Universidade de São Paulo

Instituto de Matemática e Estatística

Departamento de Ciência da Computação

Flexibilidade e Escalabilidade em Sistemas Operacionais

MAC 5701

Tópicos em Ciência da Computação

Aluno: Livio Baldini Soares

Orientadora Principal: Dilma Menezes da Silva

Co-orientador: Arnaldo Mandel

Sumário

1 Introdução 1

2 Entidades de escalonmento 2

2.1 Artigo "Scheduler Activations: Effective Kernel Support fot the User-Level Management of Parallelism" 2

2.2 Artigo "Kernel-Scheduled Entities for FreeBSD" 5

3 Mecanismos de entrega de eventos 8

3.1 Artigo "A Scalable and explicit event delivery mechanism for UNIX" 8

3.2 Artigo "Kqueue: A generic and scalable event notification facility" 11

4 Escalonamento de disco 15

4.1 Artigo "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O" 15

5 Sistemas de arquivos 20

5.1 Artigo "HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions" 20

5.2 Artigo "File System Development with Stackable Layers" 25

5.3 Artigo "Improving the Performance of Log-Structured File Systems with Adaptive Methods" 29

5.4 Artigo "The Vesta Parallel File System" 30

5.5 Artigo "Scability in the XFS File System" 31

1 Introdução

Esse estudo serviu para uma melhor formação do aluno com relação à área escolhida como tema da dissertação de mestrado, **Sistemas Operacionais**, que versará sobre flexibilidade e escalabilidade em sistemas de arquivos. Além disso, o estudo auxiliou na delineação inicial da dissertação de mestrado.

Com o desenvolvimento de arquiteturas de computadores cada vez mais complexas, que disponibilizam múltiplos recursos ao usuário, percebe-se que existe uma carência de algoritmos e arquiteturas de sistemas operacionais que conseguem, de forma eficiente, gerenciar todos os recursos da máquina. Recursos como processadores, discos, rede, etc., são cada vez mais numerosos dentro de um mesmo computador, e para que as aplicações possam tirar proveito de todos os recursos é *essencial* que o sistema operacional saiba gerenciar os recursos de forma eficiente.

O uso de algoritmos e arquiteturas de sistemas operacionais tradicionais nesses sistemas complexos têm se mostrado não ideal, e muitas vezes, decepcionante. Dessa forma, existe uma grande preocupação na área de sistemas operacionais com a **escalabilidade** e **flexibilidade** de algoritmos e técnicas utilizadas para gerenciar os recursos disponíveis.

Durante o semestre foram lidos uma diversidade de artigos relativamente recentes, apresentados em algumas conferências de renome na área de sistemas operacionais. Os artigos escolhidos tratam, em sua maioria, de inovações que solucionam alguns problemas de **escalabilidade** e **flexibilidade**, ou novas arquiteturas, em que esses problemas são amenizados ou inexistentes.

Perto do final do semestre foi redigida uma proposta de projeto de pesquisa para o pedido de bolsa para a FAPESP. O título da proposta é **Incorporação de Flexibilidade ao Linux VFS**, que está intimamente ligado ao artigo apresentado na seção 5.1. Após a redação da proposta de projeto de pesquisa para a dissertação final tentamos focar nossos esforços no estudo de sistemas de arquivos flexíveis, e escaláveis.

2 Entidades de escalonmento

2.1 Artigo "Scheduler Activations: Effective Kernel Support fot the User-Level Management of Parallelism"

Esse artigo foi publicado na revista *ACM – Transactions on Computer Systems* (TOCS), Volume 10, Número 1, Fevereiro de 1992, páginas 53-79. Também foi apresentado na conferência, *ACM – Symposium on Operating System Principles* (SOSP), sendo selecionado como um *Award Paper*.

Tipo	Artigo em revista e conferência
Autores	T. E. Anderon, B. N. Bershad, E. D. Lazowska e H. M. Levy
Revista	ACM Transactions on Computer Systems
Conferência	ACM Symposium on Operating System Principles
Data	Fevereiro de 1992
Link	`http://www.cs.berkeley.edu/~brewer/cs262/Scheduler.pdf`

Resumo A eficiência de computação paralela depende muito do desempenho das primitivas usadas para controlar o paralelismo dentro do programa. Uma das maneiras mais eficientes e utilizadas para fazer o controle de programas altamente paralelos é o uso de *threads*. Uma *thread* separa a noção de execução seqüencial do programa de outros aspectos como espaço de endereçamento, descritores de entrada e saída (E/S), etc. Essa separação resulta em um ganho significativo no desempenho com relação a processos comuns, pois exige menos interferência do kernel (núcleo do sistema operacional) e custo adicional para realização de troca de contexto.

Existem, basicamente, duas abordagens para implementação e suporte a *threads*: em espaço de usuário e as kernel threads[1]. Nenhuma das duas abordagens têm dado resultados completamente satisfatórios.

Threads em espaço de usuário são gerenciadas por rotinas de bibliotecas linkadas em cada aplicação, portanto operações de gerenciamento das *threads* não requerem intervenção do kernel. Elas são executadas dentro do contexto de processos tradicionais, e portanto, geralmente são construídas sem modificações ao sistema operacional. O resultado pode ser um desempenho excelente; o custo de operações de *threads* em espaço de usuário é da mesma ordem de grandeza de chamadas a procedimentos. *Threads* em espaço de usuário também são flexíveis e facilmente customizáveis. Muitos sistemas paralelos, por exemplo, podem usar políticas simples para escalonamento de suas *threads*, ao invés de escalonamento preemptivo com prioridades utilizados na maior parte dos sistemas operacionais. Portanto o escalonador de *threads* em espaço de usuário pode ser customizado facilmente para esse tipo de aplicação.

Entretanto as *threads* em espaço de usuário apresentam sérios problemas com relação à integração com o restante do sistema. Toda vez que o sistema operacional remove uma *thread* de um processador (devido a chamada de E/S, *page fault*, preempção, etc.) todas as *threads* daquele

[1] Kernel threads são *threads* com suporte do kernel. Geralmente executam em cima de processos distintos que compartilham espaço de endereçamento.

processo são removidas também. Esse comportamento é muitas vezes errôneo já que podemos ter *threads* prontos para execução.

Além disso, o escalonador do kernel não tem como levar em conta as *threads*, já que estão todas dentro do mesmo processo. Isso degrada em muito o desempenho dessas *threads*, ou faz com que elas apresentem um comportamento errôneo, nessas situações em que o kernel precisa intervir de alguma forma.

Diversos sistemas operacionais possuem suporte dentro do kernel para múltiplas *threads* por espaço de endereçamento. O uso de "kernel threads"[2] resolve esse problema de integração com o sistema, já que o kernel tem ciência dessas *threads* (o que não acontecia no caso da implementação em espaço de usuário) e portanto sabe escalonar cada *thread* separadamente em um processador.

Infelizmente, kernel threads são muito pesadas (como processos comuns) para uso em sistemas paralelos. O desempenho de kernel threads embora tipicamente uma ordem de grandeza menor do que o uso de processos tradicionais, é tipicamente uma ordem de grandeza *maior* do que o desempenho do melhor caso das *threads* em espaço de usuário (casos em que intervenção do kernel não é necessário ou é bem pequena). Muitos fatores como chamadas ao sistema e escalonamento por prioridades, são os maiores contribuidores para o alto custo de processos comuns tipo UNIX. Isso implica que qualquer custo adicional adicionado pelo kernel, mesmo que muito pequeno, será significativo, e um sistema de gerenciamento de *threads* em espaço de usuário terá o desempenho significativamente melhor.

O resultado é que muitas vezes as *threads* em espaço de usuário têm sido implementadas em cima de kernel threads. Entretanto essa solução não resulta em desempenho diferente de implementá-las em cima de processos tradicionais, e sofrem dos mesmos problemas.

Com esses problemas em vista, este artigo descreve uma interface para o kernel e um pacote para *threads* em espaço de usuário, que juntos combinam a funcionalidade de kernel threads com o desempenho e flexibilidade de *threads* em espaço de usuário. Especificamente essa implementação tem como objetivos: no caso comum, quando não existe muita intervenção pelo kernel, ter desempenho tão bom quanto as *threads* em espaço de usuário; e no caso incomum, quando o kernel está envolvido, ter um comportamento igual a do sistema de gerenciamento de kernel threads.

A solução proposta é providenciar informações para o gerenciador de *threads* do espaço de usuário sobre quantos e quais processadores estão disponíveis, para que a escalonador em espaço de usuário possa decidir corretamente considerando os recursos reais disponíveis. Para isso o kernel notifica o escalonador em espaço de usuário todos os eventos afetando seu espaço de endereçamento, analogamente, o sistema em espaço de usuário notifica o kernel as operações que podem afetar decisões de alocação de processadores.

O mecanismo utilizado para realizar essas idéias é chamado de ***scheduler activations***. Uma *scheduler activation* programa o controle de execução do kernel para o escalonador das *threads* em espaço de usuário avisando sobre um determinado evento; o escalonador da *thread* pode usar essa *scheduler activation* para modificar suas estruturas, executar uma nova *thread* e fazer uma

[2]Parece que o uso de kernel threads nesse artigo está errada, como aponta o próximo artigo (sobre o FreeBSD). O correto seria *threads* baseadas em processos, isto é, cada *thread* é um processo diferente, porém compartilham seu endereçamento de memória são escalonados separadamente pelo kernel.

requisição ao kernel.

Foi implementado um protótipo dessa idéia na estação de trabalho DEC SRC FireFly multi-processador. As kernel threads implementadas no Topaz (sistema operacional nativo da Firefly) foram modificadas, além da porção em espaço de usuário, a biblioteca FastThreads que implementa o suporte a *threads* em espaço de usuário executando em cima das kernel threads do Topaz.

A comunicação entre o alocador de processadores do kernel e o sistema de *threads* em espaço de usuário é estruturada em termos das *scheduler activations*. O termo *scheduler activation* foi escolhido pois cada evento programado causa o sistema de *threads* em espaço de usuário reconsiderar a decisão de escalonamento de quais *threads* executar em quais processadores.

A estrutura de dados da *scheduler activations* é bastante similar à estrutura do kernel thread tradicional. Cada *scheduler activation* tem duas pilhas de execução – uma mapeada no espaço de endereçamento do kernel e outra mapeada no espaço de endereçamento da aplicação. Cada *thread* em espaço de usuário tem sua própria pilha de execução em espaço de usuário, quando começa a executar. Quando uma *thread* faz uma chamada ao sistema, ela usa a pilha mapeada no espaço de endereçamento do kernel da *scheduler activation*.

Quando um programa é iniciado, o kernel cria uma *scheduler activation* atribuindo-a a um processador e faz um *upcall*(redirecionamento de execução) para dentro do espaço de endereçamento da aplicação em um ponto de entrada pré determinado. O sistema de gerenciamento das *threads* em espaço de usuário recebe o *upcall* e usa essa *scheduler activation* como contexto para se inicializar e executar a *thread* principal da aplicação. Quando a primeira *thread* executar ela pode criar mais *threads* em espaço de usuário e pedir mais processadores.

Analogamente, quando o kernel precisa notificar o espaço de usuário sobre algum evento, o kernel cria uma *scheduler activation*, atribuindo-a a um processador, a faz um *upcall* para dentro do espaço de endereçamento da aplicação.

A principal diferença entre *scheduler activations* e kernel threads é que uma vez que uma *thread* em espaço de usuário de uma *scheduler activation* é parada pelo kernel, essa *thread* nunca mais é reiniciada pelo kernel. Ao invés disso, uma nova *scheduler activation* é criada para notificar o sistema de *threads* em espaço de usuário de que aquela *thread* foi parada. Assim, o sistema em espaço de usuário pode decidir como escalonar suas *threads* prontas para executar. Ao contrário das kernel threads, quando o kernel pára uma *thread*, ele nunca notifica o espaço de usuário desse evento.

Usando *scheduler activations* o kernel consegue manter a invariante de que existem tantos *scheduler activations* (estruturas para executar *threads* em espaço de usuário) executando quanto processadores atribuídos ao espaço de endereçamento.

Uma questão muito relevante sobre esse processo é que uma *thread* pode estar executando em uma região crítica quando for bloqueada ou parada pelo kernel. Existem dois males possíveis: desempenho pobre (devido a outras *threads* que ficam testando por um *lock* que está na *thread* parada), ou um impasse. Problemas podem ocorrer mesmo quando as regiões críticas não estão protegidas por um *lock*.

Para evitar esse problema, é usado uma abordagem baseada em **recuperação** (ao invés de prevenção, por exemplo). Quando um *upcall* avisa ao sistema de *threads* de espaço de usuário de que uma *thread* foi bloqueada ou desbloqueada, o sistema verifica se a *thread* estava executando em uma seção crítica; se sim, a *thread* é continuada temporariamente via uma troca de contexto em espaço de usuário. Quando a *thread* sai de sua seção crítica ela devolve o controle para o *upcall* original, novamente via uma troca de contexto em espaço de usuário. Nesse momento, já é seguro colocar a *thread* de volta na lista dos "prontos". É utilizado esse mesmo mecanismo para continuar uma *scheduler activation* se ela for bloqueada durante a processamento de um evento do kernel.

Essa técnica é livre de impasses. Pelo fato de continuar a execução da *thread* que possuir o *lock*, é garantido que uma vez que o *lock* foi adquirido ele é sempre eventualmente solto. Além do mais, essa técnica dá suporte a *spin-locks* em espaço de usuário, já que o sistema de *threads* em espaço de usuário é sempre notificado quando ocorrer preempção, deixando-o continuar a *thread* que possui o *lock*.

Foram feitos alguns testes para comparação entre os diversos sistemas de *threads* vistos. São comparados os sistemas: FastThreads on Topaz Threads são *threads* em espaço de usuário executando em cima de kernel threads, FastThreads on Scheduler Activations são as *threads* em espaço de usuário executando em cima de *scheduler activations*, Topaz Threads são as kernel threads e Ultrix processes são processos comuns.

Inicialmente foi comparado o **custo** das operações para gerenciamento de *threads* dos diversos sistemas vistos. Dois testes foram feitos com esse propósito, o "Null Fork" – tempo de criar, escalonar executar e completar uma *thread*/processo que executa um procedimento nulo, e o "Signal-Wait" – tempo para o processo/*thread* enviar um sinal a um processo/*thread* esperando e esperar por uma condição.

O custo das operações de *threads* executando em cima de *scheduler activations* está na mesma ordem de grandeza que o custo de operações de *threads* em espaço de usuário comuns, que por sua vez é tem custo uma ordem de grandeza menor do que o Topaz Threads, e duas ordens de grandezas menor do que o Ultrix Processes.

Para ilustrar o efeito no desempenho de uma aplicação, foi medido a mesma aplicação paralela usando os 3 sistemas de *threads* visto. A aplicação usada foi a resolução em tempo $O(N \log N)$ para um problema de tamanho N. Foi medido o **ganho** de desempenho de executar 2 dessas aplicações em paralelo, em 6 processadores, com os sistemas de *threads*, em relação ao sistema seqüencial (sem paralelismo). O ganho observado usando Topaz Threads e o FastThreads original é relativamente o mesmo, só que é apenas a metade do ganho observado usando o FastThreads rodando com *scheduler activations*.

2.2 Artigo "Kernel-Scheduled Entities for FreeBSD"

Este artigo aparentemente não foi publicado nem apresentado em nenhuma conferência. Escrito por desenvolvedores do **FreeBSD**, descreve uma implementação da idéia de *scheduler activations* apresentado na seção 2.1.

Tipo	Artigo na *Internet*
Autores	Jason Evans e Julian Elisher
Data	Agosto de 2001
Link	`http://people.freebsd.org/~jasone/refs/freebsd_kse.ps`

Resumo O suporte para aplicações que fazem uso de *threads* (*multi-threaded*) no FreeBSD não tem sido ideal. No momento existem duas bibliotecas disponíveis; a **libc_r** que é invisível ao kernel e combina diversas *threads* em um mesmo processo; e a tradução do **LinuxThreads**, para o FreeBSD que cria um processo distinto para cada *thread* via a chamada `rfork()`, e mais uma *thread* para gerenciar sincronização entre as demais. Ambas as implementações possuem problemas de escalabilidade.

O suporte a *threads* em espaço de usuário é dado pela biblioteca **libc_r** que não explora as vantagens de múltiplos processadores e não consegue evitar bloquear quando fazendo E/S em dispositivos "rápidos" (i.e. discos rígidos, placa de rede, etc.).

Threads baseadas em processos são freqüentemente erroneamente referenciadas como kernel threads. As verdadeiras kernel threads são *threads* que executam dentro do espaço de endereçamento do kernel para executar pedaços do kernel em paralelo. *Threads* baseadas em processos são aquelas que têm como base algum número de processos que compartilham seu espaço de endereçamento, e são escalonadas como processos comuns pelo kernel. **LinuxThreads** implementa *threads* baseadas em processos. Por isso as **linuxthreads** estressam o escanolador, e também requerem mais recursos do kernel. Além do mais a troca entre *threads* só pode ser tão rápida quanto troca de contexto entre processos comuns.

Scheduler activations (SA) é uma tentativa de reunir as vantagens de *threads* em espaço de usuário e baseadas em processos, enquanto evitando as desvantagens de ambas. Colocado em termos simples, SAs permitem com que o kernel a o escalonador de *threads* em espaço de usuário (user-level threads scheduler – UTS) façam seu trabalho sem ter que "adivinhar" o que o outro está fazendo.

Kernel-scheduled entities (KSE) são similares em conceito a *scheduler activations*, mas adicionam suporte a competição com o sistema no escalonador. Com *scheduler activations* tradicionais, cada *thread* só compete por processador entre as threads da mesma aplicação, agora estão competindo com todos os processos e *threads* do sistema. A competição de escalonamento é controlada via grupos de KSE (KSE groups – KSEGs). Um processo tem um ou mais KSEGs, e cada KSEG tem um nível de concorrência que controla o número máximo de KSEs que podem executar naquele KSEGs. Cada KSEGs é uma entidade separada da perspectiva de *time-sharing*.

Um processo começa com um KSEG que tem nível de concorrência igual a um. Conforme novos *threads* são criados, o nível de concorrência daquele KSEG pode ser ajustado até o número máximo de processadores disponíveis para aquele processo. Se uma *thread* quiser competir no escalonador com o restante do sistema, um novo KSEG com nível de concorrência igual a um pode ser criado para aquela *thread*. Limites para KSEGs são implementados, ao invés de limites para processos, para que um processo não crie um número arbitrário de KSEGs, assim, ganhando uma vantagem no escalonamento.

KSEs requerem a habilidade de fazer *upcalls* ao espaço de usuário. Isso é um controle de fluxo muito diferente do que a execução normal usada em processos, pois com KSEs o processo é notificado de cada preempção e resumo, o que requer as *upcalls*.

Uma série de mudanças ao escalonador do kernel são necessárias para o suporte de KSEs, as principais são: KSEs são colocadas na fila de escalonamento, ao invés de processos, isso habilita execução concorrente de KSEs que estão associadas ao **mesmo** processo; o cálculo da divisão de tempo de processador são feitas usando KSEGs, ao invés de processos, isso permite múltiplos KSEGs no mesmo processo competir por tempo de processador com o restante do sistema, não só dentro do processo. Além disso o suporte às seguintes características é desejada: afinidade leve – o escalonador deve tentar manter processos no mesmo processador se existe a possibilidade de ganhar com localidade de *cache*; afinidade dura – alguns programas podem querer atribuir um KSE a um determinado processador para melhorar localidade de *cache*. KSEs atribuídos são executados em apenas um processador, ao menos que herança de prioridade o força a executar em outro processador para evitar um impasse; aumento no paralelismo – o escalonador atual só pode ser executado por um processador por vez, o que faz com ele seja um gargalo dependendo da carga do sistema.

Para implementar esse escalonamento, a solução encontrada foi implementar uma fila de escalonamento por processador. Cada processador tem uma fila associada para afinidade leves de KSEs, e outra para afinidade dura. Além disso existe uma fila global para KSEs com prioridades fixas, tais como, *threads* de interrupção de KSEs de tempo real.

O artigo não apresentou nenhum resultado para ilustrar o possível ganho em desempenho desse sistema.

Estado atual A idéia foi bem aceita pela comunidade que desenvolve o FreeBSD e a versão de desenvolvimento já inclui um suporte mínimo a KSEs. Segundo Julian Elischer, um dos autores dos artigos, o KSE já está funcionando para um processador. O próximo passo é permitir com que KSEs possam rodar em diversos processadores.

Houve uma exposição do trabalho desenvolvido na conferência BSDCon 2002 (`www.usenix.-org/publications/library/proceedings/bsdcon02/`), na seção Work-in-Progress.

3 Mecanismos de entrega de eventos

3.1 Artigo "A Scalable and explicit event delivery mechanism for UNIX"

Este artigo foi apresentado na conferência **USENIX 1999 Technical Conference**, ganhando o prêmio de *Outstanding paper award for research excellence.*

Tipo	Artigo apresentado em conferência
Autores	Gaurav Banga, Jeffrey C. Mogul e Peter Druschel
Conferência	USENIX 1999 Technical Conference
Data	Abril de 1999
Link	`http://www.cs.rice.edu/~gaurav/papers/usenix99.ps`

Resumo Aplicações UNIX não desejando serem bloqueadas quando efetuando E/S comumente utilizam a chamada ao sistema *select()* para aguardar eventos em diversos descritores de arquivos. O mecanismo usado no *select()* funciona bem para aplicações de pequena escala, mas não escala bem conforme o aumento do número de descritores de arquivos. Muitas aplicações modernas, como servidores da Internet, utilizam centenas ou milhares de descritores de arquivos (representando conexões de rede, arquivos de disco, etc) e sofrem muito da má escalabilidade do *select()*. Trabalhos anteriores demonstram que as implementações tradicionais de *select()* podem ser melhoradas, mas a má escalabilidade é inerente ao projeto do *select()*.

O principal problema de escalabilidade do *select()* é decorrente de sua interface com a aplicação. A aplicação precisa passar ao *select()* três "mapas" de bits, indicando quais descritores devem ser checados para escrita, leitura e exceções. Ao retornar do kernel os "mapas" de bits são sobrescritos com novos valores, correspondentes a subconjuntos dos conjuntos de entrada, indicando quais descritores de arquivos estão disponíveis para fazer E/S. A aplicação precisa então verificar cada um dos "mapas" de bits para descobrir informações sobre os descritores de arquivos.

Assim vemos que os "mapas" de bits precisam ser criados para o espaço do kernel, varridos pelo kernel, processados, copiados para o espaço da aplicação, e finalmente, varridos pela aplicação. Todos esses custos claramente aumentam conforme o aumento de número de descritores de arquivos. Outro aspecto da má escalabilidade do *select()* é que aplicação deve checar cada bit no mapa de entrada. Caso não exista nenhum descritor pronto para E/S, o *select()* bloqueia. Posteriormente, quando um descritor fica "pronto", o kernel acorda o processo bloqueado. Na implementação tradicional, esse processo não tem como saber qual descritor mudou de estado, então tem que repetir a varredura. *Essa varredura poderia ser evitada já que o kernel certamente sabia qual descritor mudou, entretanto essa informação não é conservada.*

Com isso em vista, desejou-se projetar um sistema eficiente e escalável para a aplicação decidir quais de seus descritores estão prontos para serem processados. Pode-se resolver isso de duas maneiras: com uma visão baseada em *estado*, na qual o kernel informa à aplicação qual é o estado atual de um descritor; ou uma visão baseada em *eventos*, na qual o kernel informa à aplicação a ocorrência de um evento significativo para um descritor.

O mecanismo utilizado no *select()* segue o método baseado em estado. Se, por exemplo, o *select()* indicar que um determinado descritor está pronto para leitura, e a aplicação ler apenas uma porção dos dados disponíveis, então na próxima chamada ao *select()*, ele indicará novamente que o descritor está pronto para leitura. Os métodos baseados em estado[3] inerentemente necessitam que o kernel verifique a cada chamada o estado de cada membro do conjunto de descritores.

Apesar do *select()* seguir um método baseado em estados, o subsistema de E/S do kernel é baseada em **eventos**: eventos sobre pacotes de dados, blocos de discos, etc. Portanto o *select()* precisa transformar notificações baseadas em eventos para uma visão externa baseada em estado. Entretando as aplicações "baseada em eventos" que usam o *select()* precisam gastar esforço transformando de volta as informações do modelo baseado em estados.

Este artigo apresenta um novo mecanismo para entrega de eventos, que permite à aplicação registrar interesse em uma ou mais fontes de eventos, e eficientemente processar novos eventos. Também é demonstrado que esse mecanismo, que requer apenas pequenas mudanças à aplicação, tem desempenho independente do número de descritores de arquivos.

A "Aplication Program Interface" (API) desenvolvida segue o modelo baseado em eventos. Nesse modelo, o kernel simplesmente notifica um fluxo de eventos para a aplicação. O kernel não verifica o estado de qualquer descritor, portanto não realiza trabalho proporcional ao número de descritores; apenas realiza trabalho proporcional ao número de eventos.

Uma API puramente baseadas em eventos possível dois problemas: chegadas frequentes de eventos podem criar um custo adicional de comunicação, especialmente para uma aplicação que não está interessada em ver cada evento individualmente; se a API promete fazer notificações sobre cada evento individualmente, é necessário alocar espaço proporcional à freqüência de eventos.

A API apresentada não faz notificação assíncrona (como um mecanismo baseado em sinais), o que ajuda a eliminar esse problema. Ao invés disso a API permite à aplicação descobrir de maneira eficente quais descritores receberam eventos. Uma vez que um evento chega para um certo descritor, o kernel agrupa eventos subsequentes para um determinado descritor até que a aplicação tome conhecimento do primeiro desses eventos; isso reduz a freqüência de comunicação e evita a necessidade de guardar informações para cada evento.

Com essas mudanças, é removida a necessidade de manter informações dentro do kernel que pode não ser útil para a aplicação. É removido também um par de transformações entre métodos baseado em eventos e baseado em estado. Isso melhora a escalabilidade da implementação do kernel e deixa a aplicação flexível para implementar algoritmos para gerenciamento dos eventos.

A idéia central da API é o uso da nova chamada:

```
int declare_interest(int fd, int interestmask, int *statemask)
```

A *thread*/processo pode chamar essa função declarando interesse nos eventos de determinado descritor (`int fd`). Esse interesse (`int interestmask`) pode ser de leitura, escrita, exceção ou falta de interesse (para não receber mais eventos desse descritor). Uma vez que uma *thread* declara interesse, o kernel rastrea a chegada de eventos para aquele descritor. Se o argumento *statemask* for não nulo, então *declare_interest()* também devolve o estado atual do descritor.

[3]Outro mecanismo também muito conhecido e utilizado é o *poll()*, que apesar de preservar o mapa de entrada e saída, também sofre dos mesmos problemas de escalabilidade que o *select()*.

Para aguardar novos eventos, a *thread* deve usar a seguinte nova chamada:
`int get_next_event(int array_max, event_descr_t *ev_array, struct timeval *timeout)`
O argumento `ev_array` é um ponteiro para um vetor, de comprimento `array_max`, de valores do tipo `event_descr_t`. Se algum evento estiver esperando para aquele *thread*, o kernel tira os eventos da fila e passa-os para a *thread* através do vetor `ev_array`.

Por deixar que a aplicação peça um número arbitrário de eventos em apenas uma chamada, o custo dessa chamada é amortizado ao longo das diversas chamadas. Se pelo menos um evento estiver na fila quando a chamada for feita, ela devolve imediatamente, isto é, a chamada não é bloqueada até encher o `ev_array`. Caso a fila de eventos esteja vazia, então a chamada é bloqueada até que pelo menos um evento é recebido, ou até que o `timeout` expire.

O desempenho desse novo mecanismo foi medido usado um "proxy" HTTP baseado em eventos, rodando a versão modificada do sistema operacional Digital UNIX V4.0D. Ele foi modificado para usar ou o *select()* ou a nova API.

Primeiramente foi realizado testes para medir o **custo** das operações básicas da nova API, medindo como esse custo escala conforme o número de conexões por processo. O teste consistiu basicamente em um proxy HTTP simples baseado em eventos. Ele foi modificado com a possibilidade de configurá-lo para usar ou o *select()*, ou o nova API.

Em todos os experimentos, a carga foi gerada automaticamente por dois tipos de clientes. As conexões "quentes" vem de um conjunto de processos rodando o programa *S-client*, criado para gerar cargas realísticas de requisições. Também foi usado um cliente para "adicionar" carga para gerar um número grande de conexões "frias"[4].

O **custo** da operação *get_next_event()* variou linearmente conforme o aumento do número de eventos exigidos (o *array_max*).

Para um dado valor de *array_max*, foi visto que variando o número de conexões "frias" de 0 até 2000 quase não tem efeito algum no custo do *get_next_event()*. Também foi averigüado que aumentando o número de conexões "quentes" aparentemente não aumentou o custo por evento do *get_next_event()*. De fato, o mecanismo de aglomeração/agrupamento de eventos reduz o custo por evento.

Para medir o custo da chamada *declare_interest()* e do *revoke_interest()* foi usado 32 processos *S-client* fazendo requisições por arquivos de 1 KByte. Apenas pequenas variações foram medidos no custo dessas chamadas quando variando o número de conexões "frias" de 0 até 2000.

Foi medido, depois, o **desempenho** do servidor proxy, usando ou o *select()* ou a nova API. Em todos os testes todas as requisições são para o mesmo arquivo de 1 KByte. Na primeira série de testes foram utilizados sempre 32 conexões "quentes", mas variando o número de conexões "frias" entre 0 e 2000. Os resultados desse teste claramente indicam que o nova API desempenha independentemente do número de conexões "frias", enquanto que o *select()* tradicional não.

Na segunda série de testes foi afixado o número de conexões "frias" em 750, e foi medido o tempo de resposta (medido nos clientes). Ao usar a nova API, o proxy exibe uma latência muito menor e só fica saturado numa carga de requisições maior.

[4] Conexões "frias" são aquelas com longa duração que simulam a demora de um grande WAN.

Por fim, foi feito um "profile" do uso do processador dos dois sistemas em condições iguais (número de conexões "frias" fixas em 750, 50 conexões "quentes" e um total de 400 requisições por segundo). O "profile" mostra que a nova API aumenta significativamente o tempo ocioso do processador. Enquanto que o *select()* tradicional consome 34% do processador, e a versão melhorada consome 12%, a nova API consome menos do que 1% do processador.

3.2 Artigo "Kqueue: A generic and scalable event notification facility"

Este artigo foi apresentado na conferência **FREENIX Track: 2001 USENIX Technical Conference**, e também na conferência **BSDCon 2000**. Escrito por um desenvolvedor do **FreeBSD**, conta uma implementação da idéia do mecanismo de notificação por eventos originalmente intoduzido pelo artigo na seção 3.1. A idéia apresentada nesse artigo, entretanto, é mais *geral* do que a do original, já que propõe uma interface para qualquer tipo de evento e não apenas eventos relacionados a descritores de arquivos.

Tipo	Artigo apresentado em conferência
Autor	Jonathan Lemon
Conferências	FREENIX Track: 2001 USENIX Technical Conference BSDCon 2000
Link	`http://people.freebsd.org/~jlemon/papers/kqueue_freenix.pdf`

Resumo Aplicações são comumente baseadas em eventos, de forma que elas realizam a sua tarefa em função de algum evento ou atividade externa disponibilizada por algum mecanismo. Portanto o desempenho de uma aplicação pode depender da eficiência da capacidade de detectar e responder a esses eventos. O FreeBSD disponibiliza duas chamadas ao sistema para detecção de atividades em descritores de arquivos, que são o `poll()` e o `select()`. Entretanto nenhuma dessas chamadas escalam muito bem conforme o número de descritores sendo monitorados fica grande.

As interface `poll()` e o `select()` sofrem da deficiência de que a aplicação precisa passar a lista inteira de descritores a serem monitorados, para cada chamada. Isso resulta no fato que o sistema precisa realizar duas cópias de memória da lista atravessando a fronteira de espaço de endereçamento usuário/kernel. Para listas contendo muitos milhares de descritores[5], a experiência prática tem mostrado que tipicamente apenas poucas centenas realmente possuem atividades, fazendo com que 95% das cópias sejam desnecessárias. Além disso é necessário duas "varreduras" das listas de descritores, uma feita pelo kernel e a outra pela aplicação. Como o kernel sabia *quais* descritores estavam "ativos", a aplicação recalcula informações que o sistema já tinha conhecimento.

Esses problemas têm origem no fato de que as chamadas `poll()` e o `select()` são "*stateless*", isto é, o kernel não guarda registro do que a aplicação está interessada entre as chamadas, e portanto precisa recalculá-la todas as vezes.

Para solucionar essas questões foi desenvolvida uma nova API. O principal objetivo era criar um sistema que seria **eficiente** e **escalável** para um número grande descritores. O segundo objetivo

[5] Caso bastante comum em servidores na Internet.

seria deixar o sistema **flexível**, para incorporar diversos tipos de eventos. Sistemas baseados em UNIX tradicionalmente têm falta de uma mecanismo robusto para notificação de eventos.

O conjunto de eventos que uma aplicação pode estar interessada não é limitada às atividades de descritores de arquivos abertos. Uma aplicação pode também querer ser notificada quando uma requisição de E/S assíncrona (AIO – Asynchronous I/O) é completada, quando um sinal é entregue à aplicação, quando um arquivo em um sistema de arquivos muda de alguma forma, ou quando um processo termina. Nenhum desses eventos são tratados eficientemente no momento; entrega de sinais é limitada e cara, e os outros eventos listados requerem um sistema de checagem repetida ("*polling*") ineficiente.

Expandir a quantidade de informações devolvidas para a aplicação além do fato de que um evento ocorreu também foi considerado desejável. O objetivo de providenciar mais informações à aplicação foi utilizado ao projetar o novo mecanismo de entrega de eventos.

O mecanismo também tem que ser *confiável*, isto é, ele nunca deveria falhar silenciosamente ou devolver dados inconsistentes ao usuário. Por isso o mecanismo faz com que diferentes eventos sejam agrupados, sempre que possível, em um único evento. Portanto eventos que são entregues à aplicação podem eventualmente designar muitas ocorrências de atividade na fonte do evento sendo monitorado. Esse agrupamento pode reduzir drasticamente o uso de memória (portanto aumentando a confiabilidade do mecanismo) e deixá-lo mais eficiente.

Uma outra característica desejada no novo mecanismo é que os eventos devem ser considerados como ativados por "nível", ao contrário de ativados por "borda", ou seja, um evento é notificado sempre que uma condição específica for verdadeira, ao contrário de apenas notificar quando a atividade é detectada na fonte do evento. Isso é necessário para que a aplicação não tenha que manter informações de todos os eventos sendo monitorados. Isso corresponde à semântica já existente das chamadas `poll()` e `select()`.

O nova API criada, satisfazendo todas as características mencionadas, chama-se *kqueue*, que consiste basicamente em duas chamadas novas:

```
int kqueue(void);
```

Essa chamada server para criação de uma nova fila de notificação (*kqueue*), devolvendo um inteiro para identificação da fila; e

```
int kevent(int kq,
         const struct kevent *changelist, int  nchanges,
         struct kevent *eventlist, int nevents,
         const struct timespec *timeout);

struct kevent {
  uintpt_t  ident;      // identifier for event
  short     filter;     // filter fot event
  u_short   flags;      // action flags for kq
  u_int     fflags;     // filter flag value
  intptr_t  data;       // filger data value
```

```
  void      *udata;     // opaque identifier
};
```

Essa chamada serve para ao mesmo tempo registrar interesse em novos eventos e ser notificado de algum evento que está esperando na *kqueue*. Mudanças que devem ser feitas à *kqueue* são dadas pela *changelist*, e qualquer novo evento são colocados na *eventlist*.

Eventos são registrados no sistema via a estrutura *struct kevent* que são unicamente identificadas pela tupla `<kq,ident,filter>`.

O parâmetro *filter* é um identificador de uma pequena porção de código do kernel que é executada quando há alguma atividade da fonte de eventos sendo monitorada para decidir se o evento deve ser devolvido à aplicação ou não. Alguns dos valores de filtros padrões são: `EVFILT_READ, EVFILT_WRITE, EVFILT_AIO, EVFILT_VNODE, EVFILT_PROC, EVFILT_SIGNAL`. Os filtros `READ` e `WRITE`, por exemplo, foram criados para funcionar em qualquer descritor de arquivos, e o campo *ident* indica o número do descritor. Esses filtros imitam o comportamento das chamdas `poll()` e `select()`; devolvem o fluxo de execução quando existe dados a serem lidos, ou se a aplicação pode fazer uma escrita sem ser bloqueada.

Na implementação, o centro das atividades do *kqueue* é uma estrutura de dado chamada *knote* que corresponde à estrutura *kevent* vista pelo aplicação. A outra estrutura principal é o próprio *kqueue* que possiu duas funções: providenciar uma file contendo os *knotes* que devem ser entregues à aplicação, e manter um registro dos *knotes* que correspondem a *kevents* que a aplicação registrou interesse.

Quando alguma atividade ocorre alguma estrutura de dado é modificada em resposta. Dentro do caminho de código onde isso ocorre, um "gancho" (*hook*) é colocar para o sistema do *kqueue*. Assim o sistema pode interceptar todas as atividades, passar pelo *filtro* e decidir se deve incluir esse evento na lista de *knotes* a ser entregue à aplicação.

Para medição do desempenho desse novo mecanismo foram feitos inicialmente experimentos para comparar o **custo** das operações da nova API, com o custo das chamadas tradicionais como o `poll()`.

Em um primeiro teste um número variado de arquivos e soquetes foram criados e passados ao *kevent* ou *poll*. O tempo necessário para que a chamada completasse foi registrado. A experiência demonstra que demora duas vezes mais tempo para adicionar um novo *knote* a um *kqueue* quando comparado à chamada do *poll*. Depois da chamada inicial `EV_ADD` para adicionar os descritores ao *kqueue*, entretanto, o tempo necessário para verificar esses descritores mostrou-se constante e perto de zero, já que nesse experimento não há atividade nos descritores registrados.

Isso providencia um limitante inferior do tempo necessário para uma chamada ao *kevent* independente do número de descritores sendo monitorados. O principal custo associado com essa chamada é o processo de registrar um novo *knote*; mas quando isso for feito, o custo para monitorar um descritor inativo é neglegível. Isso não ocorre com o *poll*, que demonstra o mesmo custo independente dos descritores terem atividades ou não.

O limitante superior do tempo necessário para uma chamada ao *kevent* depois do registro dos descritores seria dado se todos os descritores tivessem atividade. Este experimento também foi feito, e nesse caso o *kevent* **também** demora menos tempo do que o *poll*[6].

Por fim foi comparado o tempo acumulado necessário para verificar um descritor entre o melhor/pior caso do *kqueue* e o *poll*. Essa comparação demonstrou que apesar do *kqueue* exigir um maior tempo na inicialização, a menor que descritor seja verificado menos do que 4 vezes, o *kqueue* tem um custo amortizado mais baixo do que o *poll*.

Foi medido depois o **desempenho** desse mecanismo em aplicação em espaço de usuário. Duas aplicações foram modificadas para utilizar o *kevent*, um servidor de proxy com cache comercial e o servidor Web `thttpd`. Similarmente aos testes com servidores de rede feitos na seção 3.1, foram utilizados dois programas clientes para gerar requisições nos servidores; um gerando conexões ativas e outro gerando conexões inativas.

Com o servidor de proxy com cache foi observado que utilizando o *kevent* o tempo de processamento gasto em cada chamada são constantes independente do número de conexões inativas. A implementação baseada na chamada *select* tem desempenho muito pior e começa atingir a saturação conforme a quantidade de tempo ocioso na processador diminiu. Foi obeservado que o sistema fica essencial sem recursos quando são utilizadas apenas 2000 conexões.

O servidor Web `thttpd` também foi modificado para usar a nova chamada. Foi medido o tempo de resposta e uso do processador para um número crescente de conexões. Os resultados indicam que a versão original, usando a chamada *poll*, consome todo poder de processamento com apenas 600 conexões, enquanto que a versão que utilizava o *kevent* ainda tinha 48% de processamente ocioso com 10,000 conexões.

[6]Lembramos que o *poll* tem desempenho independente de atividade nos descritores, portanto os resultados nesses dois testes são muito semelhantes para o *poll*.

4 Escalonamento de disco

4.1 Artigo "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O"

Este artigo foi apresentado na conferência ACM – Symposium on Operating Systems Principles (SOSP).

Tipo	Artigo apresentado em conferência
Autores	Sitaram Iyer e Peter Druschel
Conferência	ACM – Symposium on Operating Systems Principles – 2001
Link	`http://www.cs.rice.edu/ ssiyer/r/antsched/`

Resumo Escanalonamento de requisições a disco tem sido uma parte importante em sistemas operacionais desde seus primórdios. Essa artigo analisa escalonamento de disco, identifica um fenômeno chamado "*deceptive idleness*" (ociosidade enganosa) e propõe "*anticipatory scheduling*" (escalonamento antecipatório) como solução.

Tradicionalmente os escalonadores de disco são "*work-conserving*" já que selecionam uma requisição logo (ou antes) que a requisição anterior termina. Se considerar processos fazendo requisições *síncronas* a disco: cada processo faz uma requisição pouco tempo depois da requisição anterior tenha terminada e portanto mantém no máximo uma requisição pendente por vez. Isso força o escalonador a tomar decisões muito cedo, assim ele assume que o processo que realizou a última requisição não tem nenhuma outra requisição pendente, e seleciona uma requisição de outro processo.

Os dados exigidos por um processo são comumente posicionados de forma sequencial no disco. Entretanto, "*deceptive idleness*" força um escalonador que otimiza busca a multiplexar entre requisições de diferentes processos. Essas buscas contínuas do cabeçote de leitura pode causar uma degradação no desempenho de um fator de até quatro vezes.

Considere o cenário onde existam dois processos intensivos no uso de disco, chamados p e q. Um escalonador que tenta minimizar o tempo de busca no disco, se tivesse todas as requisições de antemão, tentaria satisfazer todas as requisições do processo p, depois faria uma única busca custosa aos dados do processo q e satisfaria todas as requisições desse processo, já que é bastante comum os dados usados por um mesmo processo estarem fisicamente pertos em disco. Agora considere esse cenário só que com as requisições *síncronas*. Nesse caso, o escalonador iria realizar uma requisição do processo p e pelo fato do processo p não ter tido tempo para realizar suas computações para criar sua próxima requisição, o escalonador escolhe uma requisição do processo q, realizando uma busca custosa à parte do disco os dados do processo q. Isso faz com que o escalonador fique com um comportamento do tipo FCFS (First Come, First Served). O problema é ainda pior se mais do que dois processos fazem requisições síncronas.

"*Deceptive idleness*" também pode afetar escalonadores que são do tipo "*proportional-share*" (divisão proporcional). O comportamento desses escalonadores é satisfazer requisições de múltiplas aplicações de acordo com uma razão fixa indicada. O comportamento errôneo FSFC faz com que o escalonador sempre obtenha proporções tendendo a 1:1.

É possível parcialmente contornar o problema de "*deceptive idleness*" ao usarmos "*asynchronous prefetch*". Isso envolve adivinhar o padrão de requisições futuro para um processo, e servir a requisição posterior antes que a atual seja completada. Cada processo mantém, portanto, múltiplas requisições pendentes, o que permite ao escalonador uma chance de satisfazer requisições consecutivas do mesmo processo. Aparentemente isso resolveria os dois problemas citados acima. "*Prefetching*" pode ser efetuado explicitamente pela aplicação, ou transparentemente pelo kernel. Entretanto as duas abordagens têm limitações fundamentais em termos de manejabilidade, acurácia e custo adicional.

Os problemas com aplicações que fazem "*prefetch*" explicíto são: (1) as aplicações não são cientes do seu padrão de acesso futuro; (2) as aplicações precisam ser escritas num paradigma de programação enfadonho; (3) aplicações existentes teriam que ser reescritas para esse fim; e (4) essas bibliotecas/chamadas (como a `aio_read` que é uma chamada opcional descrita na extensão de tempo real do POSIX) podem não estar implementadas ou disponíveis em alguns sistemas operacionais.

Existem soluções transparentes em sistemas de arquivos modernos para fazer o "*prefeching*" de dados automaticamente e portanto transparentemente para a aplicação. O problema com essa solução é que o sistema de arquivos é até menos capaz de predizer os padrões de acesso futuros do que as próprias aplicações. "*Prefetch*" precisa ter uma noção exata da localização da próxima requisição, e as penalidades por errar podem ser altas.

Em suma, "*prefetching*" pode potencialmente aliviar e até mesmo eliminar "*deceptive idleness*", mas as limitações em sua possibilidade de implementação e desempenho sob muitas condições, descartam-no como uma solução geral.

O artigo então apresenta uma solução simples, prática, geral e transparente para a aplicação e com baixo custo, para o problema de "*deceptive idleness*". Existem três condições básicas para que "*deceptive idleness*" se manifeste: (a) múltiplas aplicações intensivas em uso de disco fazendo requisições síncronas, (b) a natureza não preemptiva das requisições a disco, e (c) um escalonador "*work-conserving*", que escalona uma requisição imediatamente após a requisição anterior tenha completada. A solução apresentada resolve o problema por eliminação da condição (c), envolvendo a política de escalonamento de disco atual em um arcabouço de "*anticipatory scheduling*" (escalonamento antecipatório).

Quando uma requisição em disco é completada, o arcabouço potencialmente espera um tempo para que requisições adicionais cheguem ao escalonador antes de despachar uma nova requisição ao disco. Aplicações que geram outras requisições rapidamente têm chance de fazê-las antes do escalonador tomar uma decisão; "*deceptive idleness*" é então evitada. O fato de que o disco fica ocioso durante um pequeno período não degrada, necessariamente, o desempenho. Ao contrário, o artigo mostra como uma aplicação cuidadosa desse método melhora o "*throughput*" (banda conseguida por unidade de tempo) e casa com mais precisão com as alocações desejadas desse serviço.

A questão de quanto tempo esperar, ou mesmo se esperar ou não em algum ponto de decisão é a chave para a eficiência e desempenho do sistema. Na prática, o arcabouço espera o menor tempo que ele espera que, em alta probabilidade, os benefícios trazidos ao ficar esperando são maiores do que o custo de deixar o disco ocioso. Para permitir tal flexibilidade, enquanto minimizando a carga em cima do desenvolvedor do escalonador de disco em questão, o arcabouço consiste de três componentes: (1) o escalonador de disco orginal, que implementa a política de escalonamento e não é ciente do "*anticipatory scheduling*"; (2) um *núcleo* de antecipação independente de escalonador; e, (3) heurísticas adaptativas, específicas ao escalonador, para escalonadores que minimizam busca e são "*proportional-share*".

Escalonadores do tipo "*Shortest Positioning-Time First*" (SPTF) calculam o tempo de posicionamento (busca) do cabeçote do disco de cada requisição na fila e escolhem aquele com menor tempo. A heurística desenvolvida para maximizar o banda desses escalonadores analisa a requisição candidata pela política de escalonamento. A idéia da heurística é a seguinte: se a requisição candidata é localizada perto da posição atual do cabeçote, não há motivo para esperar; caso contrário, se o processo que fez a requisição anterior tem alta probabilidade de fazer outra requisição em pouco tempo (i.e., seu tempo médio esperado de computação é pequeno[7]), *e* a esperança dessa requisição ser perto da posição atual do cabeçote é alta, então a heurística decide esperar por essa requisição.

A política SPTF tem o problema potencial de sofrer de "*starvation*" (abandono de requisições), já que as requisições para locais distantes do disco podem nunca serem servidas. Uma solução para isso é o "*Aged-SPTF*", que associa as requisições da fila com prioridades que são incrementadas conforme o passar do tempo. A mesma heurística usada com a política SPTF, também funciona com a política "*Aged-SPTF*".

A política CSCAN (também conhecida como C-LOOK) é extremamente popular e é implementada em diversos sistemas operacionais baseados em UNIX. É a versão unidirecional da política Elevador/SCAN/LOOK; move-se a cabeça em uma direção, servindo todas as requisições no seu caminho, e recomeça na primeira requisição disponível. A heurística de antecipação para essa política é baseada na heurística para o SPTF, mas com um cláusula adicional. O módulo de coleção de estatísticas também mantém uma espectativa da direção da busca: se o candidado atual envolve uma busca para frente e a próxima requisição tem alta probabilidade de ocasionar uma busca para trás, então a equação de custo benefício é ignorada e é decidido não esperar.

Uma heurística também foi feita para escalonadores que usam as políticas baseadas em "*proportional-share*". Essas políticas usam relógios virtuais com pesos para lembrar a quantidade de serviço de disco que cada processo recebeu. A heurística é simplesmente esperar pelo processo atual se ele satisfizer três condições: (a) não tem requisições pendentes, (b) tem um "tempo de computação" esperado menor do que 3ms, e (c) tem o menor relógio virtual entre os relógios dos processos que têm requisições pendentes.

Esse arcabouço de escalonamento antecipatório foi implementado no kernel do FreeBSD-4.3. Usando essa implementação foram feitos alguns experimentos para medir o efeito da implementação em comparação ao sistema padrão, sem o escalonamento antecipatório.

Primeiramente foi estudado o efeito do escalonamento antecipatório em requisições síncronas com diferentes tipos de padrões de acesso. Dois processos radidamente requisitavam 64KB de um arquivo de 1GB em três padrões diferentes: sequencialmente, alternando os blocos de 64KB, e

[7]Define-se aqui tempo de computação de um processo como sendo o intervalo de tempo entre duas requisições consecutivas.

randomicamente. Além disso foi usado dois métodos de leitura, um usando a chamada `read` e outro usando a chamada `mmap`. Em todos os três casos, o escalonamento antecipatório gerou um aumento na banda entre 30 a 400%, dependendo do experimento. O único caso em que não houve aumento significativo da banda foi no caso de leitura sequencial usando o `read`, já que o FreeBSD utiliza "*prefetch*" assíncrono conservativo para o `read`. Em todos os casos foi observado uma diminuição na utilização do disco no sistema com escalonamento antecipatório, proporcional ao aumento da banda.

O próximo experimento ilustra o impacto de esperar aplicações que possuem diferentes tempo paras realizar a próxima requisição (i.e. diferentes "tempos de computação"). Dois processos mapeam dois grandes arquivos distintos na memória e requisitam páginas da memória (portanto sem "*prefetch*" assíncrono). Depois de cada 64KB, os processos pausam por um tempo determinado. Foi experimentado quatro tempos de pausa distintos.

No primeiro caso, o de processos simétricos, foi medido a banda recebida variando o tempo de pausa entre cada requisição de 0 a 25ms. Os resultados demonstram que de 0 a 8ms o sistema com escalonamento antecipatório tem um banda maior do que o sistema original, e utilização de disco menor. De 8ms em diante as quantidade de memória recebida por segundo é quase idêntico nos dois sistemas, já que o tempo de espera se torna comparável ao tempo da requisição concluir.

No caso de uso de processo assimétricos, foi medido a banda recebida por um processo variando o tempo de pausa entre 0 e 25ms, e o outro processo criando requisições logo que a requisição anterior termina. Nesse caso, a banda obtida com espera antecida excede a do sistema original por uma grande margem.

No caso dos processos variarem seus tempo de espera randomicamente, o sistema com escalonamento antecipatório se comporta similarmente ao sistema original, ou seja, o escalonamento antecipatório não gerou nenhum custo adicional indesejado.

Nesse último caso, já que a heurística lida bem com tempos de espera variando randomicamente, foi exercitado o comportamento do caso patalógico escrevendo-se um "adversário inteligente". Dois processos criam n requisições rapidamente, depois esperam um tempo que excede o tempo calculado pela heurística, e repetem o processo. Para $n = 0$ o escalonamento antecipatório consegue lidar bem com as requisições lentas, mas para n entre 1 e 4, a heurística antecipatória tem desempenho um pouco pior do que o sistema original: cerca de 20%. Esse resultado indica que mesmo com aplicações maliciosas, a degradação de desempenho possível é aceitável.

A seguir foram feitos experimentos para medir o impacto do escalonamento antecipatório em aplicações reais. Foram escolhidos dois programas, o servidor Apache e o *linker* LD, e dois "*benchmarks*" padrões que devem refletir o comportamento de uma larga gama de aplicações.

Foi medido a quantidade de banda que um servidor Apache consegue servir, utilizando-se o sistema original e com o escalonador antecipatório, ambos usando duas chamadas diferentes para fazer leitura de disco, o `read` e o `mmap`. Os resultados obtidos mostrou que o sistema com escalonamento antecipatório melhorou a banda em 29% com o `read` e 71% com o `mmap` (que não implementa um "*prefetch*"). Além disso foi observado uma diminuição no uso do disco em 12% e 16%, respectivamente.

O tempo de execução necessário para fazer a *linkage* durante a compilação do kernel FreeBSD, usando o utilitário `ld`, foi medido com o sistema original e com escalonamento antecipatório, ambos utilizando duas políticas de escalonamento, o Aged-SPTF e o CSCAN. Quando foi rodado apenas um `ld`, ambos os sistemas, com ambos os escalonadores deram tempos bem parecidos. Ao utilizar dois `lds` concorrentes seria esperado que esse valor dobrasse, entretanto "*deceptive idleness*" faz com que o tempo de execução no sistema original aumentasse por um fator de 5.5. Nesse caso, o escalonamento antecipatório diminui o tempo de execução em 68% no caso do Aged-SPTF e faz com que o desempenho quase escale para o valor esperado (i.e., duas vezes o tempo da execução do processo único). Para a política CSCAN, entretanto, a melhora é de apenas 48%, já que o CSCAN sempre serve requisições na direção atual, mas arquivos objetos geralmente são acessados em ordem arbitrária.

Um "*benchmark*" para servidor de sistemas de arquivos distribuídos é o "*Andrew filesystem benchmark*" que tenta simular uma carga típica de servidor de sistemas de arquivos para um ambiente de desenvolvimento de software. Quando o experimento foi feito com apenas um cliente, não foi observado diferença de desempenho entre o sistema original e com escalonamento antecipatório. Com dois clientes, entretanto, o escalonamento antecipatório tem tempo de execução 8.4% menor do que o sistema original. Foi testado também o sistema com 8 clientes, e os resultados são bem semelhantes ao caso com 2 clientes, comprovando a escalabilidade do escalonamento antecipatório para servidores de arquivos ocupados.

O "*TCP-B benchmark*" tem como objetivo exercitar um banco de dados com comportamento semelhante a de transações bancárias. Para tal, foi utilizado o banco de dados MySQL, e dois processos clientes. Os clientes faziam requisições de atualizações (que envolve uma leitura e uma escrita assíncrona), e seleções (que envolve uma leitura síncrona), no mesmo banco de dados e em banco de dados distintos. No caso em que os dois clientes acessavam o mesmo bando de dados o aumento de desempenho obtido usando o escalonamento antecipatório foi de apenas 2% para as atualizações e 5% para as seleções, já que os dados estão fisicamente pertos um do outro. No caso em que os dois clientes acessavam banco de dados distintos o aumento foi de 30% para as atualizações e 60% para as seleções.

Por fim, alguns dos testes foram repetidos em máquinas com processadores mais rápidos, e discos mais modernos e com mais rápidos, para determinar o efeito que o escalonamento antecipatório teria nessas máquinas mais modernas. O resultado foi que esses sistemas também sofrem de "*deceptive idleness*" e portanto demonstram melhoria no desempenho com o arcabouço de escalonamento antecipatório.

5 Sistemas de arquivos

5.1 Artigo "HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions"

Este artigo foi publicado na revista ACM – Transactions on Computer Systems (TOCS) 1997, e também apresentado na conferência Workshop on I/O in Parallel and Distributed Systems (IOPADS) 1996.

Tipo	Artigo apresentado em conferência e publicado em revista
Autores	Orran Krieger e Michael Stumm
Conferência	Workshop on I/O in Parallel and Distributed Systems – 1996
Revista	ACM – Transactions on Computer Systems – 1997
Link	`http://www.eecg.toronto.edu/~okrieg/papers/hfs_tocs.ps.gz`

Resumo Um sistema de arquivos paralelo difere de um sistema de arquivos para um único processador devido aos recursos que ele deve gerenciar e pelo suporte a requisitos exigidos. Sistemas com multiprocessamento geralmente têm um número grande de discos e módulos de memória distribuídos pelo sistema. A fim de otimizar o desempenho de E/S o sistema de arquivos precisa explorar de forma apropriada esses recursos. Por exemplo, dados devem ser distribuídos nos discos para melhorar a banda de E/S, guardados em "cache" de memória para reduzir a demanda dos discos, e lidos com antecedência ("*prefetched*") com finalidade de esconder das aplicações a latência da E/S.

Muitas aplicações de supercomputadores têm exigências massivas de E/S. Essas aplicações paralelas possuem necessidades que são diferentes daquelas de aplicações sequenciais típicas do Unix para as quais muitos sistemas de arquivos foram otimizados/especializados. Otimizar um sistema de arquivos para aplicações paralelas que fazem E/S intensivamente é complicado pelo fato de que as exigências dessas aplicações não são bem compreendidas, e portanto não existe conhecimento sobre as estruturas de arquivos, políticas, e interfaces de E/S para melhor satisfazer essas necessidades.

Para suprir as necessidades dessas aplicações paralelas foi desenvolvido um sistema de arquivo chamado "Hurricane File System" (HFS). O HFS tem suporte a um grande e expandível conjunto de estruturas de arquivos, interfaces e políticas, e permite à aplicação configurar os arquivos que usa para suprir suas necessidades de E/S. Esse alto grau de flexibilidade é atingido pelo uso de muitos pequenos blocos de construção, cada um definindo uma porção da estrutura de um arquivo, ou implementando um conjunto simples de políticas. A aplicação pode controlar a composição dos blocos de construção usados para para cada arquivo, ou para cada *instância* do arquivo aberto. Isso permite que a implementação do arquivo possa ser configurada para atender melhor às necessidades de E/S de cada aplicação.

Uma aplicação usando o HFS pode escolher como os dados de um arquivo serão distribuídos pelos discos, como o dado deve ser distribuído em cada disco, se deverão ser armazenada cópias redundantes para tolerância a falhas, como esconder latências, qual politíca de sincronização deve ser usada, e se deverá ser invocado um algoritmo transparente de compressão/descompressão nos dados. O artigo argumenta que esse nível de de flexibilidade é necessário para acomodar as exigências de E/S diversas de aplicações paralelas. Primeiramente, a flexibilidade é crucial para permitir aplicações em supercomputadores desfrutar a banda total de E/S dos discos conectados ao sistema. Em segundo lugar, dado o pouco conhecimento das exigências de aplicações paralelas que são intensivas em E/S, a flexibilidade é necessária para garantir que o sistema de arquivos não restringa o desenvolvedor. Em terceiro lugar a flexibilidade permite ao HFS adaptar-se facilmente a novas interfaces de E/S, novos sistemas operacionais e novas demandas de funcionalidade no sistema de arquivos, e novas plataformas de hardware. Em quarto lugar a flexibilidade permite com que as exigências de diferentes cargas, como a de aplicações seqüenciais, sejam atendidas sem adicionar sobrecarga. Finalmente, a flexibilidade é importante para experimentação de diferentes estruturas de arquivos e políticas.

Para processamento paralelo, flexibilidade e customização não é considerado útil se vier associado a custos adicionais que levam a um desempenho pobre. É demonstrado que a flexibilidade do HFS vem com custo adicional desprezível quando comparado com os outros custos associados a E/S. Além disso é visto que em muitos casos a habilidade de customizar a implementação de um arquivo para sua demanda exigida pode resultar em um custo menor do que uma implementação mais genêrica.

O enfoque principal desse sistema foi a de aplicações paralelas intensas em E/S, mas o HFS foi projetado para ser um sistema de arquivos para uso genérico que pode eficientemente atender às necessidades de uma grande variedade de aplicações. Como resultado, o HFS também tem suporte às interfaces padrões usadas em aplicações seqüenciais.

Como já mencionado, cada arquivo no HFS é implementado combinando um conjunto de blocos de construção ("*building blocks*"), que pode definir a estrutura do arquivo, ou implementar um conjunto simples de políticas. Esses blocos são implementados como objetos e portanto contêm estado e um conjunto de operações para manipular esse estado.

Um objeto de bloco de construção exporta uma interface que especifica as operações que podem ser invocadas por outros blocos. Ele pode também importar (uma ou mais) interfaces que estão sendo exportadas por outros blocos. Dois blocos são ditos *conectados* se um deles invoca operações do outro, e o bloco também *referencia* o outro. A composição dos blocos de contrução que implementam um arquivo (isto é, o conjunto de blocos e a maneira em que estão conectados) determina o comportamento e desempenho do arquivo.

Nesse sistema de blocos de construção, a flexibilidade é atingida de diversas maneiras. Primeiramente, dada uma composição, é possível trocar um dos blocos por outro desde que as interfaces importadas e exportadas sejam aa mesmas. Portanto para cada tipo de bloco, é possível ter diversas implementações, cada uma implementando uma política diferente ou otimizada para comportamentos de aplicação diferente. Depois, novos blocos podem ser adicionados a uma estrutura existente se as interfaces dos blocos conectados casarem, assim modificando a topologia. Isso pode ser usado para adicionar uma nova funcionalidade.

Por fim, é possível dar suporte a novos blocos de construção introduzindo novos blocos que exportam essas interfaces, mas importam interfaces existentes para que elas possam ser conectadas com as estruturas existentes.

Em geral, quanto mais "fina" a granularidade dos blocos de construção usados na composição (e portanto um número de blocos maior na composição), maior será o grau de flexibilidade. Na implementação, observou-se que havia uma tendência em usar muitos blocos "finos", ao contrário de poucos blocos "grossos".

No modelo usado, o controle é passado de um bloco de contrução para outro, fazendo com que o primeiro invoque uma operação exportada pela outra. O fluxo através da composição de blocos é inicializada ou pela aplicação invocando uma operação de algum dos blocos, ou como resultado de uma *page fault* ou interrupção de disco.

Um aspecto importante do trabalho é que o sistema permite às aplicações especificar a composição inicial de arquivos que elas criam, e permite à aplicações modificar (em parte) a composição de blocos dos arquivos que elas estão acessando. Na construção nos blocos é especificado como ele deve se conectar aos demais blocos da composição, e além disso é verificado se cada bloco referenciado é do tipo correto (isto é, possui a interface correta), e que qualquer outra exigência é satisfeita.

Alguns blocos, tais como aqueles que definem estrutura de arquivos, são persistentes e existem (em disco) sempre que o arquivo existir. Outrso blocos, tais como os que representam o estado de instâncias de arquivos abertos, são transitórios e existem apenas quando o arquivo está aberto. Esse blocos podem ser modificados em qualquer momento, por exemplo para adaptar-se a padrão de acesso de alguma aplicação específica.

Já que desempenho é o objetivo final desse trabalho, o custo adicional da arquitetura de blocos de contrução é uma preocupação importante. Se um arquivo for implementado usando muitas camadas de blocos "finos" e simples, então é natural suspeitar que o custo adicional de atravessar essas camadas seria um problema. Na prática, o custo foi considerado pequeno pelas seguintes razões:

1. Mesmo que o arquivo seja implementado com muitas camadas de blocos, requisições são geralmente satisfeitas atravessando um número pequeno de camadas. Muitos blocos apenas fazem *cache* dos dados, portanto as requisições poderão ser satisfeitas usando esse cache.

2. A implementação naturalmente reduz a quantidade de comunicação entre espaços de endereçamentos, já que a customização ocorre no servidor providenciando o serviço.

3. É possível definir interfaces de blocos de construção que possuem custo adicional muito baixo. As interfaces podem ser desenvolvidas sem que haja a necessidade de realizar cópias de dados.

4. Já que as funções e dados usados nos blocos são específicos à política que está sendo implementada, é comum conseguir eliminar condicionais que um mecanismo mais genérico exigeria. A ausência de condicionais abaixa o custo adicional e como os dados são específicos, e portanto, mais compactos, podem usufruir de efeitos de localidade (principalmente o *cache* de memória).

5. Se o uso de muitos blocos resulta em um custo adicional excessivo para um certa carga, então é trivial ao desenvolvedor do sistema definir e adicionar um novo tipo de bloco específico para essa carga, possivelmente combinando um conjunto de blocos em uma bloco mais complexo.

Com relação a arquitetura usada, o HFS é logicamente dividido em três camadas: a camada da aplicação, a camada física, e a camada lógica entre as outras duas. O Alloc Stream Facility (ASF) é uma biblioteca de E/S que compõe a camada da aplicação do HFS. A camada física que implementa arquivos e controla os discos do sistema, e a camada lógica do HFS que disponibiliza autenticação de serviços, são servidores de sistema do sistema operacional Hurricane. A seguir será delineado cada uma das três camadas.

Requisições de E/S feitas por aplicações são comumente gerenciadas por alguma biblioteca em espaço de usuário, como o *stdio*, que serve como intermediário entre a aplicação e os servidores de E/S do sistema. O ASF é uma biblioteca desse tipo, e foi desenvolvida usando o método de blocos de contrução. O ASF é um exemplo em que foi considerado vantajoso definir um número grande de blocos simples, ao invés de um número pequeno de blocos genêricos. A maioria dos blocos exportam e importam a mesma interface portanto existe muita flexibilidade nas composições. Essa interface comum, a *Alloc Stream Interface* (ASI) foi projetada cuidadosamente para minimizar a quantidade de dados copiados de um bloco para outro.

A implementação do ASF suporta três tipos de blocos de construção: (1) blocos ASI, (2) blocos específicos ao serviço, e (3) blocos para construção de interfaces.

Blocos de contrução ASI exportam e importam interfaces ASI, portanto eles podem ser conectados de diversas formas arbitrárias. Foram construídos blocos ASI para implementação de políticas para esconder latência, compressão/descompressão e "*advisory locking*" (para aplicações com muitas *threads*). Blocos de construção específicos ao serviço importam interfaces particulares ao serviço de E/S (por exemplo, terminal, arquivo, conexão de rede) and exportam interfaces ASI. Blocos para construção de interfaces importam interfaces ASI mas exportam uma interface padrão de E/S para a aplicação, tais como as interfaces (emuladas) padrões de E/S do Unix.

É importante notar que qualquer bloco de construção ASF pode ser mudado durante tempo de execução para se adaptar aos diferentes padrões de acesso que a aplicação pode manifestar durante diferentes fases de execução.

A camada física do HFS implementa arquivos e controla os discos do sistema. Todos os blocos de construção dessa camada importam e exportam a mesma interface, exceto aquelas que interagem diretamente com os discos. A interface usada aqui é similar à do ASI no sentido de que as operações de escrita e leitura não envolvem cópia de dados conforme o controle vai passando através dos blocos.

Foram implementados diversos blocos de construção que organizam os dados dos arquivos em disco. Alguns blocos que foram implementados incluem blocos para dados "fatiados", distribuição dos dados, otimizados para escrita, replicação dos dados, cálculo de paridade para tolerância a falhas, e para dados pequenos.

A camada lógica do HFS implementa todas as funcionalidades do sistema de arquivos que não precisam ser implementadas na camada física e, por razões de segurança e desempenho, não deveriam ser implementadas na camada da aplicação. A camada lógica difere das outras duas camadas pelo fato de ser menos flexível no composição dos blocos de construção.

Como técnica para desenvolvimento de um sistema flexível, composição de blocos de construção tem uma vantagem de engenharia de software, reduzindo em grande parte a complexidade do código. Uma medida de flexibilidade é a facilidade em adaptar o sistema a novos ambientes. O

ASF, biblioteca em nível de aplicação, foi portado para diversos sistemas com facilidade. Outra boa medida de flexibilidade do sistema é a facilidade de satisfazer exigências imprevistas por seus desenvolvedores. Inicialmente não houve intenção de fazer recuperação rápida de quedas, entretanto adicionar recuperação de quedas ao HFS exigiu pequenas modificações aos blocos que guardavam blocos de construção persitentes em disco.

O artigo faz um estudo bastante profundo sobre a medição de desempenho do HFS. Para isso foi descrito em detalhe a infraestrutura usada para a medição. Neste resumo grande parte dos detalhes foram omitidos, e resumimos apenas as partes interessantes e necessárias para o entendimento das conclusões descritas.

Os objetivos dos experimentos foi demonstrar que (1) é factível implementar um sistema de arquivos baseado em composições de blocos de constução, (2) o custo adicional no desempenho é baixo, e (3) o objetivo básico de flexibilidade é de fato importante.

Para medir o quantidade de banda de disco que o sistema de arquivos poderia disponibilizar a uma aplicação simples, criou-se um arquivo em um único disco. Foi, então, medido a quantidade de dados lidos desse arquivo por uma *thread*, que obteve 640 a 712 KBytes por segundo, o que corresponde a 100% da banda de disco usando o Hurricane.

Para demonstrar o custo adicional do sistema e concorrência, foi rodado um teste simples com n *threads* fazendo requisições repetidas a um arquivo para obter o seu comprimento. Quando todas as requisições envolvem o mesmo arquivo o sistema fica saturado por volta de 4 *threads*. É possível concluir desse experimento que é improvável que *locks* no sistema de arquivos vão estar concorrendo durante acessos comuns a arquivos, pois mesmo nesse caso em que todas as *threads* estão fazendo requisições para o mesmo arquivo o sistema de arquivos pode manejar mais de 50000 requisições por segundo. Essa taxa de requisições é uma ordem de magnitude maior do que a taxa máxima de requisições que o **disco** consegue responder.

Outro experimento foi realizado para demonstrar a importância de casar as necessidades da aplicação, especificamente com E/S paralela. Considere dois padrões de acessos de discos distintos; em um padrão A, sete *threads* de uma aplicação paralela lêem um arquivo concorrentemente, cada uma lendo sequencialmente de uma região **distinta** do arquivo, e em um padrão B essas sete *threads* cooperam para ler o arquivo todo sequencialmente, cada uma lendo uma pequena quantidade de dados por vez.

Claramente, em termos de estrutura do arquivo, para melhorar o acesso descrito pelo padrão A é melhor ter o arquivo particionado em sete regiões, cada uma guardada em um disco diferente, chamaremos essa estrutura de estrutura A. A estrutura mais adequada para o padrão B entretanto é "picar" o arquivo em pedacinhos e espalhar em todos os discos no sistema para balancear as requisições entre os discos, chamaremos essa estrutura de estrutura B.

A semelhança entre as estruturas A e B ilustram o poder de expressão da composição por blocos de construção. As duas composições apenas diferem em **um único** bloco, entretanto implementam estruturas de arquivos completamente diferentes.

Os resultados dos experimentos demostraram que quando a aplicação com padrão de acesso A é executada usando a estrutura B a aplicação recebe cerca de um quarta da banda se comparada com o caso em que a estrutura A é usada. Para o padrão de acesso B, a diferença de desempenho entre as duas estruturas de arquivos não é significativa, entretanto será mostrado que ao usar "*prefetching*" (leitura antecipada) dos dados a diferença se torna bastante significativa.

Se a banda completa dos discos no sistema deve ser entrega às aplicações, então o "*prefetching*" é necessário; o sistema de arquivos precisa gerar requisições para manter os discos ocupados a todo momento. No HFS isso é conseguido ao usar um bloco no AFS para realizar requisições de "*prefetch*" ao gerenciador de memória.

Com uma política simples de "*prefetching*" o padrão de acesso A usando a estrutura de arquivo A conseguiu uma banda de 4526 KBytes por segundo o que corresponde a 100% da banda disponível no sistema. Quando a mesma política é usada no padrão A usando a estrutura B a aplicação recebe apenas um quarto da banda de disco.

Para o padrão de acesso B na estrutura B usando uma política de "*prefetching*" agressiva, então novamente temos 100% da banda disponível à aplicação, enquanto que ao utilizar a estrutura A a aplicação recebe apenas um sétimo da banda.

A conclusão desse experimento é que é crucial que *as estruturas de arquivos e políticas de acesso casem com o padrão de acesso da aplicação.* Para os dois padrões descritos, A e B, o HFS conseguiu entregar para a aplicação 100% da banda do sistema.

Finalmente foi feito um experimento para medir o custo adicional que os blocos de construção poderiam introduzir ao sistema. Para demonstrar que esse custo adicional é baixo foi construído um arquivo distribuído que implementa a estrutura de arquivos A (descrito acima), mas adicionou-se 10 camadas de blocos de *distribuição*. O resultado demonstrou que essa estrutura conseguiu entregar apenas 1.2% menos banda do que a estrutura A original. Conclui-se que a impacto de implementar um arquivo que usam muitas camadas de blocos de construção é pequeno. O caso considerado é extremo já que com onze blocos de distribuição é possível criar um arquivo distribuído através de 8 trilhões de discos.

5.2 Artigo "File System Development with Stackable Layers"

Este artigo foi publicado na revista ACM – Transactions on Computer Systems (TOCS) 1994. Juntamente com a dissertação de mestrado do autor, esse artigo foi que primeiro introduziu a idéia de sistemas de arquivos empilháveis que ganhou bastante atenção nos últimos anos na área de sistemas de arquivos/sistema operacionais.

Tipo	Artigo publicado em revista
Autores	John S. Heidemann e Gerald J. Povek
Revista	ACM – Transactions on Computer Systems – 1994
Link	`http://fmg-www.cs.ucla.edu/classes/239_2.fall98/papers/stack.ps`

Resumo Sistemas de arquivos representam um dos importantes serviços de um sistema operacional. Tradicionalmente, os sistemas de arquivos têm sido integrados dentro do sistema operacional, e como resultado, a evolução desse serviço tem sido relativamente lento. Por exemplo, muitos sistemas de arquivos sendo utilizados em sistemas do tipo Unix são os mesmos desenvolvidos há mais de uma década. Esse cenário existe embora existam muitas melhorias em serviços de arquivamento conhecidas. Em contraste a isso, as aplicações em muitas áreas evoluíram rapidamente, dando muitos benefícios aos usuários. Esse atrofiamento na inovação e falta de evolução atrasa o acesso a uma série de benefícios aos usuários, e força , por exemplo, desenvolvedores de banco de dados a construir um serviço de arquivamento próprio, dentro de suas aplicações.

Existem vários motivos por essa situação adversa. Primeiro, é muito difícil para qualquer pessoa introduzir um sistemas de arquivos, ou inserir melhorias incrementais em um sistema operacional. Não existe nenhuma interface bem definida a ser utilizada. Embora alguns sistemas do tipo Unix usem uma interface como o VFS pela qual um sistema de arquivos *inteiro* possa ser instalado, na prática isso não tem funcionado bem. O VFS é inflexível ao abordar uma série de problemas, forçando desenvolvedores a estendê-lo de maneiras incompatíveis. Além disso, qualquer sistema de arquivos real é um serviço complexo que requer soluções funcionais para uma variedade de requerimentos, e portanto é um esforço excessivo sobre o desenvolvedor. Por fim, o serviço de arquivamento deve interagir com outros serviços essenciais do núcleo do sistema operacional, por exemplo, o gerenciador de memória virtual para compartilhamento de páginas e *buffers*.

Apesar desses problemas, muitos benefícios seriam disponibilizados se fosse possível adicionar serviços novos ao um sistema de arquivos já existente, de uma maneira simples e de maneira análoga à adição de serviços no nível de usuário. O objetivo desse trabalho é, portanto, construir uma infra-estrutura que torne tão fácil adicionar funcionalidades a um sistema de arquivo quanto é instalar uma nova aplicação em um computador pessoal.

Para isso, é necessário a criação de um ambiente no qual fragmentos de sistemas de arquivos podem ser montados de forma fácil e eficiente, e cujo resultado não compromete a funcionalidade nem o desempenho do sistema. O ambiente deve ser estendível para que ele perdure por gerações de sistemas computacionais. Operações de módulos criados no passado devem continuar a funcionar, e adições proprietárias devem conseguir coexistir com esse ambiente, alguns dos quais podem disponibilizar serviços que não foram previstos anteriormente.

Como proposta para solução das questões abordadas, o artigo apresenta uma interface e infra-estrutura para disponiblizar o desenvolvimento de *stackable file systems* (sistemas de arquivos empilháveis).

Stackable file systems constrói serviços de arquivamento complexos através de uma variedade de camadas desenvolvidas independentemente, cada uma potencialmente disponível apenas de forma binária. Novos serviços podem ser providenciados como camadas separadas; a divisão em camadas permite uma fronteira nítida para modificações.

As camadas são juntadas (ou "empilhadas"[8]) por uma interface *simétrica*, sintaticamente idêntica para cima ou para baixo da pilha de camadas. Devido a essa simetria não existem restrições sintáticas à configuração das camadas. Camadas podem ser facilmente removidas ou adicionadas à pilha do sistema de arquivos. Essa configuração do sistema de arquivamento não exige mudanças no núcleo, permitindo um ambiente propício à experimentação.

Como serviços novos comumente fornecem e exportam funcionalidades adicionais, é necessário que a interface entre as camadas seja *estendível*. Qualquer camada pode adicionar novas operações,

[8] *Empilhar* é na verdade um nome ruim, já que "pilhas" não lineares (como bifurcação em alguma camada, por exemplo) são comuns, mas o nome é mantido por razões históricas.

e as camadas existentes se adaptam automaticamente de modo a suportar essas camadas. Se a camada encontra uma operação que ela não reconhece, a operação pode ser redirecionada para a camada abaixo na pilha.

Os objetivos de interfaces simétricas e estensíveis podem parecer incompatíveis. Configuração é mais flexível quando todas as interfaces são literalmente sintaticamente idênticas, já que todas as camadas acessam os mesmos mecanismos. Mas estensibilidade implica que as camadas podem ter diferenças semânticas, restringindo como as camadas podem ser combinadas.

A extensibilidade da interface das camadas é conseguida através da utilização da interface desenvolvida pelos autores na UCLA. Isso é feito pela interface da UCLA já que ela mantém toda informação de definição de interface até o início da execução, e então constrói a interface dinamicamente. Cada sistema de arquivos providencia uma lista de todas as operações que suporta, que essas operações são guardadas em um vetor de operações. Quando uma camada adiciona uma operação nova, é esperado que as camadas que não implementam essa operação apenas repassem a operação à uma camada inferior na pilha.

Independência de espaço de endereçamento é outra característica importante no desenvolvimento de *stackable file systems*. As camadas muitas vezes executam num mesmo domínio de proteção, mas é vantajoso permitir as camadas rodarem em espaço de endereçamento distintos. Uma *camada de transporte* faz a ponte entre espaços de endereçamento transferindo todas as operações e seus resultados de um lado para o outro.

A artigo apresenta uma série de técnicas para desenvolvimento de sistemas de arquivos que foram permitidas ou simplificadas por empilhamento de camadas, que não serão apresentadas a fundo aqui.

Um dos objetivos do sistema de arquivos em camadas é permitir a construção de serviços de arquivamento complexos a partir de um número de camadas simples e desenvolvidas independentemente. A experiência dos autores mostra que camadas mais reutilizáveis e flexíveis são as que encapsulam apenas uma única abstração.

Uma vantagem de desenvolvimento através de *micro-kernel* é a habilidade de mover grandes porções de código do sistema operacional para fora do núcleo. Empilhamento de camadas se encaixa perfeitamente com essa abordagem. Cada camada pode ser considerada como um servidor, e operações como sendo mensagens RPC (Remote Procedure Calls) entre os servidores. Com essa infra-estrutura as camadas podem ser desenvolvidas e executadas como código em espaço de usuário. Apesar do RPC entre espaços de endereçamento ter um custo real, eventualmente alto, é possível fazer *caching* para a melhoria de desempenho, ou então a camada de transporte entre duas camadas pode ser removida sem afetar a implementação de nenhuma das camadas. Portanto, por separar os conceitos de modularidade e de proteção de espaço de endereçamento, empilhamento de camadas desfrutas das vantagens de desenvolvimento de *micro-kernel* e a eficiência de um ambiente de execução integrado.

Enquanto o desenvolvimento de um sistema de arquivos empilháveis oferece diversas vantagens, ele não será bem aceito se o custo adicional no desempenho é tal que os sistemas de arquivos monolíticos têm desempenho significativamente melhores. Para verificar o desempenho desse sistema, o custo adicional foi avaliado de diversos pontos de vista.

Uma das maneiras para medir o custo adicional para a realização das chamadas da nova interface foi a medição do número de instruções da linguagem asssembly que foram geradas pela implementação. Apesar de ser apenas uma indicação pouco precisa do custo real, essa medição pode prover uma comparação da ordem de magnitude. Os resultados indicaram que o custo de invocação de operações na interface *vnode* original e da UCLA são bastante parecidos. O "custo" de repassar uma operação para uma outra camada na pilha também foi medido, e foi medido que é necessário cerca de 54 instruções assembly. Apesar disso, o artigo argumenta que esse alto custo não é significativo com relação a uma operação média em um arquivo do sistema.

Contagem de instruções podem ser úteis, mas medidas do desempenho do sistema em execução é essencial para avaliá-lo. Para tal os núcleos contendo a interface da UCLA e o padrão foram comparados utilizando-se dois *benchmarks*: o Andrew benchmark e cópia recursiva de grandes árvores de diretórios. Além disso é examinado o efeito de adicionar diversas camadas na nova interface.

O Andrew benchmark tem diversas fases que tentam simular o padrão comum que um desenvolvedor faz ao sistema de arquivos. Nesses testes o sistema utilizando a interface UCLA obteve um custo adicional de apenas cerca de 1 a 2%.

Para exercitar de forma mais estressante a interface do sistema, foi examinado o tempo de cópia e remoção recursiva. Os resultados também foram animadores e indicaram um custo adicional de quase 2% para a cópia e pelo de 1% para a remoção.

Já que a filosofia de empilhamento de camadas estimula o uso de diversas camadas para implementar o que tradicionalmente é implementado em um núcleo monolítico, o custo de transações entre as camadas deve ser minimal. Para examinar o custo global do uso de um sistema de arquivos com múltiplas camadas, é analizado o desempenho desse sistema conforme o número de camadas utilizadas cresce. Duas conclusões foram deduzidas das experiências realizadas: primeiro, o tempo de espero pelo usuário é virtualmente imperceptível quando a carga é "padrão"; e quando é feito um uso "pesado" do sistema de arquivos e existem diversas camadas, o tempo do sistema[9] sofre uma pequena sobrecarga.

Por fim, o artigo relembra que um dos principais objetivo dessa infra-estrutura é facilitar o desenvolvimento de sistemas de arquivos. Para tal o artigo tenta analizar o custo de desenvolvimento de sistemas de arquivos com e sem a infra-estrutura de empilhamento de camadas. Essa análise foi feita das seguintes maneiras: foi medido o número de linhas do código necessário para implementar sistemas de arquivos muito simples (como um *loopback-fs*), foi medido o tempo de implementação de sistemas de arquivos por alunos de graduação da UCLA, e por fim foi analisado a criação de um sistema de arquivos distribuído com replicação desenvolvido pela UCLA, considerado um sistema robusto e bastante eficiente. Segundo o artigo esses experimentos indicam facilidade para criação das camadas do sistema de arquivos, possibilidade de reutilização de camadas, portanto minimizando futuro serviço, e uma melhor organização e modularização que auxilia na produção sistemas complexos.

A conclusão desse artigo foi que o uso da infra-estrutura para empilhamento de camadas no desenvolvimento de sistemas de arquivos simplifica tanto projetos grandes quanto pequenos projetos de serviços de arquivamento.

[9]Tempo gasto dentro do núcleo realizando tarefas para um determinado processo.

5.3 Artigo "Improving the Performance of Log-Structured File Systems with Adaptive Methods"

Este artigo foi apresentado na conferência ACM – Symposium on Operating Systems Principles (SOSP) 1997.

Tipo	Artigo apresentado em conferência
Autores	J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang and T. E. Anderson
Revista	ACM – Symposium on Operating Systems Principles – 1997
Link	`http://now.cs.berkeley.edu/Papers2/Abstracts/sosp97.ps`

Resumo Os projetos de sistemas de arquivos têm sido dirigidos por mudanças no custo e desempenho dos sistemas computacionais[10]. É necessário considerar o custo relativo de cada *byte* da memória versus do disco, o desempenho relativo do processador versus um acesso a rede versus um acesso a disco, a ordem de grandeza relativa dos tempo de procura, demora rotacional e banda de discos, sem contar com as mudanças na carga que o usuário coloca no sistema de arquivos.

Além disso, as constantes mudanças e melhorias que os sistemas computationais vêm sofrendo criam desafios para os desenvolvedores de sistemas de arquivos. Mesmo sendo possível implementar um sistema de arquivos que seja eficiente para os sistemas computacionais e com os padrões de carga observados no presente, implementações de sistemas de arquivos são comumente usadas durante muitos anos ou até mesmo décadas, muito tempo depois que as suposições sobre os sistemas computacionais feitas pelos desenvolvedores deixaram de serem válidas.

Nesse artigo é proposto e examinado um princípio de desenvolvimento para sistemas de arquivos, chamado de *self-tuning* (reconfiguração dinâmica). O sistema *self-tuning* (1) mede as características físicas da sistema computacional atual, (2) mede a carga colocada no sistema de arquivos, e (3) adapta o comportamento do sistema de arquivos para casar com as medições. A construção de um sistema *self-tuning* requer novos algoritmos que monitoram o ambiente e ajustam seu comportamento adequadamente.

O artigo explora o *self-tuning* via uma conjunto de quatro melhorias no projeto de um *log-structured file system* (LFS). Pesquisa em LFS tem sido um bom caso de estudo para a necessidade de métodos adaptativos já que demonstrou a dificuldade em projetar um sistema de arquivos eficiente para um larga gama de cargas de usuários, mesmo ao utilizar um técnica fixa.

No LFS, os dados do disco são organizados em um *log* segmentado, e que é escrito apenas ao final; escritas a disco são agrupadas no final do *log*. Periodicamente um coletor de lixo chamado *cleaner* localiza espaços livre no final do *log* e os agrupa para uma nova escrita. Essa arquitetura permite ao LFS fornecer um desempenho superior quando a carga é composta de freqüentes e pequenas escritas, e tráfego de leitura que é absorvida pelos *caches*. Entretanto, o LFS tem desempenho bastante inferior para outras cargas, quais como aquelas dominadas por atualizações aleatórias a um disco cheio com pouco tempo de ociosidade para a realização de limpeza.

Essa dicotomia no desempenho do LFS têm induzido os pesquisadores a concluir que o LFS é uma idéia interessante, mas impraticável. O artigo não deseja focar na comparação de LFS

[10] Nesse texto, sistema computacional será usado principalmente para denotar o conceito de *hardware*

com outros sistemas de arquivos, mas sim em como projetar um sistema de arquivos com bom desempenho através de uma larga gama de cargas e especificações de sistemas computacionais.

São feitas quatro contribuições para a melhoria do desempenho do LFS, e depois essas melhorias são avaliadas utilizando-se simulações *trace-driven* de cargas sintéticas e medidas.

Três das otimizações se concentram em melhorar o desempenho de escrita do LFS. Primeiro, é demonstrado como escolher o tamanho de segmento por trocar eficiência de transmissão com eficiência de limpeza do *log*. Segundo, é demonstrado como combinar a limpeza tradicional do LFS com um mecanismo de coleta de lixo alternativo chamado *hole-plugging*. Terceiro, é considerado como reduzir preço de limpeza tomando vantagem de dados em *cache*. Essas três otimizações no LFS podem melhorar as escritas por um fator de quatro em cargas de alto uso.

A otimização final visa melhorar o desempenho do LFS no caso da leitura que erra o *cache*. O trabalho original do LFS foi motivado pela predição de que sistemas computacionais no futuro teriam grandes *caches* de memória comparando com as capacidades dos discos, portanto menos leituras atingiriam o disco, entretanto essa tendência tem se revertido desde aquela época. Interessantemente, o LFS pode ser facilmente adaptado para melhorar o desempenho de leitura; o *cleaner* do LFS só possui um mecanismo para atomicamente mover dados no disco e para a criação de grandes regiões de espaço contíguo que pode ser agrupado com dados relacionados. É demonstrado que um reorganizador dinâmico do disco pode ser usado para melhorar o desempenho de leitura por um fator de quase dois para uma carga na qual os padrões de leitura não casam com os padrões de escrita.

5.4 Artigo "The Vesta Parallel File System"

Este artigo foi publicado na revista ACM – Transactions on Computer Systems (TOCS) 1996. Esse pequeno resumo foi usado como parte da proposta de de projeto para o pedido de bolsa da FAPESP, sendo incluso na seção "Trabalhos Relacionados".

Tipo	Artigo publicado em revista
Autores	Peter F. Corbett e Dror G. Feitelson
Revista	ACM – Transactions on Computer Systems – 1996
Link	`http://www.cs.huji.ac.il/~feit/papers/vesta.ps.gz`

Resumo O sistema de arquivos paralelo Vesta, desenvolvido pelo IBM para o sistema operacional AIX, tem como principal objetivo oferecer um sistema de arquivos com alta disponibilidade para sistemas maciçamente paralelos, com múltiplos processadores e discos. Como E/S de dados em discos é reconhecido como um gargalo em sistemas paralelos, alguns sistemas paralelos tentam resolver a situação usando alguns nós do sistema exclusivamente para E/S. Esses nós usam, entretanto, sistemas de arquivos baseados em sistemas para Unix, que são projetados e otimizados para uniprocessamento, e portanto apresentam diversos problemas quando utilizados em sistemas altamente paralelos.

O Vesta tenta resolver esse problema disponibilizando à aplicação uma interface para realizar um particionamento de seus arquivos através dos diversos discos disponíveis, conforme as necessidades da própria aplicação. A principal inovação do Vesta é que os arquivos têm estrutura com duas dimensões: uma dimensão representa o particionamento, e a outra os dados seqüenciais como em sistemas tradicionais.

Esse sistema oferece alto grau de paralelismo já que a aplicação pode decidir fazer o particionamento do arquivo de tal forma que cada partição do arquivo esteja em um nó de E/S distinto, evitando serialização dos acessos. Além disso o sistema disponibiliza alta escalabilidade já que não exige uma busca centralizada para encontrar o nó responsável por cada partição.

Um dos grandes problemas desse sistema, apontado pelos próprios autores, foi o abondono da compatibilidade com o Unix. A incompatibilidade com a semântica já estabelecida do Unix reduziu o interesse de usuários reais, já que exigiria um esforço adicional a todas as aplicações que usam o sistema, mesma aquelas que não desejam tirar proveito do particionamento de arquivos.

5.5 Artigo "Scability in the XFS File System"

Este artigo foi apresentado na conferência **USENIX 1996 Technical Conference**. Como o resumo anterior, foi usado como parte da proposta de de projeto para o pedido de bolsa da FAPESP, sendo incluso na seção "Trabalhos Relacionados".

Tipo	Artigo apresentado em conferência
Autores	A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto e G. Peck
Conferência	USENIX – Annual Technical Conference – 1996
Link	`http://www.xfs.org/papers/xfs_usenix/index.html`

Resumo O sistema de arquivos XFS foi desenvolvido pelo SGI, por volta de 1994, originalmente para o sistema operacional IRIX, mas atualmente disponível também para o Linux. A motivação para a criação desse sistema de arquivos foi a de resolver problemas relacionados a escalabilidade que os sistemas de arquivos na época, e ainda hoje, sofrem. Com esse intuito de escalabilidade, o XFS foi construído como um sistema de arquivos de 64 bits, o que facilita o suporte a sistema de arquivos muito grandes, arquivos muito grandes, esparsos ou contíguos, diretórios grandes, e um grande número de arquivos no sistema. A preocupação de utilizar a banda completa que um conjunto de discos oferece foi levada em conta.

Muitos algoritmos usados em sistemas de arquivos tradicionais foram modificados para permitir essa escalabilidade. Em particular, muitos sistemas de arquivos utilizam buscas lineares. O XFS, entretanto, faz uso extensivo de árvores B+ permitindo uma resposta rápida, mesmo para sistemas de arquivos complexos, já que reduz a complexidade de tais algoritmos de linear para logarítmica. Além disso, é um sistema de arquivos com *journalling* para recuperação rápida de quedas em sistemas de arquivos muito grande. O suporte a ACL (POSIX Access Control Lists) também foi incluído para um controle de permissões mais fino do que as permissões tradicionais do Unix.