CS 241: programação do sistema

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

Tópicos

Se você acha que seus programas não estavam funcionando antes, espere até que eles caiam dez vezes mais rápido - **Bhuvy**

Um encadeamento é curto para 'thread-of-execution'. Representa a sequência de instruções que a CPU tem e irá executar. Para lembrar como retornar de chamadas de função e armazenar os valores de variáveis e parâmetros automáticos, um thread usa uma pilha. Quase que estranhamente, um thread é um processo, o que significa que criar um thread é semelhante ao fork , exceto que não **há cópia que** signifique nenhuma cópia na gravação. O que isso permite é que um processo compartilhe o mesmo espaço de endereço, variáveis, heap, descritores de arquivos e etc. A chamada real do sistema para criar um encadeamento é semelhante ao fork . É clone Não entraremos nos detalhes, mas você pode ler o http://man7.org/linux/man-pages/man2/clone.2.html tendo em mente que está fora do escopo direto deste curso. LWP ou Lightweight Processos ou encadeamentos são preferidos para bifurcar muitos cenários, pois há muito menos sobrecarga criando-os. Mas, em alguns casos, notavelmente o python usa isso, o multiprocessamento é o caminho para tornar seu código mais rápido.

Processos vs threads

Criar processos separados é útil quando

- Quando mais segurança é desejada. Por exemplo, o navegador Chrome usa diferentes processos para diferentes guias.
- Ao executar um programa existente e completo, um novo processo é necessário, por exemplo, iniciando 'gcc'.
- Quando você está executando em primitivos de sincronização e cada processo está operando em algo no sistema.
- Quando você tem muitos threads - o kernel tenta agendar todos os threads próximos uns dos outros, o que poderia causar mais mal do que bem.
- Quando você não quer se preocupar com condições de corrida
- Se um thread bloquear em uma tarefa (digamos, IO), todos os encadeamentos serão bloqueados. Processos não têm essa mesma restrição.
- Quando a quantidade de comunicação é mínima o suficiente, é necessário usar o IPC simples.

Por outro lado, a criação de threads é mais útil quando

- Você quer aproveitar o poder de um sistema multi-core para fazer uma tarefa
- Quando você não consegue lidar com a sobrecarga de processos
- Quando você quer comunicação entre os processos simplificados
- Quando você quer threads para fazer parte do mesmo processo

Internos de Rosca

Sua função principal e outras funções que você pode chamar tem variáveis automáticas. Vamos armazená-los na memória usando uma pilha e acompanhar o tamanho da pilha usando um ponteiro simples (o "ponteiro da pilha"). Se o encadeamento chamar outra função, moveremos o ponteiro da pilha para baixo, para que tenhamos mais espaço para parâmetros e variáveis automáticas. Quando ele retorna de uma função, podemos mover o ponteiro da pilha de volta para o valor anterior. Mantemos uma cópia do antigo valor do ponteiro da pilha - na pilha! É por isso que retornar de uma função é muito rápido - é fácil "liberar" a memória usada por variáveis automáticas - só precisamos mudar o ponteiro da pilha.
Em um programa multiencadeado, existem várias pilhas, mas apenas um espaço de endereço. A biblioteca pthread aloca algum espaço de pilha e usa a chamada de função clone para iniciar o encadeamento nesse endereço de pilha.
+-------+ | | <- Maybe some reserved space +-------+ | | <- The first stack is the main thread stack | Stack | | 1 | | | +-------+ | | <- This is the first additional thread stack | Stack | | 2 | | | +-------+ | ... |
Você pode ter mais de um thread em execução dentro de um processo. Você recebe o primeiro tópico de graça! Ele executa o código que você escreve dentro de 'main'. Se você precisar de mais threads, poderá chamar pthread_create para criar um novo thread usando

a biblioteca pthread. Você precisará passar um ponteiro para uma função para que o thread saiba onde começar.
Todos os threads criados por você residem na mesma memória virtual, porque fazem parte do mesmo processo. Assim, todos eles podem ver o heap, as variáveis globais e o código do programa, etc.
+-------+ | | +-------+ | | # Stack 1 |int *a2|-----------+ | | | | | +-------+ | | | # Stack 2 | |int *a1|--+ | | | | | | | | | +-------+ | | | ... | | | +-------+ | | | | | | | 1 |<-+--------+ | | | | # Heap
Assim, você pode ter duas (ou mais) CPUs trabalhando em seu programa ao mesmo tempo e dentro do mesmo processo. Cabe ao sistema operacional atribuir os encadeamentos às CPUs. Se você tiver mais encadeamentos ativos do que CPUs, o kernel irá atribuir o encadeamento a uma CPU por um curto período de tempo ou até que fique sem coisas para fazer e, em seguida, alternará automaticamente a CPU para trabalhar em outro encadeamento. Por exemplo, uma CPU pode estar processando o jogo AI enquanto outro segmento está computando a saída gráfica.

Uso Simples

Para usar pthreads você precisará incluir pthread.h e compilar e vincular com a opção -pthread ou -lpthread compilador. Esta opção informa ao compilador que seu programa requer suporte de threading. Para criar um segmento, use a função pthread_create . Essa função leva quatro argumentos:
int pthread_create (pthread_t * thread , const pthread_attr_t * attr , void * (* start_routine) (void *), void * arg);

- O primeiro é um ponteiro para uma variável que conterá o id do segmento recém-criado.
- O segundo é um ponteiro para os atributos que podemos usar para ajustar e ajustar alguns dos recursos avançados dos pthreads.
- O terceiro é um ponteiro para uma função que queremos executar
- Quarto é um ponteiro que será dado à nossa função

O argumento void *(*start_routine) (void *) é difícil de ler! Significa um ponteiro que pega um ponteiro void * e retorna um ponteiro void * . Parece uma declaração de função, exceto que o nome da função é empacotado com (* ....)
#include <stdio.h> #include <pthread.h> void * busy (void * ptr) { // ptr will point to "Hi" puts ("Hello World"); return NULL ; } int main () { pthread_t id ; pthread_create (& id , NULL , busy , "Hi"); void * result ; pthread_join (id , & result); }
No exemplo acima, o result será null porque a função ocupada retornou null . Precisamos passar o resultado do endereço porque o pthread_join estará escrevendo no conteúdo do nosso ponteiro.
Nas man pages, ele avisa que os programadores devem usar pthread_t como um tipo opaco e não olhar para os internos. Nós ignoramos isso muitas vezes, no entanto.

Funções Pthread

Aqui estão algumas funções pthread comuns para ajudá-lo a seguir em frente.

- pthread_create . Cria um novo segmento. Cada thread que é criado recebe uma nova pilha. Por exemplo, se você chamar pthread_create duas vezes, seu processo conterá três pilhas - uma para cada thread. O primeiro thread é criado quando o processo é iniciado e você criou mais dois. Na verdade, pode haver mais pilhas do que isso, mas vamos simplificar. A idéia importante é que cada thread requer uma pilha, porque a pilha contém variáveis automáticas e o antigo PC CPU register, de modo que possa voltar a executar a função de chamada após a conclusão da função.
- pthread_cancel interrompe um thread. Observe que o segmento não pode ser interrompido imediatamente. Por exemplo, pode ser terminado quando o encadeamento faz uma chamada ao sistema operacional (por exemplo, write). Na prática, pthread_cancel é raramente usado porque não dá a um thread uma oportunidade de limpar depois de si mesmo (por exemplo, ele pode ter aberto alguns arquivos). Uma implementação alternativa é usar uma variável booleana (int) cujo valor é usado para informar outros segmentos que eles devem concluir e limpar.
- pthread_exit(void *) interrompe o thread de chamada, o que significa que o thread nunca retorna depois de chamar pthread_exit . A biblioteca pthread terminará automaticamente o processo se não houver outros threads em execução. pthread_exit(...) é equivalente a retornar da função do thread; ambos terminam o thread e também definem o valor de retorno (void * pointer) para o thread. Chamar pthread_exit no thread main é uma maneira comum de programas simples garantirem que todos os threads sejam concluídos. Por exemplo, no programa a seguir, os threads do myfunc provavelmente não terão tempo para começar. Por outro lado, exit() sai do processo inteiro e define o valor de saída dos processos. Isso é equivalente a return (); no método principal. Todos os threads dentro do processo são interrompidos. Observe que a versão pthread_exit cria zumbis thread, no entanto, isso não é um processo de longa duração, por isso não nos importamos.
  int main () { pthread_t tid1 , tid2 ; pthread_create (& tid1 , NULL , myfunc , "Jabberwocky"); pthread_create (& tid2 , NULL , myfunc , "Vorpel"); if (keep_threads_going) { pthread_exit (NULL); } else { exit (42); //or return 42; } // No code is run after exit }
- pthread_join() espera que um segmento termine e registre seu valor de retorno. Tópicos concluídos continuarão a consumir recursos. Eventualmente, se threads suficientes forem criados, o pthread_create falhará. Na prática, isso é apenas um problema para processos de execução longa, mas não é um problema para processos simples e de curta duração, pois todos os recursos de encadeamento são liberados automaticamente quando o processo é encerrado. Isso equivale a transformar seus filhos em zumbis, portanto, lembre-se disso para os processos de longa duração. No exemplo de saída, podemos também esperar por todos os encadeamentos.
  // ... void * result ; pthread_join (tid1 , & result); pthread_join (tid2 , & result); return 42 ; // ...

Para ser claro, agora temos toda uma maneira de sair dos tópicos. Aqui está uma lista incompleta.

- Retornando da função de thread
- Chamando pthread_exit
- Cancelando o encadeamento com pthread_cancel
- Terminando o processo através de um sinal.
- chamando exit() ou abort()
- Retornando do main
- exec outro programa
- Desconectando seu computador
- Algum comportamento indefinido pode terminar seus tópicos, é comportamento indefinido

Condições de corrida

Condições de corrida são sempre que o resultado de um programa é determinado por sua seqüência de eventos determinada pelo processador. Isso significa que a execução do código não é determinista. Isso significa que o mesmo programa pode ser executado várias vezes e, dependendo de como o kernel agenda os encadeamentos, pode produzir resultados imprecisos. O seguinte é a condição de corrida canônica.

```
void * thread_main ( void * data ) { int * int_ptr = data ; * int_ptr += * int_ptr ; return NULL ; } int main () { int data = 1 ; pthread_t one , two ; pthread_create ( & one , NULL , thread_main , & data ); pthread_create ( & two , NULL , thread_main , & data ); pthread_join ( one , NULL ); pthread_join ( two , NULL ); printf ( "%d \n " , data ); return 0 ; }
```

Quebrando a montagem há muitos acessos diferentes do código. Assumiremos que os dados são armazenados no registro eax . O código para incrementar é o seguinte sem otimização (suponha que int_ptr contenha eax).

```
mov eax, DWORD PTR [rbp-4] # Loads int_ptr add eax, eax # Does the addition mov DWORD PTR [rbp-4], eax # Stores it back
```

Este é um ótimo padrão de acesso

Tópico 1	**Tópico 2**
mov eax, DWORD PTR [rbp-4]	
adicionar eax, eax	
mov DWORD PTR [rbp-4], eax	
	mov eax, DWORD PTR [rbp-4]
	adicionar eax, eax
	mov DWORD PTR [rbp-4], eax

Este padrão de acesso fará com que os data variáveis sejam 4. O problema é quando as instruções são executadas em paralelo.

Tópico 1	**Tópico 2**
mov eax, DWORD PTR [rbp-4]	mov eax, DWORD PTR [rbp-4]

adicionar eax, eax	adicionar eax, eax
mov DWORD PTR [rbp-4], eax	mov DWORD PTR [rbp-4], eax

Esse padrão de acesso fará com que os data variáveis sejam 2. Esse é um comportamento indefinido e uma condição de corrida. O que queremos é um thread para acessar a parte do código de cada vez.
Você também pode perguntar, bem, quando eu compilar o código com -O2 então eu recebo shl dword ptr [rdi] # Optimized way of doing the add
Isso não deveria resolver isso? É um único instruções de montagem, então não intercalar?
Bem, isso não corrige os problemas que o *próprio hardware* pode experimentar uma condição de corrida, porque nós, como programadores, não informamos ao hardware para verificá-lo. A maneira mais fácil é adicionar o prefixo de *bloqueio* (Guide # ref-guide2011intel , 1120).
Mas não queremos codificar em assembly! Precisamos criar uma solução de software para esse problema.

Um dia nas corridas

Aqui está outra pequena condição de corrida. O código a seguir deve iniciar dez threads com os inteiros de 0 a 9, inclusive. No entanto, quando executar imprime 1 7 8 8 8 8 8 8 8 10 ! Ou muito raramente imprime o que esperamos. Você pode ver por quê?
#include <pthread.h> void * myfunc (void * ptr) { int i = * ((int *) ptr); printf ("%d " , i); return NULL ; } int main () { // Each thread gets a different value of i to process int i ; pthread_t tid ; for (i = 0 ; i < 10 ; i ++) { pthread_create (& tid , NULL , myfunc , & i); // ERROR } pthread_exit (NULL); }
O código acima sofre de uma race condition - o valor de i está mudando. Os novos encadeamentos iniciam mais tarde na saída de exemplo que o último encadeamento inicia após o término do loop. Para superar essa condição de corrida, daremos a cada segmento um ponteiro para sua própria área de dados. Por exemplo, para cada thread, podemos querer armazenar o id, um valor inicial e um valor de saída. Em vez disso, vamos tratar i como um ponteiro e lançá-lo por valor.
void * myfunc (void * ptr) { int data = ((int) ptr); printf ("%d " , data); return NULL ; } int main () { // Each thread gets a different value of i to process int i ; pthread_t tid ; for (i = 0 ; i < 10 ; i ++) { pthread_create (& tid , NULL , myfunc , (void *) i); } pthread_exit (NULL); }
As condições da corrida não estão apenas no nosso código. eles podem estar no código fornecido Algumas funções como asctime , getenv , strtok , strerror não são thread-safe.
Vamos ver uma função simples que também não é 'thread-safe'. O buffer de resultados pode ser armazenado na memória global. Isso é bom em um único programa encadeado. Não queremos retornar um ponteiro para um endereço inválido na pilha, mas há apenas um buffer de resultado na memória inteira. Se dois segmentos fossem usá-lo ao mesmo tempo, um corromperia o outro.
char * to_message (int num) { static char result [256]; if (num < 10) sprintf (result , "%d : blah blah" , num); else strcpy (result , "Unknown"); return result ; }
Existem maneiras de contornar isso, como o uso de bloqueios de sincronização, mas primeiro vamos fazer isso por design. Como você consertaria a função acima? Você pode alterar qualquer um dos parâmetros e qualquer tipo de retorno. Aqui está uma solução válida.

int to_message_r (int num , char * buf , size_t nbytes) { size_t written ; if (num < 10) { written = snprintf (buf , nbtytes , "%d : blah blah" , num); } else { strncpy (buf , "Unknown" , nbytes); buf [nbytes] = '\0' ; written = strlen (buf) + 1 ; } return written <= nbytes ; }
Em vez de tornar a função responsável pela memória, tornamos o responsável pela chamada responsável! Muitos programas, e esperamos que seus programas, tenham uma comunicação mínima necessária. Muitas vezes, uma chamada malloc é muito menos trabalhosa do que bloquear um mutex ou enviar uma mensagem para outro thread.

Não cruze os córregos

No caso de você estar se perguntando, você pode bifurcar dentro de um processo com vários segmentos! No entanto, o processo filho tem apenas um único thread, que é um clone do thread que chamou fork . Podemos ver isso como um exemplo simples, onde os threads de fundo nunca imprimem uma segunda mensagem no processo filho.
#include <pthread.h> #include <stdio.h> #include <unistd.h> static pid_t child = - 2 ; void * sleepnprint (void * arg) { printf ("%d:%s starting up... \n " , getpid (), (char *) arg); while (child == - 2) { sleep (1);} /* Later we will use condition variables */ printf ("%d:%s finishing... \n " , getpid (), (char *) arg); return NULL ; } int main () { pthread_t tid1 , tid2 ; pthread_create (& tid1 , NULL , sleepnprint , "New Thread One"); pthread_create (& tid2 , NULL , sleepnprint , "New Thread Two"); child = fork (); printf ("%d:%s \n " , getpid (), "fork()ing complete"); sleep (3); printf ("%d:%s \n " , getpid (), "Main thread finished"); pthread_exit (NULL); return 0 ; /* Never executes */ }
8970:New Thread One starting up... 8970:fork()ing complete 8973:fork()ing complete 8970:New Thread Two starting up... 8970:New Thread Two finishing... 8970:New Thread One finishing... 8970:Main thread finished 8973:Main thread finished
Na prática, a criação de encadeamentos antes do bifurcação pode levar a erros inesperados porque (como demonstrado acima) os outros encadeamentos são encerrados imediatamente ao bifurcar. Outro segmento pode ter apenas bloquear um mutex chamando malloc e nunca desbloqueá-lo novamente. Usuários avançados podem achar o pthread_atfork útil, mas sugerimos que você geralmente tente evitar a criação de threads antes de bifurcar, a menos que você entenda completamente as limitações e dificuldades dessa abordagem.

Problemas paralelos constrangedores

O estudo de algoritmos paralelos explodiu nos últimos anos. Um problema embaraçosamente paralelo é qualquer problema que precise de pouco esforço para se tornar paralelo. Muitos deles têm alguns conceitos de sincronização com eles, mas nem sempre. Você já conhece um algoritmo parallelizable, Merge Sort!
void merge_sort (int * arr , size_t len){ if (len > 1){ //Mergesort the left half //Mergesort the right half //Merge the two halves }
Com sua nova compreensão de threads, tudo o que você precisa fazer é criar um thread para a metade esquerda e outro para a metade direita. Dado que sua CPU tem múltiplos núcleos reais, você verá um aumento de velocidade de acordo com https://en.wikipedia.org/wiki/Amdahl's_law . A análise da complexidade do tempo também fica interessante aqui. O algoritmo paralelo é executado em
O(

log
3
(*n*))
tempo de execução porque nós a análise assume que temos muitos núcleos.
Na prática, porém, normalmente fazemos duas alterações. Uma vez que a matriz fica pequena o suficiente, nós descartamos o algoritmo paralelo do Mergesort e fazemos um quicksort ou outro algoritmo que trabalha rápido em matrizes pequenas, geralmente regras de coerência de cache neste nível. A outra coisa que sabemos é que as CPUs não possuem núcleos infinitos. Para contornar isso, normalmente mantemos um pool de trabalho. Você não verá a aceleração imediatamente por causa de coisas como coerência de cache e programação de threads extras. No longo prazo, você começará a ver acelerações.
Outro problema embaraçosamente paralelo é o mapa paralelo. Digamos que queremos aplicar uma função a um array inteiro, um elemento por vez.
int * map (int (* func)(int), int * arr , size_t len){ int * ret = malloc (len * sizeof (* arr)); for (size_t i = 0 ; i < len ; ++ i) { ret [i] = func (arr [i]); } return ret ; }
Uma vez que nenhum dos elementos depende de qualquer outro elemento, como você iria paralelizar isso? Qual você acha que seria a melhor maneira de dividir o trabalho entre os tópicos.

Extra: Agendamento

Existem algumas maneiras de dividir o trabalho. Estes são comuns ao framework OpenMP (Silberschatz, Galvin e Gagne # ref-silberschatz2005operating).

- static scheduling os problemas em blocos de tamanho fixo (predeterminados) e faz com que cada thread trabalhe em cada um dos blocos. Isso funciona bem quando cada um dos subproblemas toma aproximadamente o mesmo tempo, porque não há sobrecarga adicional. Tudo o que você precisa fazer é escrever um loop e atribuir a função map a cada sub-array.
- dynamic scheduling como um novo problema torna-se disponível ter um segmento atendê-lo. Isso é útil quando você não sabe quanto tempo o agendamento levará
- guided scheduling Esta é uma mistura dos itens acima com uma combinação dos benefícios e das compensações. Você começa com um agendamento estático e move-se lentamente para a dinâmica, se necessário
- runtime scheduling Você não tem absolutamente nenhuma ideia de quanto tempo os problemas vão levar. Em vez de decidir por si mesmo, deixe o programa decidir o que fazer!

Não há necessidade de memorizar qualquer uma das rotinas de agendamento. O OpenMP é um padrão que é uma alternativa aos pthreads. Por exemplo, aqui está como paralelizar um loop for
#pragma omp parallel for for (int i = 0 ; i < n ; i ++) { // Do stuff } // Specify the scheduling as follows // #pragma omp parallel for scheduling(static)
O escalonamento estático dividirá o problema em partes de tamanho fixo O cronograma dinâmico fornecerá um trabalho quando o ciclo terminar O cronograma guiado é Dinâmico com blocos O tempo de execução é um pacote inteiro de worms.

Outros problemas

De https://en.wikipedia.org/wiki/Embarrassingly_parallel

- Servindo arquivos estáticos em um servidor da web para vários usuários de uma só vez.
- O conjunto Mandelbrot, o ruído Perlin e imagens semelhantes, em que cada ponto é calculado de forma independente.
- Renderização de computação gráfica. Na animação por computador, cada quadro pode ser renderizado independentemente (consulte renderização paralela).
- Pesquisas de força bruta em criptografia.
- Exemplos notáveis do mundo real incluem os sistemas distributed.net e proof-of-work usados em criptomoedas.
- O BLAST pesquisa em bioinformática para várias consultas (mas não para grandes consultas individuais)
- Sistemas de reconhecimento facial em larga escala que comparam milhares de rostos adquiridos arbitrariamente (por exemplo, um vídeo de segurança ou vigilância via circuito fechado de televisão) com um número similarmente grande de rostos previamente armazenados (por exemplo, uma galeria falsa ou uma lista de observação similar).
- Simulações computacionais comparando muitos cenários independentes, como modelos climáticos.
- Meta-heurística de computação evolutiva, como algoritmos genéticos.
- Cálculos conjuntos de previsão numérica do tempo.
- Simulação e reconstrução de eventos em física de partículas.
- O algoritmo dos quadrados de marcha
- Peneiramento da peneira quadrática e da peneira de campo numérico.
- Etapa de crescimento de árvores da técnica de aprendizado de máquina florestal aleatória.
- Transformada Discreta de Fourier onde cada harmônico é calculado independentemente.

Extra: threads.h

Temos muitas bibliotecas de threads discutidas na seção extra. Nós temos os encadeamentos padrão POSIX, encadeamentos OpenMP, também temos uma nova biblioteca de encadeamento C11 que é integrada ao padrão. Esta biblioteca fornece funcionalidade restrita.
Você pode estar se perguntando, por que eu usaria a funcionalidade restrita? A chave está no nome. Como esta é a biblioteca padrão C, ela deve ser implementada em todos os sistemas operacionais compatíveis, que são praticamente todos eles. Isso significa que há portabilidade de primeira classe ao usar encadeamentos.
Nós não vamos falar sobre as funções. A maioria deles está apenas renomeando funções pthread de qualquer maneira. Se você perguntar por que não ensinamos isso, há algumas razões

1. Eles são muito novos. Mesmo que o padrão tenha sido lançado em 2011, os encadeamentos POSIX estiveram por aí para sempre. Muitas de suas peculiaridades foram resolvidas.
2. Você perde expressividade. Esse é um conceito sobre o qual falaremos nos próximos capítulos, mas quando você faz algo portátil, perde alguma expressividade com o hardware do host. Isso significa que a biblioteca threads.h é bem simples. É difícil definir afinidades de CPU. Agendar tópicos juntos. Examine eficientemente os internos por motivos de desempenho.
3. Um monte de código legado já está escrito com threads POSIX em mente. Outras bibliotecas como o OpenMP, CUDA, MPI usarão processos POSIX ou encadeamentos POSIX com uma porta de recusa para o Windows.

Extra: Processos leves?

Anteriormente no início do capítulo, mencionamos que os threads são apenas processos. O que queremos dizer com isso? Você pode criar um thread como um processo. Dê uma olhada no código de exemplo abaixo
// 8 KiB stacks #define STACK_SIZE (8 * 1024 * 1024) int thread_start (void * arg) { // Just like the pthread function puts ("Hello Clone!") // I share the same heap and address space! return 0 ; } int main () { // Allocate stack space for the child char * child_stack = malloc (STACK_SIZE); // Remember stacks work by growing down, so we need // to give the top of the stack char * stack_top = stack + STACK_SIZE ; // clone create thread pid_t pid = clone (thread_start , stack_top , SIGCHLD , NULL); if (pid == - 1) { perror ("clone"); exit (1); } printf ("Child pid %ld \n " , (long) pid); // Wait just like any child if (waitpid (pid , NULL , 0) == - 1) { perror ("waitpid"); exit (1); } return 0 ; }
Parece bem simples né? Bem, por que você não faz isso todas as vezes? Primeiro, há um decente código boilerplate. Você tem que criar a pilha toda vez e passar os flags certos, e esperar como um processo que requer um monte de bandeiras definidas para serem definidas a partir das páginas de manual. Além disso, quase todo mundo usa pthreads por padrão. Pthreads permitem que você defina vários atributos - alguns que lembram a opção em clone - para personalizar seu segmento. Mas como mencionamos anteriormente, a cada segundo de abstração por razões de portabilidade, perdemos alguma funcionalidade. O clone pode fazer algumas coisas interessantes, como manter diferentes partes do heap enquanto cria cópias de outras páginas. Você tem um controle de programação mais simples do Linux, porque é um processo, apenas com os mesmos mapeamentos.
Em nenhum momento neste curso você deve estar usando clone. Mas no futuro, saiba que é uma alternativa perfeitamente viável ao fork. Você só precisa ter cuidado com o que está fazendo.

Leitura Adicional

- http://man7.org/linux/man-pages/man3/pthread_create.3.html
- http://man7.org/linux/man-pages/man7/pthreads.7.html
- http://www.thegeekstuff.com/2012/04/terminate-c-thread/

Tópicos

- ciclo de vida pthread
- Cada segmento tem uma pilha
- Capturando valores de retorno de um encadeamento
- Usando pthread_join
- Usando pthread_create
- Usando pthread_exit
- Em que condições um processo sairá

Perguntas

- O que acontece quando um pthread é criado?
- Onde está o stack de cada thread?
- Como você obtém um valor de retorno dado um pthread_t ? Quais são as maneiras como um thread pode definir esse valor de retorno? O que acontece se você descartar o valor de retorno?
- Por que pthread_join é importante (acha o espaço da pilha, registra, retorna valores)?
- O que o pthread_exit faz em circunstâncias normais (ou seja, você não é o último segmento)? Quais outras funções são chamadas quando você chama pthread_exit?
- Dê-me três condições sob as quais um processo multithread será encerrado. Você pode pensar em mais nada?
- O que é um problema embaraçosamente paralelo?

Guia, parte 2011. “Manual do Desenvolvedor de Software das Arquiteturas Intel 64 e Ia-32.” * Volume 3B: Guia de Programação do Sistema, Parte * 2.
Silberschatz, A., PB Galvin e G. Gagne. 2005. * Operating System Concepts *. Wiley https://books.google.com/books?id=FH8fAQAAIAAJ .