Uma introdução a Pthreads em linguagem C

Guilherme O'Connor de Lungarzo

Junho de 2003

Sumário

Nota

A primeira seção deste documento trata de alocação de memória para programas escritos em C, o leitor que conheça bem este assunto, pode pular diretamente para a segunda seção, sem prejuízos.

Este texto é distribuído junto com o arquivo programas.tgz que contém todos os códigos fonte dos programas exibidos em exemplos neste texto.

1 Alocação de memória

Quando executamos um programa que foi escrito em C, o sistema operacional carrega na memória o código executável do programa em uma área da memória. A esta área damos o nome de *área de texto*. Em seguida é alocada memória para todas as variáveis globais, e então uma área para as variáveis que são declaradas pela função principal do programa, função main. Esta é chamada de *área de dados*.

Depois, inicia-se a execução do programa, isto é feito porque o sistema operacional guarda um ponteiro para a área de texto, de onde retira o código executável e o envia para o processador, que se encarrega da execução.

O programa também possui uma pilha que é chamada de *pilha de execução*, que é onde o sistema operacional guarda as informações a respeito do estado do programa (valores dos registradores, ponteiros para instruções, etc.) quando a função suspende sua execução.

O programa pode suspender sua execução por diversos motivos, o mais óbvio é que em um sistema multitarefa preemptivo o sistema operacional distribui *fatias de tempo* para cada processo, ou seja, cada instância de um programa em execução. Quando o tempo ao qual o processo tinha direito se esgota, o sistema operacional, guarda o seu estado na pilha e o suspende, colocando outro processo em execução. Isto é feito exatamente do modo inverso, os valores na pilha de execução deste processo são restaurados e o programa é posto para rodar.

Outra maneira de uma função ter sua execução suspensa é quando ela faz uma chamada a outra função, o estado dela é guardado na pilha, memória é alocada para a área de dados da função, o sistema operacional guarda um ponteiro para o lugar, na área de texto, onde está o código executável do programa e a execução começa. Analogamente ao que ocorre com o main, que afinal de contas também é uma função.

Sempre que uma função chama outra os dados desta são armazenados na pilha, em espera, e uma nova linha de execução é criada. Observe por exemplo o seguinte programa:

fatorial.c

```
1  # include <stdio.h>
2
3  int fatorial(int a){
4          int fat=1;
5          if(a>1)
6                  fat = a * fatorial(a-1);
```

```
7         else
8                 fat = 1;
9         return fat;
10 }
11
12 int main(){
13         int fat,a;
14         scanf("%d",&a);
15         fat = fatorial(a);
16         printf("%d",fat);
17 }
```

No início da execução o código compilado do programa é colocado na área de texto, depois, a memória correspondente às variáveis fat e a é separada, criando a área de dados da função *main* e o processo começa a ser executado. Ao chegar à atribuição na linha 13 o valor de fatorial(a) deve ser avaliado antes de ser atribuído a fat.

Isto ocorre suspendendo-se a execução da *main*, guardando seu contexto, e repetindo o procedimento para a função fatorial. Cria-se a área de dados para guardar a variável fat, recupera-se o código executável na área de texto e executa-se a função, que, eventualmente, vai ser interrompida por outra instância da função fatorial que após sua execução vai permitir a retirada da função no topo da pilha e assim sucessivamente.

O programa acima, é tipicamente seqüencial, já que cada vez que uma chamada de função é feita o programa *depende* deste valor para continuar sua execução. Mas tome por exemplo o seguinte programa[1]:

hipotenusa.c

```
1  #include<stdio.h>
2  #include<math.h>
3
4  int quadrado(int a){
5          int quad;
6          quad = a*a;
7          return quad;
8  }
9
10 int main(){
11         int hip, hip_quad;
```

[1] O leitor observará que os programas aqui exibidos levam uma série de etapas desnecessárias, todo o cálculo da hipotenusa poderia ter sido feito em uma linha, porém, motivos didáticos justificam esta construção.

```
12          int cateto1,cateto1_quad;
13          int cateto2, cateto2_quad;
14          scanf("%d %d",&cateto1,&cateto2);
15          cateto1_quad=quadrado(cateto1);
16          cateto2_quad=quadrado(cateto2);
17          hip_quad = cateto1_quad + cateto2_quad;
18          hip = sqrt(hip_quad);
19          printf("%d\n",hip);
20  }
```

Quando a primeira chamada à função quadrado[2] é executada na linha 15, o seu resultado só será necessário na linha 17. Isto quer dizer que não haveria necessidade de interromper a execução da linha 16, enquanto a função executa embora seja isto que acontece.

2 O que é uma thread?

Em um programa em C, poderíamos executar as chamadas a funções concorrentemente, se pudéssemos conseguir que o sistema operacional criasse uma nova linha de execução, de certa maneira um novo processo. Isto é possível nos sistemas operacionais modernos através do suporte a threads, uma thread é um processo que contém:

- *PID* (Process ID);
- pilha de execução;
- registradores;
- propriedades de escalonamento

Como a thread é a execução de uma função, ela precisa ter acesso ao código executável na área de texto, bem como às variáveis globais. Para conseguir isto, o sistema operacional cria a thread como um processo que mantém os mesmos ponteiros para a área de dados globais, e um ponteiro para o lugar, dentro da área de texto, onde está o código da função.

[2]Não se esqueça que para compilar este programa com o gcc é necessário passar o argumento -lm

Além disto, para que a thread possa receber parâmetros ela mantém um único ponteiro para um bloco contíguo de memória que contém uma struct com todos seus argumentos.

Em linguagem C, um ponteiro guarda um endereço de memória e o formato dos dados armazenados em tal ponteiro, por exemplo, quando fazemos,

```
1 int a;
2 int *ponteiro;
3 ponteiro = &a;
```

estamos não só armazenando o lugar onde a variável a está sendo guardada, mas também quantos bytes ocupa, podendo assim usar a construção *ponteiro; de maneira idêntica à variável a.

Existe, no entanto, uma maneira de se ter um ponteiro não formatado, ou seja, uma variável que guarde pura e simplesmente um endereço da memória, sem ter nenhuma informação a respeito do formato dos dados ali guardados.

Um ponteiro deste tipo é declarado como

```
1 void *ponteiro
```

Embora possa parecer estranho a primeira vista, ele é um velho conhecido de qualquer programador C. A função malloc é uma função que retorna meramente um endereço de memória. O início de um bloco contíguo de memória, não formatado, que foi reservado para a função que a chamou.

É só lembrar que a função malloc, recebe como único argumento um número inteiro. O fato de, freqüentemente, usarmos a função sizeof para determinar este número não altera o fato de que a função malloc não recebe nenhuma informação a respeito dos dados que serão armazenados lá.

Outro esclarecimento que se faz necessário é que em C, o nome de uma função representa o endereço de memória, dentro da área de texto, no qual o código executável referente a ela está localizado.

Assim, a thread pode receber o ponteiro para sua área de texto (que é um subconjunto da área de texto do programa) a partir do nome da função.

Em sistemas operacionais modernos, nem todo o código executável é carregado de uma vez na área de texto, no entanto o sistema se encarrega destes detalhes e fornece um endereçamento que é válido para fins de uso do programa.

3 O que são PThreads?

Pthreads é um apelido para *Posix* Threads. *Posix*, *Portable Operating System, Interface for Unix*, é um padrão, definido pela IEEE (Institute of Electrical and Electronics Engineers) e pela ISO (International Organization for Standartization), como o próprio nome diz é um padrão que define como, no mundo do *nix os programas devem ser interfaceados, visando portabilidade.

Ocorre que inicialmente, cada fabricante de hardware tinha seu próprio padrão de threads, o que fazia com que programas multithreaded (ou seja, programas usando threads) fossem muito pouco portáveis de plataforma para plataforma.

4 Criando PThreads

Fazer uma chamada a uma pthread não é muito diferente de fazer uma chamada a uma função. De fato, um programa "Hello World"[3] pode ser feito da seguinte maneira.

hellosequencial.c

```
1  #include <stdio.h>
2
3  void hello(){
4          printf("Hello World\n");
5  }
6
7  int main(){
8          hello();
9  }
```

hellothreaded.c

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *hello(){
5          printf("Hello World\n");
6  }
7
8  int main(){
9          pthread_t id;
```

[3]Como (quase) todo mundo sabe, um "Hello World" é um programa (normalmente o primeiro programa feito dentro de um paradigma), e tudo o que ele faz é dizer "Hello World" com pequenas variações

```
10          pthread_create(&id , NULL , hello , NULL);
11
12  }
```

Para compilar e executar este programa com o gcc, você deve fornecer os comandos

```
gcc hellothread.c -o hellothread -lpthread
./hellothread
```

Os quatro parâmetros para a função pthread_create são:

1. o endereço do identificador da thread;
2. o endereço dos atributos;
3. o endereço do código a executar (passado, como vimos, através do nome da função);
4. o endereço da estrutura de argumentos.

No entanto, apenas o primeiro e terceiro argumentos são obrigatórios. O primeiro, porque a função vai tentar escrever nele, caso NULL seja passado, haverá um acesso inválido de memória. O terceiro, porque ela tentará executar o código no endereço passado. Se o endereço for inválido o mesmo problema ocorrerá.

É importante levar em conta que a função pthread_create retorna um valor inteiro que é o código de erro encontrado na criação da thread, ou zero, caso contrário.

Também é importante levar em conta que as threads só existem enquanto a thread principal existir, a menos que a função pthread_exit seja usada. Esta função leva um argumento que é um ponteiro para o valor de status da thread. Convenciona-se que um processo que finalizou sua execução adequadamente deve retornar 0, enquanto que quando um erro ocorre, um valor diferente de 0 deve ser retornado.

hellothreadedcorreto.c

```
1   #include <stdio.h>
2   #include <pthread.h>
3
4   void *hello(){
5           printf("Hello World\n");
```

```
6         pthread_exit(NULL);
7 }
8
9 int main(){
10        pthread_t id;
11        int status;
12        status = pthread_create(&id , NULL , hello , NULL);
13        if(status!=0)
14                exit(-1);
15        pthread_exit(NULL);
16 }
```

5 Passando Argumentos para PThreads

Podemos fazer uma variação do "Hello World" com um laço que execute um número arbitrário de threads, neste caso seria mais interessante que cada thread imprimisse algo como: "Eu sou a thread i de n"

Note que para saber o valor de n é muito fácil, basta a variável ser global, já o mesmo não pode ser feito com a variável i porque ela está constantemente sendo alterada pela thread executando o main. Assim quando uma determinada thread for criada, digamos com $i = 3$, ela pode não ler imediatamente o valor de i, mas fazê-lo um pouco mais tarde quando o main já a alterou. O i deve ser passado como parâmetro.

Para fazer isto, basta passar para a thread o endereço do local onde o valor de i foi armazenado, para tal, deve existir um local de armazenamento para estes valores onde eles não mudem. Como por exemplo um vetor de argumentos. Onde cada argumento é um struct.

argumentos.c

```
1  #include<stdio.h>
2  #include<pthread.h>
3  #define MAX_THREADS 100
4
5  typedef struct _ARGS{
6          int i;
7  }ARGS;
8
9  int n;
10
11 void *hello(void *p){
12         int i;
13         i = ((ARGS *)p)->i;
```

```
14
15          printf("Eu sou a thread %d de %d\n", i, n);
16          pthread_exit(NULL);
17  }
18
19  int main(){
20          int i;
21          ARGS args[MAX_THREADS];
22          int status;
23          pthread_t id[MAX_THREADS];
24
25          printf("Quantas Threads? ");
26          scanf("%d",&n);
27          for(i=0 ; i<n ; i++){
28                  args[i].i = i;
29                          status = pthread_create(&id[i],NULL,
30                                    hello,(void *)&args[i]);
31                  if(status)
32                          exit(-1);
33          }
34          pthread_exit(NULL);
35  }
```

Note a necessidade de se fazer *typecasting* nas linhas 13 e 30. Na linha 30 a função pthread_create recebe um void* por isso o formato deve ser jogado fora. Na linha 13 o formato da struct ARGS deve ser recuperado para poder acessar a variável i a partir do endereço de memória fornecido.

6 Juntando e controlando PThreads

Nem sempre é interessante que cada thread siga o seu caminho independentemente até o fim. Muitas vezes os resultados obtidos por cada thread precisam ser juntados para fazer processamento seqüencial, para outra rodada de processamento paralelo ou meramente para imprimir os resultados em uma ordem específica. Para isto existe a diretiva pthread_join que se encarrega de esperar o término da execução de uma thread, antes de continuar o processamento.

A função pthread_join recebe dois argumentos, o identificador de uma thread (declarado como pthread_t) e um ponteiro para ponteiro desformatado (void **ret) para o valor de retorno da thread. A função pthread_join, analoga-

mente a pthread_create retorna um valor zero se a thread foi adequadamente juntada[4] e um valor diferente de zero caso tenha ocorrido um erro.

No seguinte exemplo, colocamos em execução N_THREADS threads. Cada uma vai somar N números aleatórios entre 0 e MAX. A thread principal, vai aguardar os resultados parciais, somar entre si e exibir o resultado. Cada thread vai guardar o seu resultado parcial em uma posição do vetor soma_parcial[N_THREADS].

somaaleat.c

```
1  #include <stdio.h>
2  #include <stdlibh>
3  #include <pthread.h>
4  #define N_THREADS 10
5  #define N 100000
6  #define MAX 10
7
8  typedef struct _ARGS{
9          int id;
10 }ARGS;
11
12 int soma_parcial[N_THREADS];
13
14 void *realiza_soma(void *p){
15         int resultado=0, i;
16         int meu_id = ((ARGS *)p)->id;
17
18         /* soma N numeros aleatorios entre 0 e MAX */
19         for(i=0 ; i<N ; i++)
20                 resultado += rand()%MAX;
21
22         /* armazena a soma parcial */
23         soma_parcial[meu_id] = resultado;
24
25         pthread_exit((void *)0);
26 }
27
28 int main(){
29         int i, rc, status, somatotal=0;
30         pthread_t id[N_THREADS];
31         ARGS args[N_THREADS];
32
33         /* cria cada uma das threads */
```

[4]Embora a palavra seja feia ela existe e está correta. Tecnicamente o termo em inglês é *joined*=juntada.

```
34          printf("\nCriando threads:");
35          for(i=0 ; i< N_THREADS ; i++){
36                  args[i].id = i;
37                  rc = pthread_create(&id[i], NULL,
38                                  realiza_soma, &args[i]);
39                  if (rc){
40                          printf("\nErro %d na criação\n",rc);
41                          exit(-1);
42                  }
43                  printf(" [%d]",i);
44          }
45
46          /* junta cada uma das threads */
47          printf("\nJuntando threads:");
48          for(i=0 ; i< N_THREADS ; i++){
49                  rc = pthread_join(id[i],(void **)&status);
50                  if (rc){
51                          printf("\nErro %d na junção\n",rc);
52                          exit(-1);
53                  }
54                  printf(" [%d->%d]",i,status);
55          }
56
57          /* conclui a soma */
58          printf("\nSomando parcelas:");
59          for(i=0 ; i< N_THREADS ; i++){
60                  printf(" [%d]", soma_parcial[i]);
61                  somatotal += soma_parcial[i];
62          }
63
64          /* imprime o resultado */
65          printf("\nSoma total: %d\n",somatotal);
66
67          pthread_exit(NULL);
68  }
```

O objetivo do laço entre as linhas 48 e 55 é esperar a execução de todas as threads antes de fazer a soma final. Isto é desejado, obviamente, para garantir que cada thread tenha colocado sua soma parcial no vetor soma_parcial antes de realizar o procedimento.

A linha 54 imprime uma mensagem indicando que a thread i foi juntada e o seu status. Neste programa, por ser muito simples, o status de todas as threads é zero, por isso a thread principal não faz mais que exibi-lo, uma

thread mais complexa poderia retornar valores diferentes testados pelo main para exibir mensagens de erro ou outras atitudes a critério do programador.

No entanto, nem sempre é possível juntar threads. O segundo argumento passado para pthread_create é um atributo que pode ser joinable ou detached. Para que uma thread possa ser juntada ela tem que ser do tipo joinable, que é o default do Posix. Mas, cuidado, isso não garante que toda implementação siga essa orientação. Asim, é sempre uma boa idéia explicitar que uma thread é joinable, quando isto é desejável. Isto faz o código mais portável e menos propenso a ter problemas.

Para fazer isto é necessário criar uma variável do tipo pthread_attr_t, inicializá-la com a função pthread_attr_init e fazê-la joinable com a função pthread_attr_setdetachstate. Finalmente esta variável de atributo pode ser passada como parâmetro para a função pthread_create. O seguinte código intercalado adequadamente no programa somaaleat.c faria o serviço.

```
1 pthread_attr_t atributo;
2 pthread_attr_init(&atributo);
3 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
4 pthread_create(&id[i], atributo, realiza_soma, &args[i]);
```

Quando uma thread é joinable, isto significa que após sua execução a memória ocupada por ela não é liberada de imediato, isto é feito após o join, quando a thread principal já recuperou o valor de status.

Por isto mesmo, quando não há necessidade de uma thread ser esperada, é mais conveniente que ela seja criada em estado detached, o que pode ser feito passando-se o parâmetro PTHREAD_CREATE_DETACHED para pthread_attr_setdetachstate.

Existe também uma maneira de transformar uma thread *joinable* em uma *detached*. Através de uma chamada à função pthread_detach, que recebe como argumento o *id* da thread e retorna zero caso o desligamento[5] tenha sido bem sucedido.

```
1 status = pthread_detach(id);
```

[5]detaching

7 O uso de *mutex*

No programa somaaleat.c cada thread escrevia em uma posição diferente do vetor soma_parcial, por isso não havia problemas de concorrência, mas imagine o caso onde mais de uma thread precisa acessar a mesma variável.

Ler a mesma variável não tem problema, mas quando desejamos alterá-la, precisamos levar em conta que isto pode requerer (e normalmente requer) mais de uma instrução de máquina. Vamos assumir, por exemplo, que sejam necessários os seguintes passos para alterar uma variável:

1. Ler o valor da variável,
2. Alterar o valor da variável localmente,
3. Recolocar o valor da variável.

Em um sistema preemptivo, como a maioria dos sistemas modernos, um processo pode perder o processador a qualquer momento sem prévio aviso, portanto, se duas threads estão tentando alterar a mesma variável pode ocorrer a interrupção da primeira, gerando inconsistência de dados.

Por exemplo, imagine que uma variável x compartilhada, inicialmente valendo 0 vá ser alterada por duas threads A e B, que guardam localmente o valor de x em variáveis locais a e b, respectivamente. Poderia ocorrer o seguinte.

Thread A	Thread B	a	b	x
Le valor de x		0	-	0
Incrementa de 5		5		0
	Le valor de x	5	0	0
	Incrementa de 10	5	10	0
	Escreve o valor	5	10	10
Escreve o valor		5	10	5

No final do processo, o valor de x é 5, quando deveria ser 10. Isto ocorre porque a alteração da variável não é atômica, ou seja, indivisível.

Para solucionar isto o sistema tem que prover uma maneira de se executar uma série de passos sem que outro processo possa interferir, para isto usamos o *mutex*, do inglês *mutual exclusion*.

Um mutex é uma variável do tipo thread_mutex_t que tem dois estados: travado e destravado[6]. Quando a função pthread_mutex_lock, que recebe uma

[6] *locked* e *unlocked*

variável mutex como parâmetro, é chamada, o valor do mutex é verificado e uma de duas atitudes é tomada:

1. se o mutex está destravado, então trava ele e continua a execução;
2. se o mutex está travado, espera ele mudar de estado, trava e continua e execução.

Assim se mais de um processo deve acessar a mesma variável, podemos proteger a parte do código que não deve ser executada simultaneamente, chamada de *região crítica*, com um mutex. Assim, o primeiro processo a alcançar o mutex trava ele, e os seguintes, ao atingirem a região crítica são mantidos em espera pela função pthread_mutex_lock até que o valor do mutex mude.

Isto só vai ocorrer quando, ao sair da região crítica, a tread que obteve o mutex chamar a função pthread_mutex_unlock, que se encarrega de destravar o mutex. Apenas a thread que possui o lock pode destravá-lo.

Em seguida, o primeiro processo escalonado após o destravamento, trava o mutex e entra na seção crítica, mantendo os outros processos em espera.

Antes de ser usado, o mutex deve ser inicializado, isto pode ser feito estaticamente, durante a declaração da variável, através de um comando como

```
1 meu_mutex = PTHREAD_MUTEX_INITIALIZER;
```

ou dinamicamente através da função pthread_mutex_init, que recebe dois argumentos, o endereço da variável mutex e os atributos. O valor NULL para o segundo atributo significa aceitar os *defaults*, o que é equivalente a usar o modo estático. O modo *default* é inicializar o mutex como destravado.

Através do uso de mutex podemos reescrever o programa somaaleat.c de maneira que não seja necessário que a thread principal execute o processo seqüencial de fazer a somatória das somas parciais. Cada thread pode fazer isto por si só.

Contudo, tome cuidado, embora pareça que a soma está sendo feita paralelamente e, portanto, sendo mais eficiente do que a seqüencial, repare que como o mutex só permite a entrada de um processo por vez à região crítica, na prática a soma é seqüencial, apenas não é ordenada, já que as threads podem entrar na região crítica em uma ordem arbitrária.

Outra coisa que pode, em um primeiro momento, passar pela nossa cabeça, é que as threads podem ser *detached*, já que cada uma vai se encarregar de somar sua parcela ao total. Isto é verdade, mas convém não se esquecer que em geral vamos querer que alguém imprima o resultado.

Claro que podemos criar uma estrutura para que cada thread descubra se é a última e, caso seja, se encarregue de imprimir o resultado, mas tal criação é uma complicação do código e desnecessária dentro do escopo deste texto.

O seguinte programa implementa as modificações discutidas, note que este programa implementa também a declaração explícita das threads como joinable. Fica a cargo do leitor implemetar a versão *detachable*, caso deseje.

somamutex.c

```
1  #define N_THREADS 10
2  #define N 100000
3  #define MAX 10
4
5  typedef struct _ARGS{
6          int id;
7  }ARGS;
8
9  pthread_mutex_t meu_mutex = PTHREAD_MUTEX_INITIALIZER;;
10 int somatotal=0;
11
12 void *realiza_soma(void *p){
13         int resultado=0, i;
14         int meu_id = ((ARGS *)p)->id;
15
16         /* soma N numeros aleatorios entre 0 e MAX */
17         for(i=0 ; i<N ; i++)
18                 resultado += rand()%MAX;
19
20         /* armazena a soma parcial */
21         pthread_mutex_lock(&meu_mutex);
22         somatotal += resultado;
23         pthread_mutex_unlock(&meu_mutex);
24         printf("\nThread %d: parcial %d",meu_id,resultado);
25
26         pthread_exit((void *)0);
27 }
28
29 int main(){
30         int i, rc, status;
31         pthread_t id[N_THREADS];
```

```
32          pthread_attr_t atributo;
33          ARGS args[N_THREADS];
34
35          /* inicializando atributo */
36          pthread_attr_init(&atributo);
37          pthread_attr_setdetachstate(&atributo, PTHREAD_CREATE_JOINABLE);
38
39          /* cria cada uma das threads */
40          printf("\nCriando threads:");
41          for(i=0 ; i< N_THREADS ; i++){
42                  args[i].id = i;
43                  rc = pthread_create(&id[i], &atributo,
44                                                  realiza_soma, &args[i]);
45                  if (rc){
46                          printf("\nErro %d na criação\n",rc);
47                          exit(-1);
48                  }
49                  printf(" [%d]",i);
50          }
51
52          /* junta cada uma das threads */
53          printf("\nJuntando threads...");
54          for(i=0 ; i< N_THREADS ; i++){
55                  rc = pthread_join(id[i],(void **)&status);
56                  if (rc){
57                          printf("\nErro %d na junção\n",rc);
58                          exit(-1);
59                  }
60          }
61
62          /* imprime o resultado */
63          printf("\nSoma total: %d\n",somatotal);
64
65          pthread_exit(NULL);
66  }
```

8 O uso de variáveis de condição

Existe um caso em particular em que desejamos ter exclusão mútua a uma região crítica, mas também desejamos que uma determinada condição seja satisfeita, como conseqüência do processamento de outra thread. Neste caso, código adicional seria necessário para fazer a verificação da condição antes

de fazer a chamada a pthread_mutex_lock. Mais que isso, uma simples verificação não é suficiente, mas uma verificação periódica até que a condição seja satisfeita.

Chamamos este tipo de construção de *Busy Waiting* e, como o próprio nome diz, mantém o processo ocupado fazendo nada. Isto é ruim pois consome recursos computacionais sem trazer nenhum benefício.

É para evitar este desperdício que são definidas as variáveis de condição. Uma variável de condição é equivalente a um recurso pelo qual o processo espera e entra em uma fila, como uma fila do spooler de impressão ou similar. Para entrar nesta fila, o processo faz uma chamada wait sobre a variável e fica esperando sua vez.

Em contrapartida, uma outra thread sabe que a primeira thread está, ou pode estar esperando por este recurso, então, quando considera a condição satisfeita faz uma chamada signal sobre a variável, que sinaliza um processo esperando por ela. Então este processo volta à ativa.

A API de pthreads implementa as funções pthread_cond_wait e pthread_cond_signal para cumprirem estes papéis sobre uma variável declarada como pthread_cond_t. Esta variável tem, necessariamente, que trabalhar associada a um mutex. O procedimento básico para implementar variáveis de condição é:

Thread Principal	Thread A	Thread B
Declara mutex		
Declara cond		
Inicializa mutex		
Inicializa cond		
Cria A e B		
	Realiza Trabalho	Realiza Trabalho
	Trava o mutex	
	Verifica condição	
	Chama wait destravando mutex	
		Satisfaz a condição
		Sinaliza
		Destrava Mutex
	Acorda travando mutex	Continua
	Faz trabalho na S.Crítica	
	Destrava mutex	
	Continua	

Veja na tabela como o procedimento ocorre. A thread A entra na seção

crítica para verificar sua condição, o que é feito chamando a função pthread_cond_wait. Esta função realiza três operações atomicamente:

1. destrava o mutex
2. espera, propriamente, ser sinalizado
3. trava o mutex

Por isto é necessário que ela receba, no seu segundo parâmetro, o endereço do mutex que ela deve alterar.

Deste modo, olhando a certa distância, tudo se passa como um mutex normal. No entanto olhado mais de perto vemos que é necessário que o mutex seja destravado e travado dentro da espera, pois isso é o que permite que a outra thread altere a condição pela qual a primeira thread está esperando.

No programa seguinte, prodcons.c, temos um caso trivial de produtor/consumidor. Isto é, temos uma thread que espera por um valor para ser exibido na tela (consumidor). Enquanto isso, uma outra thread é encarregada de colocar este valor na variável.

prodcons.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <unistd.h>
5
6  int recurso=42;
7  pthread_mutex_t meu_mutex;
8  pthread_cond_t minha_cond;
9
10 void *produtor(){
11         /* espera um pouco para permitir
12          * o consumidor iniciar primeiro */
13         sleep(3);
14
15         /* executa seção crítica */
16         pthread_mutex_lock(&meu_mutex);
17         recurso = rand();
18         pthread_cond_signal(&minha_cond);
19         pthread_mutex_unlock(&meu_mutex);
20
21         pthread_exit(NULL);
22 }
23
```

```
24 void *consumidor(){
25         pthread_mutex_lock(&meu_mutex);
26         pthread_cond_wait(&minha_cond, &meu_mutex);
27         printf("Valor do recurso: %d\n",recurso);
28         pthread_mutex_unlock(&meu_mutex);
29 }
30
31 int main(){
32         pthread_t prod_id, cons_id;
33
34         /* inicializa o gerador de numeros aleatorios */
35         srand(time(NULL));
36
37         /* inicializa mutex e cond */
38         pthread_mutex_init(&meu_mutex, NULL);
39         pthread_cond_init(&minha_cond, NULL);
40
41         /* cria threads */
42         pthread_create(&cons_id, NULL, consumidor, NULL);
43         pthread_create(&prod_id, NULL, produtor, NULL);
44
45         pthread_exit(NULL);
46 }
```

Note que o valor inicial do recurso é 42 e note também que na linha 13, o produtor dorme por 3 segundos antes de fazer qualquer coisa. Caso este fosse um mutex normal, o consumidor, em 3 segundos teria tempo de sobra para travar o mutex, ler o valor de recurso, imprimi-lo na tela, destravar o mutex e sair, antes mesmo do produtor acordar, o valor impresso seria 42.

No entanto, na linha 26, o consumidor espera um sinal do produtor. Como o produtor só enviará este sinal depois de ter atualizado o recurso, garantimos que a linha 27 só vai ser executada depois que a linha 17 o for e o valor impresso sempre será 14.

A execução deste programa pode ser resumida assim:

Produtor	Consumidor
Dorme 3s	Entra na RC
	Espera sinal, destravando RC
Acorda	
Atualiza recurso	
Sinaliza	
Destrava RC	
Termina	Recebe sinal, travando RC
	Imprime o valor
	Destrava RC
	Termina

Sugestão para o leitor: modifique o programa, comentando as linhas 13, 18 e 26, eliminando o sincronismo e a espera, e execute o programa, o valor fica quase imprevisível, depende de quem travar o mutex primeiro. Digo quase porque na verdade é muito mais provável que o consumidor execute primeiro, simplesmente pela ordem em que as threads são criadas, mas só por este fato.

Agradecimentos

Quero agradecer ao professor Alfredo Goldman Vel Lejbman por me dar a oportunidade de dar a palestra à qual este documento, em princípio, se destina, ao professor Marco Dimas Gubitoso por tão atenciosamente ter respondido às inúmeras perguntas que eu lhe fiz no decorrer da composição deste texto e a Iomar Zaia por me ajudar com a Revisão.