Programação de Arquiteturas com Memória Compartilhada Utilizando Pthreads

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de Programação Paralela na UFABC.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em: https://www.cs.usfca.edu/~peter/ipp/
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Algumas figuras foram obtidas em: http://pngimg.com

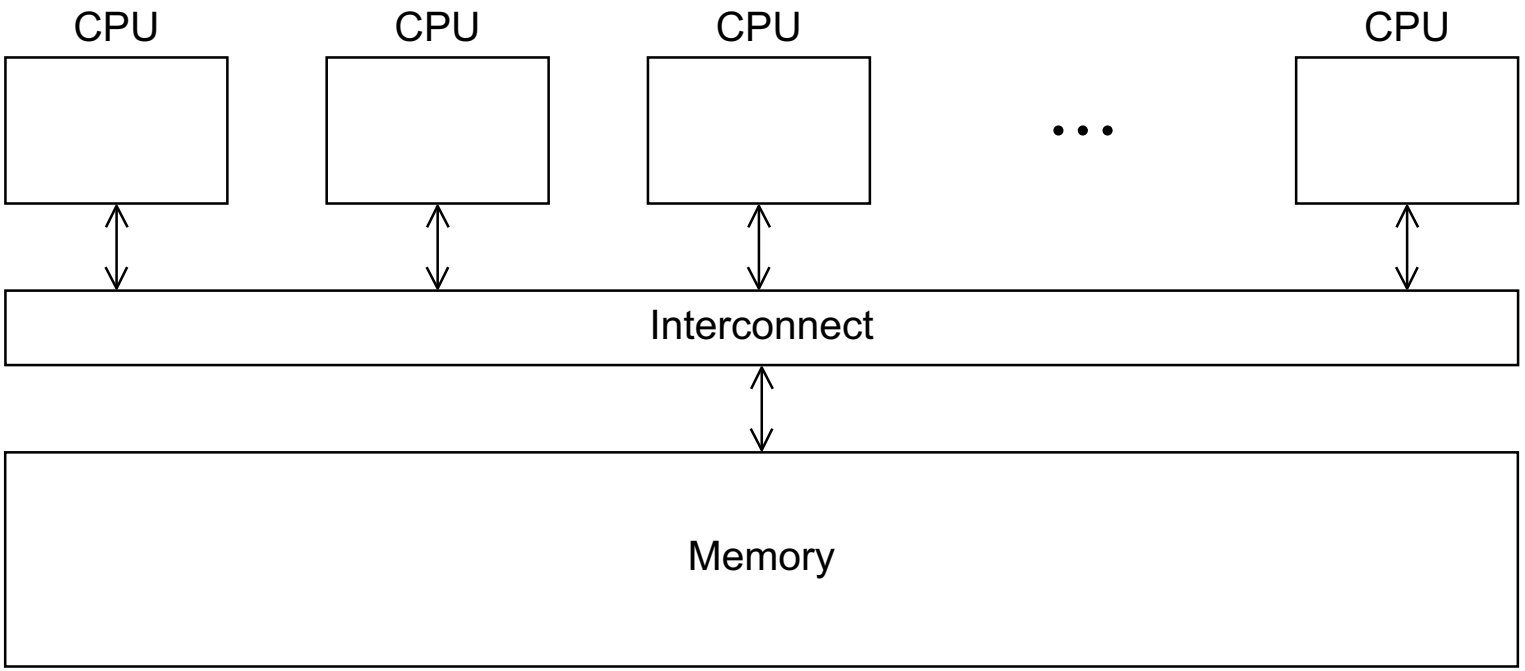
Roteiro

UFABC

- Processos, Threads e Pthreads
- Hello world!
- Multiplicação Vetor/Matriz
- Sessões críticas
- Laços de espera ocupada
- Mutexes
- Produtores-Consumidores
- Barreiras e variáveis de condição
- Travas de leitura/escrita
- Uma nota sobre false sharing e caches
- Thread-Safety

Processos, Threads e Pthreads

Um sistema com memória compartilhada



UFABC

- Um processo é uma instância de um programa em execução
  - Pode estar executando ou suspenso por diversos motivos
- Threads são semelhantes a processos, mas são mais leves
  - São, inclusive, chamadas de processos leves por alguns
- Em um programa que se utiliza de memória compartilhada, um processo pode ter múltiplos threads de controle

UFABC

- POSIX Threads ou Ptreads
  - Padrão para sistemas operacionais estilo Unix
  - Biblioteca que pode ser utilizada por programas em C
  - Especifica uma interface de programação (API) para programação multi-threaded

Atenção!

UFABC

- A API Pthreads só está disponível em sistemas POSIX como
  - Linux
  - BSD
  - MacOS X
  - Solaris
  - HPUX
- No Windows há uma API equivalente que não será coberta neste curso
  - Contudo, a maior parte do que for falado aqui tem o seu equivalente direto

Hello world!

Hello world multi-threaded!

UFABC

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  /* Global variable:  accessible to all threads */
6  int thread_count;
7
8  void *Hello(void* rank) { /* Thread function */
9      /* Use long in case of 64-bit system */
10     long my_rank = (long) rank;
11
12     printf("Hello from thread %ld of %d\n",
13            my_rank, thread_count);
14
15     return NULL;
16 }
17 ...
```

Hello world multi-threaded!

UFABC

```
1  ...
2
3  int main(int argc, char* argv[]) {
4    long thread;  /* Use long if 64-bit system */
5    pthread_t* thread_handles;
6
7    /* Get number of threads from command line */
8    thread_count = strtol(argv[1], NULL, 10);
9
10   thread_handles = malloc(thread_count *
     ↪  sizeof(pthread_t));
11
12   for (thread = 0; thread < thread_count; thread++)
13       pthread_create(&thread_handles[thread], NULL,
14                      Hello, (void*) thread);
15
16   ...
```

Hello world multi-threaded!

UFABC

```
1     ...
2
3     printf("Hello from the main thread\n");
4
5     for (thread = 0; thread < thread_count; thread++)
6         pthread_join(thread_handles[thread], NULL);
7
8     free(thread_handles);
9     return 0;
10 }
```

Compilando um programa com Pthreads

UFABC

- É preciso informar ao GCC (ou ao seu compilador preferido) que utilizaremos a biblioteca pthread
  - `-lpthread`

```
1 gcc -g -Wall -o hello hello.c -lpthread
```

Executando o nosso Hello World!

UFABC

- `./hello <Número de threads>`

```
1  $ ./hello 1
2  Hello from the main thread
3  Hello from thread 0 of 1
4  $ ./hello 4
5  Hello from thread 0 of 4
6  Hello from the main thread
7  Hello from thread 1 of 4
8  Hello from thread 3 of 4
9  Hello from thread 2 of
10 $
```

Variáveis globais

UFABC

- Podem introduzir no seu programa problemas sutis e confusos!
- Restrija o uso de variáveis globais ao máximo[1]
- Em um ambiente multi-threaded, variáveis que são acessíveis por mais de uma thread (globais ou não) são chamadas variáveis compartilhadas (*shared variables*)

[1] A princípio é uma dica válida para quaisquer programas, paralelos ou não!

Iniciando threads

UFABC

- Os processos em MPI são criados pelo ambiente MPI, tipicamente pelo `mpirun` ou `mpiexec`
- Em Phthreads, cada uma das threads são criadas explicitamente pelo próprio programa

Iniciando threads

UFABC

- O cabeçalho `pthread.h` contém as declarações dos protótipos utilizados para programação multi-threaded
- Cada uma das threads é representada por uma struct `pthread_t`
  - Esta estrutura é opaca, ou seja, varia de implementação para implementação (de SO para SO e, em alguns casos, dentro de um mesmo próprio SO)

```
1  int pthread_create (
2      pthread_t*             thread_p   /* out */,
3      const pthread_attr_t* attr_p      /* in */,
4      void* (*start_routine ) ( void ) /* in */,
5      void*                  arg_p      /* in */);
```

`pthread_t`

- Opaco
  - Os dados que ele contém são específicos e variam de sistema para sistema
  - Não são diretamente acessíveis para o código do usuário
  - Contudo, o padrão Phtread especifica que o um `pthread_t` pode ser utilizado para identificar unicamente o thread ao qual está associado

Vamos olhar um pouco mais de perto

UFABC

```
1  int pthread_create (
2      pthread_t*              thread_p   /* out */,
3      const pthread_attr_t* attr_p     /* in */,
4      void* (*start_routine ) ( void ) /* in */,
5      void*                   arg_p      /* in */);
```

- `pthread_t* thread_p /* out */,`
  - Onde serão armazenados os dados relativos ao thread. Aloque antes de passar como parâmetro

Vamos olhar um pouco mais de perto

UFABC

```
1  int pthread_create (
2      pthread_t*              thread_p   /* out */,
3      const pthread_attr_t* attr_p     /* in */,
4      void* (*start_routine ) ( void ) /* in */,
5      void*                   arg_p      /* in */);
```

- `const pthread_attr_t* attr_p /* in */`
  - Não utilizaremos no momento. Mas é onde pode-se estabelecer prioridades, tamanhos de heap, ...
  - As opções disponíveis dependem do SO

Vamos olhar um pouco mais de perto

UFABC

```
1  int pthread_create (
2      pthread_t*              thread_p   /* out */,
3      const pthread_attr_t* attr_p       /* in */,
4      void* (*start_routine ) ( void ) /* in */,
5      void*                   arg_p      /* in */);
```

- `void* (*start_routine ) ( void )`
  - Ponteiro para a função a ser executada pelo thread quando ele foi iniciado

Vamos olhar um pouco mais de perto

UFABC

```
1  int pthread_create (
2      pthread_t*              thread_p   /* out */,
3      const pthread_attr_t* attr_p     /* in */,
4      void* (*start_routine ) ( void ) /* in */,
5      void*                   arg_p      /* in */);
```

- `void* arg_p /* in */`
  - Ponteiro a ser passado como parâmetro para a função `start_routine`

`pthread_create` - Função inicial

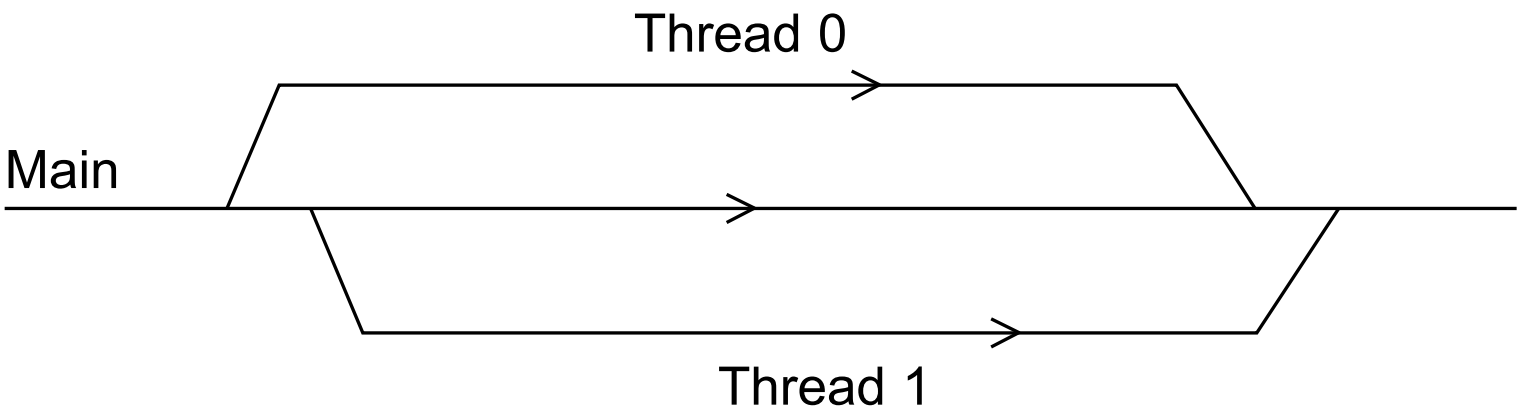
UFABC

- Protótipo:

```
1  void* thread_function (void* args_p);
```

- `void*` pode sofrer um *cast* de e para qualquer outro tipo de ponteiro em C
- Logo, `args_p` pode ser um ponteiro para qualquer estrutura de dados, incluindo vetores
- De modo equivalente, o retorno do tipo `void*` pode ser utilizado da mesma maneira

A thread principal faz um *fork* e um *join* de dois threads.



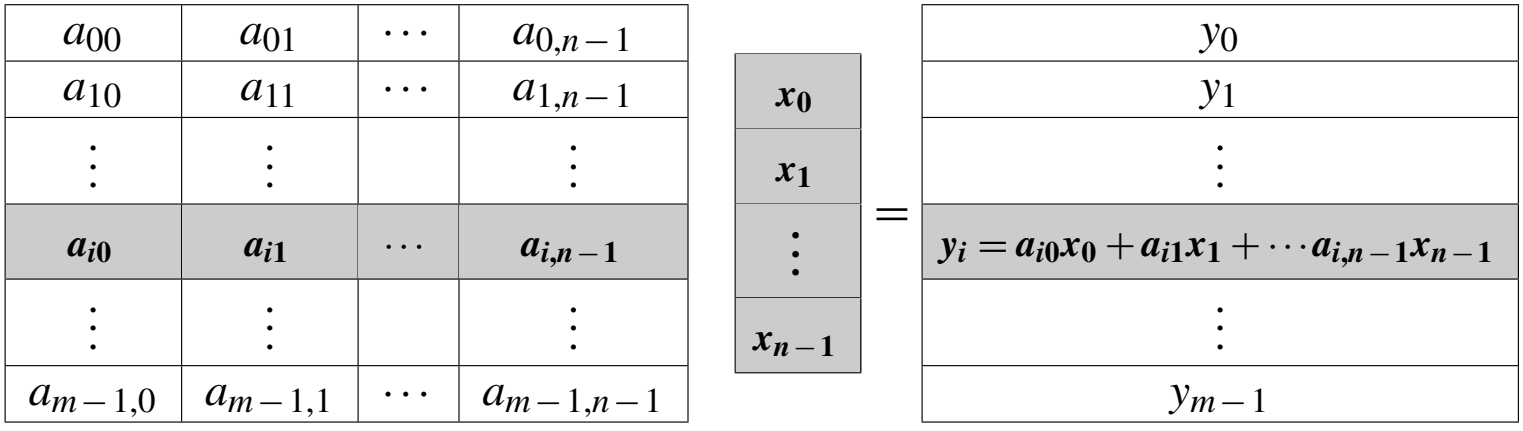
Parando os threads

UFABC

- Podemos chamar a função `pthread_join` uma vez para cada thread
- Uma chamada para a função `pthread_join` fica bloqueada até que o thread, representado por `pthread_t` tenha completado a execução
- **Não há** uma maneira segura de interromper a força uma thread durante a sua execução. Tipcamente o que fazemos é escrever em uma variável compartilhada, que é checada de tempos em tempos pela thread em execução, que indica que ela deve finalizar a sua execução.

Multiplicação Vetor/Matriz

Multiplicação Vetor/Matriz



Pseudocódigo sequencial

UFABC

```
1  /* For each row of A */
2  for (i = 0; i < m; i++) {
3      y[i] = 0.0;
4      /* For each element of the row and each element of x
       ↪  */
5      for (j = 0; j < n; j++)
6          y[i] += A[i][j] * x[j];
7  }
```

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

UFABC

Thread	Componentes de y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

Assim, a thread 0 deverá executar:

```
1  y[0] = 0.0;
2  for (j = 0; j < n; j++)
3    y[0] += A[0][j]* x[j];
```

E de maneira mais geral, a thread que tiver recebido `y[i]` deverá executar:

```
1  y[i] = 0.0;
2  for (j = 0; j < n; j++)
3    y[i] += A[i][j]*x[j];
```

Multiplicação matriz/vetor - Código

UFABC

```
1  void *Pth_mat_vect(void* rank) {
2      long my_rank = (long) rank;
3      int i, j;
4      int local_m = m/thread_count;
5      int my_first_row = my_rank*local_m;
6      int my_last_row = (my_rank+1)*local_m - 1;
7
8      for (i = my_first_row; i <= my_last_row; i++) {
9          y[i] = 0.0;
10         for (j = 0; j < n; j++)
11             y[i] += A[i*n+j]*x[j];
12     }
13
14     return NULL;
15 }
```

Seções críticas

Estimando π

UFABC

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \ldots + (-1)^n \frac{1}{2n+1} + \ldots\right)$$

```
1  double factor = 1.0;
2  double sum = 0.0;
3  for (i = 0; i < n; i ++, factor = -factor) {
4      sum += factor/(2*i +1);
5  }
6  pi = 4.0*sum;
```

- Não é a fórmula que converge mais rápido, ela precisa de muitos termos para chegar a uma boa precisão.

	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

  - Note que conforme **n** cresce, a estimativa com 1 thread melhora

- Contudo, é boa para nossos propósitos! Podemos paralelizar o código anterior da mesma maneira que fizemos para a multiplicação vetor/matriz: particionando os índices

Uma função para π utilizando threads

UFABC

```
1  void* Thread_sum(void* rank) {
2    long my_rank = (long) rank;
3    double factor;
4    long long i;
5    long long my_n = n/thread_count;
6    long long my_first_i = my_n*my_rank;
7    long long my_last_i = my_first_i + my_n;
8
9    if (my_first_i % 2 == 0)
10       factor = 1.0;
11   else
12       factor = -1.0;
13   for (i = my_first_i; i<my_last_i; i++,factor=-factor)
14       sum += factor/(2*i+1);
15
16   return NULL;
17 }
```

Possível condição de corrida

UFABC

Há uma região crítica (ou seção crítica, *critical section*) do código que precisa ser protegida.

Tempo	Thread 0	Thread 1
1	Iniciado pela thread principal	
2	Chama `Compute()`	Iniciado pela thread principal
3	Atribui `y = 1`	Chama `Compute()`
4	Põe `x=0` e `y=1` em registradores	Atribui `y = 2`
5	Soma x + y (0 + 1)	Põe `x=0` e `y=2` em registradores
6	Armazena 1 em `x`	Soma x + y (0 + 2)
7		Armazena 2 em `x`

Laços de espera ocupada

Laços de espera ocupada

UFABC

- Laços de espera ocupada (*busy-waiting*) são uma das soluções mais simples para lidar com condições de corrida
- Um thread testa repetidamente uma condição, e não faz nada até que aquela condição seja verdadeira
  - Efetivamente não faz nada durante a execução do laço a não ser *queimar ciclos*
- Cuidado com compiladores com otimização! Eles podem reordenar o seu código!

```
1  y = Compute(my_rank);
2  while (flag != my_rank);
3  x = x + y;
4  flag++;
```

Versão com espera ocupada

UFABC

```
1  void* Thread_sum(void* rank) {
2    long my_rank = (long) rank;
3    double factor;
4    long long i;
5    long long my_n = n/thread_count;
6    long long my_first_i = my_n*my_rank;
7    long long my_last_i = my_first_i + my_n;
8    if (my_first_i % 2 == 0)
9        factor = 1.0;
10   else
11       factor = -1.0;
12   for (i = my_first_i; i<my_last_i; i++,factor=-factor){
13     while (flag != my_rank);
14     sum += factor/(2*i+1);
15     flag = (flag+1) % thread_count;
16   }
17   return NULL;
18 }
```

Versão com espera ocupada - Busy-wait fora do laço

UFABC

```
1  void* Thread_sum(void* rank) {
2    long my_rank = (long) rank;
3    double factor, my_sum = 0.0;
4    long long i;
5    long long my_n = n/thread_count;
6    long long my_first_i = my_n*my_rank;
7    long long my_last_i = my_first_i + my_n;
8    if (my_first_i % 2 == 0)
9        factor = 1.0;
10   else
11       factor = -1.0;
12   for (i = my_first_i; i<my_last_i; i++, factor=-factor)
13       my_sum += factor/(2*i+1);
14   while (flag != my_rank);
15   sum += my_sum;
16   flag = (flag+1) % thread_count;
17   return NULL;
18 }
```

Mutexes

UFABC

- Apesar de funcionais, os controles de acesso utilizando busy-wait são um desperdício de tempo do processador
- Um core rodando um thread enquanto no busy-wait não faz nenhum trabalho útil
  - Apesar de gastar tempo e energia!
- Mutex (*mutual exclusion*, plural *mutexes*) é um tipo especial de variável que pode ser usada para restringir acesso à uma seção crítica do código a um único thread por vez

UFABC

- Um mutex pode ser usado para garantir que uma thread ”exclua”todas as outras threads daquela seção enquanto a estiver executando
- O padrão Pthreads inclui um tipo especial para mutexes: `pthread_mutex_t`

```
1 int pthread_mutex_init(
2     pthread_mutex_t*           mutex_p /* out */,
3     const pthread_mutexattr_t* attr_p  /* in */);
```

UFABC

- Quando um programa acaba de usar um mutex, ele deve descartá-lo usando:

```
1 int pthread_mutex_destroy(pthread_mutex_t* mutex_p
  ↪  /* in/out */);
```

- Quando um programa deseja **entrar** em uma região crítica ele deve chamar:

```
1 int pthread_mutex_lock(pthread_mutex_t* mutex_p /*
  ↪  in/out */);
```

- Quando um programa deseja **sair** de uma região crítica ele deve

```
1 int pthread_mutex_unlock(pthread_mutex_t* mutex_p /*
  ↪  in/out */);
```

UFABC

- Um mutex só pode[2] ser desbloqueado pela mesma thread que o bloqueou!

[2] Talvez o melhor termo seja "*deveria*"

Versão da soma usando mutexes

UFABC

```
1  void* Thread_sum(void* rank) {
2    long my_rank = (long) rank;
3    double factor; long long i;
4    long long my_n = n/thread_count;
5    long long my_first_i = my_n*my_rank;
6    long long my_last_i = my_first_i + my_n;
7    double my_sum = 0.0;
8    if (my_first_i % 2 == 0)
9        factor = 1.0;
10   else
11       factor = -1.0;
12   for (i = my_first_i; i<my_last_i; i++,factor=-factor)
13       my_sum += factor/(2*i+1);
14   pthread_mutex_lock(&mutex);
15   sum += my_sum;
16   pthread_mutex_unlock(&mutex);
17   return NULL;
18 }
```

UFABC

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

$$\frac{T_{\text{Sequencial}}}{T_{Paralelo}} \approx \text{ThreadCount}$$

Comparação com busy-wait

UFABC

Table 4.2 Possible Sequence of Events with Busy-Waiting and More Threads than Cores

Time	flag	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
0	0	crit sect	busy-wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy-wait	susp
2	2	—	terminate	susp	busy-wait	busy-wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy-wait

Pergunta

O busy-wait só é menos eficiente no caso onde temos mais threads do que cores?

Produtores-Consumidores

Motivação

UFABC

- Laços de espera ocupada forçam uma ordem para que threads acessem a seção crítica
- Quando se usa mutexes não ajuda muito já que a ordem é deixada ao acaso e depende do SO
- Há casos onde é preciso controlar a ordem que threads acessam a seção crítica

Problemas com os mutexes

UFABC

```
1  /* n and product matrix are shared and initialized by
   ↪  the main thread */
2  /* product matrix is initialized to be the identity
   ↪  matrix */
3  void* Thread work(void* rank) {
4      long my_rank = (long) rank;
5      matrix_t my_mat = Allocate_matrix(n);
6      Generate_matrix(my_mat);
7      pthread_mutex_lock(&mutex);
8      Multiply_matrix(product_mat, my_mat);
9      pthread_mutex_unlock(&mutex);
10     Free_matrix(&my_mat);
11     return NULL;
12 }
```

Uma primeira tentativa - Troca de mensagens com Pthreads

UFABC

```
1  /* messages has type char*. It's allocated in main. */
2  /* Each entry is set to NULL in main. */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      long source = (my_rank + thread_count - 1) %
       ↪  thread_count;
7      char* my_msg = malloc(MSG_MAX*sizeof(char));
8      ...
```

```
1       ...
2       sprintf(my_msg, "Hello to %ld from %ld", dest,
        ↪  my_rank);
3       messages[dest] = my_msg;
4       if (messages[my_rank] != NULL)
5           printf("Thread %ld > %s\n", my_rank,
            ↪  messages[my_rank]);
6       else
7           printf("Thread %ld > No message from %ld\n",
            ↪  my_rank, source);
8       return NULL;
9   }
```

Tentativas de conserto

UFABC

- Busy wait

```
1  while (messages[my_rank] == NULL);
2  printf("Thread %ld > %s\n", my_rank,
   ↪  messages[my_rank]);
```

- Mutexes

```
1  ...
2  pthread_mutex_lock(mutex[dest]);
3  ...
4  messages[dest] = my_msg;
5  pthread_mutex_unlock(mutex[dest]);
6  ...
7  pthread_mutex_lock(mutex[my_rank]);
8  printf("Thread %ld > %s\n", my_rank,
   ↪  messages[my_rank]);
9  ...
```

  - Funciona?

Semáforos

UFABC

- Maneira alternativa a mutexes
- Um semáforo pode ser entendido como um tipo especial de contador
- O contador pode assumir números positivos: 0, 1, 2, ...
- Quando o semáforo é usado apenas com valores 0 e 1 dizemos que é um semáforo binário
- Grosseiramente falando, 0 indica travado e qualquer número maior que um destravado
- `sem_wait` decrementa (se possível) o contador (equivalente ao `lock`)
- `sem_post` incrementa o contador (equivalente ao `unlock`)

UFABC

```c
/* Semáforos não são parte de pthreads */
#include <semaphore.h>

int sem_init(
             sem_t* semaphore_p /* out */,
             int shared /* in */,
             unsigned initial val /* in */);

int sem_destroy(sem_t* semaphore_p /* in/out */);

int sem_post(sem_t* semaphore_p /* in/out */);

int sem_wait(sem_t* semaphore_p /* in/out */);
```

Usando semáforos para enviar mensagens

UFABC

```
1  /* messages is allocated and initialized to NULL in main
   ↪  */
2  /* semaphores is allocated and initialized to 0 (locked)
   ↪  in main */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      char* my_msg = malloc(MSG_MAX * sizeof(char));
7      ...
```

Usando semáforos para enviar mensagens

UFABC

```
1       ...
2       sprintf(my_msg, "Hello to %ld from %ld", dest,
        ↪  my_rank);
3       messages[dest] = my_msg;
4       sem_post(&semaphores[dest]);
5       /* ‘‘Unlock’’ the semaphore of dest */
6       /* Wait for our semaphore to be unlocked */
7       sem_wait(&semaphores[my_rank]);
8       printf("Thread %ld > %s\n", my_rank,
        ↪  messages[my_rank]);
9       return NULL;
10  }
```

UFABC

- Note que não há uma região crítica que queremos proteger
- Precisamos apenas que um thread espere pelo outro ou, em outras palavras, que um thread espere que algo seja feito por outro
- Este tipo de sincronização onde um thread é impedido de prosseguir até que outro thread tenha feito alguma ação é tipicamente chamado de sincronização produtor-consumidor

Barreiras e variáveis de condição

Barreiras

UFABC

- Sincronizar os threads para que todos estejam no mesmo ponto da execução é chamado de barreira
- Nenhum thread pode passar pela barreira até que todos os threads a tenham alcançado

Usando barreiras para medir o tempo do thread mais lento

UFABC

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
...
Synchronize threads;
Store current time in my start;

/* Execute timed code */

Store current time in my finish;
my_elapsed = my_finish - my_start;
elapsed = Maximum of my_elapsed values;
```

Usando barreiras para depuração

UFABC

```
point in program we want to reach;
barrier;
if (my rank == 0) {
    printf("All threads reached this point\n");
    fflush(stdout);
}
```

Espera ocupada e mutex

UFABC

- Implementar uma barreira usando espera ocupada e mutexes é trivial
- Utiliza-se um contador compartilhado protegido pelo mutex
- Quando o contador indicar que todos os threads entraram na seção crítica, os threads podem sair da seção crítica

Espera ocupada e mutex

UFABC

```
1  /* Shared and initialized by the main thread */
2  int counter; /* Initialize to 0 */
3  int thread_count;
4  pthread_mutex_t barrier_mutex;
5  ...
6  void* Thread work(...) {
7      ...
8      /* Barrier */
9      pthread_mutex_lock(&barrier_mutex);
10     counter++;
11     pthread_mutex_unlock(&barrier_mutex);
12     while (counter < thread_count);
13     ...
14 }
```

Pergunta

Precisamos de um contador por barreira para evitar problemas. Por quê?

Implementando uma barreira com semáforos

UFABC

```
1  /* Shared variables */
2  int counter; /* Initialize to 0 */
3  sem_t count_sem; /* Initialize to 1 */
4  sem_t barrier_sem; /* Initialize to 0 */
5  ...
6
7  void* Thread work(...) {
8      ...
```

Implementando uma barreira com semáforos

UFABC

```
1       /* Barrier */
2       sem_wait(&count_sem);
3       if (counter == thread_count - 1) {
4           counter = 0;
5           sem_post(&count_sem);
6           for (j = 0; j < thread_count - 1; j++)
7               sem_post(&barrier_sem);
8       } else {
9           counter++;
10          sem_post(&count_sem);
11          sem_wait(&barrier_sem);
12      }
13      ...
14  }
```

Variáveis de condição

UFABC

- Uma variável de condição é um objeto de dados que permite um thread suspender a execução até que uma certa condição ou evento ocorra.
- Quando o evento ou condição ocorrer, outro thread pode ser avisar (*signal*) o thread suspenso para que acorde.
- Uma variável de condição é sempre associada a um mutex.

Variáveis de condição

```
lock mutex;
if condition has occurred
    signal thread(s);
else {
    unlock the mutex and block;
    /* when thread is unblocked,
       mutex is relocked */
}
unlock mutex;
```

Interface de variáveis de condição

UFABC

```
1  int pthread_cond_init(pthread_cond_t* cond_p /* out */,
2                        const pthread_condattr_t*
                         ↪  cond_attr_p /* in */);
3  int pthread_cond_destroy(pthread_cond_t* cond_p /*
   ↪  in/out */);
4
5  int pthread_cond_wait(pthread_cond_t* cond_var_p /*
   ↪  in/out */,
6                        pthread_mutex_t* mutex_p   /*
                         ↪  in/out */);
7
8  int pthread_cond_signal(pthread_cond_t* cond_var_p /*
   ↪  in/out */);
9  int pthread_cond_broadcast(pthread_cond_t* cond_var_p /*
   ↪  in/out */);
```

Implementando uma barreira com variáveis de condição

UFABC

```
1  int counter = 0;
2  pthread_mutex_t mutex;
3  pthread_cond_t cond_var;
4  ...
5  void* Thread work(...) {
6    ...
7    /* Barrier */
8    pthread_mutex_lock(&mutex);
9    counter++;
10   if (counter == thread count) {
11       counter = 0;
12       pthread_cond_broadcast(&cond_var);
13   } else {
14       while (pthread_cond_wait(&cond_var, &mutex) != 0);
15   }
16   pthread_mutex_unlock(&mutex);
17   ...
18 }
```

Travas de leitura/escrita

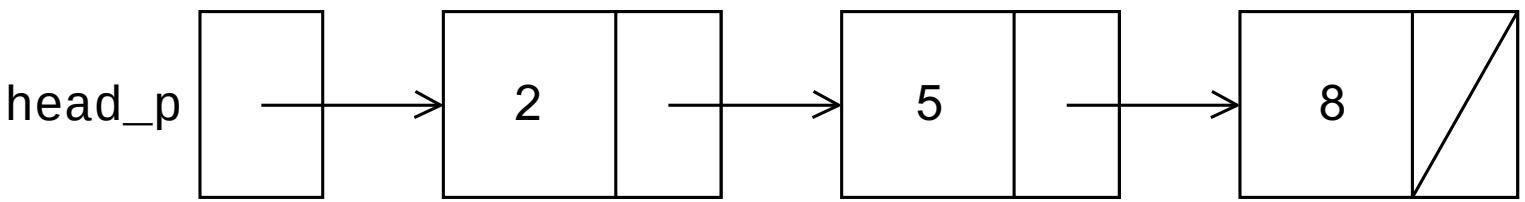
Controlando o acesso a uma grande estrutura de dados

UFABC

- Vamos usar um exemplo
- Suponha que a estrutura em questão é uma lista ligada de inteiros mantida em ordem
- Queremos operações `Member`, `Insert` e `Delete` paralelas

Listas ligadas

```
1  struct list_node_s {
2      int data;
3      struct list_node_s* next;
4  }
```

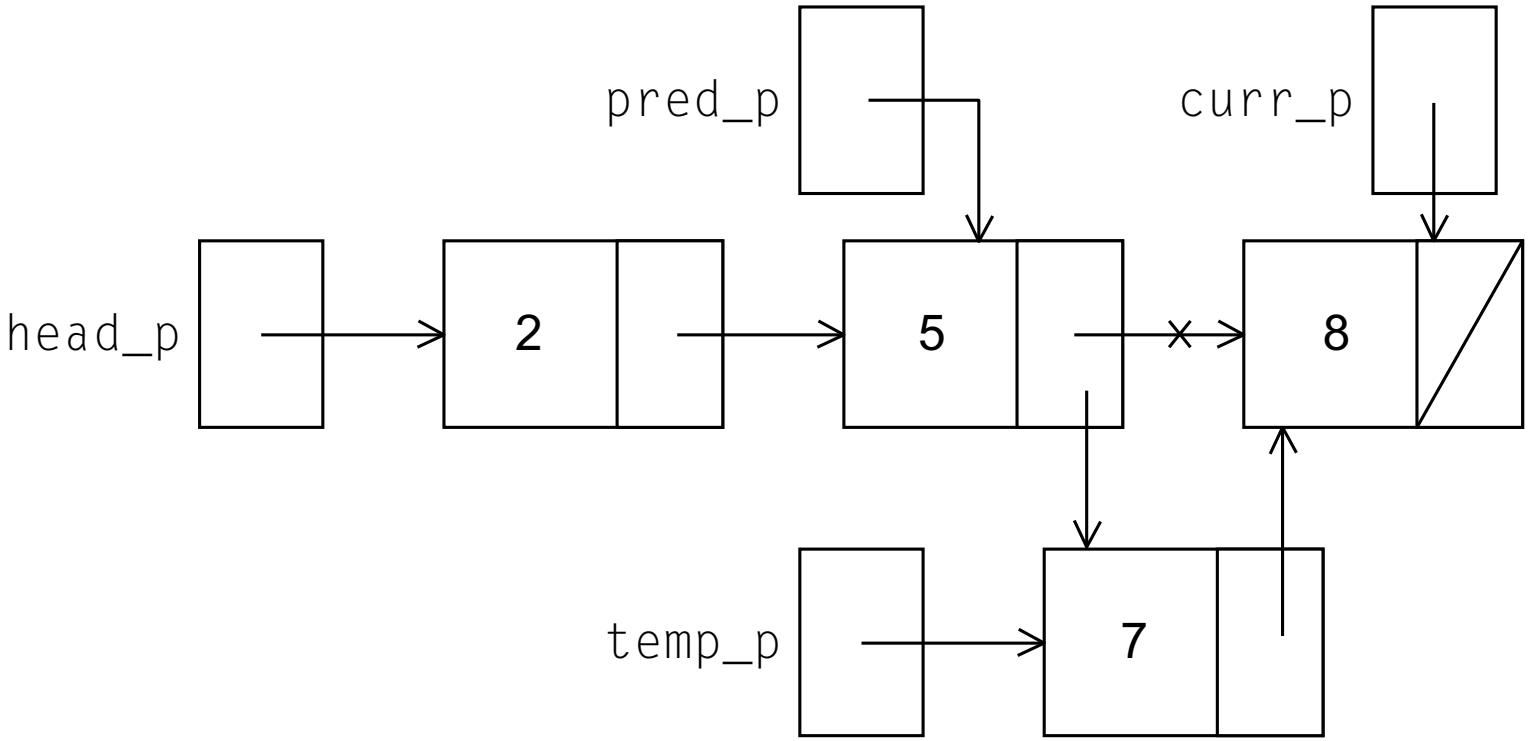


Verificando a pertinência

UFABC

```
1  int  Member(int value, struct list_node_s* head_p) {
2      struct list_node_s* curr_p;
3
4      curr_p = head_p;
5      while (curr_p != NULL && curr_p->data < value)
6          curr_p = curr_p->next;
7
8      if (curr_p == NULL || curr_p->data > value) {
9          printf("%d is not in the list\n", value);
10         return 0;
11     } else {
12         printf("%d is in the list\n", value);
13         return 1;
14     }
15 }
```

Inserção

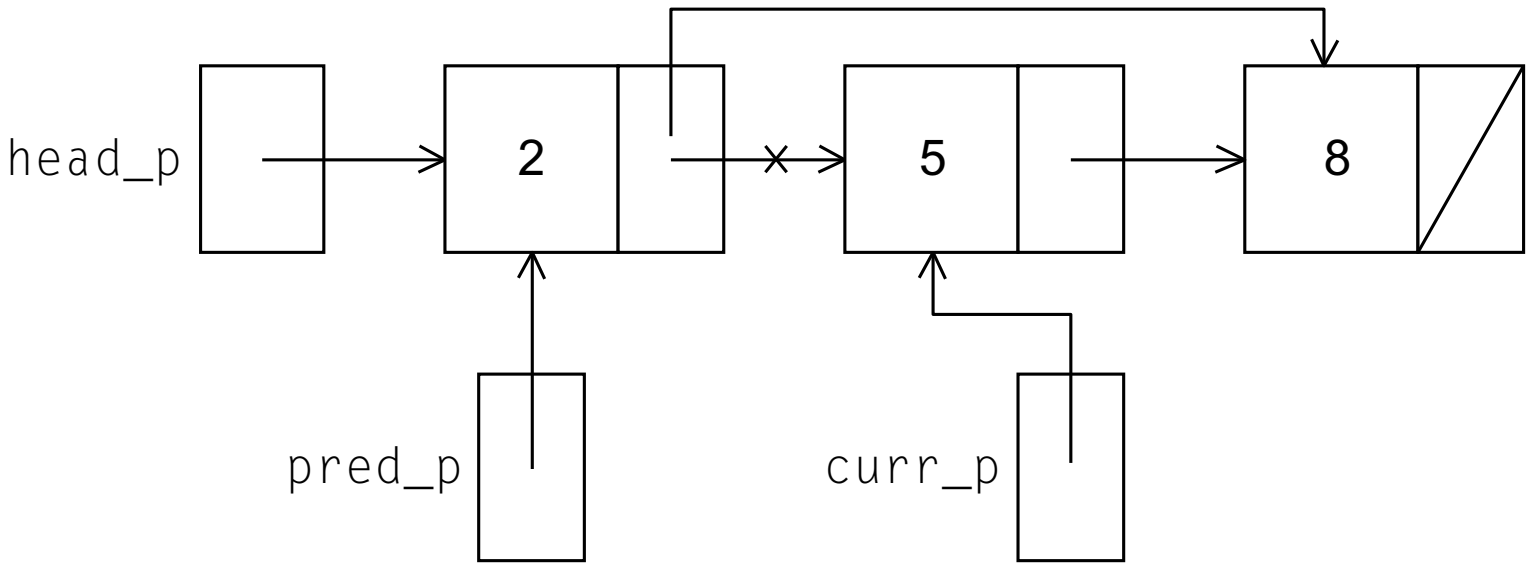


UFABC

```
1  int Insert(int value, struct list_node_s** head_pp) {
2      struct list_node_s* curr_p = *head_pp;
3      struct list_node_s* pred_p = NULL;
4      struct list_node_s* temp_p;
5
6      while (curr_p != NULL && curr_p->data < value) {
7          pred_p = curr_p;
8          curr_p = curr_p->next;
9      }
10     ...
```

UFABC

```
1       ...
2       if (curr_p == NULL || curr_p->data > value) {
3           temp_p = malloc(sizeof(struct list_node_s));
4           temp_p->data = value;
5           temp_p->next = curr_p;
6           if (pred_p == NULL)
7               *head_pp = temp_p;
8           else
9               pred_p->next = temp_p;
10          return 1;
11      } else { /* value in list */
12          printf("%d is already in the list\n", value);
13          return 0;
14      }
15  }
```


head_p
2
5
8
pred_p
curr_p

Remoção

```
1  int Delete(int value, struct list_node_s** head_pp) {
2      struct list_node_s* curr_p = *head_pp;
3      struct list_node_s* pred_p = NULL;
4
5      /* Find value */
6      while (curr_p != NULL && curr_p->data < value) {
7          pred_p = curr_p;
8          curr_p = curr_p->next;
9      }
10
11     ...
```

UFABC

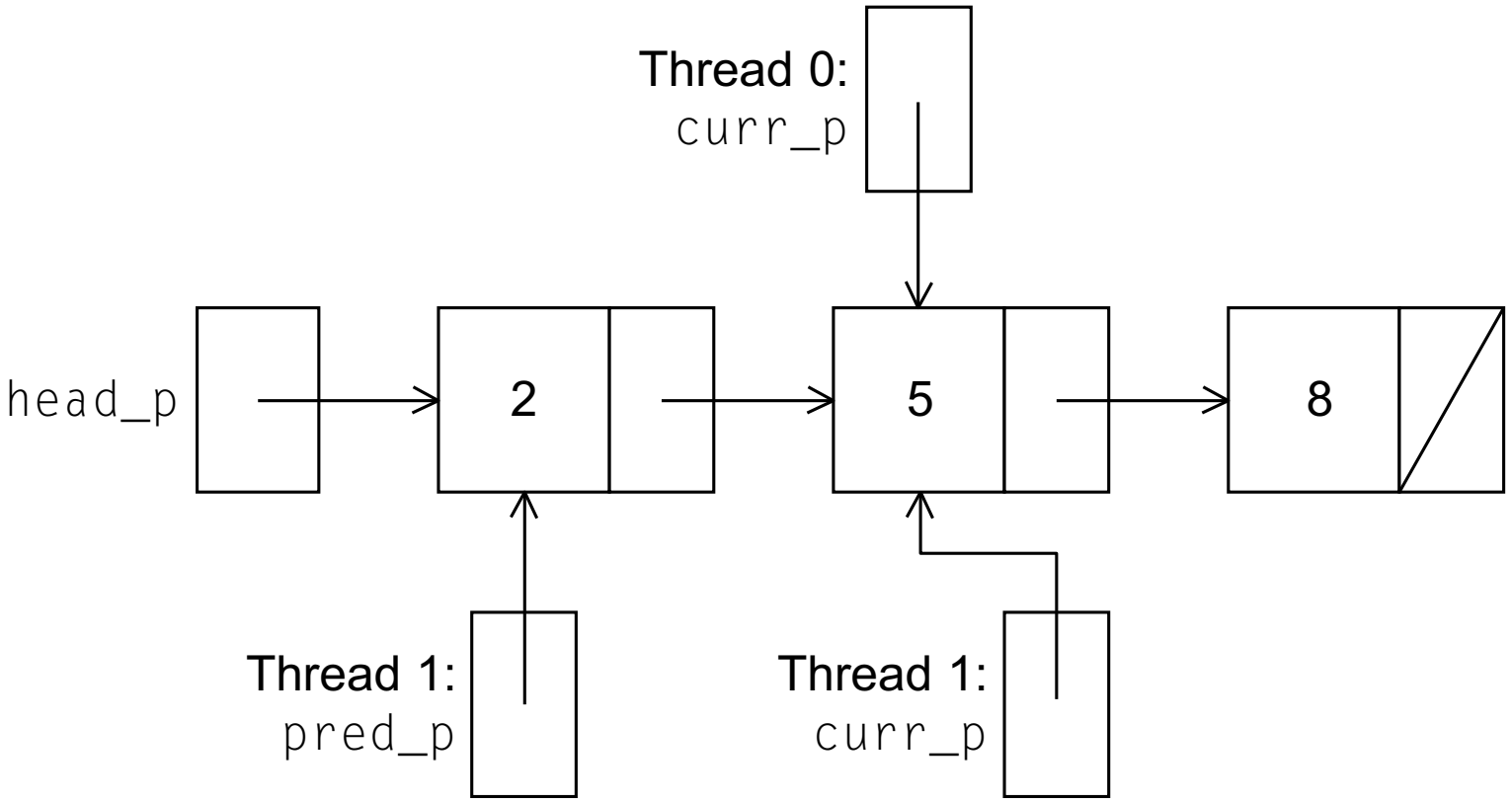
```
1       if (curr_p != NULL && curr_p->data == value) {
2           if (pred_p == NULL) { /* first element in list
            ↪  */
3               *head_pp = curr_p->next;
4               free(curr_p);
5           } else {
6               pred_p->next = curr_p->next;
7
8               free(curr_p);
9           }
10          return 1;
11      } else {
12          printf("%d is not in the list\n", value);
13          return 0;
14      }
15  }
```

Uma lista ligada multi-threaded

UFABC

- Vamos tentar usar essas funções em um programa com threads
- Para compartilhar a lista, vamos definir a variável `head_p` como uma global
- Isto vai simplificar as funções `Member`, `Insert` e `Delete` já que não precisaremos passar `head_p` ou um ponteiro para `head_p`: basta passar o valor de interesse

Acesso simultâneo por várias threads



Solução 1

UFABC

- Travamos a lista toda a toda vez que algum thread queira acessar
- Cada chamada para cada uma das funções precisa ser protegida por um mutex

```
1 Pthread_mutex_lock(&list_mutex);
2 Member(value);
3 Pthread_mutex_unlock(&list_mutex);
```

Problemas da solução 1

UFABC

- Estamos serializando o acesso à lista
- Se a maior parte das nossas chamadas à lista for à função `Member`, estamos desperdiçando a oportunidade de paralelismo
- Por outro lado, se a maior parte das operações forem chamadas à `Insert` e a `Delete`, esta pode ser a melhor solução já que precisaríamos de qualquer forma serializar o acesso à lista para a maior parte dos acessos e esta solução é fácil de implementar
- Solução de granularidade grossa

Solução 2

UFABC

- Em vez de travar o acesso à lista toda, podemos travar nós individualmente
- Solução de granularidade fina

```
1  struct list node s {
2      int data;
3      struct list_node_s* next;
4      pthread_mutex_t mutex;
5  }
```

Problemas da solução 2

UFABC

- Usar essa ideia é muito mais complicada que a ideia original
- Também é muito mais lenta já que, em geral, cada vez que um nó for acessado um mutex precisa ser travado e destravado
- A adição de um mutex a cada nó da lista vai aumentar consideravelmente a quantidade de memória necessária para armazenar a lista

Implementação de **Member** com um mutex por nó

UFABC

```
1  int  Member(int value) {
2      struct list_node_s *temp, *old_temp;
3
4      pthread_mutex_lock(&head_mutex);
5      temp = head;
6      if (temp != NULL)
       ↪  pthread_mutex_lock(&(temp->mutex));
7      pthread_mutex_unlock(&head_mutex);
8      while (temp != NULL && temp->data < value) {
9          if (temp->next != NULL)
10             pthread_mutex_lock(&(temp->next->mutex));
11         old_temp = temp;
12         temp = temp->next;
13         pthread_mutex_unlock(&(old_temp->mutex));
14     }
15     ...
```

Implementação de **Member** com um mutex por nó

UFABC

```
1       ...
2       if (temp == NULL || temp->data > value) {
3
4           if (temp != NULL)
5               pthread_mutex_unlock(&(temp->mutex));
6           return 0;
7       } else { /* temp != NULL && temp->data <= value */
8
9           pthread_mutex_unlock(&(temp->mutex));
10          return 1;
11      }
12  }
```

UFABC

- Nenhuma das soluções apresentadas explora o potencial para acesso simultâneo a qualquer nó da lista quando estamos executando a função `Member`
- A primeira solução só permite um por vez na lista toda
- A segunda solução só permite um por vex por nó

UFABC

- Uma trava de leitura e escrita é parecida com um mutex, contudo ela têm duas funções de trava
- A primeira função trava para leitura enquanto a segunda trava para escrita

- Assim múltiplos threads podem obter o lock simultaneamente chamando a função de trava para leitura enquanto apenas um thread pode obter a trava para escrita em um dado instante
- Logo, se threads possuem a trava para leitura, quaisquer threads que tentarem obter a trava para escrita ficarão bloqueados na chamada à função de trava

UFABC

```
1  int pthread_rwlock_init(
2          pthread_rwlock_t* rwlock_p          /* out */,
3          const pthread_rwlockattr_t* attr_p /* in */);
4
5  int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p);
6
7  int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p);
8  int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock_p);
9
10 int pthread_rwlock_unlock(pthread_rwlock_t* rwlock_p);
```

Protegendo a lista ligada

UFABC

```
1  pthread_rwlock_rdlock(&rwlock);
2  Member(value);
3  pthread_rwlock_unlock(&rwlock);
4  ...
5  pthread_rwlock_wrlock(&rwlock);
6  Insert(value);
7  pthread_rwlock_unlock(&rwlock);
8  ...
9  pthread_rwlock_wrlock(&rwlock);
10 Delete(value);
11 pthread_rwlock_unlock(&rwlock);
```

UFABC

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 99.9% `Member`, 0.05% `Insert`, 0.05% `Delete`

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

UFABC

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

Uma nota sobre false sharing e caches

UFABC

- Lembre-se que os designers de chips incluíram memórias relativamente rápidas nos processadores chamadas memórias cache
- O uso eficiente das memórias cache pode ter um impacto significativo no desempenho de programas rodando em máquinas de memória compartilhada
- Um write-miss ocorre quando um core tenta atualizar uma variável que não está na cache e, consequentemente, é obrigado a acessar a memória principal.

Multiplicação matrix vetor - Pthreads

UFABC

```
1  void *Pth_mat_vect(void* rank) {
2      long my_rank = (long) rank;
3      int i, j;
4      int local_m = m/thread_count;
5      int my_first_row = my_rank*local_m;
6      int my_last_row = (my_rank+1)*local_m - 1;
7
8      for (i = my_first_row; i <= my_last_row; i++) {
9          y[i] = 0.0;
10         for (j = 0; j < n; j++)
11             y[i] += A[i*n+j]*x[j];
12     }
13
14     return NULL;
15 }
```

Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

Thread-Safety

UFABC

- Um bloco de código é thread-safe se ele pode *ser executado por múltiplos threads sem que isto cause problemas*

Exemplo

UFABC

- Suponha que queiramos usar múltiplas threads para fazer a *"tokenização"*do conteúdo de um arquivo texto
- Cada *token* é simplesmente uma sequência contígua de caracteres separadas do resto do texto por espaços (espaços, tabs, quebras de linhas, ...)

Abordagem simples

UFABC

- Divida a entrada (arquivo) em linhas e atribua conjuntos de linhas a cada uma das threads usando *round-robin*
  - A linha 1 vai para a thread 0
  - A linha 2 vai para a thread 1
  - ...
  - A linha t vai para a thread $t - 1$
  - A linha $t + 1$ vai para a thread 0
  - ...

Abordagem simples

UFABC

- Podemos, por exemplo, serializar o acesso às linhas do arquivo usando mutexes ou semáforos
- Logo depois que uma thread tiver lido uma linha do arquivo ela pode *tokenizar* a linha usando a função `strtok`

A função `strtok`

UFABC

- A primeira chamada deve conter a string a ser tokenizada e o separador
  - Entrada: uma linha do arquivo, separador espaço
- Nas chamadas seguintes à `strtok`, o primeiro argumento (a string a ser tokenizada) deve ser `NULL`

```
1  char* strtok(
2                char*       string     /* in/out */,
3                const char* separators /* in     */);
```

A função `strtok`

UFABC

- A ideia por trás da função `strtok` é a de que após a primeira chamada a função faça um cache do ponteiro para a string a ser tokenizada e para cada chamada seguinte ela devolva como resposta os tokens obtidos a partir da {string apontada pelo ponteiro em cache}

Um tokenizador multi-threaded

UFABC

```
1  void* Tokenize(void* rank) {
2      long my_rank = (long) rank;
3      int count;
4      int next = (my_rank + 1) % thread_count;
5      char *fg_rv;
6      char my_line[MAX];
7      char *my_string;
8      sem_wait(&sems[my_rank]);
9      fg_rv = fgets(my_line, MAX, stdin);
10     sem_post(&sems[next]);
11     while (fg_rv != NULL) {
12         printf("Thread %ld > my_line = %s", my_rank,
           ↪  my_line);
```

Um tokenizador multi-threaded

UFABC

```
1           ...
2           count = 0;
3           my_string = strtok(my_line, " \t\n");
4           while ( my_string != NULL ) {
5               count++;
6               printf("Thread %ld > string %d = %s\n",
                ↪  my_rank, count,
7                       my_string);
8               my_string = strtok(NULL, " \t\n");
9           }
10      }
11      sem_wait(&sems[my_rank]);
12      fg_rv = fgets(my_line, MAX, stdin);
13      sem_post(&sems[next]);
14      return NULL;
15  }
```

Executando: 1 thread

UFABC

- Com uma só thread o funcionamento é conforme esperávamos

```
Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.
```

Executando: 2 threads

UFABC

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old
```

O que aconteceu?

UFABC

- `strtok` faz um cache do ponteiro para a string a ser tokenizada
- Uma cópia do ponteiro é mantida em uma variável `static`
- Assim, um valor armazenado nessa variável persiste de uma chamada à outra
- Por outro lado, a variável é compartilhada entre as threads e não privada

O que aconteceu?

UFABC

- A thread 0 chama `strtok` com a 3 linha da entrada e sobrescreve o conteúdo que havia sido colocado na variável pela thread 1
  - Há uma condição de corrida (*race condition*)
- Uma thread influencia a outro! A função `strtok` não é thread-safe!
  - Em outras palavras, chamadas simultâneas à `strtok` podem dar resultados incorretos

Outras funções dignas de nota

UFABC

- Infelizmente um bom número das funções em C não são thread-safe
  - Exemplos:
    - Gerador de números aleatórios `random` em `stdlib.h`
    - Rotina de conversão de horários `localtime` em `time.h`

Funções reentrantes (*thread-safe*)

UFABC

- Uma função é reentrante (*re-entrant*) quando pode-se invocá-la simultaneamente em vários threads sem a necessidade de tomarmos cuidados especiais para garantir que não haja uma condição de corrida
- Algumas funções da biblioteca padrão C têm uma implementação reentrante disponível

```
1  char* strtok_r(
2              char*       string     /* in/out */,
3              const char* separators /* in     */
4              char**      save_ptr   /* in/out */);
```

Conclusão

Comentários finais

UFABC

- Uma thread em programação de memória compartilhada é semelhante à um processo em programação com memória distribuída
- Contudo, uma thread é normalmente muito mais leve que um processo completo
- Em Pthreads, todos os threads tem acesso a toda a memória do processo pai (variáveis globais, arquivos abertos, ...)
  - Variáveis locais (declaradas em funções ou em seus parâmetros são locais à thread. Por que?)

UFABC

- Quando
  - O resultado final da execução de um programa não for determinístico
    - O não determinismo for o resultado de múltiplas threads em execução e os seus efeitos (operações de escrita) em uma variável compartilhada (ou a um recurso compartilhado)
    - O resultado da execução for um erro ou um resultado incorreto
- Dizemos que há uma condição de corrida.

UFABC

- Uma seção crítica / região crítica (*critical section*) é um bloco de código que atualiza um recurso compartilhado que só poderia ser atualizado por uma thread por vez
- Portanto, a execução de uma região crítica deveria ser feita exclusivamente por uma thread por vez ou, em outras palavras, sequencialmente

UFABC

- Um laço de espera ocupada (*busy-waiting*) pode ser usado para evitar acessos conflitantes a uma região crítica do código utilizando uma variável de flag e um laço com o seu corpo vazio
- Pode desperdiçar ciclos da CPU
- Pode ser inócuo se as otimizações do compilador estiverem ativas

Comentários finais

- Um mutex pode ser usado para evitar acessos conflitantes a uma região crítica do código
- Pode-se pensar em um mutex como uma trava (eles também são chamados comumente de locks) na região crítica já que mutexes garantem acesso exclusivo a uma região crítica

Comentários finais

UFABC

- Um semáforo é a terceira maneira que vimos de garantir que acessos conflitantes não ocorram em uma região crítica
- É um inteiro sem sinal que tem duas operações
  - `sem_wait`
  - `sem_post`
- Semáforos são mais gerais e mais poderosos que mutexes pois
  - Podem ser inicializados travados
  - Podem controlar acessos à uma região até um número > 1 de threads
  - Não têm o conceito de propriedade - um thread diferente daquele que o travou pode liberá-lo

UFABC

- Uma barreira (*barrier*) é um ponto do programa no qual todos as threads ficam bloqueadas até que todas o tenham alcançado
- Uma trava de leitura e escrita (*read-write lock*) é utilizada quando é seguro que diversas threads leiam a partir de um recurso compartilhado mas quando é exigido que apenas uma thread faça escritas por vez e que nenhuma outra thread tenha acesso enquanto as escritas estiverem sendo feitas.

Comentários finais

- Algumas funções guardam ”estado”entre suas chamadas (variáveis `static`, globais, ...)
- Se diversas threads chamarem essas funções (simultaneamente ou intercaladamente) pode ocorrer uma condição de corrida
- Funções assim são chamadas de thread-unsafe ou não-reentrantes.