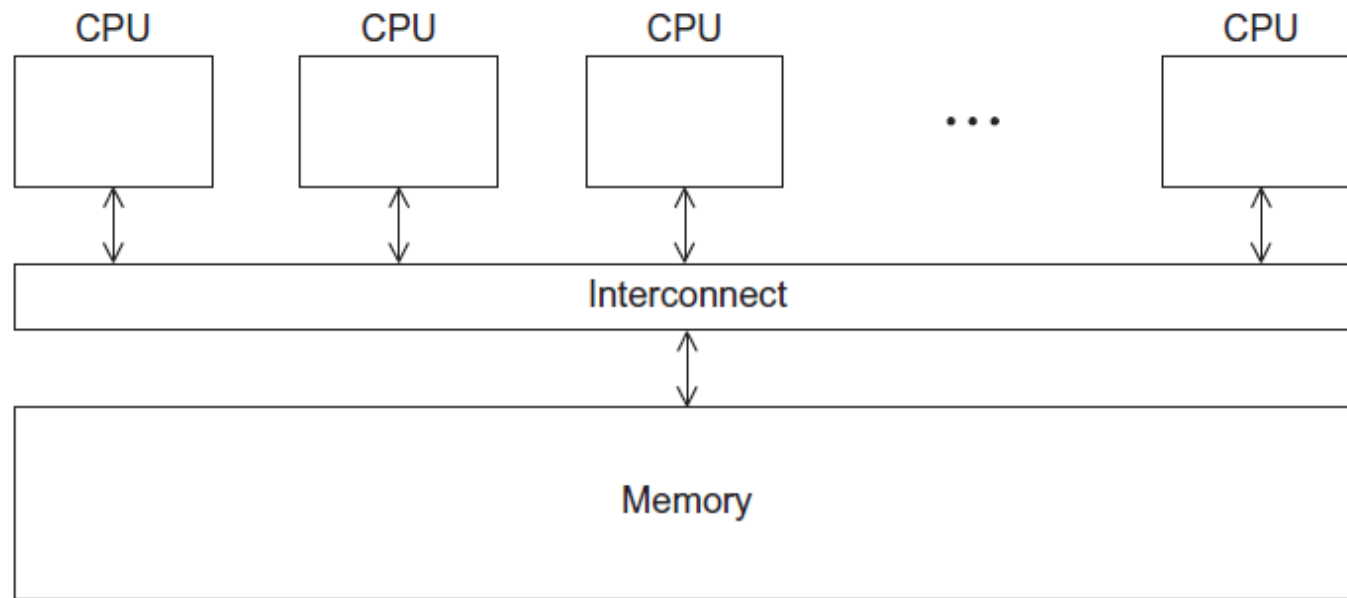


# MO644/MC900

## Programação em Memória Compartilhada usando Pthreads

Prof. Guido Araujo  
[www.ic.unicamp.br/~guido](http://www.ic.unicamp.br/~guido)

# Sistema de Memória Compartilhada



# Processos e *Threads*

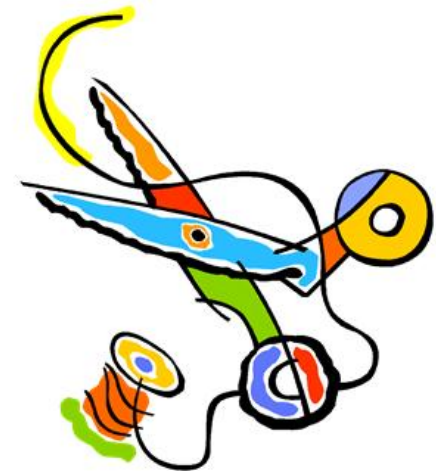
- Um processo é uma instância de um programa que está executando (ou em suspensão).
- Threads são análogas à processos “leves”.
- Em um programa de memória compartilhada um único processo pode ter várias threads em execução.

# POSIX® Threads

- Também chamadas de Pthreads.
- É um padrão para SOs baseados em Unix.
- É uma biblioteca que pode ser ligada a programas em C.
- Especifica uma *Application Programming Interface* (API) para programação *multi-threaded*.

# Cuidado

- A API de Pthreads está somente disponível em sistemas POSIX — Linux, MacOS X, Solaris, HPUX, ...



# Hello World! (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


/* Global variable: accessible to all threads */
int thread_count;

void *Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

    /* Get number of threads from command line */
    thread_count = strtol(argv[1], NULL, 10);

    thread_handles = malloc (thread_count*sizeof(pthread_t));
```



declara várias funções com  
Pthreads, constantes, tipos, etc.

# Hello World! (2)

```
for (thread = 0; thread < thread_count; thread++)  
    pthread_create(&thread_handles[thread], NULL,  
        Hello, (void*) thread);  
  
printf("Hello from the main thread\n");  
  
for (thread = 0; thread < thread_count; thread++)  
    pthread_join(thread_handles[thread], NULL);  
  
free(thread_handles);  
return 0;  
} /* main */
```

# Hello World! (3)

```
void *Hello(void* rank) {  
    long my_rank = (long) rank;  /* Use long in case of 64-bit system */  
  
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);  
  
    return NULL;  
} /* Hello */
```



# Compiling a Pthread program

```
gcc -g -Wall -o pthread_hello pthread_hello.c -lpthread
```

liga a biblioteca de Pthreads



# Executando um programa em Pthreads

`./ pthread_hello <número de threads>`

`./ pthread_hello 1`

Hello from the main thread

Hello from thread 0 of 1

`./ pthread_hello 4`

Hello from the main thread

Hello from thread 0 of 4

Hello from thread 1 of 4

Hello from thread 2 of 4

Hello from thread 3 of 4

# Variáveis globais

- Pode introduzir *bugs* sutis e confusos!
- Limite o uso de variáveis globais a situações em que elas são realmente necessárias.
  - Ex: variáveis compartilhadas (*shared*).



# Inicializando as *Threads*

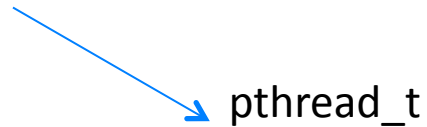
- Processos em MPI são normalmente inicializados por um *script*.
- Em Pthreads as *threads* são inicializadas por um programa executável.

# pthread\_t objects

- Opaco
- Os dados que ele armazena são específicos do sistema.
- Os campos de dados não são acessíveis pelo código de usuário.
- No entanto, o padrão de Pthreads garante que um objeto pthread\_t armazene informação suficiente para identificar univocamente a thread com a qual ele está associado.

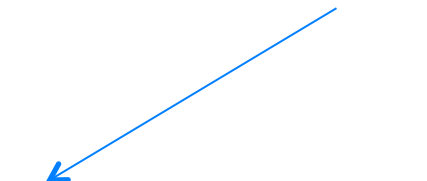
# Inicializando as *Threads*

pthread.h



pthread\_t

Um objeto para  
cada thread.



```
int pthread_create (  
    pthread_t* thread_p          /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p                  /* in */ );
```

# Olhando de perto (1)


```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

Não usaremos isto, então passamos NULL.

Alocar antes de chamar.

# Olhando de perto (2)

```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```



Apontador para o argumento que precisa ser  
passado para a função *start\_routine*.

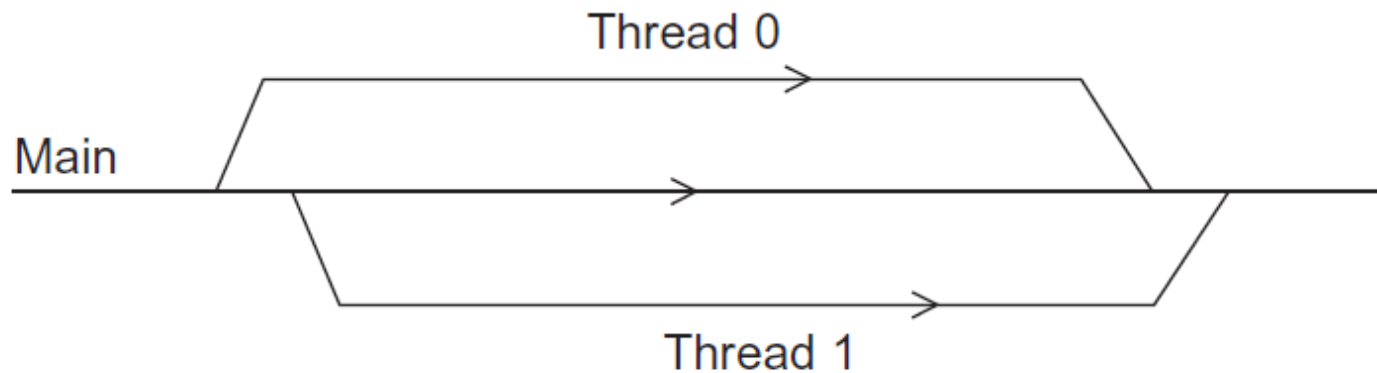
A função que a thread vai executar.



# Função inicializada por pthread\_create

- Protótipo:  
`void* thread_function ( void* args_p ) ;`
- `void*` pode ser *cast* para qualquer tipo de apontador em C.
- Deste modo, `args_p` pode apontar para qualquer lista contendo um ou mais valores necessários à `thread_function`.
- Do mesmo modo, o valor de retorno da `thread_function` pode apontar para uma lista de um ou mais valores.

# Executando as Threads



Main thread forks and joins two threads.

# Parando as Threads

- Chamamos a função `pthread_join` uma vez para cada thread.
- Uma única chamada para `pthread_join` espera pela conclusão da thread associada ao objeto `pthread_t`.

# Alguns detalhes

criando threads a medida que vai precisando ?

```
for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL,
        Hello, (void*) thread);

printf("Hello from the main thread\n");

for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);

free(thread_handles);
return 0;
} /* main */

void *Hello(void* rank) {
    long my_rank = (long) rank; /* Use long in case of 64-bit system */

    printf("Hello from thread %ld of %d\n", my_rank, thread_count);

    return NULL;
} /* Hello */
```

Tem que ser igual ao do tipo inteiro para permitir cast

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$


$x_0$
$x_1$
$\vdots$
$x_{n-1}$

 $=$ 

$y_0$
$y_1$
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
$y_{m-1}$

# MULTIPLICAÇÃO MATRIZ-VETOR USANDO PTHREADS

# Pseudo-código serial

```
/* For each row of A */  
for (i = 0; i < m; i++) {  ← paralelizar quem?  
    y[i] = 0.0;  
    /* For each element of the row and each element of x */  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]* x[j];  
}
```

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

compartilhar quem?    A e x

como era em MPI?    precisava broadcast linhas de A e todo x !!!

# Usando 3 Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

n = m = 6

thread 0

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

caso geral

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

# Multiplicação matriz-vetor em Pthreads

```
void *Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i][j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
```





# SEÇÕES CRÍTICAS

# Estimando $\pi$

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

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

# Uma função de thread para calcular $\pi$

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) /* my_first_i is even */
        factor = 1.0;
    else /* my_first_i is odd */
        factor = -1.0;

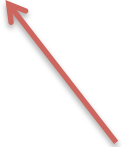
    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }

    return NULL;
} /* Thread_sum */
```

# Usando um processador dual core

	$n$			
	$10^5$	$10^6$	$10^7$	$10^8$
$\pi$	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 à medida que aumentamos  $n$ , a estimativa com uma única thread vai melhorando.



mas o que está ocorrendo aqui?

# Condições de corrida possíveis

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread
3	Assign $y = 1$	Call Compute ()
4	Put $x=0$ and $y=1$ into registers	Assign $y = 2$
5	Add 0 and 1	Put $x=0$ and $y=2$ into registers
6	Store 1 in memory location $x$	Add 0 and 2
7		Store 2 in memory location $x$

```
y = Compute(my_rank);
```

```
x = x + y;
```

$y$  privada

$x$  compartilhada



# Busy-Waiting

- Uma thread repetidamente testa uma condição, mas, efetivamente não faz qualquer trabalho útil até que a condição seja satisfeita.

flag inicializada para 0 na thread principal

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```

o que ocorre se eu usar -O3?

# Cuidado 1

- Compiladores otimizantes podem ser um problema!!

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```

depois de otimizado!!

```
y = Compute(my_rank);  
x = x + y;  
while (flag != my_rank);  
flag++;
```

# Cuidado 2

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```



qual o problema aqui?

se número threads é 4

último core faz flag++

resultar em 4 !!



## Cuidado 2 (cont.)

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag = (flag+1) % thread_count;
```



Solução !!

# Soma global usando Pthreads com busy-waiting

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */
```

qual o problema aqui?

para  $n = 10^{18}$

serial: 2.8s

2 threads: 19.5s

como melhorar?

Função para soma global com seção crítica depois do laço (1)

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor, my_sum = 0.0;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
  
    if (my_first_i % 2 == 0)  
        factor = 1.0;  
    else  
        factor = -1.0;
```

## Função para soma global com seção crítica depois do laço (2)

```
for (i = my_first_i; i < my_last_i; i++, factor = -factor)
    my_sum += factor/(2*i+1);
```

```
while (flag != my_rank);
sum += my_sum;
flag = (flag+1) % thread_count;
```

```
return NULL;
```

```
} /* Thread_sum */
```

faça muitas somas locais!

2 threads: 1.5s !!

# Mutexes

- Uma thread que está em *busy-waiting* pode usar a CPU continuamente sem fazer qualquer trabalho.
- Mutex (*mutual exclusion*) é um tipo especial de variável que pode ser usado para acessar a região crítica de uma única thread a qualquer tempo.

# Mutexes



- Usada para garantir que uma thread “exclui” todas as outras threads enquanto ela executa a seção crítica.
- O padrão Pthreads inclui um tipo especial de mutexes: `pthread_mutex_t`.

```
int pthread_mutex_init(  
    pthread_mutex_t*      mutex_p    /* out */  
    const pthread_mutexattr_t* attr_p  /* in  */);
```

# Mutexes

- Quando um programa usando Pthreads termina de usar um mutex, ele deve chamar

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

- Para poder ganhar acesso à seção crítica que uma thread chama usar

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

# Mutexes

- Quando uma thread termina de executar o código em uma seção crítica ele deve chamar

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



## Função de soma global usando mutex (1)

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
    double my_sum = 0.0;  
  
    if (my_first_i % 2 == 0)  
        factor = 1.0;  
    else  
        factor = -1.0;
```

## Função de soma global usando mutex (2)

```
for (i = my_first_i; i < my_last_i; i++, factor = -factor) {  
    my_sum += factor/(2*i+1);  
}  
pthread_mutex_lock(&mutex);  
sum += my_sum;  
pthread_mutex_unlock(&mutex);  
  
return NULL;  
} /* Thread_sum */
```

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

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \text{thread\_count}$$

qual o problema aqui?

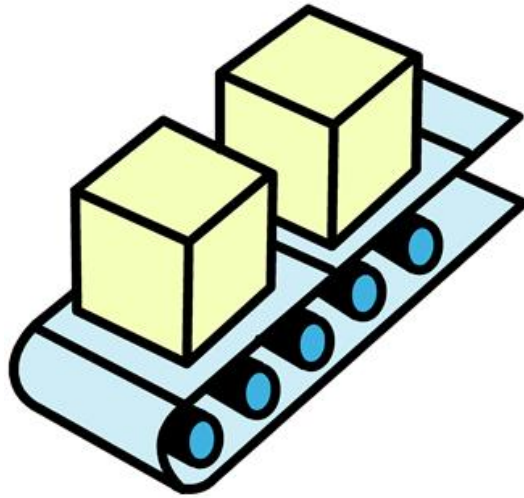
Tempos de execução (em segundos) de programas para o cálculo do  $\pi$  usando  $n = 108$  termos em um sistema com 2 processadores de 4 núcleos cada (8 núcleos).

Time	flag	Thread				
		0	1	2	3	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

Possível seqüência de eventos para *busy-waiting* e mais threads (5) que núcleos (2).

OS volta 3 para execução; qual o problema aqui?

3 poderia já ter entrado, mas precisa esperar pelo retorno de 2!!



# SINCRONIZAÇÃO DE PRODUTOR- CONSUMIDOR E SEMÁFOROS

# Problemas

- Busy-waiting força a ordem em que as threads acessam a região crítica.
- Usando mutexes, a ordem é deixada para o sistema e à “sorte”.
- Existem sistemas em que precisamos controlar a ordem em que as threads acessam a região crítica.

# Problemas com uma solução baseada em mutex

```
/* n and product_matrix are shared and initialized by the main thread */  
/* product_matrix is initialized to be the identity matrix */  
void* Thread_work(void* rank) {  
    long my_rank = (long) rank;  
    matrix_t my_mat = Allocate_matrix(n);  
    Generate_matrix(my_mat);  
    pthread_mutex_lock(&mutex);  
    Multiply_matrix(product_mat, my_mat);  
    pthread_mutex_unlock(&mutex);  
    Free_matrix(&my_mat);  
    return NULL;  
} /* Thread_work */
```

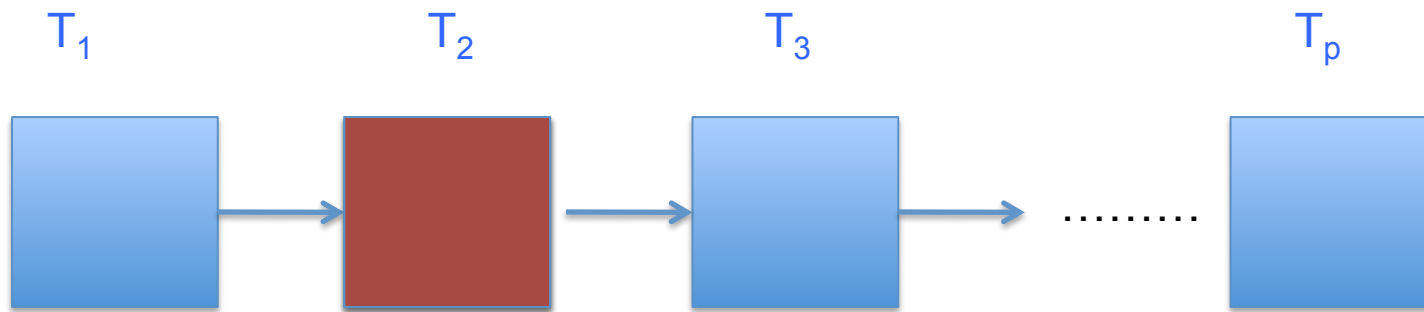
Código faz:  $\prod$  my\_mat

Qual o problema aqui?

Multiplicação de matrizes não é comutativa. A ordem importa!!

# Enviando mensagens entre threads

- Uma thread envia mensagem para a seguinte.



Qual o problema aqui?

$T_i$  somente pode receber se  $T_{i-1}$  estiver pronta. A ordem importa!!



## Uma primeira tentativa de enviar mensagens usando pthreads

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

    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
    messages[dest] = my_msg;

    if (messages[my_rank] != NULL)
        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
    else
        printf("Thread %ld > No message from %ld\n", my_rank, source);

    return NULL;
} /* Send_msg */
```

Qual o problema aqui?

Pode tentar imprimir um buffer ainda não preenchido. A ordem importa!!

# E se usar busy-wait?

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

problema já conhecido com busy-wait!!

O que queremos?

```
. . .  
messages[dest] = my_msg;  
Notify thread dest that it can proceed;
```

```
Await notification from thread source  
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);  
. . .
```

Que uma thread notifique  
a outra que a mensagem  
está pronta. Queremos  
ordem nas threads!

E qual a solução?

# Sintaxe das funções que usam semáforos

```
#include <semaphore.h>
```

← Você precisa acrescentar isto pois  
semáforos não são parte de Pthreads.

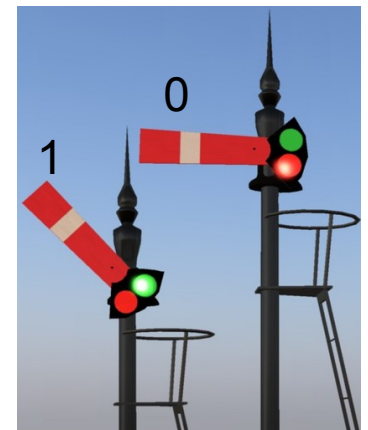
```
int sem_init(  
    sem_t*      semaphore_p    /* out */,  
    int         shared         /* in  */,  
    unsigned    initial_val    /* in  */);
```

```
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 */);
```

Para que serve semáforo então?

objetivo do semáforo **não**  
é resolver corrida, mas  
sim garantir ordem!!

A vantagem é que o  
código não possui posse  
do semáforo!!



# Usando semáforo

```
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
8      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9      messages[dest] = my_msg;
10     sem_post(&semaphores[dest])
        /* “Unlock” the semaphore of dest */
11
12     /* Wait for our semaphore to be unlocked */
13     sem_wait(&semaphores[my_rank]);
14     printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16     return NULL;
17 } /* Send_msg */
```



# **BARREIRAS E CONDIÇÕES DE VARIÁVEIS**

# Barreiras

- Sincronizar as threads de modo a garantir que todas elas chegam no mesmo ponto do programa é chamado barreira.
- Nenhuma thread pode cruzar a barreira até que todas as threads a tenham alcançado.

## Usando barreira para determinar o tempo da thread mais lenta

```
/* 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;
```

Como era mesmo?

Alguma outra aplicação?

# Usando barreiras para depuração

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





# Busy-waiting e Mutex

- Implementar barreira usando busy-waiting e mutex é bem direto.
- No fundo, estamos usando um contador compartilhado e protegido pelo mutex.
- Quando o contador indicar que todas as threads entraram a seção crítica, as threads podem deixar a seção crítica.

# Busy-waiting e Mutex

```
/* Shared and initialized by the main thread */
int counter; /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

Imagine que você quer fazer uma outra barreira em seguida. Podem ocorrer 2 problemas. Quais são?

# Problemas com um único contador (1)

```
/* Barrier */  
pthread_mutex_lock(&barrier_mutex);  
counter++;  
pthread_mutex_unlock(&barrier_mutex);  
while (counter < thread_count);  
. . .
```

precisa zerar counter!!

qual o valor de counter aqui  
quando todas passarem?

```
/* Barrier */  
pthread_mutex_lock(&barrier_mutex);  
counter++;  
pthread_mutex_unlock(&barrier_mutex);  
while (counter < thread_count);  
. . .
```

E aqui?

O que está faltando?

# Problemas com um único contador (2)

```
/* Barrier */
pthread_mutex_lock(&barrier_mutex);
counter++;
pthread_mutex_unlock(&barrier_mutex);
while (counter < thread_count);
. . .
```

Testa aqui se última thread a entrar e faz counter = 0; funciona?

```
/* Barrier */
pthread_mutex_lock(&barrier_mutex);
counter++;
pthread_mutex_unlock(&barrier_mutex);
while (counter < thread_count);
. . .
```

# Problemas com um único contador (2)

```
/* Barrier */  
pthread_mutex_lock(&barrier_mutex);  
counter++;  
pthread_mutex_unlock(&barrier_mutex);  
while (counter < thread_count);  
. . .
```

←  $T_a$  ainda não zerou

counter = 0; E aqui?

```
/* Barrier */  
pthread_mutex_lock(&barrier_mutex);  
counter++;  
pthread_mutex_unlock(&barrier_mutex);  
while (counter < thread_count);  
. . .
```

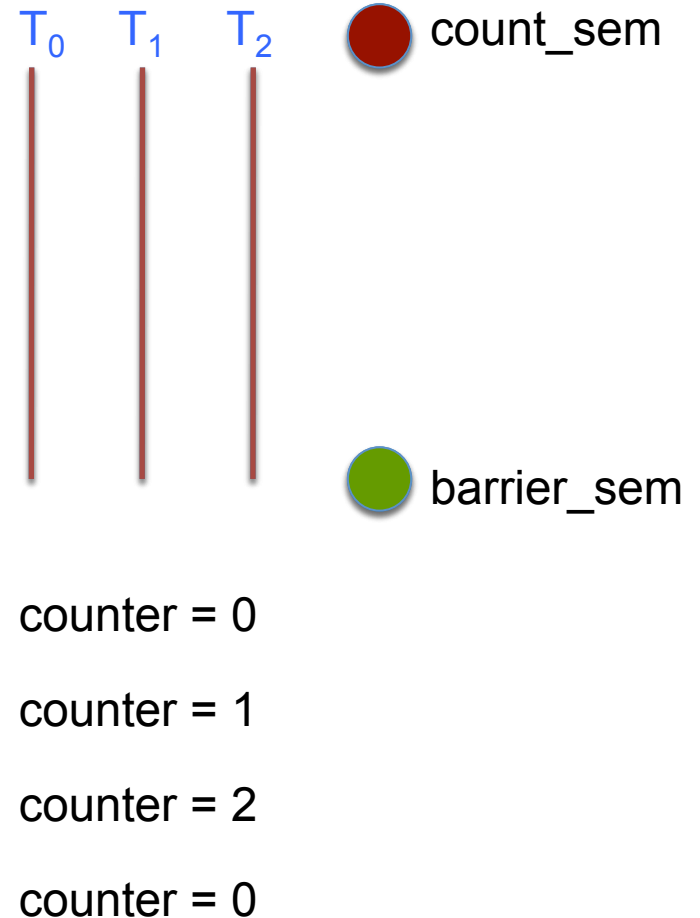
←  $T_b$  entra antes de  $T_a$  zerar!!

Perde um incremento, nunca  
chegando a thread\_count!!  
Qual a solução?

Precisa de um contador por  
barreira

# Implementando uma barreira com semáforos (3 threads)

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count - 1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count - 1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    . . .
}
```



# Variáveis de Condição

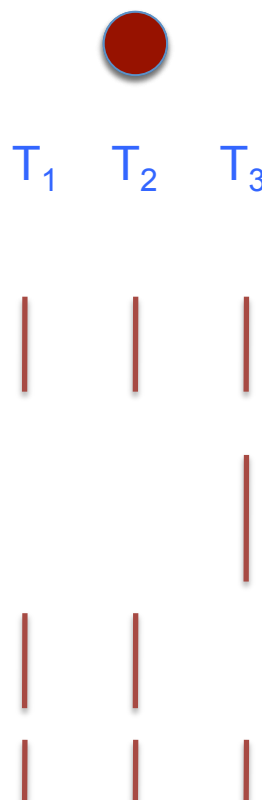
- Uma variável de condição é um objeto que permite uma thread suspender a execução até que um certo evento ou condição ocorra.
- Quando o evento ou condição ocorrer, uma outra thread pode (sinalizar) para despertar a thread.
- Variáveis de condição estão sempre associadas à mutex.

# 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;
```



## Implementando uma barreira com variáveis de condição



```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
}
```

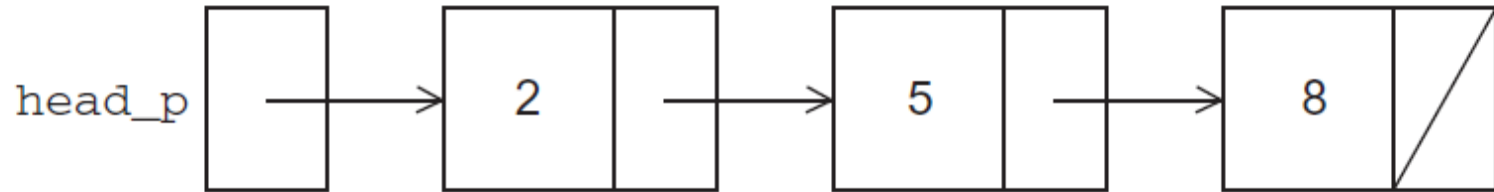


# READ-WRITE LOCKS

# Controlando o acesso a uma estrutura de dados grande e compartilhada

- Vamos ver um exemplo.
- Suponha que a estrutura que estamos interessados é uma lista ligada ordenada e que as operações de interesse são **Member**, **Insert**, e **Delete**.

# Linked Lists



```
struct list_node_s {  
    int data;  
    struct list_node_s* next;  
}
```

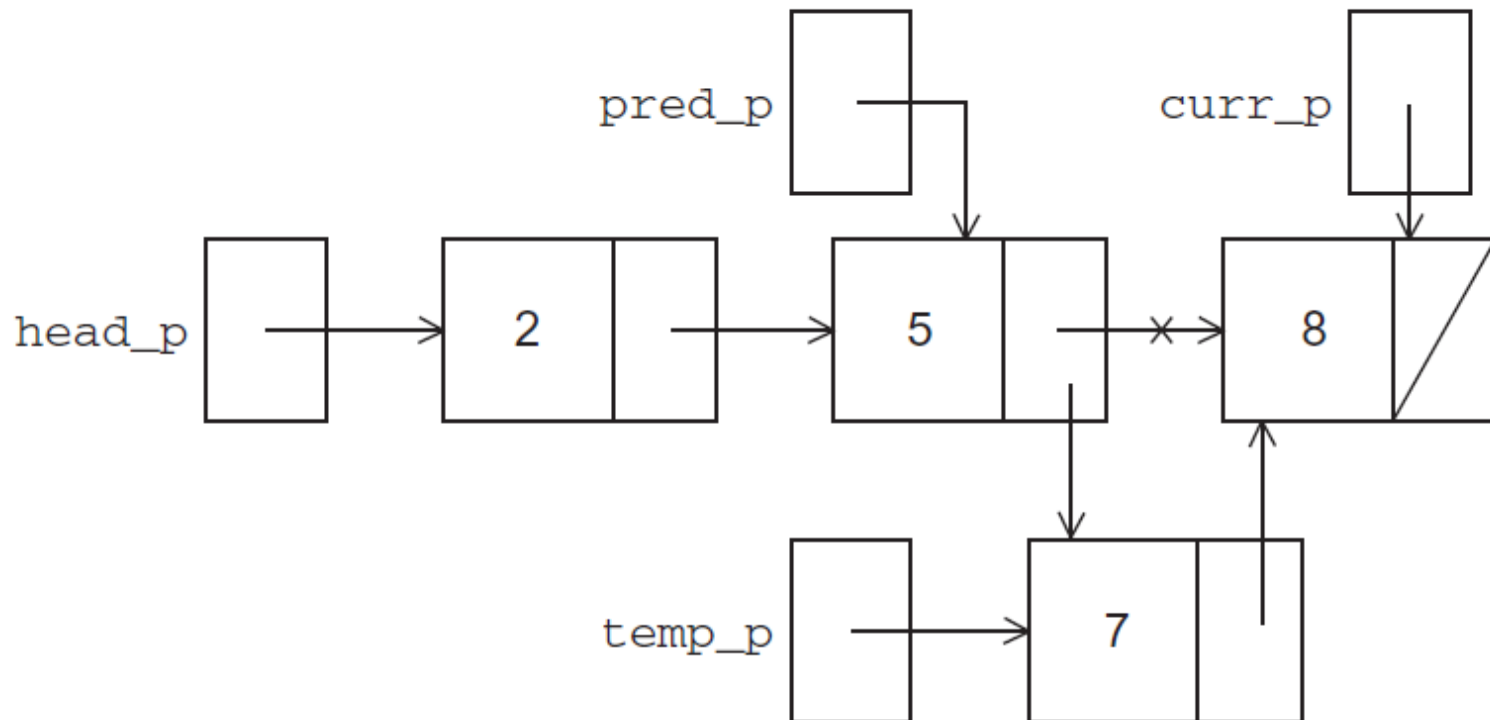
# Linked List Member

```
int Member(int value, struct list_node_s* head_p) {
    struct list_node_s* curr_p = head_p;

    while (curr_p != NULL && curr_p->data < value)
        curr_p = curr_p->next;

    if (curr_p == NULL || curr_p->data > value) {
        return 0;
    } else {
        return 1;
    }
} /* Member */
```

# Inserindo um novo nó na lista (1)



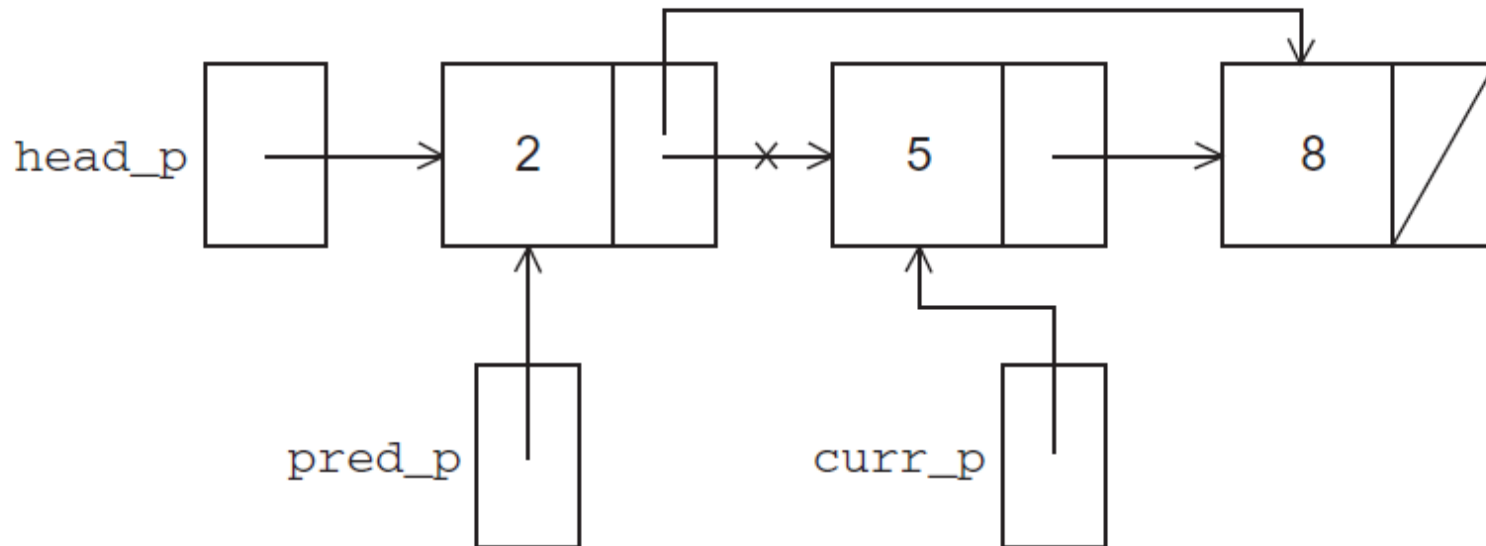
# Inserindo um novo nó na lista (2)

```
int Insert(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;
    struct list_node_s* temp_p;

    while (curr_p != NULL && curr_p->data < value) {
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

    if (curr_p == NULL || curr_p->data > value) {
        temp_p = malloc(sizeof(struct list_node_s));
        temp_p->data = value;
        temp_p->next = curr_p;
        if (pred_p == NULL) /* New first node */
            *head_pp = temp_p;
        else
            pred_p->next = temp_p;
        return 1;
    } else { /* Value already in list */
        return 0;
    }
} /* Insert */
```

# Apagando um nó da lista (1)





# Apagando um nó da lista (2)

```
int Delete(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;

    while (curr_p != NULL && curr_p->data < value) {
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

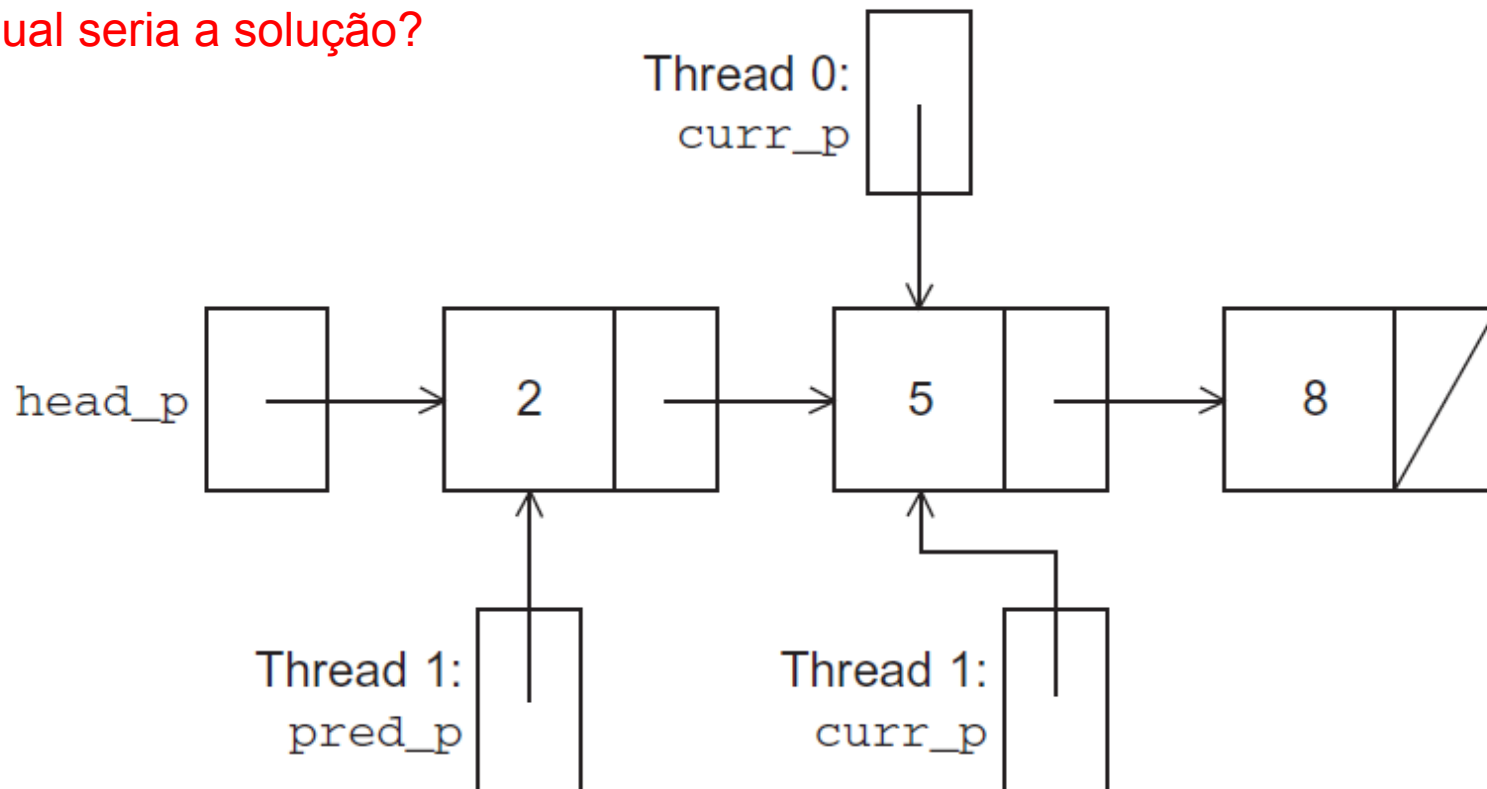
    if (curr_p != NULL && curr_p->data == value) {
        if (pred_p == NULL) { /* Deleting first node in list */
            *head_pp = curr_p->next;
            free(curr_p);
        } else {
            pred_p->next = curr_p->next;
            free(curr_p);
        }
        return 1;
    } else { /* Value isn't in list */
        return 0;
    }
} /* Delete */
```

# Uma lista ligada Multi-Threaded

- Vamos tentar usar estas funções em um programa Pthreads.
- Visando compartilhar o acesso à lista, podemos definir `head_p` como uma variável global.
- Isto vai simplificar os *headers* das funções `Member`, `Insert`, and `Delete`, já que não precisamos mais passar `head_p` ou mesmo um apontador para `head_p`; somente precisamos passar o valor do nó.

# Acesso simultâneo a duas threads

Qual seria a solução?



# Solução #1

- Uma solução óbvia é simplesmente usar lock na lista inteira toda vez que uma thread tentar acessá-la.
- Uma chamada para cada uma das funções pode ser protegida por um mutex.

```
Pthread_mutex_lock(&list_mutex);  
Member(value);  
Pthread_mutex_unlock(&list_mutex);
```

Qual o problema com esta estratégia?

# Problemas

- No fundo, serializamos o acesso à lista.
- Se a grande maioria de nossos acessos forem chamadas a **Member**, não conseguiremos explorar o paralelismo. **Por que?**
- Por outro lado, se a maioria de nossas chamadas forem para **Insert** e **Delete**, então esta é a melhor solução já que precisaremos serializar o acesso à lista na maior parte das vezes, e esta solução é certamente fácil de implementar. **Por que?**

**Tem como melhorar isto?**

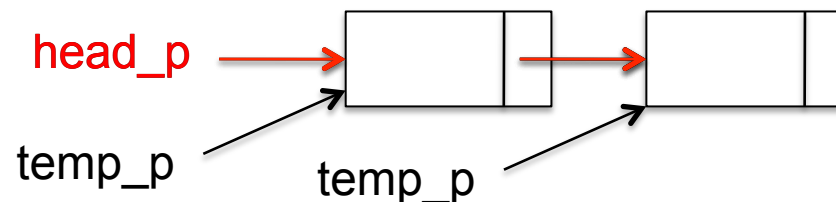
## Solução #2

- Ao invés de usar lock para toda a lista poderíamos usá-lo somente nos nós individuais.
- Esta é uma abordagem chamada “fine-grained”

```
struct list_node_s {  
    int data;  
    struct list_node_s* next;  
    pthread_mutex_t mutex;  
}
```

# Implementação de **Member** usando um mutex por nó (1)

```
int Member(int value) {  
    struct list_node_s* temp_p;  
  
    pthread_mutex_lock(&head_p_mutex);  
    temp_p = head_p;  
    while (temp_p != NULL && temp_p->data < value) {  
        if (temp_p->next != NULL)  
            pthread_mutex_lock(&(temp_p->next->mutex));  
        if (temp_p == head_p)  
            pthread_mutex_unlock(&head_p_mutex);  
        pthread_mutex_unlock(&(temp_p->mutex));  
        temp_p = temp_p->next;  
    }  
}
```



## Implementação de **Member** usando um mutex por nó (2)

O que significa isto?

```
if (temp_p == NULL || temp_p->data > value) {  
    if (temp_p == head_p)  
        pthread_mutex_unlock(&head_p_mutex);  
    if (temp_p != NULL)  
        pthread_mutex_unlock(&(temp_p->mutex));  
    return 0;  
} else {  
    if (temp_p == head_p)  
        pthread_mutex_unlock(&head_p_mutex);  
    pthread_mutex_unlock(&(temp_p->mutex));  
    return 1;  
}  
} /* Member */
```

Não achou ☹

E isto?

Achou ☺ !!



# Problemas

- Isto é muito mais complexo que a função **Member** original.
- É também muito mais lento, já que, no geral, cada vez que um nó é acessado, um mutex precisa ser usado (*locked* e *unlocked*).
- Adicionar um campo de mutex vai aumentar substancialmente o tamanho necessário para armazenar a lista.

# Problemas

- Nenhuma das soluções para acesso multi-threaded à lista permite explorar o acesso simultâneo aos membros da listas pelas threads que estão executando **Member**.
- A primeira solução somente permite uma thread acessar a lista por vez.
- A segunda solução permite somente uma thread acessar qualquer nó em um dado instante.

# Pthreads Read-Write Locks

- Uma *lock read-write* é como um mutex exceto que ela fornece duas funções de lock.
- A primeira função trava a *lock* para **leitura**, enquanto a segunda função trava ela para **escrita**.

# Pthreads Read-Write Locks

- Deste modo, várias threads podem ao mesmo tempo obter a lock chamando a função *read-lock*, enquanto somente uma thread pode obter a lock chamando a função *write-lock*.



# Pthreads Read-Write Locks

- Se qualquer thread possuir a lock de escrita, quaisquer outras threads que quiserem obter a lock para leitura ou escrita vai bloquear quando fizer a chamada de sua função.
- Além disto, se quaisquer locks possuírem a lock para leitura, qualquer thread que desejar obter a lock para escrita vai bloquear durante a chamada da função *write-lock*.

# Protegendo a nossas chamadas à lista ligada

```
pthread_rwlock_rdlock(&rwlock);  
Member(value);  
pthread_rwlock_unlock(&rwlock);  
.  
.  
.  
pthread_rwlock_wrlock(&rwlock);  
Insert(value);  
pthread_rwlock_unlock(&rwlock);  
.  
.  
.  
pthread_rwlock_wrlock(&rwlock);  
Delete(value);  
pthread_rwlock_unlock(&rwlock);
```

# Desempenho da lista ligada

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

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

# Desempenho da lista ligada

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

100,000 ops/thread

80% Member

10% Insert

10% Delete

Pior que serial !!

O que fazer?





# THREAD-SAFETY

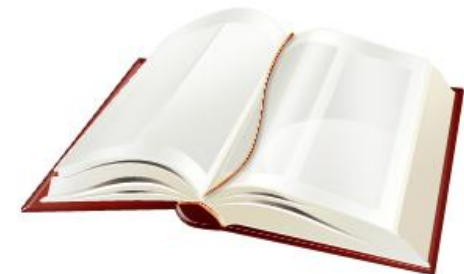
# Thread-Safety

- Um bloco de código é **thread-safe** se ele pode ser executado simultaneamente por múltiplas threads sem causar problemas.



# Exemplo

- Assuma que desejamos utilizar múltiplas threads para “tokenize” um arquivo contendo texto em Inglês.
- As tokens são sequências de caracteres separados do resto do texto por espaço em branco —*space*, *tab*, ou *newline*.



# Uma abordagem simples (1)

- Divida o arquivo de entrada em linhas de texto e atribua as linhas às *threads*, usando *round-robin*.
- A primeira linha é atribuída à thread 0, a segunda à thread 1, ..., t à thread t, t+1 à 0, etc.

# Uma abordagem simples (1)

- Podemos serializar o acesso à linhas da entrada usando semáforos.
- Depois que uma thread lê uma única linha da entrada ela pode transformar a linha em tokens usando a função `strtok`.

# A função `strtok`

- A primeira vez que ela é chamada o argumento da cadeia deve ser o texto a ser transformado em tokens.
- Nas chamadas subsequentes, o primeiro argumento deve ser NULL.

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

# The strtok function

- A idéia é que na primeira chamada `strtok` , armazena uma cópia do apontador para a cadeia; nas chamadas seguintes ele retorna tokens sucessivos a partir da cópia armazenada.

# Multi-threaded tokenizer (1)

```
void *Tokenize(void* rank) {
    long my_rank = (long) rank;
    int count;
    int next = (my_rank + 1) % thread_count;
    char *fg_rv;
    char my_line[MAX];
    char *my_string;

    sem_wait(&sems[my_rank]);
    fg_rv = fgets(my_line, MAX, stdin);
    sem_post(&sems[next]);
    while (fg_rv != NULL) {
        printf("Thread %ld > my line = %s", my_rank, my_line);
```



# Multi-threaded tokenizer (2)

```
count = 0;
my_string = strtok(my_line, " \t\n");
while ( my_string != NULL ) {
    count++;
    printf("Thread %ld > string %d = %s\n", my_rank, count,
        my_string);
    my_string = strtok(NULL, " \t\n");
}

sem_wait(&sems[my_rank]);
fg_rv = fgets(my_line, MAX, stdin);
sem_post(&sems[next]);
}

return NULL;
} /* Tokenize */
```

# Executando com um única thread

- Gera tokens corretamente.

Pease porridge hot.	1	$T_0$
Pease porridge cold.	2	$T_1$
Pease porridge in the pot	3	$T_0$
Nine days old.	4	$T_1$

# Executando com duas threads

```
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.
```

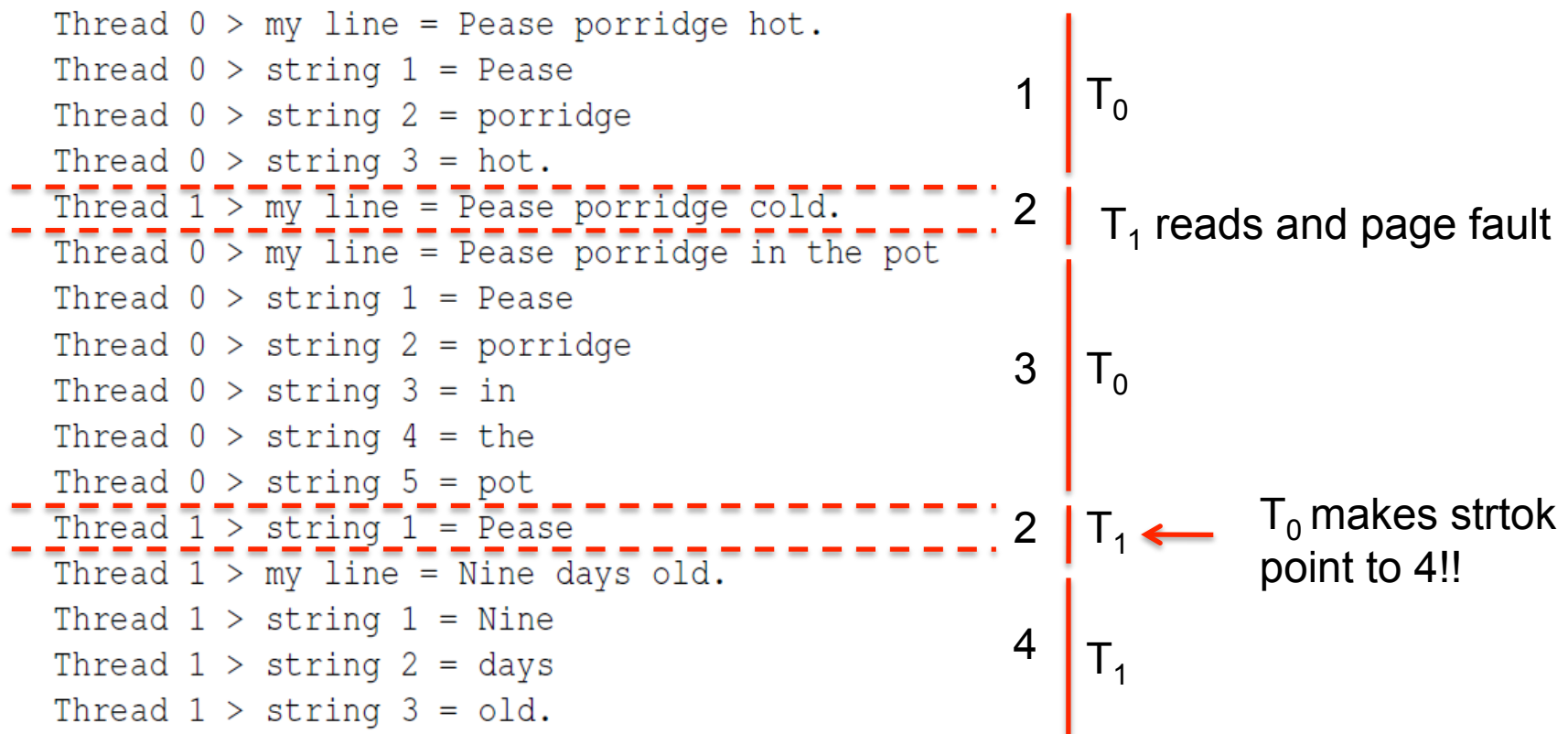
Oops!



O que ocorreu?

# Executando com duas threads

- 1 Pease porridge hot.
- 2 Pease porridge cold.
- 3 Pease porridge in the pot
- 4 Nine days old.



# O que ocorreu?

- `strtok` armazena a linha de entrada declarando uma variável como estática.
- Com isto, o valor da variável persiste de uma chamada para a seguinte.
- Infelizmente, esta cadeia é compartilhada e não privada.

# O que ocorreu?

- Deste modo, a chamada da thread 0 à `strtok` na 3ª linha da entrada aparentemente sobrescreveu o conteúdo da chamada da thread 1's na 2ª linha.
- Assim, a função `strtok` não é thread-safe.  
Se múltiplas threads a chamam simultaneamente a saída pode não ficar correta.



# Outras funções inseguras da biblioteca de C

- Infelizmente, não é incomum para funções da biblioteca de C não serem thread-safe.
- O gerador de números aleatórios `random` em `stdlib.h`.
- A função de conversão de tempo `localtime` in `time.h`.

# Funções *thread-safe* “re-entrantes”

- Em alguns casos, o padrão C especifica uma versão que é *thread-safe*.

```
char* strtok_r(  
    char*          string      /* in/out */,  
    const char*   separators, /* in */,  
    char**         saveptr_p   /* in/out */);
```



# Composição de código com locks

```
public boolean Move(List new, List old, int item)
{
    old.Delete(item);
    new.Insert(item);
}
```

- Imagine que nossa aplicação precise utilizar as operações da lista ligada para implementar uma outra operação de nível mais alto, como mover um elemento de uma lista para outra
- Não temos acesso ao código fonte
  - Apenas sabemos que cada operação é atômica

# Composição de código com locks

```
public boolean Move(List new, List old, int item)
{
    old.insert(item);
    new.delete(item);
}
```



Atômicas?

- Imagine que nossa aplicação precise utilizar as operações da lista ligada para implementar uma outra operação de nível mais alto
- Não temos acesso ao código fonte
  - Apenas sabemos que cada operação é atômica

# Composição de código com locks

```
public boolean Move(List from, List to, int item)
{
    newlock.lock();
    from.Delete(item);
    to.Insert(item);
    newlock.unlock();
}
```

Funciona?

← E se ocorrer Member (item,from)  
em outra thread neste ponto?

Como resolver?

- Colocar um lock global?
- Esta solução requer que todas as operações atômicas da lista sejam envoltas pelo novo lock

# Composição de código com locks

```
public boolean Move(List from, List to, int item)
{
    from.lock(); to.lock();
    from.Delete(item);
    to.Insert(item);
    from.unlock(); to.unlock();
}
```

- Uma solução alternativa seria quebrar o encapsulamento e expor a implementação da lista (quais locks foram usados)
- Cada lista expõe seu lock global
- Esta solução funciona?

# Composição de código com locks

## Thread 1

Move (customer\_list, debtors\_list, USER1);

from.lock();

to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

## Thread 2

Move (debtors\_list, customer\_list, USER2);

from.lock();

to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

# Composição de código com locks

Thread 1

Move (customer\_list, debtors\_list, USER1);

→ `from.lock();` Runs slower!

`to.lock();`

`from.Delete(USER1);`

`to.Insert(USER1);`

`from.unlock();`

`to.unlock();`

Thread 2

Move (debtors\_list customer\_list, USER2);

→ `from.lock();` Runs faster!

`to.lock();`

`from.Delete(USER1);`

`to.Insert(USER1);`

`from.unlock();`

`to.unlock();`

# Composição de código com locks

## Thread 1

Move (customer\_list, debtors\_list, USER1);

→ from.lock();

to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

## Thread 2

Move (debtors\_list, customer\_list, USER2);

from.lock();

→ to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

blocked

# Composição de código com locks

## Thread 1

Move (customer\_list, debtors\_list, USER1);

from.lock();

→ to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

## Thread 2

Move (debtors\_list, customer\_list, USER2);

from.lock();

→ to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

blocked





# Deadlock

## Thread 1

Move (customer\_list, debtors\_list, USER1);

from.lock();

to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

## Thread 2

Move (debtors\_list, customer\_list, USER2);

from.lock();

to.lock();

from.Delete(USER1);

to.Insert(USER1);

from.unlock();

to.unlock();

deadlock



# Explique este agora?

Thread 1

```
TestInsert (list_X, list_Y, item);  
  Repeat for all list members  
    if ( list_X.Member (item) )  
      list_Y.Insert (item);
```

Thread 2

```
TestInsert (list_Y, list_X, item);  
  Repeat for all list members  
    if ( list_Y.Member (item) )  
      list_X.Insert (item);
```

# Comentários finais (1)

- Uma thread em sistemas de memória compartilhada é análogo a um processo em sistemas de memória distribuída.
- Entretanto, uma thread é tipicamente mais “leve” do que um processo completo .
- Em programas Pthreads, todas as threads têm acesso a variáveis globais, enquanto variáveis locais usualmente são privadas à thread executando a função.

# Comentários finais (2)

- Condição de corrida ocorre quando
  - Múltiplas threads tentam acessar um recurso compartilhado, como uma variável, e
  - Pelo menos um dos acessos é uma atualização
- A corrida pode resultar em erro ou não.

# Concluding Remarks (3)

- Um **semáforo** é uma terceira maneira de acessar conflitos em seções .
- Ele é operado através de um ponteiro sem sinal e das funções: `sem_wait` and `sem_post`.
- Semáforos são mais poderosos que mutexes uma vez que eles podem ser inicializados para qualquer valor não negativo.

# Comentários finais (4)

- Uma **barrier** é um ponto em um programa no qual a thread bloqueia até que todas as threads tenha alcançado-a.
- Uma **read-write lock** é usada quando é seguro para múltiplas threads ler uma estrutura de dados, mas se uma thread precisa modificar ou escrever nela, então a somente aquela thread pode acessá-la.

# Comentários finais (5)

- Algumas funções em C armazenam dados entre chamadas ao declarar variáveis estáticas, causando erros quando múltiplas threads chamam a função.
- Este tipo de função não é **thread-safe**.
- Estudamos um pouco de composição de locks.