1.Conceitos

A interface entre o sistema operacional e os programas de usuário é definida pelo conjunto de “instruções estendidas”, fornecidas pelo próprio sistema operacional. Essas instruções são tradicionalmente conhecidas como chamadas de sistema. As chamadas disponíveis na interface variam de um sistema operacional para outro, mas os conceitos são semelhantes.

Os sistemas operacionais têm chamadas de sistema para ler arquivos (generalidades vagas).

O MINIX tem uma chamada de sistema *read* com três parâmetros, um para especificar o arquivo, um para dizer onde os dados devem ser transferidos e um para informar quantos bytes devem ser lidos (sistema específico).

As chamadas de sistema correspondentes do UNIX e do Linux são baseadas no POSIX, na maioria dos casos.

As chamadas de sistema do MINIX 3 dividem-se em duas categorias amplas: aquelas que tratam com processo e aquelas que tratam com o sistema de arquivos.

1.1 Processos

Um conceito importante em todos os sistemas operacionais é o processo. Um processo basicamente é um programa em execução e, associado a cada processo está o espaço de endereçamento, uma lista de posições de memória a partir de um mínimo, normalmente 0, até um máximo, que o processo pode ler e escrever. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha. Também associado a cada processo está um conjunto de registradores, incluindo o contador de programa, o ponteiro da pilha e outros registradores de hardware e todas as outras informações necessárias para executar um programa.

Periodicamente, o sistema operacional decide interromper a execução de um processo e iniciar a execução de outro. Quando um processo é temporariamente suspenso, posteriormente ele deve ser reiniciado exatamente no mesmo estado em que estava quando foi interrompido. Isso significa que durante a suspensão, todas as informações sobre o processo devem ser explicitamente salvas em algum lugar. Associado a cada um desses arquivos, existe um ponteiro fornecendo a posição corrente. Quando um processo é temporariamente interrompido, todos esses ponteiros devem ser salvos para que a chamada *READ* executada posteriormente que o processo for reiniciado leia os dados corretos.

Em muitos sistemas operacionais, todas as informações sobre cada processo são armazenados em uma tabela do sistema operacional chamada de tabela de processos, um *array* de estruturas, um para cada processo correntemente existente.

Um processo suspenso consiste em seu espaço de endereçamento, normalmente chamado de imagem do núcleo, e sua entrada na tabela de processos, que contém seus registradores, entre outras coisas.

As principais chamadas de sistema de gerenciamento de processos são aquelas que tratam da criação e do término de processos. Um processo chamado *shell*, lê comandos de um terminal. O *shell* deve criar um novo processo que executará o compilador. Quando esse processo termina a compilação, executa uma chamada de sistema para ele próprio terminar.

No Windows e em outros sistemas operacionais que possuem uma GUI, dar um clique em um ícone na área de trabalho ativa um programa, exatamente como aconteceria se seu nome fosse digitado no *prompt* de comandos.

Se um processo pode criar um ou mais processos e esses processos por sua vez podem criar novos processos filhos, rapidamente chegamos em uma estrutura de árvore. Os processos relacionados que estão cooperando para fazer algum trabalho frequentemente precisam se comunicar uns com os outros e sincronizar suas atividades, isto é referido como comunicação entre processos.

São disponíveis outras chamadas de processos de sistema para os processos solicitarem mais memória, esperarem que um processo filho termine e substituírem seu próprio código por outro diferente. Ocasionalmente, há necessidade de transmitir informações para um processo em execução que não está preparado por elas. Para evitar a possibilidade de perda de uma mensagem, ou de sua resposta, o remetente pode solicitar que seu próprio sistema operacional o notifique, após alguns segundos especificados, para que ele retransmita a mensagem caso nenhum sinal de confirmação tenha sido recebido. Após configurar esse tempo limite (*timeout*), o programa pode continuar a fazer outro trabalho.

Quando tiver decorrido o tempo em segundos especificado, o sistema operacional envia um sinal de alarme para o processo. O sinal faz com que o processo suspenda temporariamente o que está fazendo, salve seus registradores na pilha e comece a executar um procedimento especial de tratamento de sinal.

Quando a rotina de tratamento de sinal tiver terminado, o processo em execução será reiniciado no estado em que estava imediatamente antes do recebimento do sinal. Estes sinais são equivalentes às interrupções de hardware, porém em software e são gerados por diversas causas, além do tempo limite. Muitas interrupções detectadas pelo hardware, como a execução de uma instrução inválida, ou o uso de um endereço inválido, também são convertidas em sinais para o processo causador.

Cada pessoa autorizada a usar um sistema MINIX 3 recebe uma UID (User Identification) do administrador do sistema. Todo processo tem a UID da pessoa que a criou. Um processo filho tem a mesma UID de seu pai. Os usuários podem ser membros de grupos, cada um dos quais com uma GID (Group IDentification).

Uma UID, denominada superusuário (UNIX), tem poder especial e pode violar muitas regras de proteção. Este usuário também é denominado de *root* do sistema.

1.2 Arquivos

Uma função importante do sistema operacional é ocultar as peculiaridades dos discos e de outros dispositivos de E/S, e apresentar ao programador um modelo abstrato, agradável e

claro, dos arquivos independentes dos dispositivos que os armazenam. Antes que um arquivo possa ser lido, ele deve ser aberto, depois de lido, ele deve ser fechado.

O MINIX 3 tem o conceito de diretório como uma maneira de agrupar os arquivos. As entradas de diretório podem ser arquivos ou outros diretórios. Esse modelo também origina uma hierarquia. Tanto o processo quanto às hierarquias de arquivo são organizadas como árvores, mas a semelhança pára por aí. As hierarquias de processos normalmente não são muito profundas, enquanto as hierarquias de arquivos normalmente têm quatro, cinco ou até mais níveis de profundidade. As hierarquias de processos normalmente têm vida curta, em geral, alguns minutos no máximo, enquanto a hierarquia de diretórios pode existir por vários anos.

Todo arquivo dentro da hierarquia de diretórios pode ser especificado por meio de seu nome de caminho a partir do topo da hierarquia (diretório-raiz). Os nomes de caminho absoluto consistem na lista dos diretórios que devem ser percorridos a partir do diretório -raiz para se chegar ao arquivo, com barras separando os componentes. No windows, o caractere de barra invertida (\) é usado como separador, em vez da barra normal (/).

Cada processo tem um diretório de trabalho corrente, no qual os nomes de caminho que não começam com uma barra são procurados. Esses caminhos são denominados de caminhos relativos.

No MINIX 3, os arquivos e diretórios são protegidos, designando-se a cada um deles um código de proteção de onze bits. Este código de proteção consiste em três campos de três bits, um para o proprietário, um para os outros membros do grupo proprietário e um para as demais pessoas. Cada campo tem um bit para acesso de leitura, um bit para acesso de escrita e um bit para acesso de execução. Esses três bits são conhecidos como bits *rwx* (*read, write, execute)*. Um traço significa que a permissão correspondente está ausente (bit 0).

Antes que um arquivo possa ser lido ou escrito, ele deve ser aberto, momento este em que as permissões são verificadas. Se o acesso for permitido, o sistema retornará um valor inteiro chamado descritor de arquivo para ser usado nas operações subsequentes. Se o acesso for proibido, será retornado um código de erro (-1).

Um conceito importante do MINIX 3 é o *mounting* de um sistema de arquivos. Quase todos os computadores pessoais têm uma ou mais unidades de CD-ROM nas quais podem ser inseridos e removidos. O MINIX 3 permite que o sistema de arquivos em um CD-ROM seja anexado à árvore principal. Antes de ser chamado de sistema *mount*, o sistema de arquivos-raiz no disco rígido e um segundo sistema de arquivos em um CD-ROM estão separados e não relacionados.

Isoladamente, o sistema de arquivos no CD-ROM não pode ser usado, pois não há como especificar nomes de caminho nele. O MINIX 3 não permite que os nomes de caminho tenham como prefixo um nome ou um número de unidade de disco, é precisamente esse o tipo de dependência de dispositivo que os sistemas operacionais devem eliminar. Em vez disso, a chamada de sistema *mount* permite que o sistema de arquivos no CD-ROM seja anexado ao sistema de arquivos raiz onde o programa quiser que ele esteja. Se um sistema contém vários discos rígidos, todos eles também podem ser montados em uma única árvore.

Outro conceito importante no MINIX 3 é o de arquivo especial. Os arquivos especiais são fornecidos para fazer os dispositivos de E/S se comportarem como se fossem arquivos convencionais. Desse modo, eles podem ser lidos e escritos usando as mesmas chamadas de sistema que são usadas para ler e escrever arquivos. Existem dois tipos de arquivos especiais.

Arquivos especiais de bloco

Os arquivos especiais de bloco normalmente são usados para modelar dispositivos que consistem em um conjunto de blocos endereçáveis aleatoriamente, como os discos. Abrindo um arquivo especial de bloco e lendo, um programa pode acessar diretamente o quarto bloco no dispositivo, sem considerar a estrutura do sistema de arquivos contida nela.

Arquivos especiais de caractere

São usados para modelar impressoras, modems e outros dispositivos que aceitam ou geram como saída um fluxo de caracteres. Por convenção, os arquivos especiais são mantidos no diretório */dev* (*device*).

Um ***pipe*** é uma espécie de pseudo-arquivo que pode ser usado para conectar dois processos. Se um processo A e outro B quiserem se comunicar usando um *pipe*, eles devem configurá-lo antecipadamente. Quando o processo A quer enviar dados para o processo B, ele escreve no *pipe* como se fosse um arquivo de saída. O processo B pode ler os dados do *pipe* como se ele fosse um arquivo de entrada. Assim, a comunicação entre processos no MINIX 3 é muito parecida com as leituras e escritas normais em arquivos.

1.3 Shell

Os editores, compiladores, montadores, ligadores e interpretadores de comandos não fazem parte do sistema operacional, ainda que sejam importantes e úteis, pois o sistema operacional é o código que executa chamadas de sistema. O *shell* é a principal interface entre um usuário sentado diante de seu terminal e o sistema operacional. Existem muitos *shells*, incluindo *csh, ksh, zsh* e *bash*.

Quando um usuário se conecta, um *shell* é iniciado. Este, tem o terminal como entrada e saída padrão. Ele começa apresentando o *prompt*, normalmente um caractere como o cifrão, que informa que o *shell* está esperando para aceitar comandos.

2.Chamadas de sistema

A mecânica de uma chamada de sistema depende muito da máquina, e frequentemente deve ser expressa em código assembly. É fornecida uma biblioteca de funções para tornar possível fazer chamadas de sistema a partir de programas escritos em C.

Qualquer computador com apenas uma CPU pode executar apenas uma instrução por vez. Se um processo estiver executando um programa no modo usuário e precisar de um serviço do sistema, como a leitura de dados de um arquivo, ele terá de executar uma instrução de interrupção ou de chamada de sistema para transferir o controle para o próprio sistema operacional. Então, o sistema operacional descobre o que o processo que fez a chamada deseja inspecionando um conjunto de parâmetros. Em seguida, ele executa a chamada de sistema e retorna o controle para a instrução que está depois da chamada de sistema.

A chamada *read* contém três parâmetros: a primeira especificando o arquivo, o segundo um buffer e o terceiro o número de bytes a serem lidos. Uma chamada para *READ* a partir de um programa em C poderia ser escrita da seguinte forma.

count = read(fd, buffer, nbytes);

A chamada de sistema retorna o número de bytes realmente lidos em *count*. Este valor é igual ao de *nbytes*, mas pode ser menor se o fim de arquivo for encontrado. Se a chamada de sistema não puder ser executada, *count* será configurado como -1 e o número indicando o código do erro será colocado em uma variável global *erro*. Os programas sempre devem verificar os resultados de uma chamada de sistema para ver se ocorreu um erro.

O MINIX 3 tem um total de 53 chamadas de sistemas principais. Os serviços oferecidos por essas chamadas determinam a maior parte do que o sistema operacional tem de fazer, pois o gerenciamento de recursos nos computadores pessoais é mínimo. O mapeamento de chamadas de função do POSIX para as chamadas de sistema não é necessariamente biunívoco. O padrão POSIX especifica várias funções que um sistema compatível deve fornecer, mas não especifica se elas são chamadas de sistema, chamadas de biblioteca ou qualquer outra coisa.

1.4 Chamadas de sistema para gerenciamento de processo

Considere a chamada de sistema *FORK*. Ele é a única maneira de criar um novo processo no MINIX 3, de forma que ele cria uma duplicata exata do processo original, incluindo todos os descritores de arquivo, registradores-tudo. Depois de *FORK*, o processo original e a cópia (o pai e o filho) seguem caminhos diferentes. Todas as variáveis têm valores idênticos no momento do *FORK*, mas como os dados do pai são copiados para criar o filho, as alterações subsequentes em um deles não afetam o outro.

FORK retorna um valor que é zero no filho e igual ao PID do filho no pai. Quando o PID retorna, os dois processos podem ver qual deles é o processo pai e qual é o processo filho.

Na maioria dos casos, após um *FORK*, o filho precisará executar um código diferente do pai. Para esperar o filho terminar, o pai executa uma chamada de sistema *waitpid*, a qual apenas espera até que o filho termine. Ele pode ser configurado para esperar um filho específico ou, configurando o primeiro parâmetro como -1 por qualquer filho. Quando *waitpid* terminar, o endereço apontado pelo segundo parâmetro, *staloc*, será configurado com o status de saída do filho.

Considere o caso do comando *cp file 1 file 2*, usado para copiar *file 1* para *file 2*. O *shell* cria um processo filho que localiza e executa o arquivo *cp* e passa para ele os nomes dos arquivos de origem e destino. O programa principal de *cp* contém a seguinte declaração, *main(argc, argv, envp)*, onde *argc* é o número de elementos da linha de comando, incluindo o nome do programa, *argv* é um ponteiro para um *array* e o terceiro parâmetro de *main* é *envp*, um ponteiro para o ambiente, um *array* de strings contendo atribuições da forma *nome=valor*, usadas para passar informações para um programa, como o tipo de terminal e o nome do diretório base.

No MINIX 3, os processos têm sua memória dividida em três segmentos, o segmento de texto, também conhecido como código do programa, o segmento de dados, isto é, as variáveis do programa e o segmento de pilha.

O segmento de dados cresce para cima e a pilha cresce para baixo. Entre eles, há um intervalo de espaço de endereçamento não utilizado. A pilha aumenta de tamanho automaticamente, conforme necessário, mas o aumento do segmento de dados é feito explicitamente por meio de uma chamada de sistema, *brk*, que especifica o novo endereço onde o segmento de dados deve terminar. Este endereço pode ser maior do que o valor corrente ou menor do que o valor corrente. O parâmetro deve ser menor do que o ponteiro da pilha, senão os segmentos de dados e de pilha iriam se sobrepor.

As chamadas *brk* e *sbrk* não são definidas pelo padrão POSIX. Os programadores devem usar a função de biblioteca *malloc* para alocar área de armazenamento dinamicamente e a sua implementação não foi considerada um assunto conveniente para a padronização.

A chamada *GETPID* retorna o PID do processo ou grupo que fez a chamada. A última chamada de sistema de gerenciamento de processos é a *ptrace*, utilizada por programas de depuração para controlar o programa que está sendo depurado. Ela permite que o depurador leia e escreva a memória do processo controlado e a gerencie de outras maneiras.

1.5 Chamadas de sistema para sinais

No MINIX 3, o usuário pode pressionar em conjunto as teclas CTRL e C que envia um sinal para o editor realizar a interrupção de um comando. Os sinais também podem ser usados para informar sobre certas interrupções detectadas pelo hardware, como uma instrução inválida ou estouro de ponto flutuante (*overflow*). Os *timeouts* também são implementados como sinais.

Quando um sinal é enviado para um processo que não anunciou seu desejo de aceitá-lo, o processo é simplesmente eliminado. Para evitar essa condição, um processo pode usar a chamada de sistema *sigaction* para anunciar que está preparado para aceitar algum tipo de sinal para fornecer o endereço de uma rotina de tratamento de sinal e um local para armazenar o endereço de execução da rotina atual.

Quando a rotina de tratamento de sinal termina, ela chama *sigaction* para que a execução continue a partir de onde parou, antes do sinal. Ela substitui a chamada mais antiga, que agora é fornecida como uma função de biblioteca por compatibilidade com versões anteriores. No MINIX 3, os sinais podem ser bloqueados. Um sinal bloqueado fica dependente até ser desbloqueado. Ele não é enviado, mas também não é perdido.

A chamada de *sigprocmask* permite que um processo define o conjunto de sinais a serem bloqueados apresentando ao núcleo um mapa de bits. Também é possível um processo solicitar o conjunto de sinais pendentes, mas que não podem ser enviados por estarem bloqueados. A chamada de *sigpending* retorna esse conjunto como um mapa de bits. A chamada *sigsuspend* que permite a um processo configurar de forma atômica o mapa de bits dos sinais bloqueados e suspender a si mesmo.

Pressionar CTRL+C não é a única maneira de enviar um sinal. A chamada de sistema *kill* permite que um processo sinalize outro processo, que envia um sinal para qualquer processo. Enviando-se o sinal 9 (SIGKILL) para um processo de segundo plano, esse processo é eliminado. SIGKILL não pode ser ignorado ou capturado.

Para muitas aplicações de tempo real, um processo precisa ser interrompido, após um intervalo de tempo específico, para fazer algo como retransmitir um pacote possivelmente perdido em um meio de comunicação não confiável. Para tratar dessa situação, foi definida a chamada de sistema *alarm*. Ela especifica um intervalo de tempo, em segundos, após o qual um sinal SIGALARM é enviado para o processo. Se for feita uma chamada de *alarm* com um parâmetro de 10 segundos e, então, 3 segundos mais tarde, for feita outra chamada *alarm* com um parâmetro de 20 segundos, somente um sinal será gerado, 20 segundos após a segunda chamada. O primeiro é cancelado.

Às vezes ocorre que um processo não tem nada para fazer até a chegada de um sinal. Uma ideia é usar *pause*, que instrui o MINIX 3 a suspender o processo até o próximo sinal.

1.6 Chamadas de sistema para gerenciamento de arquivo

Muitas chamadas de sistema estão relacionadas com o sistema de arquivos. Para criar um arquivo novo é usada a chamada *creat*. Seus parâmetros fornecem o nome do arquivo e o modo de proteção.

fd = creat("abc", 0751);

Assim, o comando acima cria um arquivo chamado *abc* com o modo 0751 octal. Os 9 bits de ordem inferior da constante 0751 especificam os bits *rwx* do proprietário. O comando também o abre para escrita, independentemente do modo do arquivo. O descritor de arquivo retornado, *fd*, pode ser usado para escrever no arquivo. Se *creat* for usado sobre um arquivo já existente, esse arquivo será truncado no comprimento 0, desde que, é claro, todas as permissões estejam corretas. A chamada de *creat* é obsoleta, pois *open* pode criar novos arquivos e foi incluída por compatibilidade com versões anteriores. Arquivos especiais são criados usando-se *mknod* em vez de *creat*.

fd=mknod("dev/ttyc2", 020744, 0x0402);

O comando acima atribui ao arquivo o modo 02744 octal. O terceiro parâmetro contém o tipo de dispositivo no byte de ordem superior, 0x04 e a identificação de uma unidade específica desse mesmo tipo de dispositivo é dada no byte de ordem inferior, 0x02. O tipo de dispositivo poderia ser qualquer um, mas um arquivo de nome */dev/ttyc2* deve ser sempre associado ao dispositivo 2.

Para ler ou escrever um arquivo existente, primeiramente o arquivo deve ser aberto com *open*. Essa chamada especifica o arquivo a ser aberto através de um nome de caminho absoluto ou de um nome relativo ao diretório de trabalho. Os códigos O_RDONLY, O_WRONLY ou O_RDWR, significam aberturas somente para leitura, somente para escrita ou ambos. O descritor de arquivo retornado pode ser então usado para operações posteriores de leitura ou escrita. Depois, o arquivo é fechado com a chamada de sistema *close*, o que

libera o descritor de arquivo disponível para ser reaproveitado por uma chamada *creat* ou *open* subsequente.

A chamada *iseek* altera o valor do ponteiro de posição, de modo que as chamadas subsequentes para *read* ou *write* podem começar em qualquer ponto no arquivo ou mesmo além do final. Este comando tem três parâmetros, onde o primeiro é o descritor de arquivo, o segundo é o ponteiro de posição e o terceiro informa se essa posição é relativa ao início dele, a posição atual ou ao final do arquivo. O valor retornado por *iseek* é a posição absoluta no arquivo depois de alterar a posição do ponteiro.

Para cada arquivo, o MINIX 3 monitora o modo do arquivo, o tamanho, o momento da última modificação e outras informações. As chamadas *stat* e *fstat* fornecem como segundo parâmetro um ponteiro para uma estrutura onde as informações devem ser colocadas.

No MINIX 3, a comunicação entre processos usa *pipes*. Quando um usuário digita o comando *cat file1 file2 | sort*, o *shell* cria um *pipe* e faz com que a saída padrão do primeiro processo escreva no *pipe* e que a entrada padrão do segundo processo leia a partir dele. A chamada de sistema *pipe* cria um *pipe* e retorna dois descritores de arquivo, um para leitura e outro para escrita. A chamada é *pipe(&fd[0])*, onde *fd* é um *array* de dois números inteiros, *fd[0]* é o descritor de arquivo para leitura e *fd[1]* para escrita. Geralmente é feito um *FORK* logo após a chamada *pipe*, o processo pai fecha o descritor de arquivo para leitura e o processo filho fecha o descritor de arquivo para escrita.

O programa abaixo representa dois processos, com a saída do primeiro *piped* para o segundo. Primeiro, o *pipe* é criado e o processo executa um *FORK* fazendo com que o processo pai se torne o primeiro processo do *pipe* e o processo filho o segundo. Como os arquivos a serem executados, *process1* e *process2* não sabem que fazem parte de um *pipe*, é fundamental que os descritores de arquivo sejam tratados de modo que a saída padrão do primeiro processo e a entrada padrão do segundo processo sejam o *pipe*.

O pai primeiro fecha o descritor de arquivo para leitura do *pipe*. A seguir, ele fecha a saída padrão e faz uma chamada de *dup* para permitir ao descritor de arquivo1 escrever no *pipe*. Após a chamada de *execl*, o processo pai terá os descritores de arquivo 0 e 2 inalterados, e o descritor de arquvio1 para de escrever no *pipe*. Os parâmetros de *execl* são repetidos porque o primeiro é o arquivo a ser executado e o segundo é o primeiro parâmetro, que a maioria dos programas espera que seja o nome do arquivo.

```
#define STD_INPUT 0
#define STD_OUTPUT 1
pipeline(process1, process2)
char *process1, *process2;

{
int fd[2];
pipe(&fd[0]);
```

```
if (FORK() != 0) {
        close(fd[0]);
        close(STD_OUTPUT);
        dup(fd[1]);
        close(fd[1]);
        execl(process1, process1, 0);
} else {
        close(fd[1]);
        close(STD_INPUT):
        dup(fd[0]);
        execl(process2, process2, 0);
}
}
```

A chamada de sistema *IOCTL* é potencialmente aplicada a todos os arquivos especiais. Utilizada por drivers. Seu principal uso é com arquivos de caracteres especiais, principalmente terminais. O padrão POSIX define diversas funções que a biblioteca transforma em chamada de *IOCTL*. As funções de biblioteca *tcgetattr* e *tcsetattr* usam *IOCTL* para alterar os caracteres utilizados para corrigir erros de digitação no terminal.

O modo processado (cooked mode) é o modo terminal normal, no qual os caracteres de apagamento e de eliminação funcionam normalmente.

No modo bruto (raw mode) todas as funções de parada são desativadas, consequentemente, cada caractere é passado diretamente para os programas sem nenhum processamento especial. Alem disso, uma leitura a partir do terminal fornecerá para o programa todos os caracteres que forma digitados, mesmo uma linha parcial, em vez de esperar que uma linha completa seja digitada, como no modo processado.

O modo cbreak é um meio-termo. Os caracteres de apagamento e eliminação são desativados para edição, mas CTRL-S, CTRL-Q, CTRL-C e CTRL-\ são ativados.

O POSIX não usa os termos processado, bruto e cbreak. O modo canônico corresponde ao modo processado. No modo não canônico, um número de caracteres a serem aceitos e um limite de tempo definido em unidades de décimos de segundo, determinado como leitura será feita.

IOCTL tem três parâmetros, onde o primeiro especifica um arquivo, o segundo uma operação e o terceiro é o endereço da estrutura do POSIX, que contém os *flags* e o *array* de caracteres de controle. Outros códigos de operação instruem o sistema a adiar as alterações até que toda saída tenha sido enviada, a fazer com que uma entrada não lida seja descartada e a retornar os valores correntes.

A chamada de sistema *access* é utilizada para determinar se certo acesso a um arquivo é permitido pelo mecanismo de proteção. Ela é necessária porque alguns programas podem ser executados usando o UID de um usuário diferente.

A chamada de sistema *rename* permite dar um novo nome a um arquivo. Os parâmetros são o nome antigo e o novo.

A chamada de sistema *fcntl* que é utilizada para arquivos de controle, mais ou menos análoga a *IOCTL*, tem várias opções, sendo a mais importante o *locking* de arquivos consultados por mais de uma aplicação. Usando *fcntl* é possível para um processo travar e destravar partes de arquivos e testar se determinadas partes de um arquivo se encontram ou não travadas. A chamada não impõe nenhuma semântica para o travamento. Os programas devem fazer isso por si mesmo.

1.7 Chamadas de sistema para gerenciamento de diretórios

As duas primeiras chamadas, *mkdir* e *rmdir* criam e removem diretórios vazios, respectivamente. A chamada seguinte é *link*, seu objetivo é permitir que o mesmo arquivo apareça com dois ou mais nomes, frequentemente em diretórios diferentes. Compartilhar um arquivo não é o mesmo que dar a cada membro da equipe uma cópia privativa, porque ter um arquivo compartilhado significa que as alterações feitas por qualquer membro da equipe são instantaneamente visíveis para os outros membros. Quando são feitas cópias de um arquivo, as alterações subsequentes feitas em uma cópia não afetam as outras.

No UNIX, cada arquivo tem um número exclusivo, número-i, que o identifica. Esse número é um índice em uma tabela de *i-nodes*, um por arquivo, informando quem é o proprietário do arquivo, onde estão seus blocos de disco e outras coisas. Um diretório é simplesmente um arquivo contendo um conjunto de pares (*i-nodes*). O que *link* faz é simplesmente criar uma nova entrada de diretório com um nome, usando o *i-node* de um arquivo já existente.

A chamada de sistema *mount* permite que dois sistemas de arquivos sejam combinados em um só. Uma situação comum é ter o sistema de arquivos-raiz, contendo as versões em binário dos comandos comuns, e outros arquivos intensamente utilizados em um disco rígido. O usuário pode então inserir um CD-ROM com arquivos a serem lidos na respectiva unidade. Uma instrução típica em C para realizar a montagem é a seguinte.

***mount(“/dev/cdrom0”, “/mnt”, 0*);**

O primeiro parâmetro é o nome de um arquivo de bloco especial da unidade de CD-ROM 0, o segundo parâmetro é o lugar na árvore onde ele deve ser montado e o terceiro indica se o sistema de arquivos deve ser montado para leitura e escrita ou somente para leitura.

Depois da chamada *mount*, um arquivo na unidade de CD-ROM 0 pode ser acessado usando-se apenas seu caminho a partir do diretório-raiz, ou do diretório de trabalho, sem considerar em qual unidade ele fisicamente está. A chamada *mount* torna possível integrar mídia removível em uma única hierarquia de arquivos, sem a necessidade de se preocupar com o dispositivo em que um arquivo está. Quando um sistema de arquivos não é mais necessário, ele pode ser desmontado com a chamada de sistema *unmount*.

O MINIX 3 mantém um cache de blocos recentemente usados na memória principal para evitar a necessidade de lê-los do disco, se eles forem utilizados outra vez em um curto espaço de tempo. Se um bloco que está na cache for modificado e o sistema falhar antes do bloco modificado ser escrito no disco, o sistema de arquivos será danificado. Para limitar o

possível dano, é importante esvaziar a cache periodicamente, para que o volume de dados perdidos diminua em caso de falha.

Duas outras chamadas relacionadas com diretórios são *chdir* e *chroot*. A primeira muda o diretório de trabalho e a última muda o diretório-raiz. Quando um processo tiver dito ao sistema para que mude seu diretório-raiz, todos os nomes de caminho absolutos começarão em uma nova raiz. Isto é feito por questões de segurança. Programas servidores que implementam protocolos FTP e HTTP fazem isso para que os usuários remotos desses serviços possam acessar apenas as partes de um sistema de arquivos que estão abaixo da nova raiz. Apenas superusuários podem executar *chroot*, e mesmo eles não podem fazer com muita frequência.

1.8 Chamadas de sistema para proteção

No MINIX 3, cada arquivo tem um modo de proteção dado em 11 bits. Nove deles são os bits de leitura-escrita-execução para o proprietário, para o grupo e para outros. A chamada de sistema *chmod* torna possível mudar o modo de proteção de um arquivo.

Os outros dois bits de proteção, 02000 e 04000 são os bits de SETGID (set-group-id) e SETUID (set-user-id), respectivamente. Quando um usuário executa um programa com o bit SETUID ativado, o UID efetivo do usuário é alterado para o do proprietário do arquivo até o término desse processo. Esse recurso é intensamente utilizado para permitir que os usuários executem programas que efetuam funções exclusivas do superusuário.

Quando um processo executa um arquivo que tem o bit SETUID ou SETGID ativado em seu modo de proteção, ele adquire um UID ou GID efetivo diferente de seu UID ou GID real. As chamadas de sistema *getuid* e *getgid* foram providenciadas para fornecer UID ou GID efetivo e real. Cada chamada retorna o UID ou GID efetivo e real, de modo que quatro rotinas de biblioteca são necessárias para extrair as informações corretas sendo elas: *getuid*, *geteuid*, *getgid* e *getegid*. A primeira e a terceira obtêm o UID/GID real e a segunda e quarta obtêm os efetivos.

Usuários normais não podem alterar seu UID, exceto executando programas com o bit SETUID ativado, mas o superusuário tem outra possibilidade, a chamada de sistema *setuid*, que configura os UIDs real e efetivo. *setgid* configura os dois GIDs, real e efetivo.

A chamada de sistema *umask* configura uma máscara de bits interna dentro do sistema, que é utilizada para mascarar bits de modo quando um arquivo é criado. Após a chamada, o modo fornecido por *creat* e *mknod* terá os bits que foram utilizados nesta mesma chamada, mascarados antes de serem utilizados.

Assim, uma chamada *umask(022)* e *creat("file", 0777)* configurará o modo como 0755, em vez de 0777. Como a máscara de bits é herdada pelos processos filhos, se o *shell* executar uma instrução *umask* imediatamente após o *login*, nenhum dos processos do usuário nessa sessão criará acidentalmente arquivos em que outras pessoas possam escrever.

Quando um programa pertencente pelo usuário *root* tem o bit SETUID ativado, ele pode acessar qualquer arquivo, pois seu UID efetivo é o superusuário. Frequentemente, é útil o programa saber se a pessoa que o ativou tem permissão para acessar determinado arquivo.

Se o programa simplesmente tentar o acesso, ele sempre terá êxito e, portanto, não saberá nada.

A chamada de sistema *access* fornece uma forma de descobrir se o acesso é permitido para o UID real. O parâmetro *mode* é 4 para verificar acesso de leitura, 2 para acesso de escrita e 1 para acesso de execução. Combinações desses valores também são permitidas. Embora os mecanismos de proteção de todos os sistemas operacionais do tipo UNIX geralmente sejam semelhantes, existem algumas diferenças e inconsistências que levam a vulnerabilidades de segurança.

1.9 Chamadas de sistema para gerenciamento de tempo

O MINIX 3 tem quatro chamadas de sistema que envolvem o tempo de relógio convencional. A chamada *time* retorna apenas a hora atual, em segundos, com 0 correspondendo à meia-noite de 1º de janeiro de 1970. A chamada *stime* foi fornecida para permitir que o relógio seja ajustado (superusuário). A terceira chamada de tempo é *utime*, que permite ao proprietário de um arquivo alterar o tempo armazenado no *i-node* de um arquivo. A aplicação desta chamada de sistema é bastante limitada, mas alguns programas precisam dela. A chamada *times* retorna as informações de contabilização de um processo, para que se possa ver quanto tempo de CPU foi utilizado diretamente e quanto tempo de CPU o sistema em si gastou em seu nome. Também são fornecidos os tempos de usuário e de sistema totais utilizados por todos os seus filhos combinados.