GNU Debian

Francisco Anderson Bezerra Rodrigues, Marcelo Bulhões Fonseca, Vitor Silva De Deus

Departamento de Ciência da Computação,
Universidade de Brasília

I. Introdução

Debian é um projeto e um sistema operacional iniciado em 16 Agosto de 1993 por Ian Murdock[1]. Cada versão possui o nome de um personagem de Toy Story. É uma distribuição utilizada por várias organizações pelo mundo[2] graças à sua estabilidade, que o torna ideal para servidores.

II. Propósito do Debian

Debian é um sistema operacional com o objetivo de ser um sistema operacional completamente livre. No qual qualquer um pode baixar, modificar e compartilhar. De acordo com o projeto GNU [3], um software para ser considerado livre, deve permitir que seus usuários possuam todas as seguintes liberdades fundamentais:

- Liberdade para rodar o software da forma que o usuário quiser, e para qualquer propósito.
- Liberdade para estudar como o programa funciona, e poder editá-lo para que o programa compute da forma que o usuário quiser.
- Liberdade para redistribuir cópias.
- Liberdade para distribuir suas modificações para outras pessoas.

Para que essas liberdades sejam possíveis é necessário que o código fonte do software seja disponibilizado.

O Advanced Packaging Tool(APT) foi criado no projeto Debian e atualmente é usando em várias distribuições. Muitas delas baseadas no próprio Debian, como o Ubuntu e o Mint(que também possui uma versão baseada no Ubuntu).

III. Requisitos de Aplicações

IV. Linux

A. Arquitetura

O Linux é um kernel monolítico criado por Linus Benedict Towards em 1991.[4] Enquanto o GNU Project ficava fazendo tentativas de criar um kernel não monolítico(o que resultou no Hurd) o Linus conseguiu criar um kernel monolítico que era compatível com o sistema GNU. Desde então quase todos os esforços de desenvolvimento do kernel do sistema operacional GNU se voltaram para o Linux.

B. Gerência de Memória

O linux usa segmentação-paginação para alocação de espaço em memória. Em sistemas Alpha AXP usa páginas de 8 kbytes e em sistemas Intel x86 usa páginas de 4kbytes. A cada uma dessas páginas é dado um único número, o PFN(page frame number)[5].

Um endereço de memória virtual é composto de duas partes. Um offset e o numero de frame da página virtual. Cada processo possui uma tabela de páginas. Cada entrada nessa tabela contém as seguintes informações:

1) flag de validade:indica se a entrada é válida
2) O frame referente a pagina dessa entrada
3) Informação de controle de acesso. Como a página deve ser usada e etc

[5]

Quando é necessário alocar novas páginas e não se tem espaços vazios o SO deve definir uma política para desalocar alguma página e colocar outra no lugar. O linux usa a política LRU(Least Recently Used). Nesse esquema, páginas pouco acessadas são boas candidatas para troca(swapping)[6].

C. gerência de E/S

A getencia de entrada e saída no linux é realizada por um escalonador chamado CFQ(Complete Fair Queuing I/O Scheduler). O CFQ trata requisições de IØbaseado no processo que originou essa requisição, ou seja, cada processos possui filas independentes. As filas são ordenadas por tipo de requisição, requisições semelhantes ficam próximas O algoritmo de escalonamento utilizado nas filas é o round robin[**?**].

D. gerência de Processos

Os primeiros escalonadores do linux eram bastante simples, sem foco em arquiteturas massivas ou com muitos processadores multithread. O escalonador do linux 1.2 usava uma fila circular com Round-Robin. No linux 2.2 foi inserido outro escalonador com políticas de escalonamento para tarefas de tempo real, tarefas não preemptivas e tarefas de tempo não real, ou seja, classes de escalonamento. Além disso também foi incluído suporte para SMP(symmetric multiprocessing)[7].

No kernel 2.4 foi inserido outro escalonador relativamente simples com complexidade O(n). Ele era relativamente ineficiente, limitava escalabilidade e era lento para processos de tempo real. Também se mostrou ruim para arquiteturas de

multiprocessador. No kernel 2.6, introduziu-se o chamado de escalonador O(1), por ter complexidade O(1). Esse possuía suporte a SMP e cada fila de execução possuía dois arrays de prioridade.

O array de tarefas ativas e o de expiradas. No caso, cada array contém uma fila de processos por nível de prioridade. Essas filas contém listas de processos prontos em cada nível de prioridade. Os arrays de prioridade também contém mapas de bits para para descobrir eficientemente a maior prioridade entre as tarefas prontas no sistema sem precisar percorrer as filas. Achar a próxima tarefa a ser executada é simplesmente selecionar o próximo elemento nas listas.

Dada uma prioridade, as tarefas são escalonadas com Roud Robin. Mantêm-se dois arrays de prioridade para cada processador, um para tarefas ativas e outro para expiradas. O de expiradas contém todas as tarefas da lista de execução associada que já exauriram o temporizador. Quando uma tarefa esgota o seu temporizador um novo tempo é atribuído antes dessa tarefa ser movida para o array de expirados. Recalcular todas as temporizações é tão simples quanto trocar o array de expirados pelo de não expirados porque os arrays são acessados apenas por ponteiro. Trocar os arrays é tão rápido quanto trocar dois ponteiros[8].

O escalonador O(1) era muito mais escalável e incorporou interatividade com várias métricas heurísticas para determinar se as tarefas eram IO-bound ou process-Bound. Apesar das vantagens o escalonador O(1) tornou-se pesado no kernel. A grande massa de código necessária para calcular heurísticas tornou-se difícil de manejar.

A nova mudança veio por meio do algoritmo RSDL(Rotating Staircase Deadline Scheduler), proposto por Con Kolivas e foi incorporado a versão 2.6.21 do kernel linux. Depois, Ingo Molnar, o criador do escalonador O(1) desenvolveu um algoritmo CFS baseado nas mesmas ideias do trabalho de Kolivas. A principal idéia por trás do CSF é manter a justiça no provimento do processador aos processos. Para manter esse balanceamento, a quantidade de tempo designada a uma tarefa no CFS é chamada de virtual-runtime. Quanto menor for o "virtual-runtime"de uma tarefa maior a necessidade que essa tarefa tem de ser executada.

O CFS também possiu uma politica de justiça processos bloqueados(sleeper fainess) para processos que estão parados esperando por I/O tenham algum tempo para execução quando precisarem, por exemplo. Entretanto, ao invés de manter as tarefas em uma fila de espera, o CFS mantem uma árvore "red-black"ordenada por tempo.

Uma árvore "red-black"é uma árvore auto-balanceável, as operações na árvore ocorrem com complexidade O(log n), onde n é o número de nós da árvore. Nessa estrutura, tarefas com maior necessidade do processador(menor virtual-runtime) são armazenadas no lado esquerdo da arvore, e tarefas com menor necessidade de serem executadas do lado direito [7]. O escalonamento ocorre como a seguir:

1) O nó mais a esquerda da árvore é escolhido e enviado à execução.
2) Se o processo conclui sua execução ele é removido do sistema e da árvore
3) Se o processo alcança seu tempo máximo de execução(quantum) ou é bloqueado(voluntariamente ou por interrupção), ele é reinserido na árvore baseado em seu novo "virtual-runtime".
4) O novo nó mais a esquerda será escolhido repetindo a iteração[9]

Outra alternativa moderna ao escalonador O(1) é o BFS(Brain fuck scheduler). Também desenvolvido por Con Kolivas, esse escalonador evita o uso de configurações e heurísticas Em alguns benchmarks apresenta desempenho levemente ou moderadamente maior que o CFS mas não chegou a tomar o seu lugar. Está presente em distribuições como Sabayon Linux[10]. [9]

E. Funcionamento de interrupções

Uma das principais tarefas do gerenciador de interrupções do linux é encaminhar as interrupções para o trecho de código correto no próprio gerenciador. Esse código deve entender a topologia do sistema. O linux utiliza um conjunto de ponteiros para estruturas de dados que contém os endereços das rotinas que tratam cada interrupção. Essas rotinas pertencem aos "drivers"dos dispositivos no sistema e é responsabilidade de cada driver requisitar a interrupção correta. Um vetor de ponteiros chamado irq_action contém as referências para as estruturas de dados citadas anteriormente, as quais se chama irqaction. Cada irqaction contém informações sobre o gerenciamento dessa interrupção, incluindo o endereço da rotina correspondente de gerenciamento da interrupção. Visto que o número de interrupções e como as mesmas são gerenciadas varia entre as arquiteturas e sistemas, o gerenciador de interrupções do linux é dependente disso. Isso significa que o tamanho do vetor irq_action varia dependendo da arquitetura ou sistema.

Quando a interrupção acontece, o Linux deve determinar a sua origem lendo o status da interrupção armazenado em um registrador em algum controlador programável de instruções de interrupções(isso é dependente de sistema e arquitetura). Depois traduzir a origem em uma posição do vetor irq_action. Caso não exista alguma posição referente a determinada origem o kernel retorna um erro, caso contrário irá chamar a rotina contida em irqaction na dada posição de irq_action. Ao fim, o driver do dispositivo é responsável por informar o kernel se a operação foi concluída com sucesso ou não[11].

F. Suporte a Threads

O linux dá suporte a threads tratando-as como processos e escalonando-os juntamente. Um processo pode ser visto como uma thread mas processos podem contem múltiplas threads que compartilham seus recursos[7].

G. Segurança

V. FreeBSD

A. Arquitetura

B. Gerência de Memória

O FreeBSD usa uma política de troca de páginas ao estilo Last Recent Used(LRU)[12]

C. gerência de E/S

Para gerenciar o acesso a dispositivos de entrada e saída, o FreeBSD emprega o C-LOOK scheduler[13]. Que também é chamado de escalonamento "elevador", que trata os cilindros do disco como se fossem andares de um prédio. Porém a a variação que esse elevador funciona de forma circular para equalizar a frequência com a qual cada cilindro/andar é visitado.

D. gerência de Processos

A gerência de processos do FreeBSD tem como objetivo priorizar processos interativos sobre processos de processamento contínuo. Isso é feito atribuindo prioridades a processos. Os processos são computados em time slices, os processos que usam todo o time slice têm sua prioridade diminuída enquanto processos que são bloqueados(quesitam i/o) durante seu time slice possuem suas prioridades mantidas. Processos que fazem tempo que não são rodados têm sua prioridade aumentada para evitar starvation. O FreeBSD também dá suporte a aplicações de tempo real, mantendo essas aplicações numa fila diferente da fila dos outros processos.[14]

E. Funcionamento de interrupções

F. Suporte a Threads

G. Segurança

VI. Hurd

A. Arquitetura

O kernel GNU/Hurd possui a arquitetura cliente/servidor e possui o microkernel GNU/Mach que contém instruções de modo kernel. No Hurd, pode-se atrelar à arquivos, máquinas de tradução. Que podem prover diversas funcionalidades, como por exemplo implementar o protocolo ftp de forma transparente, prover gerenciamento de histórico de arquivos entre outras coisas muito interessantes.[15]

B. Gerência de Memória

C. gerência de Processos

D. gerência de E/S

No GNU/Hurd as operações de entrada e saída são feitas com o uso de portas de comunicação que implementam pelo menos o protocolo de arquivo ou o protocolo de socket. As operações são realizadas a partir de chamadas de procedimento remoto(RPC's) nessas portas de comunicação[16].

E. Funcionamento de interrupções

F. Suporte a Threads

O GNU/Hurd faz uso agressivo de Threads como o objetivo de extrair o máximo do hardware.[17] Todas as bibliotecas do GNU/Hurd são thread-safe, ou seja, chamadas concorrentes aos serviços da biblioteca não fazem ela se comportar de forma inesperada.

G. Segurança

Um dos objetivos do Hurd é minimizar a quantidade de código que é executada no modo kernel através do uso de microkernel GNU Mach. O que aumenta a confiabilidade do sistema como um todo.

Referências

[1] A brief history of debian. https://www.debian.org/doc/manuals/project-history/index.en.html#contents. Acessado em 16/11/2016.

[2] Who's using debian? https://www.debian.org/users/. Acessado em 16/11/2016.

[3] Freesoftware. https://www.gnu.org/philosophy/free-sw. Acessado em 16/11/2016.

[4] History of linux. https://en.wikipedia.org/wiki/History_of_Linux. Acessado em 20/11/2016.

[5] Chapter 3 memory management. http://www.tldp.org/LDP/tlk/mm/memory.html. Acessado em 20/11/2016.

[6] Cfq. https://en.wikipedia.org/wiki/CFQ. Acessado em 20/11/2016.

[7] Inside the linux 2.6 completely fair scheduler. https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/. Acessado em 19/11/2016.

[8] The linux process scheduler. http://www.informit.com/articles/article.aspx?p=101760&seqNum=2. Acessado em 19/11/2016.

[9] Completely fair scheduler. https://en.wikipedia.org/wiki/Completely_Fair_Scheduler. Acessado em 19/11/2016.

[10] Brain fuck scheduler. https://en.wikipedia.org/wiki/Brain_Fuck_Scheduler. Acessado em 19/11/2016.

[11] Chapter 7 interrupts and interrupt handling. http://www.tldp.org/LDP/tlk/dd/interrupts.html. Acessado em 20/11/2016.

[12] Freebsd architecture handbook. ftp://ftp.freebsd.org/pub/FreeBSD/doc/en_US.ISO8859-1/books/arch-handbook/book.pdf. Acessado em 18/11/2016.

[13] Hibrid - freebsd wiki. https://wiki.freebsd.org/Hybrid. Acessado em 20/11/2016.

[14] The design and implementation of the freebsd operation system. http://ptgmedia.pearsoncmg.com/images/9780321968975/samplepages/9780321968975.pdf. Acessado em 20/11/2016.

[15] Towards a new strategy of os design, an achitectural overview by thmos bunshnell, bsg. https://www.gnu.org/software/hurd/hurd-paper.html. Acessado em 17/11/2016.

[16] Hurd reference manual: 4. input and output. https://www.gnu.org/software/hurd/doc/hurd_5.html. Acessado em 17/11/2016.

[17] Hurd. www.gnu.org/software/hurd/doc/hurd.ps. Acessado em 16/11/2016.