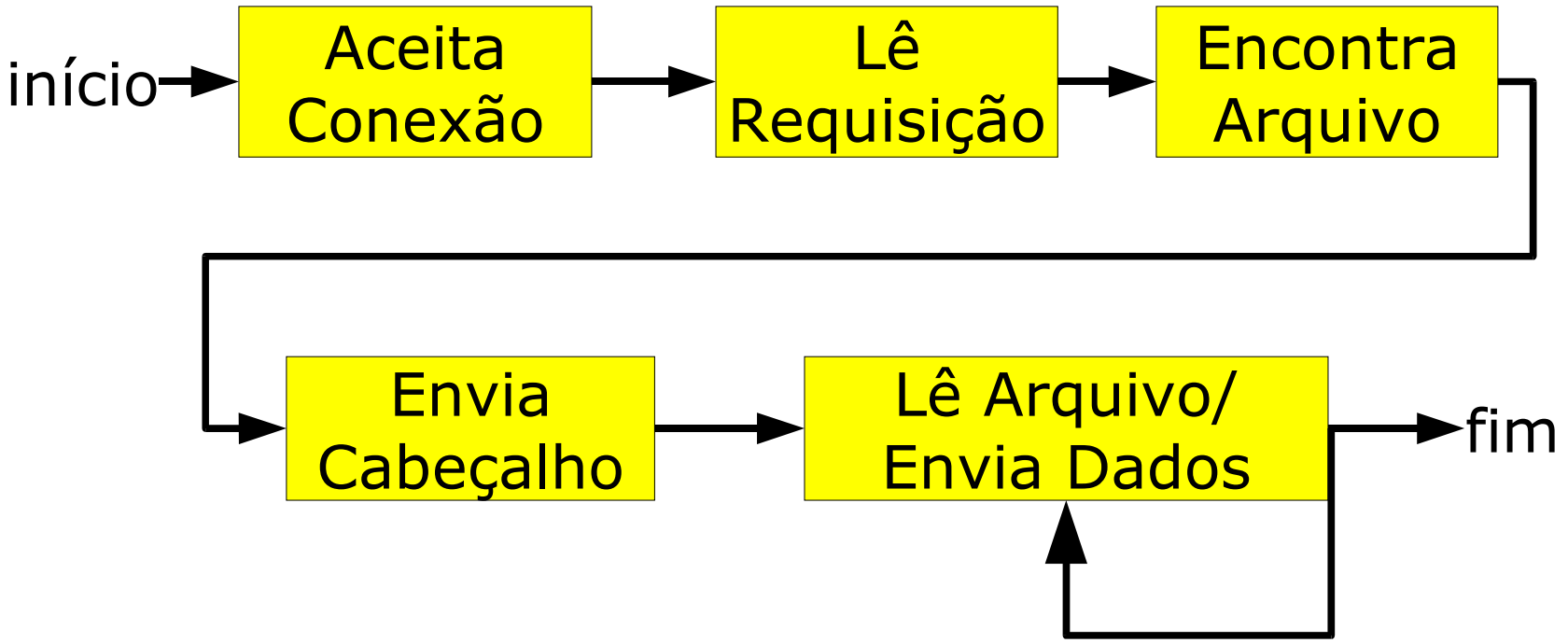
- Aceita conexão:
  - executa a chamada de sistema *accept* no *socket* que monitora a porta de entrada para aceitar a conexão do cliente.
- Lê requisição:
  - lê e processa o cabeçalho da requisição HTTP do *socket* do cliente.
- Encontra arquivo:
  - verifica se o arquivo requisitado existe, se o cliente possui permissão de acesso e lê o tamanho do arquivo e a data de modificação.

Etapas do processamento de uma requisição HTTP (cont.)

- Envia cabeçalho de resposta:
  - envia o cabeçalho HTTP de resposta pelo *socket* do cliente.
- Lê arquivo:
  - lê o conteúdo do arquivo (ou parte dele, no caso de arquivos muito grandes).
- Envia dados:
  - transmite o conteúdo requisitado (ou parte dele) pelo *socket* do cliente. Caso o arquivo seja muito grande, as duas últimas etapas são repetidas até que tudo seja enviado.

Etapas do processamento de uma requisição HTTP (cont.)

Esquematicamente:



Problemas de Escalabilidade

- Todas essas etapas podem, potencialmente, bloquear o processo:
  - operações que lêem ou aceitam conexões podem bloquear se os dados esperados ainda não chegaram do cliente;
  - operações que escrevem em *sockets* podem bloquear se os *buffers* de envio do TCP estiverem cheios (devido a grande utilização da rede);
  - o *stat()*, que valida a existência do arquivo e o *open()* podem ser bloqueados até que o acesso ao disco termine.

Problemas de Escalabilidade (cont.)

- Além disso:
  - a leitura de um arquivo, usando *read()*, e o acesso aos dados em uma região mapeada da memória podem bloquear enquanto os dados são lidos do disco.

- Por isso um servidor web escalável deve intercalar as etapas de forma que seja possível utilizar o processador enquanto operações de acesso ao disco ou à rede estejam executando.

Arquiteturas de Servidores WEB

- A arquitetura de sistema adotada define a estratégia utilizada para realizar a intercalação das etapas.
- Veremos as arquiteturas:
  - Multi-process (MP);
  - Multi-thread (MT);
  - Single-process event-driven (SPED);
  - Asymmetric multi-process event-driven (AMPED).

Arquitetura Multi-process

- Nesta arquitetura, todas as etapas são executadas, seqüencialmente, por um único processo.
- Quando um processo executa uma operação bloqueante, outro processo é automaticamente eleito pelo sistema operacional para usar a CPU.
- Como múltiplos processos podem ser utilizados (tipicamente algo entre 20 e 200 processos), requisições são atendidas simultaneamente.
- Não é necessário realizar sincronização entre processos. Entretanto é mais difícil realizar otimizações que dependam de memória global.

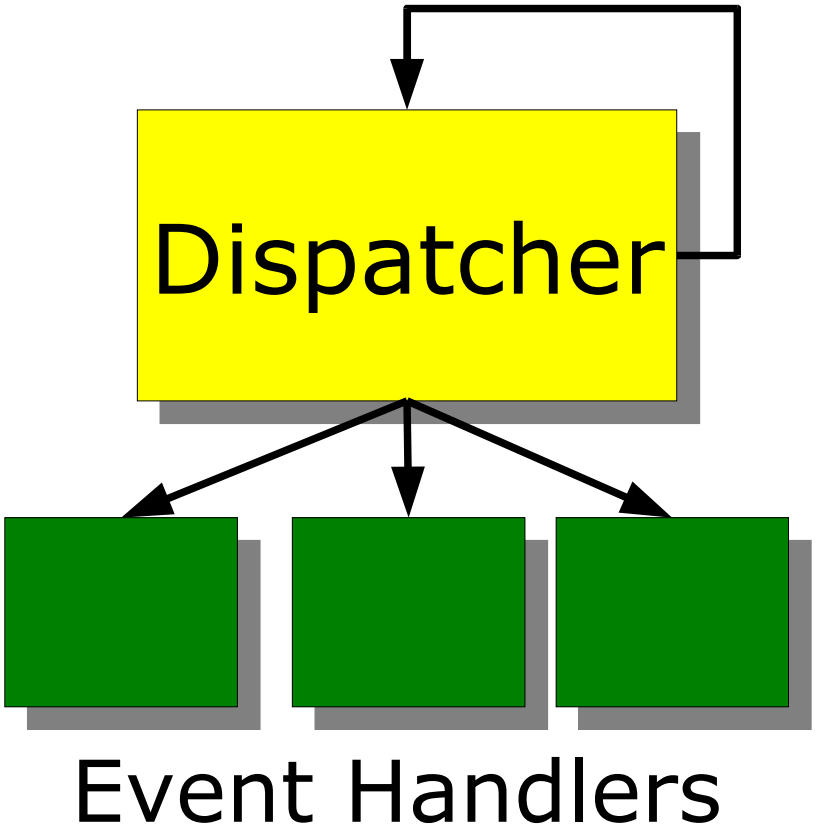
Arquitetura Multi-threaded

- A arquitetura MT emprega múltiplas *threads* independentes que utilizam um espaço de endereçamento compartilhado. Cada *thread* executa todas as etapas de uma requisição.
- A diferença em relação a arquitetura MP é que neste modelo é mais fácil realizar otimizações que necessitem de memória compartilhada (por exemplo, um cache de URLs válidas).
- Entretanto, é necessário utilizar controles de acesso a memória compartilhada e é imprescindível que o sistema operacional forneça suporte a *threads.*

Arquitetura Single-process event-driven

- Utiliza um único processo, implementado utilizando o paradigma de programação baseado em eventos.

  - O fluxo de execução é controlado pelos eventos ocorridos;
  - Possui uma única linha de execução (não há concorrência);
  - Não há preempção nos *handlers.*



Arquitetura SPED (cont.)

- O servidor utiliza operações bloqueantes para realizar operações de E/S assíncronas (utiliza as operações *select()* ou *poll()*).
- A arquitetura SPED pode ser vista como uma máquina de estados. Em cada iteração, o servidor realiza um *select()* para determinar se alguma operação de E/S foi concluída e, se foi concluída, gera um novo evento que será tratado pelo *dispatcher*.
- Não necessita de nenhuma forma de sincronização, mas não é adequado para ambientes multi-processados ou que possuam problemas com as operações assíncronas.

Arquitetura Asymmetric Multi-Process Event-Driven

- A arquitetura AMPED combina a abordagem baseada em eventos da arquitetura SPED com múltiplos processos (ou *threads*) ajudantes.
- Quando uma operação potencialmente bloqueante precisa ser realizada, o *dispatcher* utiliza um ajudante existente ou cria um novo. Assim que a operação termina, o ajudante notifica o *dispatcher* utilizando canal de comunicação inter-processos (ex: pipe).
- AMPED resolve o problema de operações assíncronas que bloqueiam e possui melhor desempenho em ambientes multi-processados.

Interferência do SO na Escalabilidade

- Os modelos apresentados anteriormente deixam claro a influência que o SO exerce sobre a escalabilidade de servidores web:
  - nas arquiteturas MP e MT o escalonamento dos processos e *threads* causam um custo extra no desempenho, além da sobrecarga devida à sincronia entre *threads* no MT;
  - na SPED os problemas com as chamadas assíncronas fazem com que o processador fique ocioso algumas vezes;
  - na AMPED o processador não fica ocioso, mas o problema do excesso de *threads* se repete.

Interferência do SO na Escalabilidade (cont.)

- Quando escalabilidade deve ser alcançada a qualquer custo, modificar o sistema operacional pode ser interessante. Mas, claro, há o compromisso entre dificuldade de programação e maior escalabilidade.

- A adaptação de sistemas operacionais para alcançar escalabilidade em algumas classes de aplicativos é um tópico importante de pesquisa hoje em dia.

Scheduler Activations (SA)

- *Thread* é a abstração de concorrência para a maior parte das aplicações paralelas.
- Entretanto, *kernel threads* são inerentemente mais lentas do que *user-level threads:*
  - embora o desempenho de *kernel threads* seja uma ordem de magnitude melhor do que o desempenho de um processo tradicional, é uma ordem de magnitude *pior* do que as bibliotecas de *threads* que rodam em nível de usuário.

Objetivos de SA

Criar uma interface para *kernel threads* e uma biblioteca que rode em nível de usuário para que, juntos:

- tenham o mesmo desempenho que a melhor biblioteca de gerenciamento de *threads* em nível de usuário no caso mais comum, quando as operações realizadas pela *thread* não necessitam de auxílio do sistema operacional, isto é, não executam nenhuma ação bloqueante.

Objetivos de SA (cont.)

- no caso em que é necessário intervenção do kernel, é necessário garantir que:
  - não haja processadores ociosos na presença de *threads* prontas para executar;
  - não exista *threads* de alta prioridade esperando por um processador enquanto uma *thread* de baixa prioridade estiver executando;
  - quando a *thread* for bloqueada (por exemplo, em uma *page-fault)* o processador que estava sendo utilizado por esta *thread* possa ser utilizado por outra *thread* do mesmo ou de outro processo;

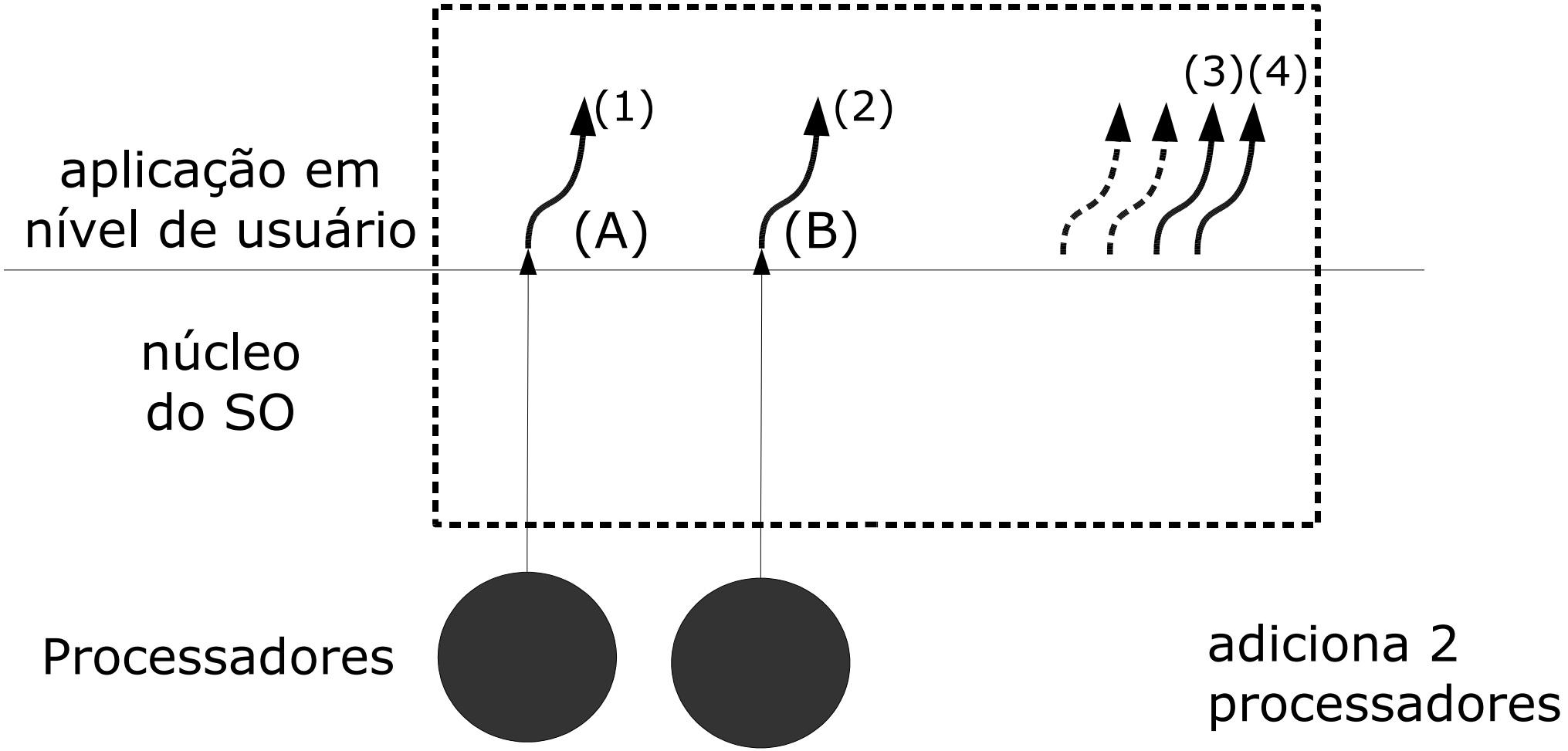
Abordagem de SA

- Cada aplicação possui um conjunto de processadores virtuais.
- Cada aplicação conhece quantos e quais processadores virtuais possui disponíveis e controla quais *threads* devem rodar em quais processadores.
- Ainda assim, o sistema operacional tem controle completo sobre quais processadores são disponibilizados para cada espaço de endereçamento de usuário. O SO pode, ainda, mudar o número de processadores disponíveis durante a execução do programa.

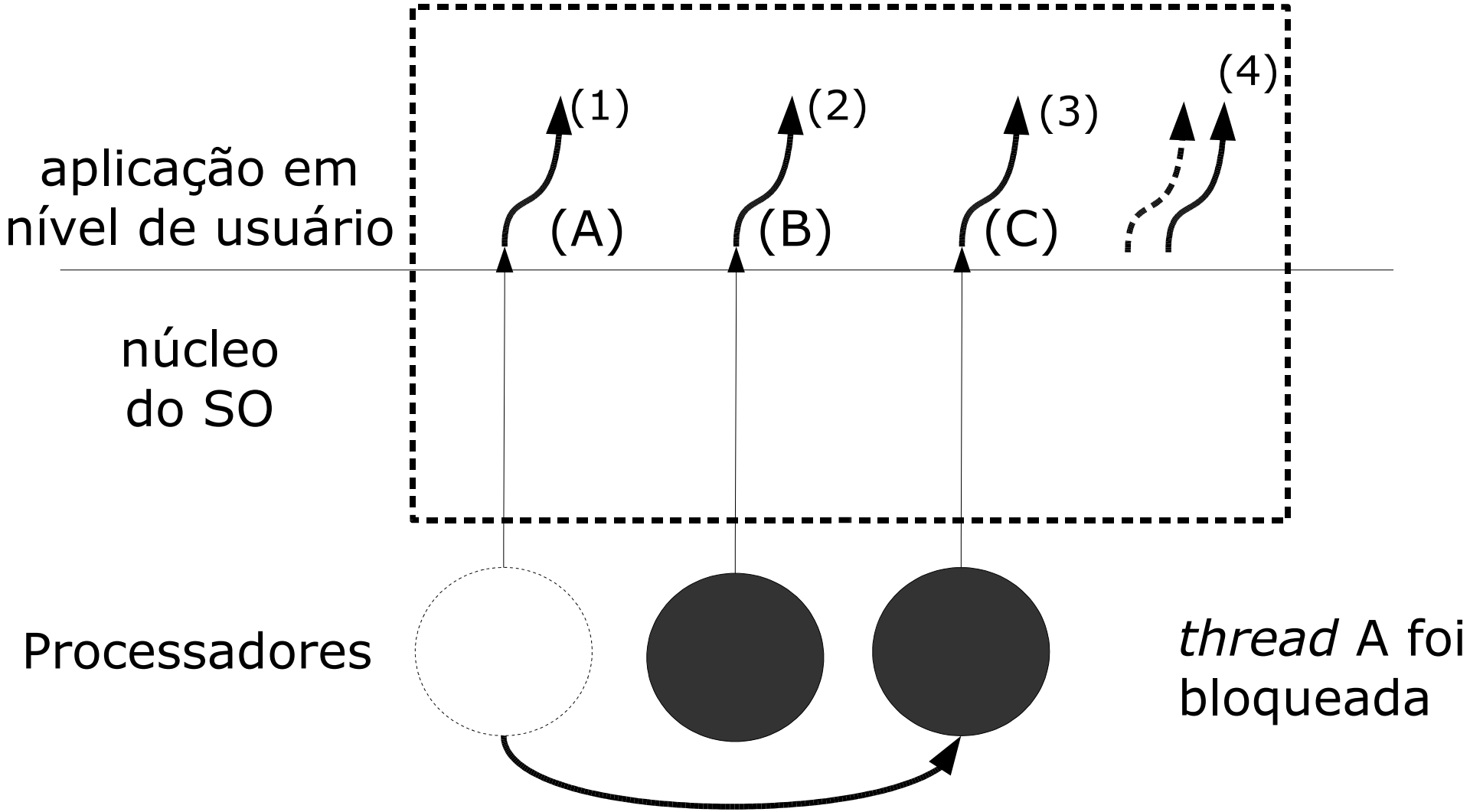
Abordagem de SA (cont.)

- O núcleo do SO notifica o escalonador no nível de usuário sobre todos os eventos que afetarão seu espaço de endereçamento, permitindo que a aplicação tenha completo conhecimento sobre o estado do escalonamento de suas *threads.*
- O mecanismo implementado no núcleo para que tudo isso seja possível é denominado de *scheduler activations.*

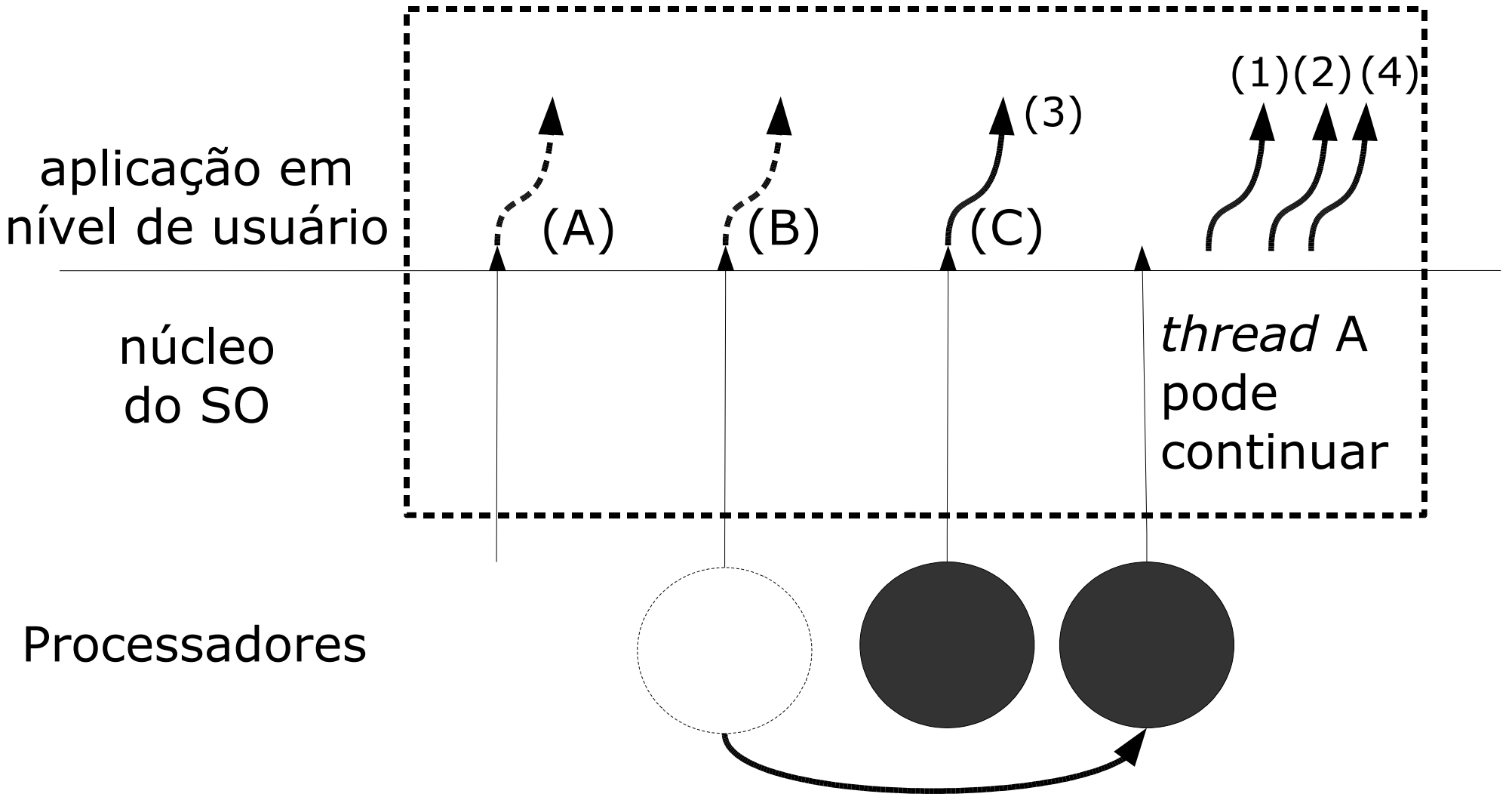
Um exemplo de SA



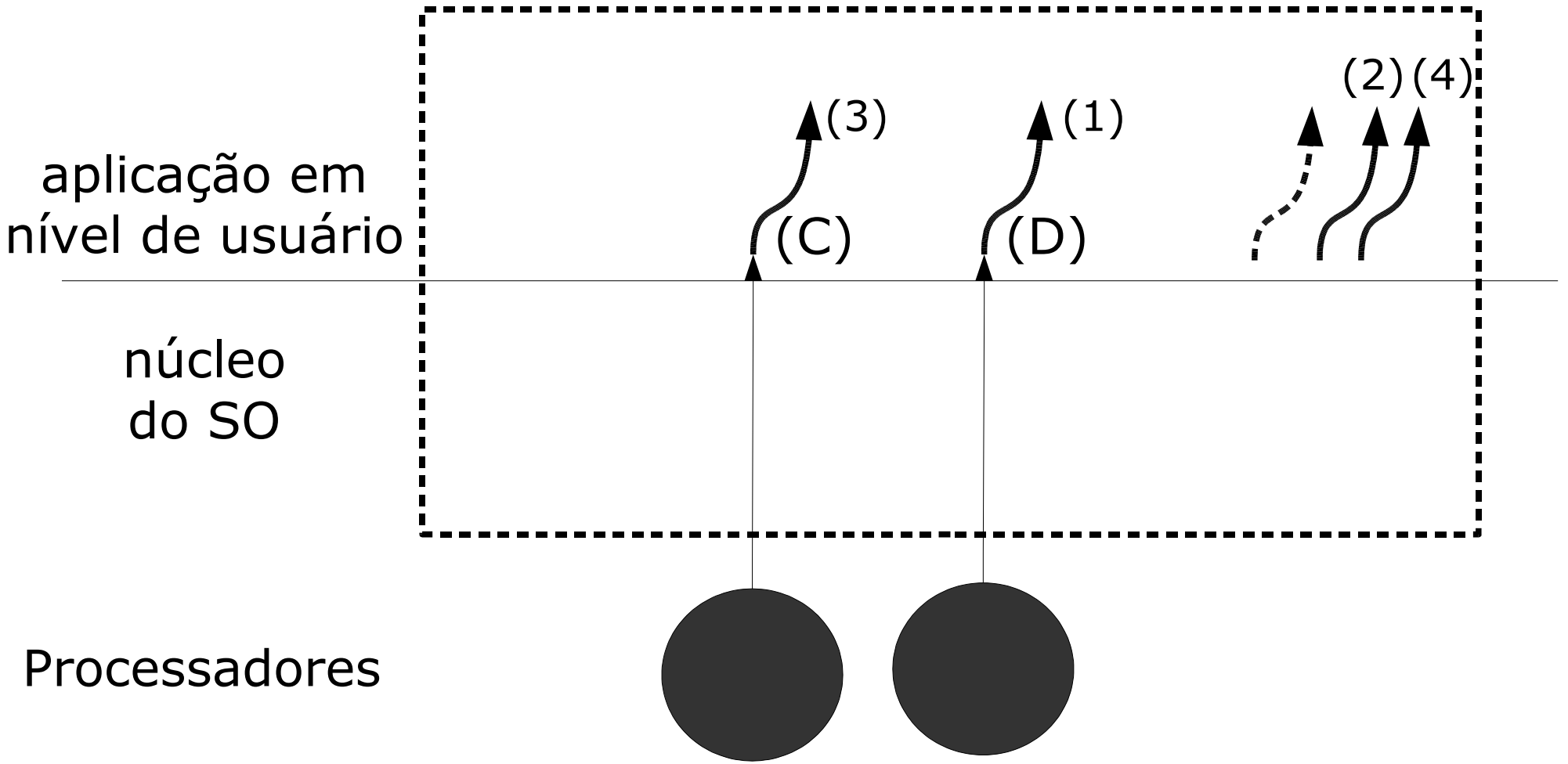
Um exemplo de SA



Um exemplo de SA



Um exemplo de SA



Seções Críticas

- Potencialmente uma *thread* pode ser bloqueada dentro de uma seção crítica.
- Duas abordagens podem ser feitas:
  - prevenção: preempções seriam evitadas através de um protocolo entre o núcleo e o nível de usuário – violaria a semântica de que o núcleo tem total controle sobre o número de processadores disponíveis;
  - recuperação: foi solução adotada; se a *thread* estiver com um *lock*, a *thread* é executada mais um pouco até que esta saia da seção crítica.

Conclusões

- Use programação baseada em eventos! Em ambientes mono-processados é uma alternativa interessante a *threads* que torna o sistema mais eficiente e mais fácil de depurar;
- Em geral, a arquitetura do sistema determina se ele será escalável ou não;
- Em ambientes onde a escalabilidade é o objetivo final, talvez seja necessário algum tipo de integração entre o programa em nível de usuário e o núcleo do sistema operacional.

Referências

Ousterhout, J. K. Why threads are a bad idea (for most purposes). Invited talk at 1996 USENIX Conference, Jan. 1996.

V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In Proceedings of the 1999 USENIX Technical Conference, Monterey, CA, June 1999.

Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. ACM Transactions on Computer Systems, 10(1):53--79, February 1992.