Universidade Federal de Uberlândia
FEELT – Faculdade de Engenharia Elétrica

Resumo Capítulo 3 Semana 3

Prof. Éder Alves de Moura

Sistemas Embarcados 2

Aluno: Pedro Jacob Favoreto - 11721EAU003

Data: 18/12/2021

3.1 Looking at Processes

Mesmo quando você se senta em frente ao computador, há processos em execução. Cada execução programa usa um ou mais processos.

3.1.1 Process IDs

Cada processo em um sistema Linux é identificado pelo seu ID de processo exclusivo, às vezes referido como pid. IDs de processo são números de 16 bits atribuídos sequencialmente pelo Linux à medida que novos processos são criados. Cada processo também tem um processo pai (exceto o processo init especial, nominados de “Processos zumbis”). Assim, você pode pensar nos processos em um Linux sistema conforme organizado em uma árvore, com o processo init em sua raiz. O ID do processo pai, ou ppid, é simplesmente o ID do processo do pai do processo. Ao referir-se a IDs de processo em um programa C ou C ++, sempre use o pid_t typedef, que é definido em <sys / types.h>. Um programa pode obter o ID do processo de o processo em execução com a chamada de sistema getpid () e pode obter o processo ID de seu processo pai com a chamada de sistema getppid ().

3.1.2 Viewing Active Processes

O comando ps exibe os processos que estão sendo executados em seu sistema. A versão GNU / Linux do ps apresenta muitas opções porque tenta ser compatível com versões do ps em várias outras variantes do UNIX. Essas opções controlam quais processos estão listados e quais informações sobre cada um são mostradas. Por padrão, invocar o ps exibe os processos controlados pelo terminal ou terminal janela na qual o ps é invocado.
A opção -e instrui o ps a exibir todos os processos em execução no sistema. -o pid, ppid, opção de comando diz ao ps quais informações mostrar sobre cada processo— neste caso, o ID do processo, o ID do processo pai e o comando em execução neste processo.

3.1.3 Killing a Process

Você pode matar um processo em execução com o comando kill. Basta especificar no com linha de comando o ID do processo do processo a ser eliminado. O comando kill funciona enviando ao processo um SIGTERM, ou terminação, signal.1 Isso faz com que o processo seja encerrado, a menos que o programa em execução explicitamente trata ou mascara o sinal SIGTERM.

3.2 Creating Processes

Duas técnicas comuns são usadas para criar um novo processo. A primeira é relativamente simples, mas deve ser usada com moderação porque é ineficiente e tem consideravelmente riscos de segurança. A segunda técnica é mais complexa, mas oferece maior flexibilidade, velocidade e segurança.

3.2.1 Using system

A função do sistema na biblioteca C padrão fornece uma maneira fácil de executar um comando de dentro de um programa, como se o comando tivesse sido digitado em um Concha. Na verdade, o sistema cria um subprocesso executando o shell Bourne padrão (/ bin / sh) e entrega o comando a esse shell para execução.

Como a função do sistema usa um shell para invocar seu comando, ela está sujeita a recursos, limitações e falhas de segurança do shell do sistema. Você não pode confiar na disponibilidade de qualquer versão específica do shell Bourne. Em muitos sistemas UNIX, / bin / sh é um link simbólico para outro shell. Portanto, é preferível usar o método fork e exec para processos de criação.

3.2.2 Using fork and exec

A API do DOS e do Windows contém a família de funções de geração. Estas funções tomam como argumento o nome de um programa a ser executado e cria uma nova instância de processo desse programa. O Linux não contém uma única função que faz tudo isso em uma única etapa. Em vez disso, o Linux fornece uma função, fork, que faz um processo filho que é um exato cópia de seu processo pai. O Linux oferece outro conjunto de funções, a família exec, que faz com que um determinado processo deixe de ser uma instância de um programa e, em vez disso, tornar-se uma instância de outro programa. Para gerar um novo processo, você primeiro usa fork para fazer uma cópia do processo atual. Então você usa o exec para transformar um destes processos em uma instância do programa que você deseja gerar.

Calling fork

Quando um programa chama fork, um processo duplicado, chamado de processo filho, é criado. O processo pai continua executando o programa a partir do ponto em que o fork foi chamado. O processo filho também executa o mesmo programa no mesmo lugar. O processo filho é um novo processo e portanto, tem um novo ID de processo, diferente do ID de processo de seu pai. Uma maneira de programa para distinguir se está no processo pai ou se o processo filho deve se chamar getpid. No entanto, a função fork fornece diferentes valores de retorno para o pai e processo filho - um processo "entra" na chamada de bifurcação e dois processos "saem", com diferentes valores de retorno. O valor de retorno no processo pai é o ID do processo filho. O valor de retorno no processo filho é zero. Porque nenhum processo nunca tem um ID do processo de zero, isso torna mais fácil para o programa se ele agora está sendo executado como o pai ou filho. A Listagem 3.3 é um exemplo de uso de fork para duplicar o processo de um programa.

Using the exec Family

As funções exec substituem o programa em execução em um processo por outro programa. Quando um programa chama uma função exec, esse processo para imediatamente de executar aquele programa e começa a executar um novo programa desde o início, assumindo que o chamado exec não encontrou um erro. Dentro da família exec, existem funções que variam ligeiramente em suas capacidades e como eles são chamados. Algumas funções que contêm a letra p em seus nomes (execvp e execlp) aceitam um nome do programa e procure um programa com esse nome na execução atual caminho; funções que não contêm op devem receber o caminho completo do programa a ser executado. Algumas funções que contêm a letra v em seus nomes (execv, execvp e execve) aceitar a lista de argumentos para o novo programa como uma matriz terminada em NULL de ponteiros para strings. Funções que contêm a letra l (execl, execlp e execle) aceitar a lista de argumentos usando o mecanismo varargs da linguagem C. n Funções que contêm a letra e em seus nomes (execve e execle) aceitam um argumento adicional, uma matriz de variáveis de ambiente. O argumento deve ser uma matriz terminada em NULL de ponteiros para cadeias de caracteres. Cada sequência de caracteres deve ter o formato “VARIÁVEL = valor”. Como exec substitui o programa de chamada por outro, ele nunca retorna, a menos que um ocorreu um erro. A lista de argumentos passada para o programa é análoga ao argumento da linha de comando que você especifica para um programa ao executá-lo a partir do shell. Eles estão disponíveis por meio dos parâmetros argc e argv para principal. Lembre-se, quando um programa é invocado a partir do shell, o shell define o primeiro elemento da lista de argumentos argv [0]) como o nome do programa, o segundo elemento

da lista de argumentos (argv [1]) para o primeiro argumento da linha de comando e assim por diante. Quando você usa uma função exec em seu programa, você também deve passar o nome da função como o primeiro elemento do argumento lista de ment.

Using fork and exec Together
Um padrão comum para executar um subprograma dentro de um programa é primeiro bifurcar o processo e, em seguida, executar o subprograma. Isso permite que o programa de chamada continue a execução no processo pai, enquanto o programa de chamada é substituído pelo subprograma no processo filho.

3.2.3 Process Scheduling
O Linux agenda os processos pai e filho independentemente; não há garantia de qual será executado primeiro, ou por quanto tempo ele executará antes que o Linux o interrompa e deixe que outro processo (ou algum outro processo no sistema) seja executado. Em particular, nenhum, parte ou todos do comando ls pode ser executado no processo filho antes que o pai seja concluído.2 Linux promete que cada processo será executado eventualmente - nenhum processo ficará completamente sem recursos de execução. Você pode especificar que um processo é menos importante - e deve receber uma prioridade mais baixa - atribuindo a ele um valor de gentileza mais alto. Por padrão, todo processo tem uma gentileza zero. Um valor de niceness mais alto significa que o processo recebe uma prioridade de execução menor; inversamente, um processo com uma gentileza menor (ou seja, negativa) obtém mais tempo de execução. Para executar um programa com uma gentileza diferente de zero, use o comando nice, especificando o valor niceness com a opção -n. Você pode usar o comando renice para alterar a gentileza de um processo em execução de a linha de comando. Para alterar a gentileza de um processo em execução programaticamente, use a função nice. Seu argumento é um valor de incremento, que é adicionado ao valor de gentileza do processo que o chama. Lembre-se de que um valor positivo aumenta o valor da gentileza e, portanto, reduz a prioridade de execução do processo. Observe que apenas um processo com privilégio de root pode executar um processo com um nice negativo valor de niceness ou reduzir o valor de niceness de um processo em execução. Isso significa que você pode especificar valores negativos para os comandos nice e renice apenas quando conectado como root, e apenas um processo rodando como root pode passar um valor negativo para a função nice. Isso evita que usuários comuns obtenham a prioridade de execução de outros usando o sistema.

3.3 Signals
Sinais são mecanismos de comunicação e manipulação de processos no Linux. Um sinal é uma mensagem especial enviada a um processo. Os sinais são assíncronos; quando um processo recebe um sinal, ele processa o sinal imediatamente, sem terminar a curva função de aluguel ou até mesmo a linha de código atual. Existem várias dezenas de sinais diferentes, mas cada um com um significado diferente. Cada tipo de sinal é especificado por seu número de sinal, mas em programas, você geralmente se refere a um sinal pelo nome. No Linux, eles são definidos em /usr/include/bits/signum.h. (Você não deve incluir este arquivo de cabeçalho diretamente no seus programas; em vez disso, use <signal.h>.) Quando um processo recebe um sinal, ele pode fazer uma de várias coisas, dependendo da disposição do sinal. Para cada sinal, há uma disposição padrão, que determina o que acontece com o processo se o programa não especifica algum outro comportamento. Para a maioria tipos de sinal, um programa pode especificar algum outro comportamento, seja

para ignorar o sinal ou para chamar uma função especial de manipulador de sinal para responder ao sinal. Se um manipulador de sinal é usado, o programa atualmente em execução é pausado, o manipulador de sinais é executado e, quando o manipulador de sinais retorna, o programa continua. O sistema Linux envia sinais para processos em resposta a condições específicas. Por instância, SIGBUS (erro de barramento), SIGSEGV (violação de segmentação) e SIGFPE (flutuante exceção de ponto) pode ser enviada para um processo que tenta realizar uma operação ilegal. A disposição padrão para esses sinais para encerrar o processo e produzir um arquivo principal. Um processo também pode enviar um sinal para outro processo. Um uso comum deste mecanismo é encerrar outro processo enviando-lhe um sinal SIGTERM ou SIGKILL.3 Outro uso comum é enviar um comando para um programa em execução. definidos ”sinais são reservados para esta finalidade: SIGUSR1 e SIGUSR2. O sinal SIGHUP às vezes também é usado para esta finalidade, normalmente para despertar um programa inativo ou fazer com que um programa releia seus arquivos de configuração. A função sigaction pode ser usada para definir uma disposição do sinal. O primeiro parâmetro é o número do sinal. Os próximos dois parâmetros são ponteiros para estruturas de ligação; o primeiro deles contém a disposição desejada para esse número de sinal, enquanto o segundo recebe a disposição anterior. O campo mais importante no primeiro ou no segundo estrutura sigaction é sa_handler. Pode ter um de três valores:

- SIG_DFL, que especifica a disposição padrão para o sinal.
- SIG_IGN, que especifica que o sinal deve ser ignorado.
- Um ponteiro para uma função de manipulador de sinal. A função deve ter um parâmetro, o número do sinal e retornar vazio.

Como os sinais são assíncronos, o programa principal pode estar em um estado muito frágil quando um sinal é processado e, portanto, enquanto uma função de tratamento de sinal é executada. Portanto, você deve evitar realizar quaisquer operações de I / O ou chamar a maioria das bibliotecas e funções de sistema de manipuladores de sinal. Um manipulador de sinais deve realizar o trabalho mínimo necessário para responder ao sinal e, em seguida, retornar o controle ao programa principal (ou encerrar o programa). Dentro na maioria dos casos, consiste simplesmente em registrar o fato de que ocorreu um sinal. o programa então verifica periodicamente se um sinal ocorreu e reage de acordo. É possível que um manipulador de sinal seja interrompido pela entrega de outro sinal. Embora possa parecer uma ocorrência rara, se ocorrer, será muito difícil diagnosticar e depurar o problema. Portanto, você deve ter muito cuidado com o que seu programa faz em um manipulador de sinal. O sinal SIGTERM pede que um processo termine; o processo pode ignorar a solicitação mascarando ou ignorando o sinal. O sinal SIGKILL sempre mata o processo imediatamente porque o processo não pode mascarar ou ignorar o SIGKILL.
Até mesmo atribuir um valor a uma variável global pode ser perigoso porque a atribuição pode realmente ser realizado em duas ou mais instruções de máquina, e um segundo sinal pode ocorrer entre eles, deixando a variável em um estado corrompido. Se você usar um global variável para sinalizar um sinal de uma função de manipulador de sinal, deve ser do tipo especial sig_atomic_t. O Linux garante que as atribuições a variáveis deste tipo sejam per- formado em uma única instrução e, portanto, não pode ser interrompido no meio. No Linux, sig_atomic_t é um int comum; na verdade, as atribuições a tipos inteiros têm o tamanho de int ou menores, ou para ponteiros, são atômicos. Se você quiser escrever um programa que seja portátil para qualquer sistema UNIX padrão, entretanto, usa sig_atomic_t para essas variáveis globais.

3.4 Process Termination

Normalmente, um processo termina de duas maneiras. O programa em execução chama a função de saída ou a função principal do programa retorna. Cada processo tem uma saída de código: um número que o processo retorna ao seu pai. O código de saída é o argumento passado para a função de saída ou o valor retornado de principal. Um processo também pode terminar de forma anormal, em resposta a um sinal. Por exemplo, os sinais SIGBUS, SIGSEGV e SIGFPE mencionados anteriormente fazem com que o processo termine antes. Outros sinais são usados para encerrar um processo explicitamente. O sinal SIGINT é enviado a um processo quando o usuário tenta encerrá-lo digitando Ctrl + C em seu terminal. O sinal SIGTERM é enviado pelo comando kill. A disposição padrão para ambos é encerrar o processo. Ao chamar a função de aborto, um processo envia a si mesmo o Sinal SIGABRT, que termina o processo e produz um arquivo principal. O sinal de término completo é SIGKILL, que termina um processo imediatamente e não pode ser bloqueado ou controlado por um programa. Qualquer um desses sinais pode ser enviado usando o comando kill especificando um extra sinalizador de linha de comando; por exemplo, para encerrar um processo problemático enviando um SIGKILL, invoque o seguinte, onde pid é o ID do processo: Para enviar um sinal de um programa, use a função kill. O primeiro parâmetro é o tar- obter o ID do processo. O segundo parâmetro é o número do sinal; use SIGTERM para simular o comportamento padrão do comando kill. Por exemplo, onde o filho pid contém o ID do processo filho, você pode usar a função kill para encerrar um filho processo do pai chamando-o assim: Inclua os cabeçalhos <sys / types.h> e <signal.h> se você usar a função kill. Por convenção, o código de saída é usado para indicar se o programa foi executado corretamente. Um código de saída zero indica a execução correta, enquanto um código de saída diferente de zero indica que ocorreu um erro. No último caso, o valor particular retornado pode dar alguma indicação da natureza do erro. É uma boa ideia ficar com esta condição de atenção em seus programas porque outros componentes do sistema GNU / Linux assumem este comportamento. Por exemplo, shells assumem esta convenção quando você conecta vários programas com o && (lógico e) e || operadores (lógicos ou). Portanto, você deve retornar explicitamente zero de sua função principal, a menos que ocorra um erro.

Com a maioria dos shells, é possível obter o código de saída do mais recentemente executado programa usando o especial $? variável. Os códigos de saída acima de 128 têm um significado especial - quando um processo é encerrado por um sinal, seu código de saída é 128 mais o número do sinal.

3.4.1 Waiting for Process Termination

O Linux é um sistema de multitarefas, ambos os processos parecem funcionar simultaneamente, e você não pode prever se o programa será executado antes ou depois da execução do processo pai. Em algumas situações, porém, é desejável que o processo pai espere até um ou mais processos filhos serem concluídos. Isso pode ser feito com a família de espera do sistema chamadas. Essas funções permitem que você aguarde a conclusão de um processo de execução e habilite o processo pai para recuperar informações sobre a rescisão de seu filho. Existem quatro diferentes chamadas de sistema na família de espera; você pode escolher obter algumas ou muitas informações informações sobre o processo que saiu, e você pode escolher se se preocupa com qual processo filho foi encerrado.

3.4.2 The wait System Calls

A mais simples dessas funções é chamada simplesmente de espera. Ele bloqueia o processo de chamada até um de seus processos filho sair (ou ocorre um erro). Ele retorna um código de status por meio de um número inteiro de argumento de ponteiro, a partir do qual você pode extrair informações sobre como o processo filho saiu.

Você pode usar a macro WIFEXITED para determinar o status de saída de um processo filho se esse processo saiu normalmente (por meio da função de saída ou retornando do principal) ou morreu devido a um sinal não tratado. No último caso, use a macro WTERMSIG para extrair de seu status de saída, o número do sinal pelo qual morreu. A função waitpid pode ser usada para esperar que um processo filho específico saia em vez de qualquer processo filho. A função wait3 retorna estatísticas de uso da CPU sobre a saída do processo filho, e o wait4 permite que você especifique opções adicionais sobre quais processos esperar.

3.4.3 Zombie Processes

Se um processo filho terminou enquanto seu pai está chamando uma função de espera, o o processo filho desaparece e seu status de encerramento é passado para seu pai por meio da chamada de espera. Quando um processo filho termina e o pai não está chamando wait? ele não desaparece simplesmente, porque então informações sobre seu encerramento - como se ele sai normalmente e, em caso afirmativo, qual é o seu status de saída - seria perdido. Em vez de, quando um processo filho termina, ele se torna um processo zumbi.

Um processo zumbi é um processo que foi encerrado, mas ainda não foi limpo. É responsabilidade do processo pai limpar seus filhos zumbis. A espera das funções fazem isso também, então não é necessário rastrear se o seu processo filho ainda está sendo executado antes de esperar por ele. Suponha, por exemplo, que um programa bifurque uma processo filho, executa alguns outros cálculos e, em seguida, chamadas em espera. Se o processo filho não terminar nesse ponto, o processo pai irá bloquear na chamada de espera até que o processo filho termine. Se o processo filho terminar antes que o processo pai chame a espera, o processo filho se torna um zumbi. Quando o processo pai chama espera, o zumbi o status de rescisão do filho é extraído, o processo filho é excluído e a chamada de espera retorna imediatamente.

% ps -e -o pid,ppid,stat,cmd lista o ID do processo, ID do processo pai, status do processo e comando do processo linha. O processo filho é marcado como <defunct>, e seu código de status é Z, para zumbi. Sem tentar executando o ps novamente, e observe que ambos os processos do make-zombie se foram. sai do programa, seus filhos são herdados por um processo especial, o programa init, que sempre é executado com ID de processo 1 (é o primeiro processo iniciado quando o Linux é inicializado). O processo init limpa automaticamente todos os processos filho zumbi que ele herda.

3.4.4 Cleaning Up Children Asynchronously

Se você estiver usando um processo filho simplesmente para executar outro programa, não há problema em chamar wait imediatamente no processo pai, que será bloqueado até que o processo filho seja concluído. Mas, frequentemente, você deseja que o processo pai continue em execução, como um ou mais filhos executar de forma síncrona. Como você

pode ter certeza de que limpará os processos filhos que foram concluídos para que você não saia dos processos zumbis, que consomem o sistema recursos, por aí? Uma abordagem seria o processo pai chamar wait3 ou wait4 periodicamente, para limpar crianças zumbis. Ligar em espera para este propósito não funciona bem porque, se nenhum filho tiver terminado, a chamada será bloqueada até que um o faça. No entanto, espere 3 e wait4 recebe um parâmetro sinalizador adicional, para o qual você pode passar o valor do sinalizador WNOHANG. Com este sinalizador, a função é executada em modo não bloqueador - ela limpará um processo filho, se houver, ou simplesmente retornar, se não houver. O valor de retorno da chamada é o ID do processo do filho encerrado no primeiro caso, ou zero no último caso. Uma solução mais elegante é notificar o processo pai quando um filho é encerrado. Quando um processo filho termina, o Linux envia ao processo pai o sinal SIGCHLD. A disposição padrão deste sinal é não fazer nada, e é por isso que você não pode já ter feito isso antes. Portanto, uma maneira fácil de limpar os processos filhos é manipulando SIGCHLD. Claro, ao limpar o processo filho, é importante armazenar seu status de encerramento se estas informações são necessárias, porque uma vez que o processo é limpo usando esperar, essa informação não está mais disponível.