Estudo das implementações de barreiras para *multithread*

Fabio Andrijauskas
André Oliveira

20 de Outubro de 2009

1 Barreiras e *multithread*

Em aplicações *multithread* pode-se ter a necessidade de ter um ponto de sincronização, um ponto onde todas as *threads* se juntam e esperam até que cada *thread* chegue ao mesmo ponto, para então, prosseguirem. Para isso existem as barreiras, elas vão definir um ponto no código no qual todas as *threads* ficarão esperando até que as demais cheguem ao mesmo ponto.

2 Objetivo

O objetivo desse trabalho é comparar a implementação de barreiras implementada da **glibc**[1], as vistas em aula e as de [2].

3 Código da glibc

A barreira da **glibc** foi feita por Martin Schwidefsky, desenvolvedor da IBM desde 1996. A explicação do código está nos comentários.

```
1  /* Desenvovlido por Martin Schwidefsky <schwidefsky@de.ibm.com>, 2003. */
2
3  #include <errno.h>
4  #include <sysdep.h>
5  #include <lowlevellock.h>
6  #include <pthreadP.h>
7
8  /* Trecho retirado do código de pthread_barrier_init.c: incialização da barreria do init. */
9  ibarrier->lock = LLL_LOCK_INITIALIZER;
10 ibarrier->left = count;
11 ibarrier->init_count = count;
12 ibarrier->curr_event = 0;
13
14 /* Espera na barreira. */
15 int
16 pthread_barrier_wait (barrier)
17      pthread_barrier_t *barrier;
```

```
18 {
19   /* Pega a estrutura de barreira (inicializada por uma chamada a pthread_barrier_init(). */
20   struct pthread_barrier *ibarrier = (struct pthread_barrier *) barrier;
21
22   /* A variável "result" armazena qual thread chegou por último
23       na barreira e liberou a passagem.
24      É o valor retornado por pthread_barrier_wait
25   */
26   int result = 0;
27
28
29   /* Quando uma thread chama barrier_wait, ela deve adquirir o
30       lock da barreira para incrementar contadores, verificar campos, etc.
31   */
32   lll_lock (ibarrier->lock, ibarrier->private ^ FUTEX_PRIVATE_FLAG);
33
34   /* ibarrier->left armazena quantas threads ainda faltam chegar à barreira.
35      Esse número foi inicializado previamente por uma chamada de pthread_barrier_init(),
36       na qual foi especificado o número de threads que usará a barreira.
37      Como mais uma thread está chegando, este contador deve ser decrementado.
38   */
39   --ibarrier->left;
40
41   /* Verifica se é a última thread a chegar na barreira. */
42   if (ibarrier->left == 0)
43     {
44       /* Agora as threads que estão dormindo na barreira
45          podem acordar e sair do loop em ibarrier->curr_event,
46          pois este valor que controla o loop está sendo modificado pela thread que
47          libera a barreira (a última a chegar).
48       */
49       ++ibarrier->curr_event;
50
51       /* Acorda todas as threads. */
52       lll_futex_wake (&ibarrier->curr_event, INT_MAX,
53                       ibarrier->private ^ FUTEX_PRIVATE_FLAG);
54
55       /* Armazena qual thread que liberou a barreira. */
56       result = PTHREAD_BARRIER_SERIAL_THREAD;
57     }
58   else /* Ainda não chegaram todas as threads. */
59     {
60       /* Armazena o valor de ibarrier->curr_event antes de liberar o lock.
61          Assim, a última thread não pode alterá-lo antes que todas armazenem o valor.
62          Isso é importante porque se as threads esperando
63           nessa barreira acordarem por algum outro motivo, como receber algum sinal,
64           elas só passarão da barreira quando a última thread alterar o valor de
65           ibarrier->curr_event, como pode ser visto no loop abaixo.
66       */
```

```
67        unsigned int event = ibarrier->curr_event;
68
69        /* Libera o lock da barreira e vai dormir. */
70        lll_unlock (ibarrier->lock, ibarrier->private ^ FUTEX_PRIVATE_FLAG);
71
72        /* Espera acordar E o contador de evento mudar (pela thread que libera a barreira).
73
74           From PTHREAD_BARRIER_WAIT Man Page: If  a  signal  is delivered
75            to a thread blocked on a barrier, upon return from
76            the signal handler the thread shall resume waiting
77            at the barrier if the barrier wait has not completed.
78            Until the thread in the signal handler returns from it,
79            it is  unspecified whether other threads may proceed past
80            the barrier once they have all reached it.
81        */
82        do
83          lll_futex_wait (&ibarrier->curr_event, event,
84                          ibarrier->private ^ FUTEX_PRIVATE_FLAG);
85        while (event == ibarrier->curr_event);
86      }
87
88    /* Agora todas as threads passaram da barreira!! */
89    unsigned int init_count = ibarrier->init_count;
90
91    /* O ibarrier->left que estava em zero (este contador guardava o número de
92        threads que faltava chegar na barreira) agora deve ser incrementado
93        até chegar no número de threads que usam a barreira.
94       A thread que liberou a barreira pegou o lock mas ainda não soltou.
95       Então a última a passar por aqui dá um unlock.
96       Com isso também, se alguma thread que sair dessa barreira for tentar
97        utilizar a barreira de novo, não conseguirá pegar o lock, fazendo com que
98        ela espere até a barreira estar pronta de novo.
99       Isso faz com que a barreira seja reutilizável sem lançar mão de algo como
100       a roleta dupla, vista em aula.
101   */
102   if (atomic_increment_val (&ibarrier->left) == init_count)
103
104     /* A última thread que passar pela barreira libera o
105         lock da barreira.
106     */
107     lll_unlock (ibarrier->lock, ibarrier->private ^ FUTEX_PRIVATE_FLAG);
108
109   /* Retorna a thread que liberou a barreira. */
110   return result;
111 }
```

4 Soluções Propostas

As soluções propostas abaixo são do livro [2].

Esta primeira solução mostrada abaixo utiliza um estilo *turnstile*, que pode ser traduzido como *estilo roleta*. Ela utiliza duas sincronizações para manter a barreira de forma estável e permitir que ele seja reutilizável. Neste caso, quando a *thread* alcança a barreira, ela ganha exclusividade para incrementar o contador de *threads* que já chegaram à barreira e verificar se ela é a última a chegar. Se não for, ela simplesmente espera na primeira roleta. Se ela for a última a chegar, ela trava a segunda roleta e libera as *threads* que estão esperando na barreira. Na segunda roleta, as *threads* novamente ganham acesso exclusivo ao contador associado à segunda roleta e vão decrementando-o. Quando chegar na última *thread* a ser liberada, a primeira roleta é travada e a segunda é liberada, impedindo assim que alguma *thread* passe pela primeira roleta antes que ela seja fechada.

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait()       # lock the second
        turnstile.signal()      # unlock the first
mutex.signal()

turnstile.wait()                # first turnstile
turnstile.signal()

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait()         # lock the first
        turnstile2.signal()      # unlock the second
mutex.signal()

turnstile2.wait()                # second turnstile
turnstile2.signal()
```

A solução abaixo difere da anterior no que diz respeito ao número de trocas de contexto. Esse número é diminuído com a solução a seguir visto que, ao contrário da solução anterior, não é mais necessário que cada *thread* execute um *signal* para deixar a roleta. Nesse caso, a última *thread* a chegar na roleta destrava-a para um número exato de *threads* que estão esperando, passando um parâmetro n para destravar a roleta. Esse parâmetro poderia ser implementado com um loop, por exemplo.

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile.signal(n)      # unlock the first
mutex.signal()

turnstile.wait()                 # first turnstile

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile2.signal(n)     # unlock the second
mutex.signal()

turnstile2.wait()                # second turnstile
```

A implementação abaixo faz uso de *futex* e roleta dupla. Agora, as *threads* que não são as últimas a chegarem na barreira encontram a primeira roleta fechada e dormem no *futex*, ao passo que a última, além de travar a próxima roleta e liberar a roleta atual, executa um *futex_wake* para acordar as *threads* que estão dormindo na primeira roleta.

```
1
2  /* Retorna -1 se o futex não bloqueou e
3              0 caso contrário */
4  int futex_wait(volatile void *addr, int val1) {
5    return syscall(SYS_futex, addr, FUTEX_WAIT,
6                   val1, NULL, NULL, 0);
7  }
8
9  /* Retorna o número de threads que foram acordadas */
10 int futex_wake(volatile void *addr, int n) {
11     return syscall(SYS_futex, addr, FUTEX_WAKE,
12                    n, NULL, NULL, 0);
13 }
14
15 volatile int roleta_entrada;
16 volatile int roleta_saida;
17 volatile int c;
18
19 void *f_thread(void* v) {
20   int id = (int) v;
21   int i, local_c;
22
23   for (i = 0; i < 2; i++) {
24     printf("Thread %d atingiu a barreira.\n", id);
25     local_c = __sync_add_and_fetch (&c, 1);
26     if (local_c == N) {
27       roleta_saida = 0;
28       roleta_entrada = 1;
29       futex_wake(&roleta_entrada, N-1);
30     }
31     else
32       futex_wait(&roleta_entrada, 0);
33
34     printf("Thread %d passou pela roleta de entrada.\n", id);
35
36     local_c = __sync_sub_and_fetch (&c, 1);
37     if (local_c == 0) {
38       roleta_entrada = 0;
39       roleta_saida = 0;
40       futex_wake(&roleta_saida, N-1);
41     }
42     else
43       futex_wait(&roleta_saida, 0);
44     printf("Thread %d passou pela roleta de saída.\n", id);
45   }
```

```
46    return NULL;
47  }
```

5 Comparação

As duas soluções do livro funcionam; a segunda implementação faz menos chamadas de sistema, com isso economizando processamento, já que existem menos trocas de contexto. A terceira solução utiliza *futex* mas, assim como as duas primeiras, continua utilizando a roleta dupla para fazer com que a barreira seja reutilizável. A variável na qual as *threads* dormem na terceira solução (*futex*) é justamente a variável que representa a roleta na qual as *threads* estão esperando.

A solução da **glibc**, assim como a terceira solução, utiliza *futex* para fazer as *threads* esperarem na barreira. No entanto, ao contrário de todas as soluções apresentadas no livro, a solução da **glibc** não utiliza o estilo *roleta*. Quando uma *thread* chega na barreira, ela ganha o *lock* e só libera quando for dormir no *futex*. Quando a última *thread* chega (esse controle é feito por um contador do número de *threads* embutido na barreira e inicializado por uma chamada a *pthread_barrier_init()*), esta acorda as demais *threads* que estão esperando no *futex*, mas o *lock* que esta última *thread* adquiriu ao chegar na barreira não é liberado. Agora, ao passar pela barreira, a **última** *thread* libera o *lock* adquirido pela *thread* que acordou as demais. Dessa forma, se alguma outra *thread* chegar à barreira novamente, não conseguirá adquirir o *lock* necessário para utilizar o contador associado à barreira. Assim, a solução da **glibc** provê uma barreira reutilizável, sem utilizar a roleta dupla.

Além disso, a variável utilizada pelo *futex* na **glibc** é um contador de eventos que só é alterado pela última *thread* que chega à barreira (e consequentemente acorda as demais), e as demais *threads* armazenam o valor deste contador antes de liberar o *lock* da barreira e dormirem no *futex*. Assim, a **glibc** trata um problema que as soluções do livro não aborda, que é a possibilidade de uma *thread* acordar devido a um outro evento (como um sinal recebido) que não seja a liberação da barreira. Isso é feito colocando a *thread* para dormir dentro de um loop cuja variável de controle é esse contador de eventos que só é incrementado pela última *thread* que chega à barreira.

O algoritmo a seguir, obtido a partir do estudo da **glibc**, implementa uma barreira reutilizável de uma forma mais adequada que as soluções propostas pelo livro, pois utiliza *futex*, mas não necessita de roleta dupla, além de tratar o problema de uma *thread* acordar devido a algum evento que não seja a liberação da barreira.

```
1
2   volatile int c = num_threads;
3   volatile int current_event = 1;
4   lock_t mutex;
5   lock(mutex);
6   --c;
7   if (c == 0)
8     ++current_event;
9     futex_wake(&current_event, num_threads-1);
10  else
11    int local_event = current_event;
12    unlock(mutex);
```

```
13    do
14      futex_wait(&current_event, local_event);
15    while (local_event == current_event);
16  int local_num_threads = num_threads;
17  if (atomic_inc(c) == local_num_threads)
18    unlock(mutex);
19
```

6 Utilizando uma glibc alternativa

Para que seja possível testar alterações realizadas na **glibc** é necessário compilar e ligar um código qualquer com a **glibc** alterada. Para isso, é preciso fazer com que o compilador e o ligador utilizem a **glibc** alterada ao invés de utilizar a **glibc** do sistema. Assim, mostraremos como atingir esse objetivo nessa seção.

Após serem feitas as alterações desejadas na **glibc**, o primeiro passo é verificar se o .c alterado já não possui um .S específico para sua arquitetura. Arquivos como *pthread_barrier_wait.S* e *pthread_cond_wait.S*, por exemplo, possuem um .S para várias arquiteturas, e podem ser encontrados em ⟨**caminho para glibc**⟩**/nptl/sysdeps/unix/sysv/linux/**⟨**arquitetura**⟩. Isso é feito com o intuito de otimizar tais funções. No entanto, na presença desses arquivos para a arquitetura para a qual a **glibc** está sendo compilada, o .c correspondente nem mesmo será compilado. Portanto, para fazer com que o .c seja utilizado, o .S correspondente deve ser apagado. Assim, o .c será utilizado automaticamente.

Em seguida, devemos executar o *configure* da **glibc**. Aqui vale ressaltar dois pontos importantes. Em primeiro lugar, existem várias opções para o *configure* e algumas delas determinam se partes da **glibc** serão compiladas ou não. Por exemplo, para habilitar o suporte a *threads*, devem ser utilizadas as opções –**with-__thread** –**enable-add-ons** –**enable-shared** –**with-tls**, além da opção –**prefix=**⟨**diretório onde será instalada a glibc**⟩, para que a **glibc** a ser compilada não apague a **glibc** do sistema ao ser instalada. O segundo ponto é que não é recomendado rodar o *configure* de dentro do diretório dos fontes da **glibc**. Portanto, crie um diretório para fazer a *build* e outro para instalar a **glibc**.

Finalmente, basta executar os famosos **make && make install**.

Agora, a **glibc** alterada está instalada em um diretório que chamaremos de **inst-dir**. Contudo, ao compilar um programa qualquer feito para testar suas alterações na **glibc**, o compilador e o ligador utilizarão as bibliotecas do sistema. Para alterar este comportamento, utilizamos a seguinte linha de comando (assumindo o compilador **gcc** em ambiente Linux):

```
gcc -nostdinc -isystem<inst-dir>/include \
    -I<caminho para os cabeçalhos do gcc do sistema> \
    -Wl,--dynamic-linker=<inst-dir>/lib/ld-linux-<arquitetura>.so.2,-R=<inst-dir>/lib \
    -L<inst-dir>/lib <teste.c>
```

A opção **-nostdinc** faz com que o **gcc** não utilize os cabeçalhos padrões do sistema; **-isystem** especifica onde estão os cabeçalhos da **glibc** a serem utilizados. Já o **-I** indica onde estão os cabeçalhos do **gcc**, que podem ser os do próprio sistema. Agora, a opção **-Wl** significa que, as opções a seguir separadas por vírgulas serão passadas para o ligador, e não serão utilizadas pelo compilador em si. Assim, –**dynamic-linker** indica onde está o **ld** recém-compilado pela compilação da **glibc** (esse **ld** será utilizado no lugar do **ld** padrão do sistema),

e **-R** diz para o **ld** utilizar as bibliotecas especificadas por esta opção ao invés de utilizar as bibliotecas do sistema. Finalmente, a opção **-L** faz com que o compilador utilize as bibliotecas especificadas no lugar das bibliotecas padrões do sistema.

Agora, o binário gerado pelo compilador, ao ser executado, utilizará a **glibc** alterada, fazendo com que esta possa ser testada por algum programa desenvolvido para esse fim.

References

[1] Glibc - http://www.gnu.org/software/libc/, October 2009.

[2] Allen B. Downey. *The little book of semaphores.*