Mac OS X

Felipe Gomes Lacerda 06/83884
Pedro Garcia Freitas 04/35597

23 de novembro de 2009

1 Kernel

O Mac OS X é um sistema operacional formado de vários componentes projetados e implementados pela Apple. Embora o sistema seja em maior parte fechado, possui uma grande parte do código aberto, denominado Darwin.

O kernel do Mac OS X, chamado XNU (X is Not Unix), foi originalmente desevolvido pela extinta empresa NeXT para ser um kernel híbrido, combinando o Mach, um microkernel de pesquisa desenvolvido na universidade de Carnegie-Mellon, com componentes do BSD 4.3. Quando a NeXT foi comprada pela Apple, o XNU foi herdado para o novo sistema da empresa, onde recebeu uma atualização para o Mach 3.0 e os componentes BSD foram substituídos pelo do FreeBSD [7], resultando no Mac OS X.

O Mach 3.0 foi um microkernel para sistemas operacionais desenvolvido para ser simples, extensível e servir outros sistemas operacionais, separando suas funcionalidades daquelas que deveriam executar em modo não-privilegiado. Todavia, diferente do Mach, o XNU não separa totalmente as suas funcionalidades em diferentes permissões de execução, deixando todos os seus componentes em modo privilegiado. Desta forma, ele modulariza as diferentes classes de funcionalidades do kernel, tornando-se mais robusto que um kernel monolítico, sem perder tanta performance pelo overhead na comunicação de diferentes camadas, como acontece em um microkernel puro. Esta organização pode ser visualizada na figura 1

Sendo assim, dentro destas duas camadas é possível encontrar os componentes que compõem a arquitetura do kernel, tais como sistema de arquivos, gerência de dispositivos, comunicação entre processos etc. Cada camada pode ser vista da seguinte maneira:

- Mach: serviços críticos—aqueles que ficariam em modo protegido em um microkernel real.
- BSD: camada que provê interface de comunicação com aplicativos (syscalls) e os serviços que ficariam em modo usuário em modelo de microkernel real.
- I/O Kit: ambiente de desenvolvimento e execução para comunicação com dispositivos

1.1 Camada Mach

Esta camada provê as funcionalidades que um sistema microkernel forneceria, se a camada BSD estivesse em espaço de usuário. Pode-se pensar na Mach como o núcleo

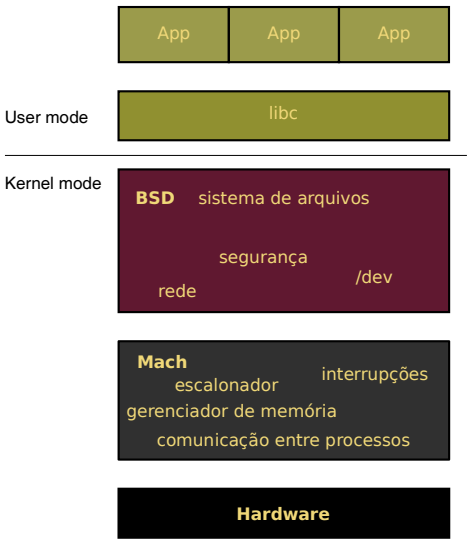

Figura 1: Estrutura do kernel

do kernel, pois provê os serviços de baixo nível de forma totalmente transparente para as aplicações do usuário. Tal como um microkernel, a base de seu funcionamento se dá por comunicação por troca de mensagem, provendo apenas as funcionalidades de gerência de processos, comunicação entre processos, gerência e suporte à memória virtual e tratamento de interrupções do hardware.

Para atender a estes serviços, a Mach possui as seguintes abstrações: portas, tasks, threads, espaço de endereçamento e mensagens. É utilizando estas abstrações que a camada opera suas atribuições escalonando processos, controlando a comunicação entre processos, gerencia de memória e relógio.

1.2 Camada BSD

Embora divida o espaço de endereçamento com a Mach, esta camada está um nível de abstração acima. É ela que é visível diretamente para o aplicativo usuário, fornecendo os recursos gerenciados pelo sistema operacional através das syscalls. Baseada no FreeBSD, esta camada provê gerência para os sistemas de arquivos (incluindo VFS), protocolos de redes, modelo de segurança UNIX—controle de permissões de escrita, leitura ou arquivos—, modelo de processos BSD, API POSIX, o acesso multiusuário e proteção de memória em nível de processo.

A camada BSD comunica-se com a Mach, servindo de interface nas ações de entrada e saída em dispositivos, operações no sistema de arquivos e comunicação entre processos de forma transparente.

2 Processos

Um sistema operacional típico representa um programa em execução como um processo. Esta abstração é utilizada tanto pelo usuário quanto pela maioria dos sistemas operacionais ao gerir recursos de hardware para os programas.

Contudo, como o kernel do Mac OS X é formado por duas camadas de abstração interna—BSD e Mach—, o programa executado também terá dois níveis de abstração respectivos à cada camada.

A primeira abstração é aquela mais próxima do usuário, feita na camada BSD que cria o ente "processo" que possui os atributos seguindo o padrão POSIX [1]:

A segunda abstração de programas deve ser feita na camada Mach. Para isto, a Mach cria uma estrutura chamada "task".

Esta estrutura serve para conter e encapsular o acesso à um espaço de endereçamento virtual referente à um processo da camada BSD, além de servir como domínio comum às threads do programa. A estrutura task em termos de código-fonte [2], embora seja extenso e formado por estruturas mais complexas que o processo, possui um atributo fundamental que é o ponteiro de referência ao processo equivalente da camada BSD.

As threads da camada Mach são as linhas de fluxo dentro de uma task. Elas são fundamentais para o escalonador de processos, pois são com elas que ele trabalha. Quando uma task é criada ao menos uma thread é iniciada. Então, o kernel endereça a referencia desta thread para o escalonador. Sendo assim, uma thread deve conter as informações:

- prioridade de escalonamento
- política de escalonamento
- estatistica de uso do processador
- direito de porta: para comunicação e compartilhamento de recursos entre threads

2.1 Escalonador de processo

O escalonamento fica a cargo da camada Mach e é feito sobre as threads. Portando, o XNU não possui "escalonamento de processos" e sim "escalonamento de threads".

Esta abordagem possui a vantagem de permitir que threads diferentes possam ser distribuídas para diferentes processadores em sistemas com vários núcleos. Além disso, ainda possibilita que o overhead interno de um processo, causado pela troca de contexto das threads, seja minimizado.

O algoritmo de escalonamento das threads utilizado pelo XNU é composto de múltiplas filas, onde cada uma possui uma prioridade diferente. Cada nível de prioridade é classificado em 4 grupos básicos, sendo que cada grupo recebe uma faixa de pontuação por prioridade [5]: Normal (0-63), alta prioridade (64-79), modo kernel (80-95) e prioridades de tempo real (98-127). Portanto, existe um total de 127 tipos de prioridades, significando que o escalonador tem que manter esta quantidade de filas.

Toda vez que uma thread ganha tempo de CPU, ela perde 1 ponto de prioridade. Por exemplo, se uma aplicação com nível de prioridade normal inicia possuindo prioridade 63, quando ela recebe a posse da CPU, cai para o nível 62. Sendo assim, cada vez que uma thread entra em estado de espera ela cai para a fila de menor prioridade.

Para evitar que as prioridades das threads fiquem indiferenciáveis após um tempo, cada tipo de thread (normal, de sistema, modo kernel ou tempo real) deve possuir uma prioridade base e outra máxima, criando bandas de prioridades para cada tipo.

Portanto, o agrupamento por banda de prioridades é necessário para garantir que uma thread que realiza uma operação crítica não fique prejudicada em favorecimento de outra que não o faz. Desta forma, uma thread do kernel sempre terá prioridade sobre uma de prioridade normal, garantindo bom funcionamento dos serviços do sistema operacional.

Embora o principal fator para a mudança de nível da thread seja a posse da CPU, o escalonador também utiliza outros fatores para determinar sua prioridade.

Como dito anteriormente, um dos atributos que a thread possui está relacionado com seu uso do processador. Esta informação permite que o escalonador verifique a demanda dos recursos necessários pela thread. Sabendo disto, é possível classificar se a thread é CPU-bound ou IO-bound, por exemplo. Dependendo da sua classe, é possível redefinir sua prioridade, privilegiando aquelas que necessitam muito mais da CPU do que aquelas que ficam em espera ocupada.

Escalonando as threads alterando suas prioridades gera alguns problemas. O primeiro deles é que, dependendo de como ela se comporta, pode tender a ser mais favorecida que outras, levando aquelas de prioridade mais baixa tender a ficar em starvation. Para contornar este problema, o escalonador analisa o "envelhecimento" da thread[1] e aumenta a sua prioridade, tornando-a candidata à posse da CPU.

Porém, a solução acima gera um novo problema: para um sistema com grande demanda de carga—muitas threads—, pouco tempo de processador vai ser dedicado a cada uma. Isso faz com que a prioridade de todas as threads tendam a aumentar. Assim, a disputa irá deteriorar o desempenho geral do sistema. Para amenizar este overhead, o XNU utiliza-se de um fator de conversão de carga para redistribuir as prioridades das threads [4].

3 IPC: Comunicação entre procesos

A comunicação entre processos é um mecanismo que permite a transferência de informação entre dois programas diferentes. Historicamente, o IPC surgiu para permitir a associação de recursos comuns à diferentes processos, de forma compartilhada. Contudo, para o Mac OS X, este recurso é fundamental para transferência de dados utilizados pela camada Mach. Portanto, é um conceito fundamental para o kernel pois é via IPC que ele se comunica com entidades como tasks, threads e outros elementos.

Por isso, naturalmente, o Mac OS X necessita de uma grande quantidade de mecanismos de IPC, distribuídas em diferentes camadas do sistema. Os mecanismos mais

[1] Calculado a partir do tempo atual (`system_timer`) - último instante marcado pelo escalonador (`sched_stamp`) [3].

comuns de IPC do sistema são: Mach IPC (sendo a abstração de nível mais baixo, cujos outros mecanismos de IPC são reduzidos), as exceções na camada Mach, Unix signals, pipes, POSIX IPC e os mecanismos de notificação[2].

3.1 Mach IPC

A camada Mach realiza todas suas atividades de forma orientada a mensagens. As entidades gerenciadas pelo kernel através de duas abstrações: ports e messagens. A primeira é uma entidade utilizada para referenciar os objetos comunicantes, enquanto que a segunda é uma coleção de dados a serem transferidos.

O tratamento dos IPCs nesta camada é feito utilizando um subsistema virtualizado que permite os dados sejam compartilhados de forma eficiente através de otimizações por copy-on-write (ver seção 4).

3.2 Mach Ports

Uma port é um elemento fundamental para permitir que a intercomunicação ocorra, além de agregar um conjunto de funcionalidades para o XNU.

A primeira é que ela serve como um canal de comunicação protegido e gerenciado pelo kernel, mantendo uma fila das mensagens envolvidas na comunicação. A operação mais básica de uma porta é a de enviar e receber mensagens.

A porta é usada para representar recursos e serviços, provendo acesso aos objetos. Por exemplo, Mach usa as portas para representar hosts, tasks, threads, objetos de memória, clocks, timers, processadores etc. Por isso, todas operações realizadas por esta camada são orientadas a portas.

Uma porta também representa as capacidades que a entidade que elas representam e servem para indicar os direitos de acesso aos recursos compartilhados. Para acessar uma porta, uma task necessita ter um direito de operação: enviar ou receber. Como ela funciona como um canal de comunicação, uma porta Mach assemelha-se com um socket BSD, mas com algumas diferenças:

- é integrada com o sistema de memória virtual
- Mach IPC visa comunicação interna entre os processos, sendo transparente através da rede
- as mensagens carregam conteúdo tipado

4 Memória

O Mac OS X possui um sistema de gerenciamento de memória flexível baseado em paginação. O sistema é dividido em duas partes:

- Um subsistema dependente de arquitetura. Esse módulo, chamado `pmap` (*Physical Map*), encapsula o código que manipula a MMU e fornece funções para o resto do sistema.
- Um módulo independente de arquitetura, que é responsável por gerenciar estruturas como *VM maps*, páginas e *VM objects*. Essas estruturas serão explicadas com mais detalhes adiante.

Cada task possui um espaço de endereçamento. No Mac OS X, esse espaço de endereçamento é em geral muito maior que o espaço ocupado pela task em si. Numa máquina de 32 bits, o espaço é de 4 gigabytes; para 64 bits, o espaço é de mais de 2 petabytes. Esse espaço de endereçamento normalmente é alocado de maneira esparsa—isto é, a task ocupa várias regiões contíguas, separadas por grandes regiões não usadas.

Quando uma task é criada, o kernel cria para ela uma *VM map*. Essa estrutura mantém uma lista de regiões de memória que a task ocupa, além de uma estrutura do tipo *pmap*, que mapeia as regiões à memória física. Cada uma dessas regiões possui um *VM object*, que mantém uma lista de páginas residentes (na memória física) e páginas que estão na memória secundária. Para pegar de volta essas páginas (page-in), a VM object também possui um *memory object*, que é uma porta utilizada pelo kernel.

No caso do Mac OS X, não é o kernel em si que realiza o page-in. Ele faz isso se comunicando com um pager, que é uma estrutura que pode existir em espaço de usuário. Quando uma falha de página ocorre, o kernel manda uma mensagem ao memory object, que pertence ao pager. O pager então recebe os dados de volta e os manda para a porta de controle do memory object, que também é mantida pela VM object.

O page-out é realizado de maneira similar. Quando não há mais memória física disponível, o kernel escolhe, usando o algoritmo da segunda chance, uma página a ser removida. Ele então envia uma mensagem ao pager, que salva a página na memória secundária.

O sistema de gerenciamento de memória do Mac OS X possui mais uma peculiaridade. Ele implementa uma técnica de otimização, chamada *copy-on-write*, que funciona da seguinte forma. Quando uma VM object é copiada, o sistema não faz a cópia efetiva. Em vez disso, as páginas ficam compartilhadas. A cópia só é feita de fato quando houver escrita em uma das páginas.

Para realizar isso, a VM object possui mais três estruturas não mencionadas anteriormente: um ponteiro para um shadow object, um para um copy object e um contador de referências. Quando uma VM object é copiada, é criado um copy object. Seu campo copy passa a apontar para a VM object original. O campo shadow da VM object original passa a apontar para o copy object. Quando ocorre uma modificação numa página da VM object original, ela é primeiro copiada para uma nova página e depois enviada para o copy object.

A maneira como a paginação é estruturada no Mac OS X permite que vários pagers sejam usados para diferentes situações. De fato, ele até permite que um usuário desenvolva seu próprio pager. Mas os desenvolvedores do Mac OS X resolveram não dar essa flexibilidade ao usuário. Ainda assim, o sistema possui três pagers: um para transferir dados entre a memória física e a memória swap, outro para transferir dados entre a memória física e arquivos, e outro que lida com a memória utilizada por dispositivos.

[2]Notificação é dependente do contexto. Por exemplo, uma thread pode notificar à Mach que outra thread está sendo criada, o que faz com que esta camada crie uma Mach port e notifique as outras.

5 Entrada e saída

O Mac OS X fornece um ambiente para interface com dispositivos de entrada e saída, além de uma arquitetura para o desenvolvimento de drivers. O conjunto ambiente e arquitetura é chamado de *I/O Kit*.

A arquitetura da I/O Kit fornece uma interface orientada a objetos em um subconjunto de C++ (*Embedded C++*) para modelar a hierarquia de hardware em software. A funcionalidade comum a diferentes dispositivos é encapsulada em classes (também chamadas de *famílias*), que são herdadas pelos drivers desses dispositivos. Por exemplo, toda controladora SCSI deve fazer coisas como verificar o barramento SCSI; em vez de ter que replicar essa funcionalidade em todos os drivers para controladoras SCSI, basta fazer os drivers herdarem da família de controladoras SCSI.

Em geral, um driver não se comunica diretamente com o hardware mas participa de uma hierarquia de classes que inclui outros drivers. Para comunicação entre drivers, classes chamadas *nubs* são usadas. Um exemplo seria um driver para uma controladora Ethernet que usa o barramento PCI. A camada de nível mais alto é a classe `IONetworkStack`, que implementa a interface para a pilha de rede do Mac OS X, presente na camada BSD. Essa classe utiliza o nub `IOEthernetInterface` para se comunicar com o driver da controladora. Esse driver herda da família `IOEthernetController`. Para usar o barramento PCI, o driver usa o nub `IOPCIDevice` para se comunicar com o driver `IOPCIBridge`.

A I/O Kit também fornece um ambiente de execução protegido para drivers. Esse ambiente é baseado na idéia de que drivers podem ser dinamicamente carregados e descarregados. Assim, todo driver deve gerenciar seu ciclo de vida. Ainda, quando um dispositivo é encontrado, o sistema deve procurar um driver que corresponda àquele dispositivo. Essa função é também realizada pelos nubs.

No exemplo do driver da controladora Ethernet, o driver `IOPCIBridge` constantemente verifica o barramento procurando por um novo dispositivo. Quando a placa Ethernet é inserida, o driver usa o nub `IOPCIDevice` para encontrar um driver adequado para o dispositivo.

I/O Kit fornece ainda uma interface para controle de dispositivos em nível de usuário. Isso permite que o uso de memória do kernel seja minimizado, o que é importante porque o kernel deve sempre estar na memória física. Essencialmente, o usuário especifica em seu programa que dispositivo ele deseja acessar. O kernel então procura o driver correspondente (usando os nubs) e executa a função requisitada de uma maneira controlada. Assim, é como se o usuário pudesse escrever seus próprios drivers—mas em todo caso, deve haver suporte do sistema operacional para as funcionalidades desejadas.

6 Sistemas de arquivos

O Mac OS X possui uma interface para sistemas de arquivos que é conceitualmente similar ao que é feito em outros sistemas Unix. A abstração para o usuário baseia-se em *descritores de arquivos* (ou `fds`), que são números visíveis aos programas do usuário e usados pelo kernel para manipular arquivos. Para cada processo, o kernel mantém uma lista de descritores de arquivos correspondentes aos arquivos abertos pelo processo. Quando o processo realiza uma chamada de sistema passando um descritor de arquivo para ela, o kernel a estrutura representando o `fd`. Essa estrutura guarda um ponteiro para um `vnode`, do qual os dados do arquivo podem ser lidos—assim, eles funcionam de maneira aos *inodes* do Unix—assim como uma lista de operações que podem ser executadas no arquivo (como `read()` e `write()`).

Por que guardar a lista de operações na estrutura, em vez de simplesmente executar as operações diretamente? A resposta está na maneira como os sistemas de arquivo são organizados. O Mac OS X implementa a camada *VFS*, que é uma interface virtual para sistemas de arquivos. Essa interface, de forma análoga a uma linguagem orientada a objetos, define uma série de operações e estruturas que um sistema de arquivos deve possuir. A implementação interna do sistema de arquivos só deve ser conhecida quando chegamos ao nível do arquivo. Dessa forma, o sistema pode suportar diversos sistemas de arquivo de maneira uniforme—os programas não precisam saber sobre o que está "debaixo dos panos".

O VFS na verdade define duas abstrações: o `vnode` e o `mount`. Como vimos, o `vnode` contém as informações específicas de onde ler o arquivo no disco, além da lista de operações que o sistema de arquivos implementa para manipulação de arquivos. Por sua vez, o `mount` possui informações sobre o sistema de arquivos em si. Ela só existe quando o sistema de arquivos está montado[3]. Essas informações ponto de montagem, uma lista de operações que podem ser executadas no sistema de arquivos, a lista de `vnodes`, e uma estrutura de dados privada que indica onde o sistema de arquivos está no disco.

Antes de discutir o HFS+, o sistema de arquivos padrão do Mac OS X, precisamos discutir a forma como os dispositivos de armazenamento são manipulados. A forma é similar aos outros sistemas Unix. Todos os dispositivos de armazenamento são representados como um arquivo especial no diretório `/dev`. Por exemplo, um disco pode ser representado como o arquivo `/dev/disk0`.

Todo dispositivo de armazenamento é dividido em partições, que também são arquivos especiais. Essas partições são também chamadas de volumes. Um volume contém um sistema de arquivos. Para montar um volume, utilizamos o comando `mount`. Assim, para montar um volume do disco em `/dev/disk0` no ponto de montagem `/mnt`, usamos `mount /dev/disk0 /mnt`.

O Mac OS X permite definir o tamanho de um bloco no momento da criação do sistema de arquivos. O tamanho padrão é de 4 KB; esse tamanho costumar fornecer o desempenho ótimo. Normalmente, é alocado ao arquivo uma região contígua de blocos. Para aumentar a eficiência, o Mac OS X possui uma estrutura chamada *extent*, que define o número de blocos contíguos da região.

Vamos agora discutir a implementação interna do HFS+. Um volume HFS+ mantém várias estruturas, dentre as quais as mais importantes serão discutidas a seguir.

Uma estrutura usada é o *Allocation File*. Esse arquivo

[3] *Montar* é um jargão Unix para indicar ao sistema operacional que o sistema de arquivos pode ser usado, e associá-lo a um ponto sistema de arquivos já montado.

é um bitmap que mantém informações sobre os blocos em uso. Uma particularidade dessa estrutura é que ela é guardada como um arquivo regular no HFS+; ela também pode ser livremente redimensionada.

Arquivos e diretórios são representados por `cnodes`, que são entradas num catálogo (o *Catalog File*) mantido pelo sistema. Essas estruturas são semelhantes aos `inodes` do Unix. O catálogo é mantido como uma *árvore B* (ou B-Tree), uma estrutura que permite executar operações como busca, inserção e remoção de forma eficiente. É importante notar que os diretórios existem apenas na árvore B.

Como vimos antes, um arquivo possui *extents*, que são regiões contíguas de blocos no disco. Por razões de eficiência, os primeiros 8 descritores dos extents são guardados no próprio catalog file. Mas para arquivos com mais de oito extents, é usada também uma outra árvore B, o *Extents Overflow File*.

O HFS+ também possui uma estrutura para manter os atributos estendidos do arquivo. Essa estrutura é o *Attributes File* e também é uma árvore B. Um exemplo de atributo mantido nesse arquivo são os dados de ACL (*access control lists*), que é uma forma de controlar as permissões dos arquivos de modo mais flexível que o usado normalmente por sistemas Unix.

Outro recurso suportado pelo HFS+ é o *journaling*. Essencialmente, journaling é uma técnica que consiste em escrever a operação a ser realizada num log especial (o journal) antes de realizar a operação no arquivo. Dessa forma, se uma falha ocorre, a consistência do sistema de arquivos pode ser garantida. No caso do HFS+, isso é feito implementando operações da camada VFS para journaling. Assim, qualquer aplicativo que trabalhe apenas com o VFS e não algum sistema de arquivos específico funcionará com o HFS+.

7 Segurança

Em um sistema computacional, segurança é formada por práticas que permitem o controle de acesso aos dados somente pelos usuários devidos. Para isso, o sistema deve verificar as identidades dos programas dos usuários e dos serviços de sistema, salvaguardando informações sensíveis.

Os recursos—dados pessoais, de grupos ou do sistema—geralmente não são mantidos isolados. Ao contrário, os recursos gerenciados pelo sistema são compartilhados entre usuários na mesma máquina ou através de uma rede.

Quando um sistema operacional permite que um processo não autorizado acesse, altere ou corrompa os dados de um recurso, diz-se que este sistema possui uma vulnerabilidade. Estas podem ser exploradas através de várias técnicas, podendo provocar diversos tipos de danos, tais como modificação ou destruição de dados sensíveis, acesso não autorizado à um serviço de sistema ou degradação da operação do sistema operacional.

Para evitar que vulnerabilidades e falhas de segurança, o Mac OS X fornece recursos que podem ser classificados em duas classes: em nível de kernel e de usuário.

A primeira segue um modelo estilo Unix, possuindo:

- Identificadores de grupo BSD (UID e GID): O arquivo possui atributos que indicam a permissão das operações de leitura, escrita e execução que os usuários e grupos podem realizar.
- Direitos de portas da camada Mach: além de ser um canal de comunicação interprocessos, uma Mach port é usada para representar recursos tais como tasks, threads e dispositivos. Por isso, as operações realizadas por estas ports precisam ser controladas e geridas pelo kernel.
- Sistema de auditoria: API usada para manter um log dos eventos relacionados com falhas de segurança.
- Encriptação de memória virtual: O kernel permite que a memória virtual seja encriptada através do algoritmo AES (*Advanced Encryption Standard*.) [6]. Esta abordagem evita os dados sejam roubados e dificulta a engenharia reversa através de uma análise dinâmica das instruções de um processo em execução.
- ACLs (Access Control List): Uma ACL especifica quais usuários e processos possuiem direitos de acessar os objetos e quais operações ele pode realizar. Diferente dos identificadores BSD, que indicam as permissões de escrita, leitura ou execução apenas para o criador ou grupo que ele pertence, ACLs permitem indicar diferentes ações para cada usuário, independentemente.
- K-auth (Kernel Authorization): É o mecanismo de avaliação das ACLs.

A segurança em espaço de usuário no Mac OS X é implementada seguindo o padrão CDSA (*Common Data Security Architecture*) [7]. Este padrão aberto consiste de um framework de criptografia e vários serviços de segurança.

O núcleo do CDSA é o CSSM (*Common Security Services Manager*), um conjunto de módulos que implementam uma API para serviços de criptografia, criação e manipulação de certificados digitais, armazenamento seguro e outros serviços de segurança que os programas em modo usuário podem precisar.

8 Conclusões

Os desenvolvedores do Mach original mostraram que é possível rodar um sistema Unix em cima de um microkernel, mas o sistema resultante se mostrou muito lento comparado a um Unix usual. Para resolver esse problema, os projetistas do Mac OS X colocaram a camada Unix em modo kernel, embora o design das duas camadas—Unix e Mach—fosse separado.

Essa decisão também veio com certos problemas. Por exemplo, a divisão entre processo e task no Mac OS X é essencialmente um artefato da definição minimalista do Mach de task—como o sistema foi feito para servir vários sistemas operacionais, ele não podia assumir muitas coisas sobre os processos. Essa divisão não é útil para a organização do Mac OS X. Outro exemplo é a separação entre pagers—que podem rodar em espaço de usuário—e o modelo de memória virtual, que é fornecido

na camada Mach. Os desenvolvedores do Mac OS X não se importam muito com essa funcionalidade (eles nem permitem que o usuário escreva seus pagers), mas o recurso estava disponível no Mach, e veio de graça para o sistema da Apple.

No entanto, as vantagens do projeto mais do que compensam seus problemas. A divisão do kernel em duas camadas torna o sistema mais modular, e por consequência mais fácil de ser desenvolvido. Ainda, o fato do XNU se basear em código já escrito e testado reduziu o esforço no desenvolvimento do sistema, deixando mais tempo para desenvolver as outras camadas do Mac OS X—por exemplo, a I/O Kit e a interface gráfica.

Graças às divisões das camadas internas do kernel—BSD e Mach—, o Mac OS X reserva um espaço de memória maior para kernel, com um endereçamento de 4GB de memória virtual para os seus processos. Isso faz com que o sistema operacional tenha que trocar todo seu espaço de memória com o dos programas usuários. Embora isto crie um overhead maior do que daqueles sistemas que reservam partes fixas de memória para cada espaço de endereçamento, as vantagens que a organização modular que o XNU fornece uma solução mais elegante que a maioria dos sistemas operacionais modernos.

A filosofia do projeto—em separar conceitualmente as camadas Mach e BSD, mas colocando-as em um mesmo modo protegido—colocam o Mac OS X como um sistema à frente dos seus atuais concorrentes, tornando o sistema próximo à um microkernel, sem prejudicar tanto a performance do sistema, como ocorre geralmente com este tipo de núcleo.

Referências

[1] `http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/sys/proc.h`

[2] `http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/osfmk/kern/task.h`

[3] `http://developer.apple.com/mac/library/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html#//apple_ref/doc/uid/TP30000905-CH211-TPXREF114`

[4] `http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/osfmk/kern/thread.h`

[5] `http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/osfmk/kern/priority.c`

[6] `http://en.wikipedia.org/wiki/Advanced_Encryption_Standard`

[7] `http://www.opengroup.org/security/cdsa.htm`

[8] `http://www.freebsd.org/`