Resumo do Capítulo 8:

Fluxo de Controle Excepcional

Lucas Batista Santos

1. Introdução

O sistema precisa ser capaz de reagir a mudanças no estado do sistema que não são necessariamente relacionadas à execução do programa. Por exemplo, pacotes chegam no adaptador de rede e precisam ser guardados na memória. Programas solicitam dados ao disco e então dormem até serem notificados que os dados estão prontos. Um processo pai que criou um processo filho precisa ser notificado quando o filho terminar.

Sistemas modernos reagem a essas situações fazendo mudanças abruptas no fluxo de controle. Em geral, essas mudanças abruptas são chamadas de *fluxo de controle excepcional* (ECF). ECF ocorrem em todos os níveis do sistema.

2. Exceções

Exceções são uma forma de ECF que são implementadas parcialmente pelo hardware e parcialmente pelo sistema operacional. Uma vez que elas são implementadas em parte pelo hardware, os detalhes variam de sistema para sistema. No entanto, as ideias básicas são as mesmas para qualquer sistema. Uma exceção é uma mudança abrupta no fluxo de controle em resposta a alguma mudança no estado do processo. Essa mudança no estado é conhecida como um evento.

Quando o processador detecta que um evento ocorreu, ele faz uma chamada de procedimento indireta (a exceção), através de uma tabela chamada de tabela de exceção, para uma sub-rotina do sistema operacional (manipulador de interrupção) que é especificamente projetada para processar esse tipo particular de evento.

2.1 Manipulando Exceções

A cada tipo possível de interrupção em um sistema é atribuído um inteiro maior que zero e único *número de exceção*. Alguns desses números são atribuídos pelos projetistas do processador, outros são atribuídos pelos projetistas do kernel do sistema.

Em tempo de boot (quando o computador está sendo ligado ou reiniciado), o sistema operacional aloca e inicializa uma tabela de salto chamada de *tabela de exceção*, onde cada entrada k contém o endereço do manipulador para a exceção k.

Em tempo de execução (quando o sistema está executando algum programa), o processador detecta que um evento ocorreu e determina o número k da exceção correspondente. O processador então dispara a exceção fazendo uma chamada indireta, através da entrada k da tabela de exceção, para o manipulador correspondente.

Uma vez que o hardware tenha disparado a exceção, o resto do trabalho é feito em software pelo manipulador de exceção. Após o manipulador processar o evento, ele opcionalmente retorna para o programa interrompido executando uma instrução especial, que restaura o estado anterior ao controle do processador e os registradores, retorna ao modo de usuário se a exceção interrompeu um programa de usuário, e então retorna o controle ao programa interrompido.

2.2 Classes de Exceções

Exceções podem ser divididas em quatro classes: interrupções, traps, faltas e abortos.

Interrupções

Interrupções ocorrem assincronamente como resultado de sinais de dispositivos de E/S que são externos ao processador. Interrupções de hardware são assíncronas no sentido de que elas não são causadas pela execução de qualquer instrução particular. Manipuladores de exceção para interrupções de hardware são frequentemente chamados de manipuladores de interrupção.

Depois que a instrução atual termina de executar, o processador percebe que o pino de interrupção foi ativado, lê o número da exceção do barramento de sistema e então chama o manipulador de interrupção apropriado. Quando o manipulador retorna, ele retorna o controle para a próxima instrução.

As classes de exceções restantes ocorrem sincronamente como resultado da execução da instrução atual. Refere-se a essa instrução como *faulting instruction.*

Traps e Chamadas de Sistema

Traps são exceções intencionais que ocorrem como resultado da execução de uma instrução. Assim como manipuladores de interrupção, manipuladores de traps retornam o controle para a próxima instrução. O uso mais importante das traps é fornecer uma interface procedural entre os programas do usuário e o kernel, conhecida como chamada de sistema.

Faltas

Faltas resultam de condições de erro que o manipulador precisa ser capaz de corrigir. Quando uma falta ocorre, o processador transfere o controle para o manipulador de faltas. Se o manipulador conseguir corrigir o erro, ele retorna o controle para a instrução causadora do erro, que vai ser reexecutada. Caso

contrário, o manipulador retorna para uma rotina no kernel que termina o programa que causou a falta.

Abortos

Abortos são resultados de erros fatais irreparáveis. Manipuladores de aborto nunca retornam o controle para o programa, ele sempre retorna o controle para uma rotina de aborto que termina o programa.

3. Processos

A definição clássica de um processo é uma *instância de um programa em execução*. Cada programa no sistema roda no contexto de algum processo. O contexto consiste no estado que o programa precisa para rodar corretamente. Esse estado inclui o código do programa e os dados em memória, sua pilha, o conteúdo dos registradores, seu contador de programa, variáveis de ambiente e o conjunto dos descritores de arquivos abertos.

Cada vez que o usuário roda um programa escrevendo o nome de um arquivo executável no shell, o shell cria um novo processo e então roda o objeto executável no contexto de um novo processo.

3.1 Fluxo de Controle Lógico

Um processo dá a cada programa a ilusão de que ele tem uso exclusivo do processador, mesmo que muitos outros programas estejam rodando ao mesmo tempo no sistema. Quando um debbuger é utilizado para fazer a execução passo a passo de um programa, é possível observar uma série de valores do contador de programa (PC) que correspondem exclusivamente às instruções contidas no arquivo executável do programa ou nos objetos linkados dinamicamente ao programa em tempo de execução. Essa sequência de valores do PC é conhecida como *fluxo de controle lógico*, ou simplesmente *fluxo lógico*.

3.2 Fluxos Concorrentes

Um fluxo lógico que executa em sobreposição de tempo com outro fluxo é chamado de fluxo concorrente e esses dois fluxos rodam concorrentemente. O fenômeno de múltiplos fluxos executarem concorrentemente é conhecido como concorrência. A noção de um processo revezando com outros processos é também chamada de multitasking. Cada período de tempo que um processo executa uma porção do seu fluxo é chamado de time slice.

3.3 Espaço de Endereçamento Privado

Cada processo possui seu próprio espaço de endereçamento privado. Esse espaço é privado no sentido de que um byte de memória associado com

um endereço particular no espaço geralmente não pode ser lido ou escrito por qualquer outro processo.

Ainda que o conteúdo de memória associado com cada endereço privado é geralmente diferente, cada espaço possui a mesma organização geral. A porção de baixo do espaço de endereço é reservada para programas de usuário, com seus segmentos de código, dados, heap e pilha. A porção de cima do espaço de endereço é reservada para o kernel, essa parte da memória contém o código, dados e pilha que o kernel utiliza quando executa instruções em nome do processo (e.g., quando o programa executa uma chamada de sistema).

3.4 Mudanças de Contexto

O kernel mantém um contexto para cada processo. O contexto é o estado que o kernel precisa para reiniciar um processo preterido. Ele consiste de valores de objetos como os registradores, contador de programa, pilha de usuário, pilha do kernel e várias estruturas de dados do kernel.

Em certos pontos durante a execução de um processo, o kernel pode decidir preterir o atual processo e reiniciar um processo previamente preterido. Essa decisão é conhecida como escalonamento e é manipulada por código no kernel, chamado de escalonador. Após o kernel ter escalonado um novo processo para rodar, ele pretere o processo atual e transfere o controle para o novo processo usando um mecanismo chamado de mudança de contexto que primeiro salva o contexto do processo atual, restaura o contexto salvo de algum processo previamente preterido e então passa o controle para o processo restaurado.

4. Manipulador de Erro de Chamada de Sistema

Quando funções de nível de sistema do Unix encontram um erro, elas tipicamente retornam -1 e configuram a variável inteira global *errno* para indicar o que deu errado. Programadores devem sempre checar os erros, mas infelizmente muitos pulam a checagem de erros porque isso incha o código e torna-o difícil de ler.

5. Controle de Processos

5.1 Obtendo ID dos Processos

Cada processo tem um ID (PID) único, inteiro e positivo (diferente de zero). A função *getpid* retorna o PID do processo que a chamou. A função *getppid* retorna o PID do pai do processo chamador (i.e., o processo que criou o processo chamador).

5.2 Criando e Terminando Processos

De uma maneira geral, um processo pode estar em um dos três seguintes estados:

- **Rodando:** O processo está executando na CPU ou esperando para ser executado e vai eventualmente ser escalonado pelo kernel.
- **Parado:** A execução do processo está suspensa e este não vai ser escalonado.
- **Terminado:** O processo está permanentemente parado. Um processo pode parar por um dos três motivos seguintes: receber um sinal cuja ação padrão é terminar o processo, retornar da rotina main, chamar a função *exit*.

A função exit termina o processo com um status de saída passado como parâmetro. (Outra maneira de configurar o status de saída é retornar um valor inteiro da rotina main.)

Um processo pai pode criar um novo processo filho chamando a função fork. O novo processo filho criado é quase idêntico ao seu pai. O filho obtém uma cópia idêntica (porém separada) do espaço de memória virtual do processo pai, incluindo segmentos de código e dados, heap, bibliotecas compartilhadas e pilha de usuário. O filho também recebe uma cópia idêntica dos descritores de qualquer arquivo aberto no processo pai. A principal diferença entre o pai e o filho é que eles possuem diferentes PIDs.

A função fork é chamada uma vez, mas retorna duas vezes: uma vez no processo chamador (pai) e uma vez no processo filho. No pai, a função retorna o PID do filho. No filho, a função retorna o valor 0. Uma vez que o PID de um processo é sempre diferente de zero, esse retorno fornece uma maneira precisa para determinar se o programa está executando no pai ou no filho.

5.3 Coletando Processo Filhos

Quando um processo termina por alguma razão, ele é mantido em uma espécie de estado terminado até que seja coletado pelo seu pai. Quando o pai coleta o filho terminado, o kernel passa o status de saída do filho para o pai e então descarta o processo terminado. Um processo terminado que ainda não foi coletado é chamado de zumbi.

Quando um processo pai termina, o kernel faz com que o processo init se torne o pai adotivo dos filhos órfãos. O processo init, que possui o PID 1, é criado pelo kernel durante a inicialização do sistema, nunca termina e é o ancestral de todos os processos. Se um processo pai termina sem coletar seus filhos zumbis, então o kernel faz com que o processo init colete eles. Contudo, programas de longa duração como shells ou servidores devem sempre coletar seus filhos zumbis. Mesmo que os zumbis não estejam rodando, eles continuam consumindo recursos de memória. Um processo espera por suas crianças terminarem ou pararem chamando a função waitpid.

5.4 Colocando Processos para Dormir

A função sleep suspende um processo por um período de tempo específico. Essa função retorna zero se período de tempo requisitado já esgotou, ou a quantidade de segundos que restam para dormir caso contrário.

5.5 Carregando e Rodando Programas

A função execve carrega e roda um novo programa no contexto do processo atual. A função retorna para o programa chamador apenas se houver algum tipo de erro, então diferente da fork, que é chamada uma vez e retorna duas, execve é chamada uma vez e nunca retorna.

5.6 Usando fork e execve para Rodar Programas

Programas como shells do Unix e servidores Web fazem uso pesado das funções fork e execve. Um shell é uma aplicação interativa que roda outros programas em nome do usuário. Um shell realize uma sequência de passos de leitura/avaliação e então termina. O passo de leitura lê a linha de comando do usuário. O passo de avaliação analisa a linha de comando e roda programas em nome do usuário.

6. Sinais

Um sinal é uma pequena mensagem que notifica um processo que um evento de algum tipo ocorreu no sistema. Cada tipo de sinal corresponde ao algum tipo de evento no sistema. Exceções de baixo nível do hardware são processadas pelo manipulador de exceção do kernel e normalmente não seriam visíveis para os processos do usuário. Sinais fornecem um mecanismo para expor as ocorrências dessas exceções para os processos do usuário.

6.1 Terminologia do Sinal

A transferência de um sinal para processo ocorre em dois passos distintos:

- **Enviando um sinal:** O kernel envia um sinal para um processo de destino atualizando algum estado no contexto no processo. O sinal é enviado por uma das seguintes razões: O kernel detectou um evento no sistema como um erro de divisão por zero ou a terminação de um processo filho, ou um processo invocou a função kill para solicitar explicitamente ao kernel para enviar um sinal para o processo de destino. Um processo pode enviar um sinal para si mesmo.

- **Recebendo um sinal:** Um processo de destino recebe um sinal quando ele é forçado pelo kernel a reagir de algum jeito à entrega do sinal. O processo pode ignorar o sinal, terminar, ou pegar o sinal executando uma função de nível de usuário chamada de manipulador de sinal.

Um sinal que foi enviado, mas ainda não foi recebido é chamado de um sinal pendente. Se um processo tem um sinal pendente do tipo k, então qualquer sinal subsequente do tipo k enviado para o processo não será enfileirado, eles são simplesmente descartados. Um processo pode seletivamente bloquear a recepção de certos sinais. Quando um sinal é bloqueado, ele pode ser enviado, porém o sinal pendente resultante não será recebido pelo processo até que ele desbloqueie o sinal.

Um sinal pendente é recebido no máximo uma vez. Para cada processo, o kernel mantém um conjunto de sinais pendentes no pending bit vector e o conjunto de sinais bloqueados no blocked bit vector. O kernel configura o bit k no pending sempre que um sinal do tipo k é enviado e limpa o bit k no pending sempre que um sinal do tipo é recebido.

6.2 Recebendo Sinais

Quando o kernel troca o modo de um processo p de kernel para usuário (e.g., retornando de uma chamada de sistema ou completando uma mudança de contexto), ele checa o conjunto de sinais pendentes desbloqueados para p. Se o conjunto está vazio, então o kernel passa o controle para a próxima instrução no fluxo de controle lógico de p. No entanto, se o conjunto não estiver vazio, então o kernel escolhe algum sinal k no conjunto (tipicamente o menor k) e força p a receber o sinal k. A recepção do sinal faz com que o processo tome alguma ação. Uma vez que o processo complete a ação, então o controle é passado de volta para a próxima instrução no fluxo de controle lógico de p. Cada sinal tem uma ação padrão pré-definida, que é uma das seguintes:

- O processo termina.
- O processo termina e despeja o núcleo.
- O processo para (é suspenso) até ser reiniciado.
- O processo ignora o sinal.

6.3 Bloqueado e Desbloqueando Sinais

O Linux fornece mecanismos implícitos e explícitos para bloquear sinais:

- **Mecanismo implícito:** Por padrão, o kernel bloqueia qualquer sinal pendente do tipo atualmente sendo processado por um manipulador.
- **Mecanismo explícito:** Aplicações podem bloquear e desbloquear explicitamente sinais selecionados usando a função sigprocmask.

6.4 Escrevendo Manipuladores de Sinais

Manipulando Sinais de Forma Segura

- **Manter os manipuladores o mais simples possível:** A melhor maneira de evitar problemas é manter os manipuladores tão pequenos e simples quanto possível.
- **Chamar apenas funções *async-signal-safe* nos manipuladores:** Uma função que é *async-signal-safe*, ou simplesmente *safe*, tem a propriedade de que ela pode ser chamada com segurança por um manipulador de sinal, isso porque ela é reentrante (e.g., acessa apenas variáveis locais) ou porque ela não pode ser interrompida por um manipulador de sinal.
- **Salvar e restaurar errno:** Muitas das funções async-signal-safe do Linux configurar o errno quando elas retornam com um erro. Chamar essas funções dentro do manipulador pode interferir com outras partes do programa que dependem de errno.
- **Proteger acessos às estruturas de dados globais compartilhadas bloqueando todos os sinais:** Se um manipulador compartilha uma estrutura de dados global com um programa ou com outros manipuladores, então esses manipuladores e o programa devem bloquear temporariamente todos os sinais enquanto acessam a estrutura de dados.

Manipuladores de Sinais Portáveis

Outro aspecto complicado da manipulação de sinais do Unix é que diferentes sistemas podem ter diferentes semânticas. Por exemplo:

- **A semântica da função de sinal varia:** Alguns sistemas Unix antigos restauram a ação de um sinal k para o seu padrão depois do sinal k ter sido pegado por um manipulador. Nesses sistemas, o manipulador precisa explicitamente reinstalar a si mesmo, chamando um sinal, cada vez que ele roda.
- **Chamadas de sistema podem ser interrompidas:** Chamadas de sistema como read, wait e accept que podem bloquear o processo por um longo período de tempo são chamadas de slow system calls. Em algumas versões antigas do Unix, slow system calls que são interrompidas quando um manipulador pega um sinal não continuam quando o manipulador de sinal retorna, ao invés disso elas retornam imediatamente para o usuário com uma condição de erro. Nesses sistemas, os programadores precisam incluir um código que manualmente reinicia as chamadas de sistema interrompidas.

Para lidar com esses problemas, o padrão Posix define a função sigaction, que permite aos usuários claramente especificar a semântica do manipulador de sinais que querem quando eles instalam um manipulador.

6.5 Esperando Explicitamente por Sinais

Algumas vezes o programa principal precisa esperar explicitamente que um certo manipulador de sinal rode. Por exemplo, quando o shell do Linux cria

um processo em primeiro plano, ele precisa esperar que o processo termine e seja coletado antes de aceitar o próximo comando do usuário.

A solução apropriada é usar a função sigsuspend, essa função substitui temporariamente o blocket set atual com uma máscara e então suspende o processo até que ele receba um sinal cuja ação padrão é rodar um manipulador ou terminar o processo. Se a ação é para terminar, então o processo termina sem retornar da sigsuspend. Se a ação é rodar um manipulador, então sigsuspend retorna depois que o manipulador retornar, restaurando o blocked set para o estado do momento em que sigsuspend foi chamada.

7. Saltos Não-Locais

O C fornece uma forma de fluxo de controle excepcional a nível de usuário, chamado de salto não-local, que transfere diretamente o controle de uma função para outra atualmente executando sem precisar passar pela sequência normal de chamada-retorno. Saltos não-locais são providos pelas funções setjmp e longjmp.

A função setjmp salva o ambiente chamador em um buffer, para ser usado posteriormente pela longjmp, e retorna 0. O ambiente chamador inclui o contador de programa, ponteiro de pilha e registradores de propósito geral.

A função longjmp restaura o ambiente chamador do buffer e então retorna para a chamada mais recente de setjmp que inicializou o buffer.

Um uso importante de saltos não-locais é possibilitar o retorno imediato de uma chamada de função profundamente aninhada, usualmente como resultado da detecção de algum erro. Se um erro for detectado em uma chamada de função profundamente aninhada, é possível usar uma salto não-local para retornar diretamente para um manipulador de erro de localização conhecida, ao invés de trabalhosamente “desenrolar” a pilha de chamadas.

8. Ferramentas para Manipulação de Processos

Os sistemas Linux fornecem ferramentas úteis para monitorar e manipular processos:

- STRACE: Imprime um rastro de cada chamada de sistema invocada por um programa rodando e seus filhos.
- PS: Lista os processos (incluindo zumbis) atuais no sistema.
- TOP: Imprime informações sobre o uso de recursos pelos processos atuais.
- PMAP: Mostra o mapa de memória de um processo.
- /proc: Um sistema de arquivos virtual que exporta o conteúdo de várias estruturas de dados do kernel em um texto ASCII que pode ser lido por programas de usuário.