CS 241: programação do sistema

- atribuições
- Quizzes
- Socorro!
- Cronograma
- Honras
- Funcionários
- Livro didático
- Links

Processos

- Processos
  - Descritores de Arquivos
  - Processos
  - Conteúdo do Processo
    - Layout de memória
    - Outros conteúdos
  - Introdução ao Fork
    - Uma palavra de aviso
    - Funcionalidade Forqueta
    - O que é uma bomba de garfo?
    - Sinais
    - Detalhes de Garfo POSIX
  - Esperando e Executando
    - Sair dos status
    - Zumbis e órfãos
    - Extra: Como posso esperar de forma assíncrona pelo meu filho usando o SIGCHLD?
  - exec
    - Detalhes do POSIX Exec
    - Atalhos
  - O padrão fork-exec-wait
    - variáveis ambientais
  - Leitura Adicional
    - Tópicos
  - Perguntas / Exercícios

Quem precisa de isolamento do processo? - **Marketing Intel no colapso e no espectro**
Para entender o processo, você precisa entender a ordem de inicialização. No começo, há um kernel. O kernel do sistema operacional é um software especial. Esta é a parte do software que é carregada antes de todos os seus outros programas considerarem a inicialização. O que o kernel faz é o seguinte, abreviado

1. O sistema operacional executa ROM ou somente código de leitura
2. O sistema operacional, em seguida, executa um boot_loader ou extensões EFI hoje em dia
3. O boot_loader carrega seus kernels
4. Seu kernel executa o init para https://en.wikipedia.org/wiki/Bootstrapping do nada
5. O kernel executa scripts de inicialização
6. O kernel executa scripts userland e você pode usar seu computador!

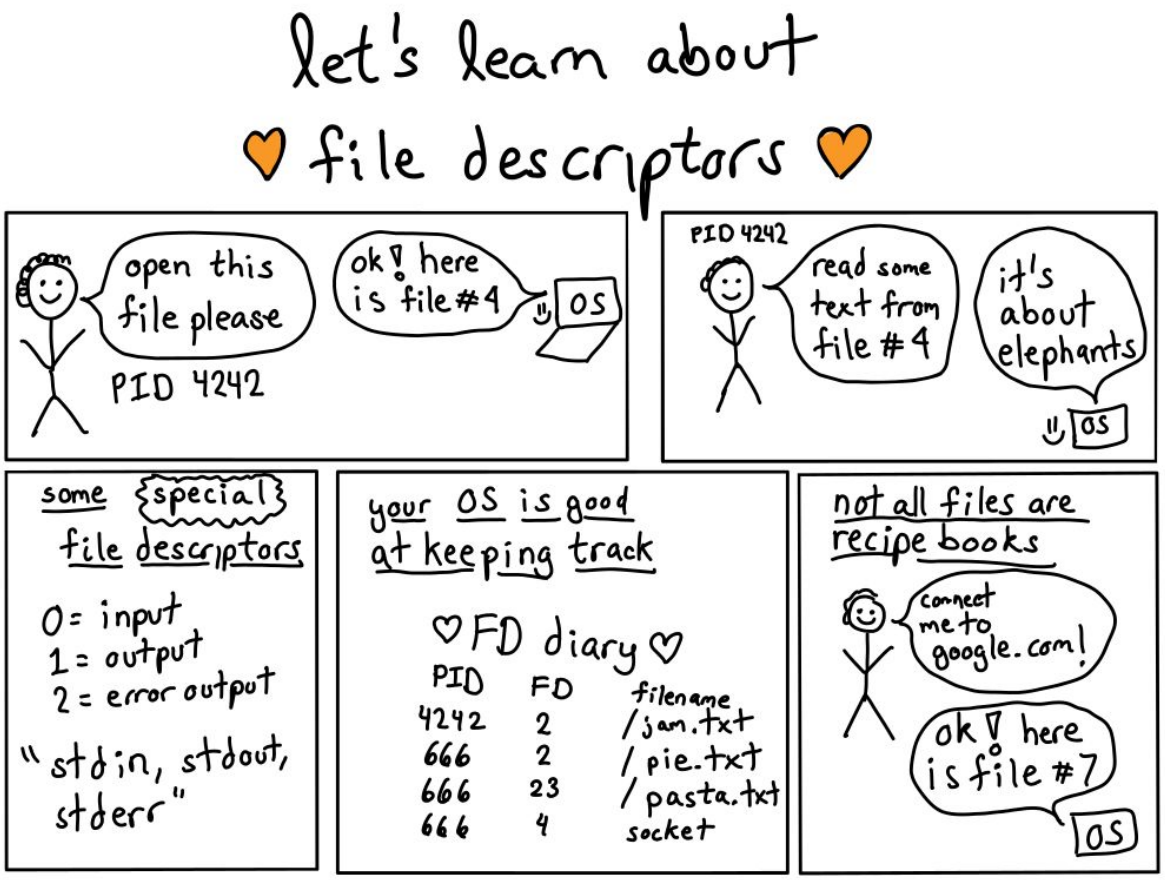
Você não precisa conhecer as especificidades do processo de inicialização, mas aí está. Quando você está executando no espaço do usuário, o kernel fornece algumas operações importantes com as quais os programas não precisam se preocupar.

- Agendando Processos e Encadeamentos
- Manipulando Primitivos de Sincronização
- Fornecendo chamadas do sistema como write ou read
- Gerencia a memória virtual e dispositivos binários de baixo nível, como drivers USB
- Lida com a leitura e compreensão de um sistema de arquivos
- Lida com comunicação em redes
- Lida com comunicações com outros processos
- Vinculando dinamicamente bibliotecas

O kernel lida com tudo isso no modo kernel. O modo kernel permite que você tenha mais poder, como executar instruções extras da CPU, mas ao custo de uma falha trava todo o seu computador. É com isso que você vai interagir nesta aula.
Mais relevante, o kernel cria o primeiro processo init.d O init.d inicializa coisas como sua GUI, terminais, etc. O que é importante é que o sistema operacional só realmente cria um processo por padrão, todos os outros processos são fork e executados a partir desse único processo.

Descritores de Arquivo

Embora tenha sido mencionado no último capítulo, vamos lhe dar um lembrete rápido sobre os descritores de arquivos. Aqui está um zine de Julia Evans que detalha um pouco (Evans # ref-evans_2018).



Como mostra o pequeno zine, o Kernel monitora os descritores de arquivos e o que eles apontam. Veremos mais adiante que os descritores de arquivos não precisam apontar para arquivos reais e o sistema operacional os acompanha para você. Além disso, observe que, entre os processos, os descritores de arquivos podem ser reutilizados, mas dentro de um processo eles são exclusivos. Os descritores de arquivos podem ter uma noção de posição. Você pode ler um arquivo no disco completamente porque o sistema operacional controla a posição no arquivo, e isso também pertence ao seu processo. Outros descritores de arquivos apontam para soquetes de rede e várias outras informações.

Processos

Um processo uma instância de um programa de computador que pode estar em execução. Os processos têm muitas coisas à sua disposição. No início de cada programa, você obtém um processo, mas cada programa pode realizar mais processos. Um programa consiste no seguinte.

- Um formato binário: Isso informa ao sistema operacional quais conjuntos de bits no binário são o que - qual parte é executável, quais partes são constantes, quais bibliotecas devem ser incluídas etc.
- Um conjunto de instruções de máquina
- Um número que indica a instrução a partir de
- Constantes

- Bibliotecas para vincular e onde preencher o endereço dessas bibliotecas

Os processos são muito poderosos, mas estão isolados! Isso significa que, por padrão, nenhum processo pode se comunicar com outro processo. Isso é muito importante porque se você tiver um sistema grande (digamos, as Estações de Engenharia da Universidade de Illinois), então você quer que alguns processos tenham privilégios mais altos do que seu usuário comum, e um certamente não quer que o usuário médio seja capaz de trazer todo o sistema de propósito ou acidentalmente, modificando um processo. Como a maioria já percebeu até agora, se você colocou o trecho de código a seguir em um programa, as variáveis não seriam compartilhadas entre duas chamadas paralelas do programa.
int secrets ; secrets ++ ; printf ("%d \n " , secrets);
Em dois terminais diferentes, como você poderia imaginar, ambos imprimiriam 1 e não 2. Mesmo se mudássemos o código para fazer algo realmente hacky, não haveria como alterar o estado de outro processo sem querer. Existem maneiras de mudá-lo propositadamente embora.

Conteúdo do processo

Layout de memória

Quando um processo é iniciado, ele recebe seu próprio espaço de endereço. Cada processo recebe o seguinte.

- **Uma pilha** . A pilha é o local onde as variáveis automáticas e os endereços de retorno de chamada de função são armazenados. Toda vez que uma nova variável é declarada, o programa move o ponteiro da pilha para baixo para reservar espaço para a variável. Este segmento da pilha é gravável, mas não é executável. Se a pilha crescer muito - o que significa que ela cresce além de um limite predefinido ou cruza o heap - você obterá um stackoverflow que provavelmente resultará em um SEGFAULT. **A pilha é alocada estaticamente por padrão, o que significa que há apenas uma certa quantidade de espaço para o qual se pode escrever**
- **Um monte** . O heap é uma região de memória contígua e em expansão (“Overview of Malloc” # ref-mallocinternals). Se você quiser alocar um objeto grande, ele vai aqui. O heap começa na parte superior do segmento de texto e cresce para cima, o que significa que, às vezes, quando você chama malloc ele pede ao sistema operacional para empurrar o limite do heap para cima. Mais sobre isso no capítulo de alocação de memória. Esta área também é gravável, mas não é executável. Pode-se ficar sem memória heap se o sistema for restrito ou se você ficar sem endereços. Isso é mais comum em um sistema de 32 bits.
- **Um segmento de dados** ou conforme tentaremos nos referir a ele como o segmento inicializado. Isso contém todos os seus globals e quaisquer outras variáveis de duração estática. Esta seção começa no final do segmento de texto e é de tamanho estático porque a quantidade de globais é conhecida em tempo de compilação. Esta seção é gravável (Van der Linden # ref-van1994expert P. 124). Mais notavelmente, esta seção contém variáveis que foram inicializadas com um inicializador estático como

int global = 1 ;
Mas não aqueles que são inicializados por padrão, esse é o próximo segmento.

- **Um Segmento BSS** ou o Segmento de Serviço Básico. Isso contém todos os seus globais e quaisquer outras variáveis de duração estática que são implicitamente zeradas.
  int assumed_to_be_zero ;
  Não é um erro supor que isso será zero, porque, do contrário, teríamos um risco de segurança de outros processos. Eles são colocados em uma seção diferente para acelerar o tempo de inicialização do processo. Esta seção começa no final do segmento de dados e também é de tamanho estático porque a quantidade de globais é conhecida em tempo de compilação. Neste ponto, o BSS e o segmento de dados são combinados e confusamente referidos como o segmento de dados (Van der Linden # ref-van1994expert P. 124). De agora em diante, por conveniência, adotaremos que o segmento de dados seja o segmento de dados
- **Um segmento de texto** . Aqui é onde todo o seu código é armazenado - todos os 1's e 0's. O contador de programa passa por este segmento executando instruções e movendo a próxima instrução. É importante observar que esta é a única seção executável do código criado por padrão. Se você tentar mudar o código enquanto ele estiver rodando, provavelmente você vai SEGFAULT. Existem maneiras de contornar isso, mas não iremos explorar as pessoas neste curso. Por que não começa no zero? Por causa de https://en.wikipedia.org/wiki/Address_space_layout_randomization . Está fora do escopo dessa classe, mas saiba que ela existe. O endereço pode ser corrigido se compilado com o sinalizador DEBUG embora.

Outros conteúdos

Para acompanhar todos esses processos, seu sistema operacional atribui a cada processo um número e esse processo é chamado de PID, ID do processo. Os processos também têm um ppid que é curto para o id do processo pai. Todo processo tem um pai, esse pai pode ser init.d
Os processos também podem conter

- Estado de Execução - Se um processo está ficando pronto, em execução, interrompido, encerrado etc. (mais sobre isso durante o capítulo Agendamento).
- Descritores de Arquivo - Lista de mapeamentos de inteiros para dispositivos reais (arquivos, sticks USB, soquetes)
- Permissões - em qual user o arquivo está sendo executado e a que group o processo pertence. O processo só pode, então, fazer isso admissível ao user ou group como abrir um arquivo que o user tenha feito exclusividades. Existem truques para fazer com que um programa não seja o usuário que iniciou o programa, ou seja, o sudo pega um programa que o user inicia e o executa como root .
- Arguments - uma lista de strings que informa ao seu programa quais parâmetros executar sob
- Lista de ambientes - uma lista de cadeias no formato NAME=VALUE que pode ser modificada.

Se queremos dar a você a definição técnica, todo o POSIX diz que um processo precisa de um espaço de thread e endereço, mas a maioria dos desenvolvedores de kernel e usuários sabem que isso não é suficiente (“Definições” # ref-process_def).

Introdução ao Fork

Uma palavra de aviso

O processo de bifurcação é uma ferramenta poderosa e perigosa. Se você estragar e causar uma bifurcação, **você pode derrubar todo o sistema** . Para reduzir as chances de isso, limite o seu número máximo de processos a um número pequeno, por exemplo, 40, digitando ulimit -u 40 em uma linha de comando. Note que este limite é somente para o usuário, o que significa que se você for uma bomba, então você não será capaz de matar todos os processos que você acabou de criar, já que chamar o killall requer o shell to fork() … irônico, certo? Uma solução é gerar outra instância de shell como outro usuário (por exemplo, root) antes da mão e matar processos a partir daí. Outra é usar o comando built in exec para matar todos os processos do usuário (cuidado você só tem uma chance nisso). Finalmente, você pode reinicializar o sistema, mas você só tem uma chance com a função exec. Ao testar o código fork (), certifique-se de ter acesso root e / ou físico à máquina envolvida. Se você precisar trabalhar no código fork () remotamente, lembre-se de que **kill -9 -1** salvará você no caso de uma emergência.
TL: DR: Garfo pode ser **extremamente** perigoso se você não estiver preparado para isso. **Você foi avisado.**

Funcionalidade Forqueta

A chamada do sistema fork separa o processo atual para criar um novo processo. Ele cria um novo processo chamado processo filho, duplicando o estado do processo existente com algumas pequenas diferenças. O processo filho não inicia a partir do principal. Em vez disso, ele executa a próxima linha após o fork() assim como o processo pai. Assim como uma observação secundária, em sistemas UNIX mais antigos, todo o espaço de endereço do processo pai era copiado diretamente, independentemente de o recurso ter sido modificado ou não. Atualmente, o kernel executa https://en.wikipedia.org/wiki/Copy-on-write , o que economiza muitos recursos, além de ser eficiente em termos de tempo (Bovet e Cesati # ref-Bovet: 2005: ULK: 1077084 Copy -on-write seção). Aqui está um exemplo muito simples
printf ("I'm printed once! \n "); fork (); // Now there are two processes running if fork succeeded // and each process will print out the next line. printf ("You see this line twice! \n ");
Aqui está um exemplo simples dessa clonagem de espaço de endereço. O seguinte programa pode imprimir 42 duas vezes - mas a fork() é posterior à printf ! Por quê?
#include <unistd.h> /*fork declared here*/
#include <stdio.h> /* printf declared here*/
int main () { int answer = 84 >> 1 ; printf ("Answer: %d" , answer); fork (); return 0 ; }
A linha printf *é* executada apenas uma vez, mas observe que o conteúdo impresso não é liberado para a saída padrão. Não há nova linha impressa, não chamamos fflush ou fflush o

modo de armazenamento em buffer. O texto de saída ainda está na memória de processo aguardando para ser enviado. Quando fork() é executado, toda a memória do processo é duplicada, incluindo o buffer. Assim, o processo filho inicia com um buffer de saída não vazio que será liberado quando o programa for encerrado. Também dizemos porque o conteúdo pode não ser escrito, dado também uma saída incorreta do programa.
Para escrever código diferente para o processo pai e filho, verifique o valor de retorno de fork() . Se fork() retorna -1, isso implica que algo deu errado no processo de criação de um novo filho. Deve-se verificar o valor armazenado em *errno* para determinar que tipo de erro ocorreu. O Commons um inclui EAGAIN e ENOMEM que são essencialmente tentar novamente e sem memória. Da mesma forma, um valor de retorno de 0 indica que estamos no processo filho, enquanto um inteiro positivo mostra que estamos no processo pai. O valor positivo retornado por fork() fornece o id do processo (*pid*) do filho.
Outra maneira de lembrar qual é que o processo filho pode encontrar seu pai - o processo original que foi duplicado - chamando getppid() - portanto, não precisa de nenhuma informação de retorno adicional de fork() . Na palavra POSIX, você só tem um processo pai. O processo pai, no entanto, só pode descobrir o id do novo processo filho a partir do valor de retorno de fork :
pid_t id = fork (); if (id == - 1) exit (1); // fork failed if (id > 0) { // I'm the original parent and // I just created a child process with id 'id' // Use waitpid to wait for the child to finish } else { // returned zero // I must be the newly made child process }
Um exemplo ligeiramente bobo é mostrado abaixo. O que vai imprimir? Experimente com vários argumentos para o seu programa.
#include <unistd.h> #include <stdio.h> int main (int argc , char ** argv) { pid_t id ; int status ; while (-- argc && (id = fork ())) { waitpid (id , & status , 0); /* Wait for child*/ } printf ("%d:%s \n " , argc , argv [argc]); return 0 ; }
Outro exemplo está abaixo. Este é o incrível paralelismo aparente-O (N) *sleepsort* é o vencedor bobo de hoje. Publicado pela primeira vez em https://dis.4chan.org/read/prog/1295544154 . Uma versão deste algoritmo de ordenação horrível mas divertido é mostrada abaixo.
int main (int c , char ** v) { while (-- c > 1 && ! fork ()); int val = atoi (v [c]); sleep (val); printf ("%d \n " , val); return 0 ; }
O algoritmo não é realmente O (N) por causa de como o planejador do sistema funciona.

O que é uma bomba de garfo?

Uma bifurcação é o que avisamos anteriormente. Uma 'fork fork' é quando você tenta criar um número infinito de processos. Isso muitas vezes trará um sistema para quase-parar enquanto ele tenta alocar tempo e memória da CPU para um grande número de processos que estão prontos para serem executados. Os administradores de sistema não gostam de garfo-bomba e podem estabelecer limites máximos no número de processos que cada usuário pode ter ou podem revogar os direitos de login, porque isso cria um distúrbio na força dos programas de outros usuários. Você também pode limitar o número de processos filhos criados usando setrlimit() . Os garfos não são necessariamente maliciosos - eles ocorrem ocasionalmente devido a erros de codificação. Abaixo está um exemplo simples que é malicioso.
while (1) fork ();

Pode até haver forkbombs sutis que ocorrem quando você está sendo descuidado durante a codificação. Você pode manchar a bomba do garfo aqui?
#include <unistd.h> #define HELLO_NUMBER 10 int main (){ pid_t children [HELLO_NUMBER]; int i ; for (i = 0 ; i < HELLO_NUMBER ; i ++){ pid_t child = fork (); if (child == - 1){ break ; } if (child == 0){ //I am the child execlp ("ehco" , "echo" , "hello" , NULL); } else { children [i] = child ; } } int j ; for (j = 0 ; j < i ; j ++){ waitpid (children [j], NULL , 0); } return 0 ; }
Nós ehco errado ehco , então não podemos exec lo. O que isto significa? Em vez de criar 10 processos, criamos **1024 processos, bombardeando nossa máquina. Como poderíamos evitar isso? Coloque uma saída logo após o exec, então, caso o exec falhe, não acabaremos com o fork da nossa máquina.** Existem várias outras maneiras. E se removêssemos o binário de echo ? E se o próprio binário criar uma forkbomb?

Sinais

Nós não entraremos completamente em sinais até o final do curso, mas é relevante trazê-lo agora porque várias semânticas com fork e algumas outras chamadas de função detalham o que é um sinal.
Um sinal pode ser considerado como uma interrupção de software. Isso significa que um processo que recebe um sinal interrompe a execução do programa atual e faz com que o programa responda ao sinal.
Existem vários sinais definidos pelo sistema operacional, dois dos quais você já deve saber: SIGSEGV e SIGINT. Um é causado por um acesso à memória ilegal e um é enviado por um usuário que deseja encerrar um programa. Em cada caso, o programa salta do que está fazendo para o manipulador de sinal. Se nenhum manipulador de sinal é fornecido pelo programa, um manipulador padrão é executado - por exemplo, finalizando, ignorando.
Aqui está um exemplo de um simples manipulador de sinal definido pelo usuário.
void handler (int signum) { write (1 , "signaled!" , 9); // we don't need the signum because we are only catching SIGINT // if you want to use the same piece of code for multiple // signals, check the signum } int main () { signal (SIGINT , handler); while (1) ; return 0 ; }
Um sinal pode estar no estado gerado, enviado, pendente e recebido. Estes referem-se a quando um processo gera, o kernel envia, o kernel está prestes a entregar, e quando o kernel entrega um sinal que tudo leva um pouco de tempo para fazer. A terminologia é importante porque fork e exec exigem operações diferentes com base no estado em que o sinal está.
Apenas para notar, geralmente é uma prática de programação ruim usar sinais na lógica do programa. Significado envia um sinal para realizar uma determinada operação. Os sinais não têm prazo de entrega nem garantia de serem entregues. Se você precisar se comunicar entre dois processos, existem outras maneiras de fazer isso.
Se você quiser ler mais, fique à vontade para pular para aquele capítulo em particular e ler sobre ele. Não é muito longo e dá a você o longo e curto sobre como lidar com sinais em processos.

Detalhes de Garfo POSIX

O POSIX determina quais são os padrões do fork (“Fork” # ref-fork_2018). Você pode ler a citação anterior, mas ela pode ser bastante detalhada. Aqui está um resumo do que é relevante.

1. Fork retornará um inteiro não negativo ao sucesso
2. Um filho herdará todos os descritores de arquivos abertos do pai. Isso significa que, se um pai ler um meio caminho do arquivo e dos garfos, o filho será iniciado nesse deslocamento. Quaisquer outras bandeiras também são transportadas.
3. Sinais pendentes não são herdados. Isso significa que se um pai tiver um sinal pendente e criar um filho, o filho não receberá esse sinal, a menos que outro processo sinalize o filho.
4. O processo será criado com um thread (mais sobre isso mais tarde, o consenso geral é não bifurcar e pthread ao mesmo tempo).
5. Como temos cópia na gravação, os endereços de memória somente leitura são compartilhados entre processos
6. Se você configurar determinadas regiões de memória, elas serão compartilhadas entre processos.
7. Os manipuladores de sinal são herdados, mas podem ser alterados.
8. O diretório de trabalho atual é herdado, mas pode ser alterado.
9. As variáveis de ambiente são herdadas, mas podem ser alteradas.

As principais diferenças entre o pai e o filho incluem:

- O id do processo retornado por getpid() . O id do processo pai retornado por getppid() .
- O pai é notificado através de um sinal, SIGCHLD, quando o processo filho termina, mas não vice-versa.
- A criança não herda sinais pendentes ou alarmes do temporizador. Para uma lista completa, veja o http://man7.org/linux/man-pages/man2/fork.2.html
- A criança tem seu próprio conjunto de variáveis de ambiente

Esperando e Executando

Se o processo pai quiser que o filho termine, waitpid (ou wait).
pid_t child_id = fork (); if (child_id == - 1) { perror ("fork"); exit (EXIT_FAILURE);} if (child_id > 0) { // We have a child! Get their exit code int status ; waitpid (child_id , & status , 0); // code not shown to get exit status from child } else { // In child ... // start calculation exit (123); }
wait é uma versão mais simples do waitpid . wait aceita um ponteiro para um inteiro e aguarda em qualquer processo filho. Depois que o primeiro muda, o estado wait retorna.
waitpid é semelhante a wait mas tem algumas diferenças. Primeiro, você *pode* esperar em um processo específico, ou você pode passar valores especiais para o pid fazer coisas diferentes (verifique as man pages). O último parâmetro para waitpid é um parâmetro de opção. As opções estão listadas abaixo
WNOHANG - Retorna se o processo pesquisado é ou não encerrado
WNOWAIT - Aguarde, mas deixe a criança esperada por outra chamada de espera
WEXITED - Espere pelas crianças que saíram

WSTOPPED - Espere por crianças paradas
WCONTINUED - Aguarde crianças continuadas
Os status de saída ou o valor armazenado no ponteiro inteiro para ambas as chamadas acima são explicados abaixo.

Sair dos status

Para encontrar o valor de retorno de main() ou valor incluído em exit()), use as Wait macros - normalmente você usará WIFEXITED e WEXITSTATUS . Veja a página do wait / waitpid man para mais informações.
[language = C] int status ; pid_t child = fork (); if (child == - 1) return 1 ; //Failed if (child > 0) { /* I am the parent - wait for the child to finish */ pid_t pid = waitpid (child , & status , 0); if (pid != - 1 && WIFEXITED (status)) { int low8bits = WEXITSTATUS (status); printf ("Process %d returned %d" , pid , low8bits); } } else { /* I am the child */ // do something interesting execl ("/bin/ls" , "/bin/ls" , "." , (char *) NULL); // "ls ." }
Um processo pode ter apenas 256 valores de retorno, o restante dos bits é informativo e a informação é extraída com deslocamento de bit. Mas, o kernel tem um modo interno de rastrear sinais, saídas ou paradas. Essa API é abstraída para que os desenvolvedores do kernel estejam livres para mudar à vontade. Lembre-se de que essas macros só fazem sentido se a condição prévia for atendida. Isso significa que o status de saída de um processo não será definido se o processo não for sinalizado. As macros não farão a verificação por você, então cabe ao programador certificar-se de que a lógica seja verificada. Como exemplo acima, você deve usar o WIFSTOPPED para verificar se um processo foi parado e, em seguida, o WSTOPSIG para encontrar o sinal que o interrompeu. Como tal, não há necessidade de memorizar o seguinte, isto é apenas uma visão geral de alto nível de como as informações são armazenadas dentro das variáveis de status. De sys/wait.h de um antigo kernel de Berkeley (“Source para Sys / Wait.h,” # ref-sys / wait.h):
[language = C] /* If WIFEXITED(STATUS), the low-order 8 bits of the status. */ #define _WSTATUS(x) (_W_INT(x) & 0177) #define _WSTOPPED 0177 /* _WSTATUS if process is stopped */
#define WIFSTOPPED(x) (_WSTATUS(x) == _WSTOPPED) #define WSTOPSIG(x) (_W_INT(x) >> 8) #define WIFSIGNALED(x) (_WSTATUS(x) != _WSTOPPED && _WSTATUS(x) != 0) #define WTERMSIG(x) (_WSTATUS(x)) #define WIFEXITED(x) (_WSTATUS(x) == 0)
Existe uma convenção sobre códigos de saída. Se o processo saiu normalmente e tudo foi bem sucedido, então um zero deve ser retornado. Além disso, não há muitas convenções, exceto as que você coloca em si mesmo. Se você souber como o programa que você irá interagir vai ser possível, você poderá fazer mais sentido dos 256 códigos de erro. Você poderia de fato escrever seu programa para retornar 1 se o programa fosse para o estágio 1 (como gravar em um arquivo) 2 se fizesse alguma outra coisa etc… Mas nenhum dos programas Unix é projetado para seguir isso por uma questão de simplicidade.

Zumbis e órfãos

É uma boa prática esperar por seus filhos! Se você não esperar por seus filhos, eles se tornarão zumbis. Zumbis ocorrem quando um filho termina e, em seguida, ocupa um lugar na tabela de processo do kernel para o seu processo. A tabela de processos mantém

informações sobre esse processo, como pid, status, como foi morto. A única maneira de se livrar de um zumbi é esperar por seus filhos. Se você nunca esperar por seus filhos e o programa tiver um longo período de execução, você poderá perder a capacidade de bifurcar-se.
Você nem sempre precisa esperar por seus filhos! Seu processo pai pode continuar a executar o código sem ter que esperar pelo processo filho. Se um pai morre sem esperar por seus filhos, um processo pode tornar seus filhos órfãos. Quando um processo pai for concluído, qualquer um de seus filhos será atribuído ao init - o primeiro processo com pid de 1. Assim, esses filhos veriam getppid() retornando um valor de 1. Esses órfãos eventualmente acabarão e por um breve momento se tornarão zumbi. O processo de inicialização automaticamente espera por todos os seus filhos, removendo assim esses zumbis do sistema.

Extra: Como posso esperar de forma assíncrona pelo meu filho usando o SIGCHLD?

Aviso: Esta seção usa sinais que ainda não foram totalmente introduzidos. O pai recebe o sinal SIGCHLD quando um filho é concluído, portanto, o manipulador de sinal pode aguardar o processo. Uma versão ligeiramente simplificada é mostrada abaixo.
pid_t child ; void cleanup (int signal) { int status ; waitpid (child , & status , 0); write (1 , "cleanup! \n " , 9); } int main () { // Register signal handler BEFORE the child can finish signal (SIGCHLD , cleanup); // or better - sigaction child = fork (); if (child == - 1) { exit (EXIT_FAILURE);} if (child == 0) { /* I am the child!*/ // Do background stuff eg call exec } else { /* I'm the parent! */ sleep (4); // so we can see the cleanup puts ("Parent is done"); } return 0 ; }
O exemplo acima, no entanto, erra alguns pontos sutis.

1. Mais de uma criança pode ter terminado, mas o pai receberá apenas um sinal SIGCHLD (os sinais não são enfileirados)
2. Os sinais SIGCHLD podem ser enviados por outros motivos (por exemplo, um processo filho é temporariamente interrompido)
3. Ele usa o código de signal depreciado

Um código mais robusto para colher zumbis é mostrado abaixo.
void cleanup (int signal) { int status ; while (waitpid ((pid_t) (- 1), 0 , WNOHANG) > 0) { } }

exec

Para fazer com que o processo filho execute outro programa, use uma das funções http://man7.org/linux/man-pages/man3/exec.3.html após o bifurcação. O conjunto de funções exec substitui a imagem do processo pela imagem do processo do que está sendo chamado. Isso significa que quaisquer linhas de código após a chamada exec são substituídas. Qualquer outro trabalho que você deseja que o processo filho faça deve ser feito antes da chamada exec . Os esquemas de nomenclatura podem ser encurtados mnemonicamente.

1. e - Uma matriz de ponteiros para variáveis de ambiente é passada explicitamente para a nova imagem do processo.
2. l - Os argumentos da linha de comando são passados individualmente (uma lista) para a função.
3. p - Usa a variável de ambiente PATH para encontrar o arquivo nomeado no argumento do arquivo a ser executado.
4. v - Os argumentos da linha de comandos são transmitidos para a função como uma matriz (vetor) de ponteiros.

Um exemplo desse código está abaixo. Este código executa ls
#include <unistd.h> #include <sys/types.h> #include <sys/wait.h> #include <stdlib.h> #include <stdio.h> int main (int argc , char ** argv) { pid_t child = fork (); if (child == - 1) return EXIT_FAILURE ; if (child) { /* I have a child! */ int status ; waitpid (child , & status , 0); return EXIT_SUCCESS ; } else { /* I am the child */ // Other versions of exec pass in arguments as arrays // Remember first arg is the program name // Last arg must be a char pointer to NULL execl ("/bin/ls" , "/bin/ls" , "-alh" , (char *) NULL); // If we get to this line, something went wrong! perror ("exec failed!"); } }
Tente decodificar o seguinte exemplo
#include <unistd.h> #include <fcntl.h> // O_CREAT, O_APPEND etc. defined here int main () { close (1); // close standard out open ("log.txt" , O_RDWR | O_CREAT | O_APPEND , S_IRUSR | S_IWUSR); puts ("Captain's log"); chdir ("/usr/include"); // execl(executable, arguments for executable including program name and NULL at the end) execl ("/bin/ls" , /* Remaining items sent to ls*/ "/bin/ls" , "." , (char *) NULL); // "ls ." perror ("exec failed"); return 0 ; }
O exemplo escreve “Captain's Log” em um arquivo e depois imprime tudo em / usr / include no mesmo arquivo. Não há nenhuma verificação de erro no código acima (assumimos próximo, aberto, chdir etc funciona como esperado).

1. open - usará o menor descritor de arquivo disponível (ou seja, 1); tão padrão agora vai para o arquivo de log.
2. chdir - Altera o diretório atual para / usr / include
3. execl - Substitua a imagem do programa por / bin / ls e chame seu método main ()
4. perror - Não esperamos chegar até aqui - se o fizermos, o exec falhou.
5. Precisamos do retorno zero porque compiladores reclamam se não o temos.

Detalhes do POSIX Exec

POSIX detalha toda a semântica que o exec precisa cobrir (“Exec” # ref-exec_2018). O que você precisa saber é os seguintes pontos.

1. Os descritores de arquivos são preservados após um exec. Isso significa que se você abrir um arquivo e se esquecer de fechá-lo, ele permanecerá aberto no filho. Isso é um problema porque geralmente a criança não sabe sobre esses descritores de arquivos e eles ocupam um slot na tabela de descritores de arquivos e podem evitar que outros processos acessem o arquivo. A única exceção a isso é se o

descritor de arquivo tiver o sinalizador Fechar Exec definido (mais sobre como defini-lo posteriormente).
2. Várias semânticas de sinal. Os processos executados preservam a máscara de sinal e o conjunto de sinal pendente, mas não preservam os manipuladores de sinal porque é um programa diferente.
3. As variáveis de ambiente são preservadas, a menos que seja usada uma versão do ambiente do exec
4. O sistema operacional pode abrir 0, 1, 2 - stdin, stdout, stderr, se eles forem fechados após exec, na maioria das vezes eles os deixam fechados.
5. O processo exec é executado como o mesmo PID e tem o mesmo pai e grupo de processos que o processo anterior.
6. O processo exec é executado no mesmo usuário e grupo com o mesmo diretório de trabalho

Atalhos

system pré-empacota o código acima (Jones # ref-jones2010wg14 P. 371). Aqui está como usá-lo:
#include <unistd.h> #include <stdlib.h> int main (int argc , char ** argv) { system ("ls"); // execl("/bin/sh", "/bin/sh", "-c", "\\"ls\\"") return 0 ; }
A chamada do system irá bifurcar, executar o comando passado pelo parâmetro e o processo pai original aguardará que isso termine. Isso também significa que o system é uma chamada de bloqueio. O processo pai não pode continuar até que o processo seja iniciado pelas saídas do system . Além disso, o system realmente cria um shell que recebe a string, que é mais sobrecarga do que apenas usando exec diretamente. O shell padrão usará a variável de ambiente PATH para procurar um nome de arquivo que corresponda ao comando. O uso do sistema normalmente será suficiente para muitos problemas simples de executar este comando, mas pode rapidamente se tornar limitante para problemas mais complexos ou sutis, e oculta a mecânica do padrão fork-exec-wait, então encorajamos você a aprender e usar o fork e waitpid vez disso. Não só isso, ele tende a ser um enorme risco de segurança. Ao permitir que alguém acesse uma versão shell do ambiente, você pode alcançar todos os tipos de problemas
int main (int argc , char ** argv) { char * to_exec = asprintf ("ls %s" , argv [1]); system (to_exec); }
Passando algo ao longo das linhas de argv [1] = “; sudo su ”é um enorme risco de segurança.

O garfo-exec-espera Padrão

Um padrão de programação comum é chamar fork seguido por exec e wait . O processo original chama fork, que cria um processo filho. O processo filho, em seguida, usa exec para iniciar a execução de um novo programa. Enquanto isso, o pai usa wait (ou waitpid) para esperar que o processo filho termine.
[language = C] #include <unistd.h> int main () { pid_t pid = fork (); if (pid < 0) { // fork failure exit (1); } else if (pid > 0) { // I am the parent int status ; waitpid (pid , & status , 0); } else { // I am the child execl ("/bin/ls" , "/bin/ls" , NULL); exit (1); // For safety. } }

Você pode perguntar por que nós não apenas executamos ls diretamente. A razão é que agora temos um programa de monitoramento - nosso pai que pode fazer outras coisas. Ele pode prosseguir depois e executar outra função, ou também pode modificar o estado do sistema ou ler a saída da chamada de função.

variáveis ambientais

Variáveis de ambiente são variáveis que o sistema mantém para todos os processos usarem. Seu sistema tem essas configurações agora mesmo! No Bash, você pode verificar alguns desses
$ echo $HOME /home/bhuvy $ echo $PATH /usr/local/sbin:/usr/bin:...
Como você pegaria e os colocaria em C / C ++? Você pode usar a função getenve setenvrespectivamente.
char * home = getenv ("HOME"); // Will return /home/bhuvy setenv ("HOME" , "/home/bhuvan" , 1 /*set overwrite to true*/);
As variáveis de ambiente são importantes porque são herdadas entre processos e podem ser usadas para especificar um conjunto padrão de comportamentos (“Variáveis de ambiente” # ref-env_std_2018), embora você não precise memorizar as opções. Outra preocupação relacionada à segurança é que as variáveis de ambiente não podem ser lidas por um processo externo enquanto a argv pode ser.

Leitura Adicional

Leia as man pages e os grupos POSIX acima!

- http://man7.org/linux/man-pages/man2/fork.2.html
- http://man7.org/linux/man-pages/man3/exec.3.html
- http://man7.org/linux/man-pages/man2/wait.2.html

Tópicos

- Uso correto de fork, exec e waitpid
- Usando exec com um caminho
- Entendendo o que fork e exec e waitpid fazem. Por exemplo, como usar seus valores de retorno.
- SIGKILL vs SIGSTOP vs SIGINT.
- Qual sinal é enviado quando você pressiona CTRL-C
- Usando kill do shell ou a chamada POSIX kill.
- Processar isolamento de memória
- Layout de memória do processo (onde está o heap, pilha etc; endereços de memória inválidos).
- O que é uma bomba garfo, zumbi e órfão? Como criar / removê-los.
- getpid vs getppid
- Como usar as macros de status de saída WAIT WIFEXITED etc.

Perguntas / Exercícios

- Qual a diferença entre executivos com ap e sem ap? O que o sistema operacional
- Como você passa em argumentos de linha de comando para execl*? Que tal execv*? Qual deve ser o primeiro argumento da linha de comando por convenção?
- Como você sabe se execou forknão?
- Qual é o int statusponteiro passado em espera? Quando espera falhar?
- Quais são algumas das diferenças entre SIGKILL, SIGSTOP, SIGCONT, SIGINT? Quais são os comportamentos padrão? Quais você pode configurar um manipulador de sinal?
- Qual sinal é enviado quando você pressiona CTRL-C?
- Meu terminal está ancorado a PID = 1337 e acaba de deixar de responder. Escreva-me o comando do terminal e o código C para enviar SIGQUITpara ele.
- Pode um processo alterar a memória de outro processo através de meios normais? Por quê?
- Onde está o heap, pilha, dados e segmento de texto? Para quais segmentos você pode escrever? Quais são os endereços de memória inválidos?
- Codifique-me uma bomba em C (por favor, não o execute).
- O que é um órfão? Como isso se torna um zumbi? Como eu sou um bom pai?
- Você não odeia quando seus pais lhe dizem que você não pode fazer alguma coisa? Escreva-me um programa que envia SIGSTOPpara seu pai.
- Grave uma função que o fork fork espere em um executável e, usando as macros de espera, informe se o processo foi encerrado normalmente ou se foi sinalizado. Se o processo sair normalmente, imprima isso com o valor de retorno. Caso contrário, imprima o número do sinal que causou o encerramento do processo.

Bovet, Daniel e Marco Cesati. 2005. * Entendendo o Kernel Linux *. Oreilly & Associates Inc.
“Definições.” 2018. * As especificações básicas do grupo aberto Edição 7, 2018 Edition *. O grupo aberto / IEEE. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_210 .
“Environment Variables.” 2018. * Variáveis de ambiente *. O grupo aberto / IEEE. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html .
Evans, Julia. 2018. “Descritores de Arquivo.” * Desenhos de Julia *. Julia Evans. https://drawings.jvns.ca/file-descriptors/ .
“Exec.” 2018. * Exec *. O grupo aberto / IEEE. https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html .
“Garfo.” 2018. * Garfo *. O grupo aberto / IEEE. https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html .
Jones, Larry. 2010. “Rascunho do comitê WG14 N1539 Iso / Iec 9899: 201x.” International Standards Organization.
“Visão geral de Malloc.” 2018. * MallocInternals - Glibc Wiki *. Fundação Software Livre. https://sourceware.org/glibc/wiki/MallocInternals .
“Fonte para Sys / Wait.h.” Nd * Sys / Wait.h Source *. superglobalmegacorp. http://unix.superglobalmegacorp.com/Net2/newsrc/sys/wait.h.html .
Van der Linden, Peter. 1994. * Especialista c Programação: Deep c Secrets *. Prentice Salão Profissional.