Resumo do Capítulo 8 do Livro *Computer Systems: A Programmer's Perspective*

Aluno: Johnatas Barbosa dos Santos Matrícula: 201610328

Professor: Leard Oliveira Fernandes

Disciplina: Software Básico Turma: 2018.1

8 Exceptional Control Flow

Durante o período em que o processador estar ativo, o contador de programa assume uma sequência de endereços de instruções. A transição entre os valores é chamada de *control transfer*. Uma sequência de tal *control transfers* é chamada de *flow of control* ou *control flow* do processador. Sistemas devem ser capazes de reagir a mudanças no estado do sistema não capturados por variáveis internas do programa e não estão necessariamente relacionados a execução do programa. Sistemas modernos reagem a essas situações realizando mudanças bruscas no *control flow*. Em geral, essas mudanças são referidas como *exceptional control flow* (ECP). ECF pode ocorrer em todos os níveis o sistema computacional.

Para o programador, há um número de razões do por que é importante entender ECF:

- Entender ECF ajudará você a compreender conceitos importantes do sistema. ECF é um mecanismo básico que o sistema operacional usa para implementar entrada e saída, processos e memória virtual.

- Entender ECF ajudará você a compreender como aplicações interagem com o sistema operacional. Aplicações requerem serviços do sistema operacional usando uma forma de ECF conhecida como *trap* ou *system call*.

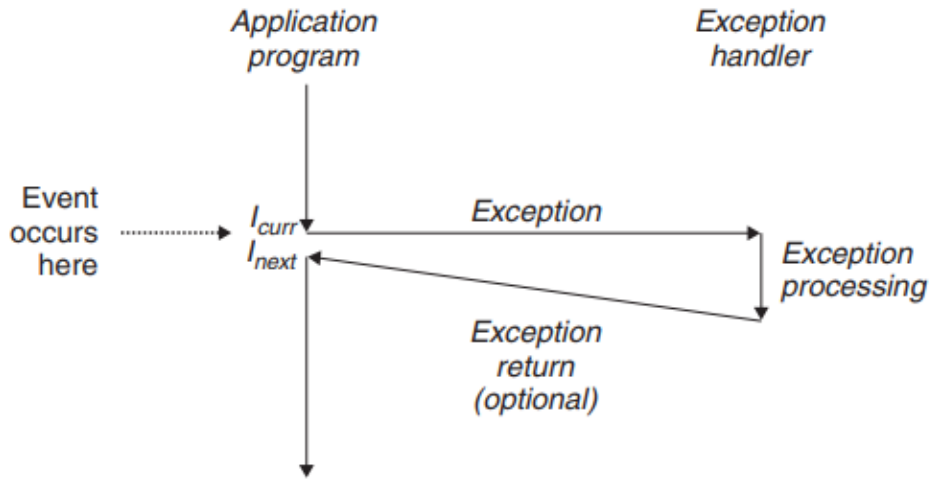
- Entender ECF ajudará você a escrever interessantes novas aplicações. O sistema operacional prover poderosos mecanismos para criar novos processos, esperando por processos terminar, notificar outros processos de eventos excepcionais, detectar e responder esses eventos.

- Entender ECF ajudará você a compreender concorrência. ECF é um manipulador de exceções que interrompe a execução de um programa, processos e threads cuja execução se sobrepõe no tempo, e um manipulador de sinal que interrompe a execução de um programa são todos exemplos de simultaneidade em ação.

- Entender ECF ajudará você a compreender como funciona as exceções de software. Exceções de software permitem a programas fazer *nonlocal jump* em resposta a condições de erro.

8.1 Exceções

As exceções são uma forma de *exceptional control flow* que são implementadas tanto pelo hardware como pelo sistema operacional. Uma exceção é uma abrupta mudança no con*trol flow* em resposta a alguma mudança no estado do processador. A figura abaixo apresenta a exemplificação da anatomia de uma exceção.



Uma alteração no estado do processador (evento) aciona uma transferência de controle abrupta (uma exceção) do programa para um manipulador de exceção. Depois de terminar o processamento, o manipulador retorna o controle ao programa interrompido ou aborta.

Quando o processador detecta que o evento ocorreu, ele faz uma chamada de procedimento indireta (a exceção), através de uma tabela de salto chamada tabela de exceção, para uma sub-rotina do sistema operacional (o manipulador de exceção) especificamente projetada para processar esse tipo específico de evento. Quando o manipulador de exceções termina o processamento, uma de três coisas acontece, dependendo do tipo de evento que causou a exceção:

1. O manipulador retorna o controle para a instrução atual, a instrução que estava sendo executado quando o evento ocorreu.

2. O manipulador retorna o controle para a próxima instrução, a instrução que seria executada a seguir teve a exceção não ocorrida.

3. O manipulador aborta o programa interrompido.

8.1.1 Tratamento de exceção

Cada tipo de exceção possível em um sistema recebe um número de exceção inteiro não-negativo exclusivo. Alguns desses números são atribuídos pelos projetistas do processador. Outros números são atribuídos pelos projetistas do kernel do sistema operacional (a parte residente na memória do sistema operacional). Exemplos do primeiro incluem dividir por zero, falhas de página, violações de acesso à memória, pontos de interrupção e estouros

aritméticos. Exemplos deste último incluem chamadas do sistema e sinais de dispositivos de E/S externos.

No momento da inicialização do sistema (quando o computador é reinicializado ou ligado), o sistema operacional aloca e inicializa uma tabela de saltos chamada tabela de exceções, de modo que a entrada k contém o endereço do manipulador para a exceção k.

No tempo de execução (quando o sistema está executando algum programa), o processador detecta que um evento ocorreu e determina o número de exceção correspondente k. O processador dispara a exceção fazendo uma chamada de procedimento indireta, através da entrada k da tabela de exceções, para o manipulador correspondente.

8.1.2 Classes de exceções

Exceções podem ser divididas em quatro classes: *interrupts*, *traps*, *faults*, and *aborts*. A figura 8.2 sumariza o processo de interrupção. As interrupções são sinalizadas em um pino no chip do processado e colocando no barramento do sistema o número da exceção que identifica o dispositivo que causou a interrupção.

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

8.1.3 Exceções nos sistemas Linux/IA32

Há 256 diferentes tipos de exceção, sendo de 0 a 31 correspondentes a exceções definidas pela arquitetura Intel e o restante correspondem a *interrupts* e *traps* definidos pelo sistema operacional. A figura 8.3 mostra alguns exemplos.

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Interrupt or trap

8.2 Processos

As exceções são os blocos básicos que permitem que o sistema operacional forneça a noção de um processo, uma das ideias mais profundas e bem-sucedidas da ciência da computação.

O processo é uma instância de um programa em execução. Cada programa no sistema roda no contexto de algum processo. O contexto consiste do estado que o programa precisa para executar corretamente. Esse estado inclui o código do programa e dados armazenados na memória, a pilha, o conteúdo dos registradores de propósito geral, o contador de programa, variáveis de ambiente e o conjunto das descrições dos arquivos abertos.

As abstrações chaves que o processo prover para a aplicação são:

- *Logical Control Flow*: Um processo fornece a cada programa a ilusão de que ele tem uso exclusivo do processador, embora muitos outros programas sejam executados simultaneamente no sistema.
- *Concurrent Flows*: Fluxos lógicos assumem diferentes formas em sistemas computacionais. Manipuladores de exceções, processos, manipuladores de sinais, encadeamentos e processos Java são exemplos de fluxos lógicos. Um fluxo lógico cuja execução se sobrepõe no tempo com outro fluxo é chamado de fluxo concorrente, e os dois fluxos são executados simultaneamente.
- *Private Address* Space: Um processo fornece a cada programa a ilusão de que ele tem uso exclusivo do espaço de endereço do sistema.
- *User and Kernel Modes*: Para que o kernel do sistema operacional forneça uma abstração de processo impermeável, o processador deve fornecer um mecanismo que restrinja as instruções que um aplicativo pode executar, bem como as partes do espaço de endereço que ele pode acessar. Um processo em execução no modo kernel pode executar qualquer instrução no conjunto de instruções e acessar qualquer localização de memória no sistema. Um processo no modo de usuário não tem permissão para executar instruções privilegiadas que fazem coisas como interromper o processador, alterar o bit de modo ou iniciar uma operação de E / S. Também não é permitido fazer referência direta a código ou dados na área do kernel do espaço de endereço.
- *Context Switches*: O kernel do sistema operacional implementa a multitarefa usando uma forma de alto nível de fluxo de controle excepcional, conhecida como switch de contexto. O kernel mantém um contexto para cada processo. O contexto é o estado que o kernel precisa para reiniciar um processo preempted. Consiste nos valores de objetos como registradores de propósito geral, registradores de ponto flutuante, contador de programa, pilha do usuário, registradores de status, pilha do kernel e várias estruturas de dados do kernel, como uma tabela de páginas que caracteriza o espaço de endereço, tabela de processos que contém informações sobre o processo atual e uma tabela de arquivos que contém informações sobre os arquivos que o processo abriu.

8.3 Tratamento de erros de chamada do sistema

Quando as funções de nível de sistema do Unix encontram um erro, elas normalmente retornam −1 e configuram a variável de inteiro global errno para indicar o que deu errado. Os programadores devem sempre verificar se há erros, mas, infelizmente, muitos ignoram a verificação de erros porque isso incha o código e dificulta a leitura.

8.4 Controle de Processo

O Unix prover um número de chamadas de sistema para manipulação de processo com programas em C. Vejamos alguns exemplos:

- pid_t getpid(void): retorna o PID do processo chamado.
- void exit(int status): termina o processo com o status de saída indicado.
- pid_t fork(void): cria um novo processo filho que é uma cópia do processo pai. É chamado uma vez, mais retorna duas e ambos os processos passam a ser tratados de forma concorrente. Duplicados, mas em espaços de endereço separados e compartilhamento de arquivos entre os dois.
- Pid_t waitpid(pid_t pid, int *status, int options): suspende a execução do processo até que o processo filho termine.
- unsigned int sleep(unsigned int secs): suspende um processo por um período de tempo especificado.
- int execve(const char *filename, const char *argv[], const char *envp[]): carrega e executa um novo programa no contexto do processo atual.

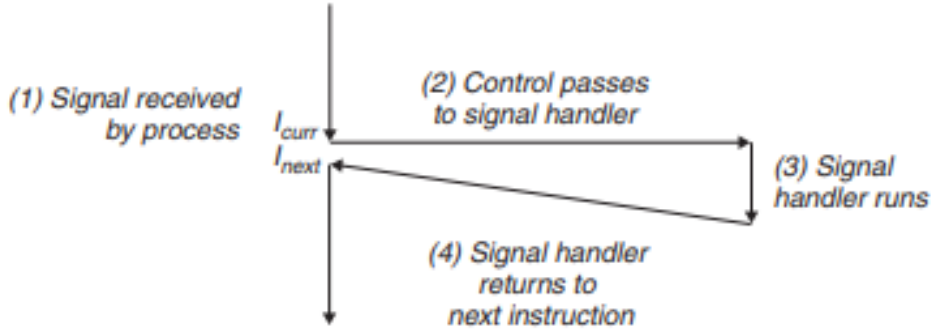
8.5 Sinais

Unix signal permite processos e o kernel interromperem outros processos. Sinais são uma pequena mensagem que notifica um processo que um evento de algum tipo ocorreu no sistema. Os sistemas Linux suportam 30 tipos diferentes de sinais com cada um correspondendo a algum tipo de evento do sistema. Sinais proveem um mecanismo para expor a ocorrência de exceções para processos do usuário.

8.5.1 Terminologia de Sinal

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

- Enviando um sinal: O kernel envia um sinal para um processo de destino atualizando alguns estados no contexto do mesmo. Esses sinais são entregues devido a ocorrência de um evento de sistema ou pela chamada da função *kill*. Um processo pode chamar a si mesmo.
- Recebendo um sinal: Um processo destino recebe um sinal quando é forçado pelo kernel a reagir de algum jeito para a entrega do sinal. O processo pode ignorar, terminar ou capturar o sinal.



8.5.2 Enviando sinais

Todos os mecanismos de envio de sinais contam com a noção de grupos de processo. Cada processo pertence a exatamente um grupo de processo, que é identificado por um inteiro positivo *process group ID*. Por padrão, um processo filho pertence ao mesmo grupo de processo que o seu pai. A mudança do grupo pode partir do próprio processo ou de um outro.

Unix shells usam a abstração de *job* para representar os processos que são criados como resultado da avaliação de uma única linha de comando. A qualquer momento, há no máximo um *job* em primeiro plano e zero ou mais trabalhos em segundo plano. Por exemplo, digitando

unix> ls | sort

cria um trabalho de primeiro plano que consiste em dois processos conectados por um *pipe Unix*: um executando o programa *ls*, o outro executando o programa sort. O shell cria um grupo de processos separado para cada *job*. Geralmente, o ID do grupo de processos é obtido de um dos processos pai no trabalho.

Processos enviam sinais para outros processos (incluindo eles mesmos) chamando a função *int kill(pid_t pid, int sig)*. O sinal pode ser enviado tanto para um processo ou para todos os processos em um grupo.

A função *unsigned int alarm(unsigned int secs)* pode ser utilizada para agendar o envio de um sinal, estilo um alarme.

8.5.3 Recebendo sinais

Quando o kernel retorna de tratar uma exceção e está pronto para passar o controle para o processo, primeiro ele checa o conjunto de sinais pendentes desbloqueados do processo. Se ele estiver vazio, então o kernel passa o controle para a próxima instrução. Caso contrário, o kernel escolhe algum sinal no conjunto e força o processo a recebê-lo. A recepção do sinal desencadeia algumas ações do processo. Uma vez que a ação é completada, então o controle passa de volta para a próxima instrução no fluxo de controle do processo. Cada processo tem uma ação padrão predefinida, que é uma das seguintes:

- O processo termina
- O processo termina e despeja o núcleo
- O processo para até ser reiniciado por um sinal SIGCONT
- O processo ignora o sinal

8.5.4 Problemas de manuseio de sinal

O manuseio de sinal é direto para programas que capturam um único sinal e depois terminam. No entanto, questões sutis surgem quando um programa captura múltiplos sinais.

- Sinais pendentes são bloqueados: Manipuladores de sinais Unix normalmente bloqueiam sinais pendentes do tipo atualmente sendo processado pelo manipulador.

- Sinais pendentes não são empilhados: A ideia chave é que a existência de um sinal pendente indica apenas que pelo menos um sinal chegou.
- Chamadas de sistemas podem ser interrompidas: As chamadas do sistema, como *read*, *wait* e *accept*, que potencialmente podem bloquear o processo por um longo período de tempo, são chamadas de *slow system calls*. Em alguns sistemas, *slow system calls* que são interrompidas quando um manipulador captura um sinal não retomam quando o manipulador de sinal retorna, mas em vez disso retornam imediatamente ao usuário com uma condição de erro.

8.5.5 Manipulador de Sinal Portátil

A diferença em manipular sinais semanticamente de um sistema para outro é um feio aspecto do manipulador de sinal do Unix. Para lidar com esse problema, o Posix padrão define a função *int sigaction(int signum, struct sigaction *act, struct sigaction *oldact)* que permite aos usuários especificar a semântica do manipulador de sinais que ele querem. Um exemplo é o *Signal wrapper*, que instala um manipulador de sinal com as seguintes semânticas:

- Apenas sinais do tipo atualmente sendo processadas pelo manipulador de sinais são bloqueadas.
- Como com todos os sinais implementados, sinais não são empilhados.
- Chamadas de sistemas de interrupção são automaticamente reiniciadas sempre que possível.

8.5.6 Bloqueando e Desbloqueando Explicitamente Sinais

Aplicações podem explicitamente bloquear e desbloquear sinais selecionados usando a função e *sigprocmask*. Ela muda o conjunto de sinais atualmente bloqueados. O comportamento dela pode ser:

- *SIG_BLOCK*: Adiciona os sinais no conjunto to blocked (blocked = blocked | set).
- *SIG_UNBLOCK:* Remove os sinais em conjunto do bloqueado (*blocked = blocked & ~set*).
- *SIG_SETMASK*: *blocked = set.*

8.5.7 Sincronizando Fluxo Para Evitar Bugs Desagradáveis de Concorrência

O problema de como programar fluxos simultâneos que leem e escrevem nos mesmos locais de armazenamento tem desafiado gerações de cientistas da computação. Em geral, o número de intercalações potenciais dos fluxos é exponencial no número de instruções. Alguns desses entrelaçamentos produzirão respostas corretas e outros não. O problema fundamental é de alguma forma sincronizar os fluxos simultâneos de modo a permitir o maior conjunto de intercalações viáveis, de tal modo que cada uma das intercalações viáveis produza uma resposta correta.

8.6 Saltos Não Locais

C prover uma forma de *exceptional control flow* de nível de usuário, chamado *nonlocal jump*, que transfere o controle diretamente de uma função para outra função atualmente executando sem ter que ir pela chamada normal e sequencia de retorno.

A função *setjmp* salva o ambiente de chamada atual no buffer *env*, para uso posterior por *longjmp*, e retorna 0. O ambiente de chamada inclui o contador de programa, ponteiro de pilha e registradores de propósito geral.

A função *longjmp* restaura o ambiente de chamada do buffer *env* e, em seguida, dispara um retorno da chamada *setjmp* mais recente que inicializou *env*. O *setjmp* retorna então com o valor de retorno diferente de zero.

As interações entre *setjmp* e *longjmp* podem ser confusas à primeira vista. A função *setjmp* é chamada uma vez, mas retorna várias vezes: uma vez quando o *setjmp* é chamado pela primeira vez e o ambiente de chamada é armazenado no buffer *env*, e uma vez para cada chamada *longjmp* correspondente. Por outro lado, a função *longjmp* é chamada uma vez, mas nunca retorna.

Uma aplicação importante de *nonlocal jump* é permitir um retorno imediato de uma chamada de função profundamente aninhada, geralmente como resultado da detecção de alguma condição de erro. Se uma condição de erro for detectada no fundo de uma chamada de função aninhada, podemos usar um *nonlocal jump* para retornar diretamente a um manipulador de erros comumente localizado em vez de desenrolar laboriosamente a pilha de chamadas.

Outra aplicação importante de *nonlocal jumps* é a ramificação de um manipulador de sinal para um local de código específico, em vez de retornar à instrução que foi interrompida pela chegada do sinal.

8.7 Ferramentas para manipulação de processos

Sistemas Linux proveem um número de ferramentas úteis para monitorar e manipular processos:

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