# 用Pthreads进行共享内存编程

任课教师：吴迪

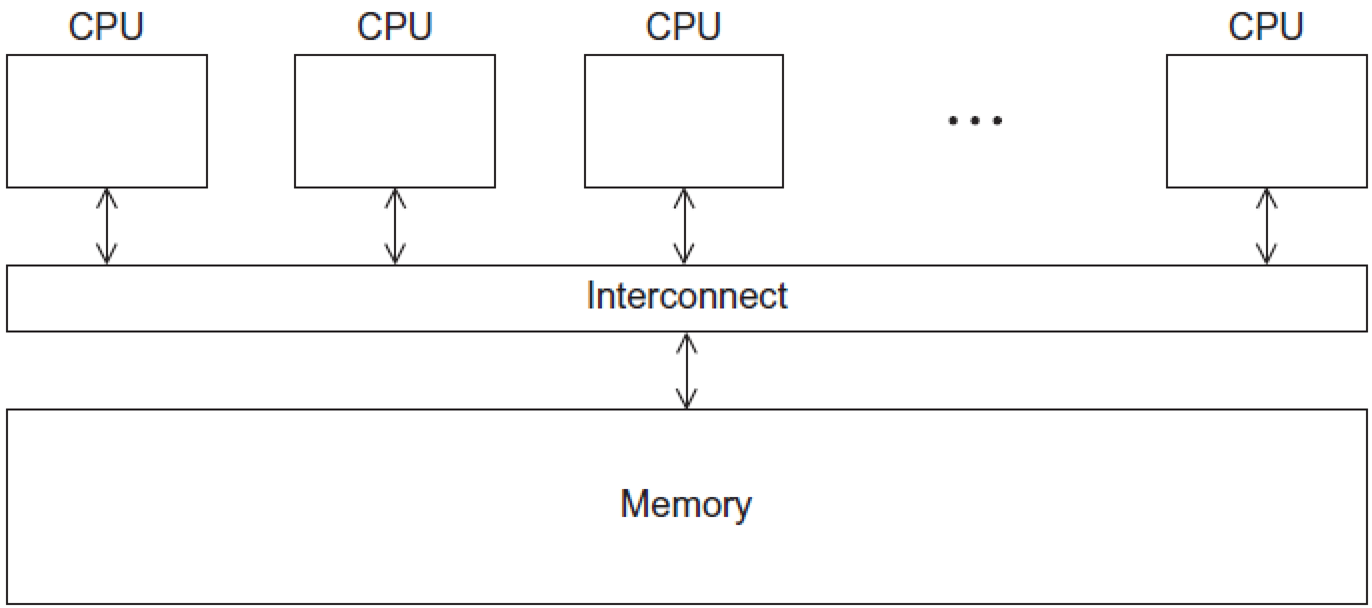
# 课程内容

- 参考资料：
  - 并行程序设计导论，Peter S Pacheco，机械工业出版社， 2016
  - Chapter 4 Shared-Memory Programming With Pthreads

# Roadmap

- Problems programming shared memory systems.
- Controlling access to a critical section.
- Thread synchronization.
- Programming with POSIX threads.
- Mutexes.
- Producer-consumer synchronization and semaphores.
- Barriers and condition variables.
- Read-write locks.
- Thread safety.

# A Shared Memory System



# Processes and Threads

- A process is an instance of a running (or suspended) program.
- Threads are analogous to a “light-weight” process.
- In a shared memory program, a single process may have multiple threads of control.

# POSIX® Threads

- Portable Operating System Interface (POSIX)
- Also known as Pthreads.
- A standard for Unix-like operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.

# Caveat

- The Pthreads API is only available on POSIXR systems — Linux, MacOS X, Solaris, HPUX, …

# Hello World! (1)

declares the various Pthreads functions, constants, types, etc.

```
#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));
```

# 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 pth_hello pth_hello . c −lpthread

link in the Pthreads library

# Running a Pthreads program

. /pth_hello <number of threads>

. /pth_hello 1

Hello from the main thread

Hello from thread 0 of 1

. /pth_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

# Global variables

- Can introduce subtle and confusing bugs!

- Limit use of global variables to situations in which they're really needed.
  - Shared variables.

# Starting the Threads

- Processes in MPI are usually started by a script.

- In Pthreads the threads are started by the program executable.

# Starting the Threads

pthread.h

pthread_t

One object for each 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 */ ) ;
```

# pthread_t objects

- Opaque
- The actual data that they store is system-specific.
- Their data members aren't directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information
  - to uniquely identify the thread with which it's associated.

# A closer look (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 */ ) ;
```

We won’t be using, so we just pass NULL.

Allocate <u>before</u> calling.

# A closer look (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 */ ) ;
```

Pointer to the argument that should be passed to the function *start_routine*.

The function that the thread is to run.

# Function started by pthread_create

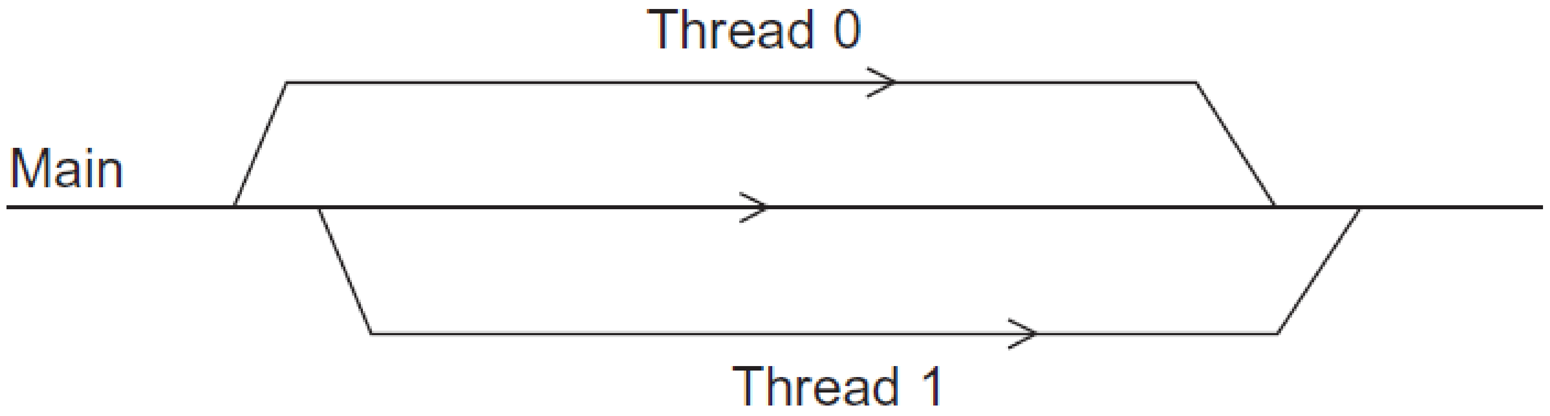
- Prototype:
  void* thread_function ( void* args_p ) ;

- Void* can be cast to any pointer type in C.

- So args_p can point to a list containing one or more values needed by thread_function.

- Similarly, the return value of thread_function can point to a list of one or more values.

# Running the Threads




Main thread forks and joins two threads.

# Stopping the Threads

- We call the function pthread_join once for each thread.

- A single call to pthread_join will wait for the thread associated with the pthread_t object to complete.

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

# Matrix-Vector Multiplication in pthreads

# Serial pseudo-code

```
/* For each row of A */
for (i = 0; i < m; i++) {
   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];
}
```

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

# Using 3 Pthreads

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

thread 0

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

general case

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

# Pthreads matrix-vector multiplication

```
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 */
```

# Critical sections

# Estimating π

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

```
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;
```

# Using a dual core processor

| | $n$ | | | |
|---|---|---|---|---|
| | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
| $\pi$ | 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 that as we increase n, the estimate with one thread gets better and better.

# A thread function for computing π

```
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 */
```

# Possible race condition

| 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()
x=x+y

x=1+2?
x=2?

# Busy-Waiting

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.

- Beware of optimizing compilers, though!

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

flag initialized to 0 by main thread

# Pthreads global sum with 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 */
```

# Global sum function with critical section after loop (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;
```

# Global sum function with critical section after loop (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 */
```

# Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.

- **Mutex** (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.

# Mutexes


- Used to guarantee that one thread "excludes" all other threads while it executes the critical section.

- The Pthreads standard includes a special type for mutexes: pthread_mutex_t.

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

# Mutexes

- In order to gain access to a critical section a thread calls

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

- When a thread is finished executing the code in a critical section, it should call

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

# Mutexes

- When a Pthreads program finishes using a mutex, it should call

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

# Global sum function that uses a 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;
```

# Global sum function that uses a 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 |

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

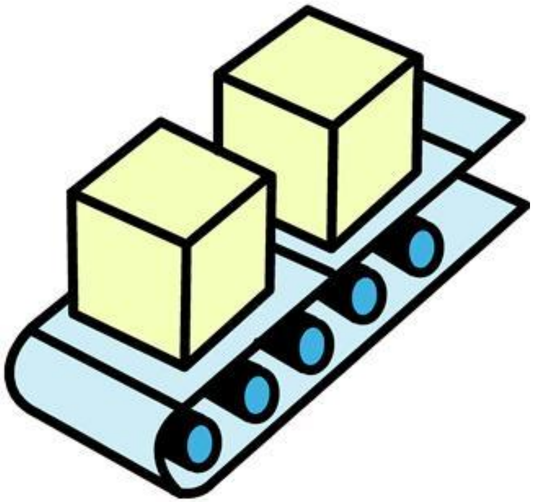
⇨ # of threads > # of cores

Run-times (in seconds) of π programs using n = $10^8$ terms on a system with two four-core processors.

# 5 threads   2 cores

| 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 |

Possible sequence of events with busy-waiting and more threads than cores.



# Producer-Consumer Synchronization and Semaphores

# Issues

- Busy-waiting enforces the order threads access a critical section.

- Using mutexes, the order is left to chance and the system.

- There are applications where we need to control the order threads access the critical section.

# Problems with a mutex solution

```
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 */
```

**Problem**: matrix multiplication is not commutative

For example, $A * B * C \neq C * B * A$

# A first attempt at sending messages using 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;
                                        while (messages[my_rank] == NULL);
   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 */
```

**Problem**: If running 2+ threads on a dual-core system, some of the messages are never received.

# Syntax of the various semaphore functions

Semaphores are not part of Pthreads;

you need to add this.

```
#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 */);
```

int sem_init(sem_t *sem,int pshared,unsigned int value);

value表示初始化信号的值

pshared为0，表示信号在当前进程的多个线程之间共享

**int sem_wait(sem_t *sem);**

- sem_wait可以用来阻塞当前线程，直到信号量的值大于0，解除阻塞。
- 解除阻塞后，sem的值-1，表示公共资源被执行减少了

**int sem_post(sem_t *sem);**

- sem_post用于增加信号量的值，信号量+1
- 当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程不再阻塞，选择机制由线程的调度策略决定。

# Using Semaphores to send msgs

semaphores[thread_count]

```
/* semaphores are initialized to 0 (locked) */
void* Send_msg(void* rank) {
        long my_rank = (long) rank;
        long dest = (my_rank + 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;
        sem_post(&semaphores[dest]);   /*unlock the semaphore of dest*/

        /*wait for our semaphore to be unlocked*/
        sem_wait(&semaphores[my_rank]);
        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
        return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  long thread;
  pthread_t* thread_handles;
  messages = malloc(MSG_MAX*sizeof(char));

  if(argc != 1) Usage(argv[0]);

  thread_count = 8;
  semaphores = malloc(thread_count*sizeof(sem_t));
  sem_init(semaphores, 0, 0);

  /* allocate array for threads */
  thread_handles = malloc(thread_count*sizeof(pthread_t));

  /* start threads */
  for(thread = 0; thread < thread_count; thread++) {
    pthread_create(&thread_handles[thread], NULL, Send_msg,
             (void*) thread);
  }

  /* wait for threads to complete */
  for(thread = 0; thread < thread_count; thread++) {
    pthread_join(thread_handles[thread], NULL);
  }
```

```
ehpc@f68c2d045077:~/data$ ./4_8_pth_msg_smp
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 5 > Hello to 5 from 4
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 0 > Hello to 0 from 7
ehpc@f68c2d045077:~/data$ ./4_8_pth_msg_smp
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 5 > Hello to 5 from 4
Thread 7 > Hello to 7 from 6
Thread 6 > Hello to 6 from 5
Thread 0 > Hello to 0 from 7
```

输出不一定严格有序，但是消息发送的顺序是有序的

# Barriers and Condition Variables

# Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a **barrier**.

- No thread can cross the barrier until all the threads have reached it.

# Using barriers to time the slowest thread

```
/* 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;
```

# Using barriers for debugging

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

# Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

# Busy-waiting and a 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);
   . . .
}
```

We need one counter variable for each instance of the barrier, otherwise problems are likely to occur.

# Implementing a barrier with semaphores

```
/* 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);
   }
   . . .
```

## int sem_wait(sem_t *sem);

- sem_wait可以用来阻塞当前线程，直到信号量的值大于0，解除阻塞。
- 解除阻塞后，sem的值-1，表示公共资源被执行减少了

信号量和互斥锁(mutex)的区别：

- 互斥锁只允许一个线程进入临界区，
- 而信号量允许多个线程同时进入临界区。

先判断，如果大于0，解除阻塞，减去1；

如果等于0，保持阻塞

# Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to “wake up.”
- A condition variable is always associated with a mutex.

# Condition Variables

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

# Implementing a barrier with condition variables

```
/* 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);
    . . .
}
```

pthread_mutex_unlock(&mutex);
wait_on_signal(&cond_var);
pthread_mutex_lock(&mutex)

**通常的应用场景下，当前线程执行pthread_cond_wait时，处于临界区访问共享资源，存在一个mutex与该临界区相关联，这是理解pthread_cond_wait带有mutex参数的关键**

- 当前线程执行pthread_cond_wait前，已经获得了和临界区相关联的mutex；**执行pthread_cond_wait会阻塞，但是在进入阻塞状态前，必须释放已经获得的mutex，让其它线程能够进入临界区**
- 当前线程执行pthread_cond_wait后，阻塞等待的条件满足，条件满足时会被唤醒；**被唤醒后，仍然处于临界区，因此被唤醒后必须再次获得和临界区相关联的mutex**
- 综上，调用pthread_cond_wait时，线程总是位于某个临界区，该临界区与mutex相关，pthread_cond_wait需要带有一个参数mutex，用于释放和再次获取mutex。

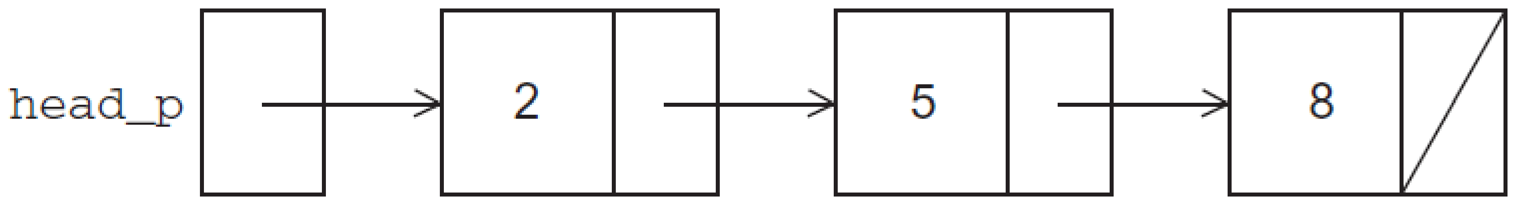
# Read-Write Locks

# Controlling access to a large, shared data structure

- Let's look at an example.

- Suppose the shared data structure is a sorted linked list of ints, and the operations of interest are Member, Insert, and Delete.

# Linked Lists

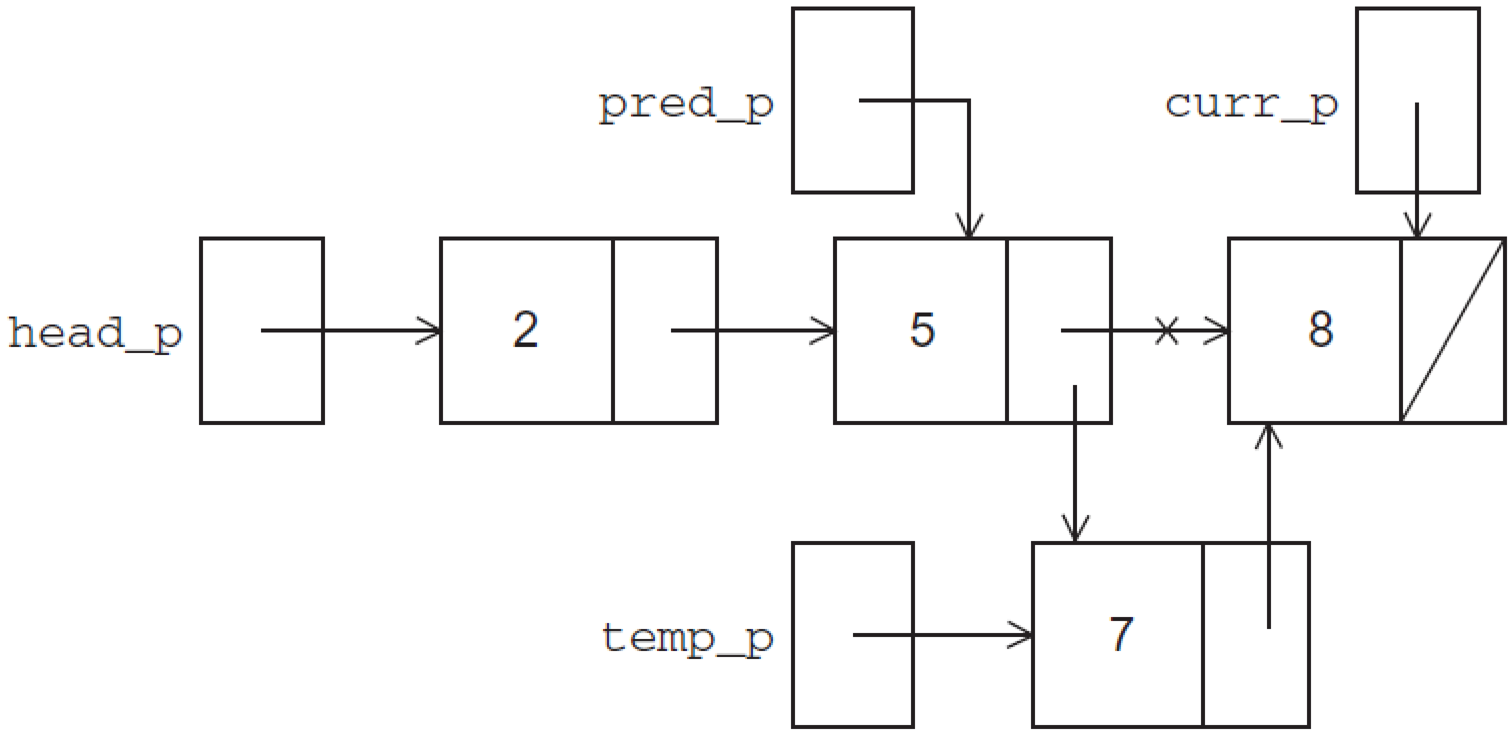



```
struct list_node_s {
    int data;
    struct list_node_s* next;
}
```

# Linked List Membership

```
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 */
```

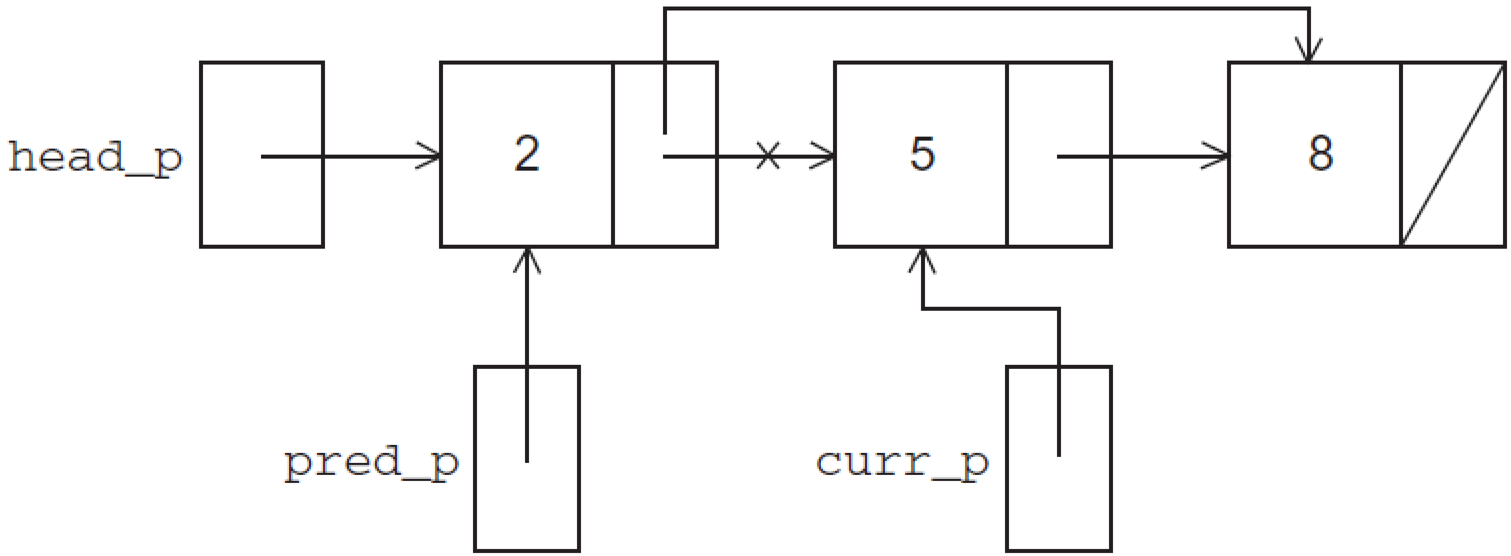
# Inserting a new node into a list



# Inserting a new node into a list

```
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 */
```

# Deleting a node from a linked list



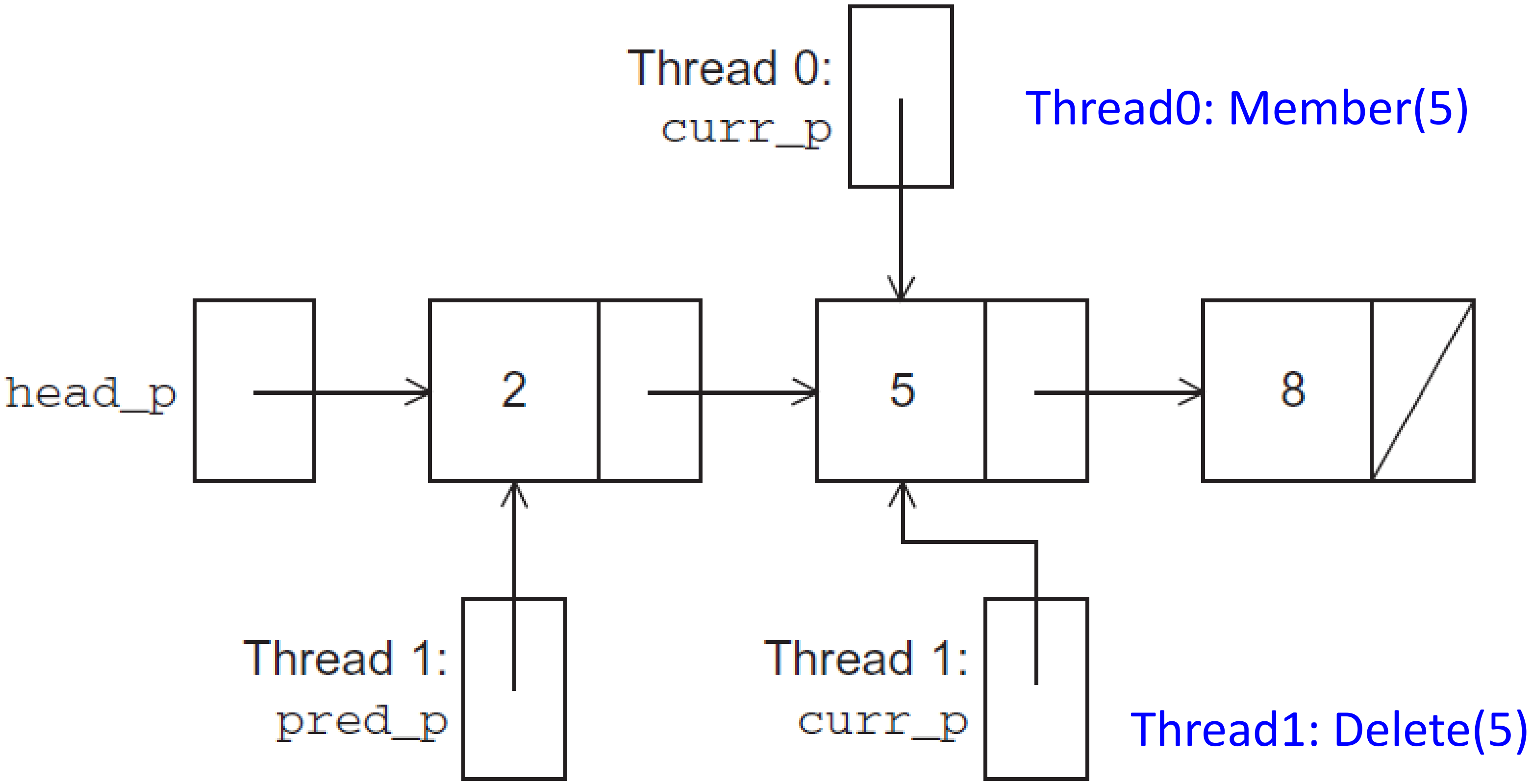
# Deleting a node from a linked list

```
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 */
```

# A Multi-Threaded Linked List

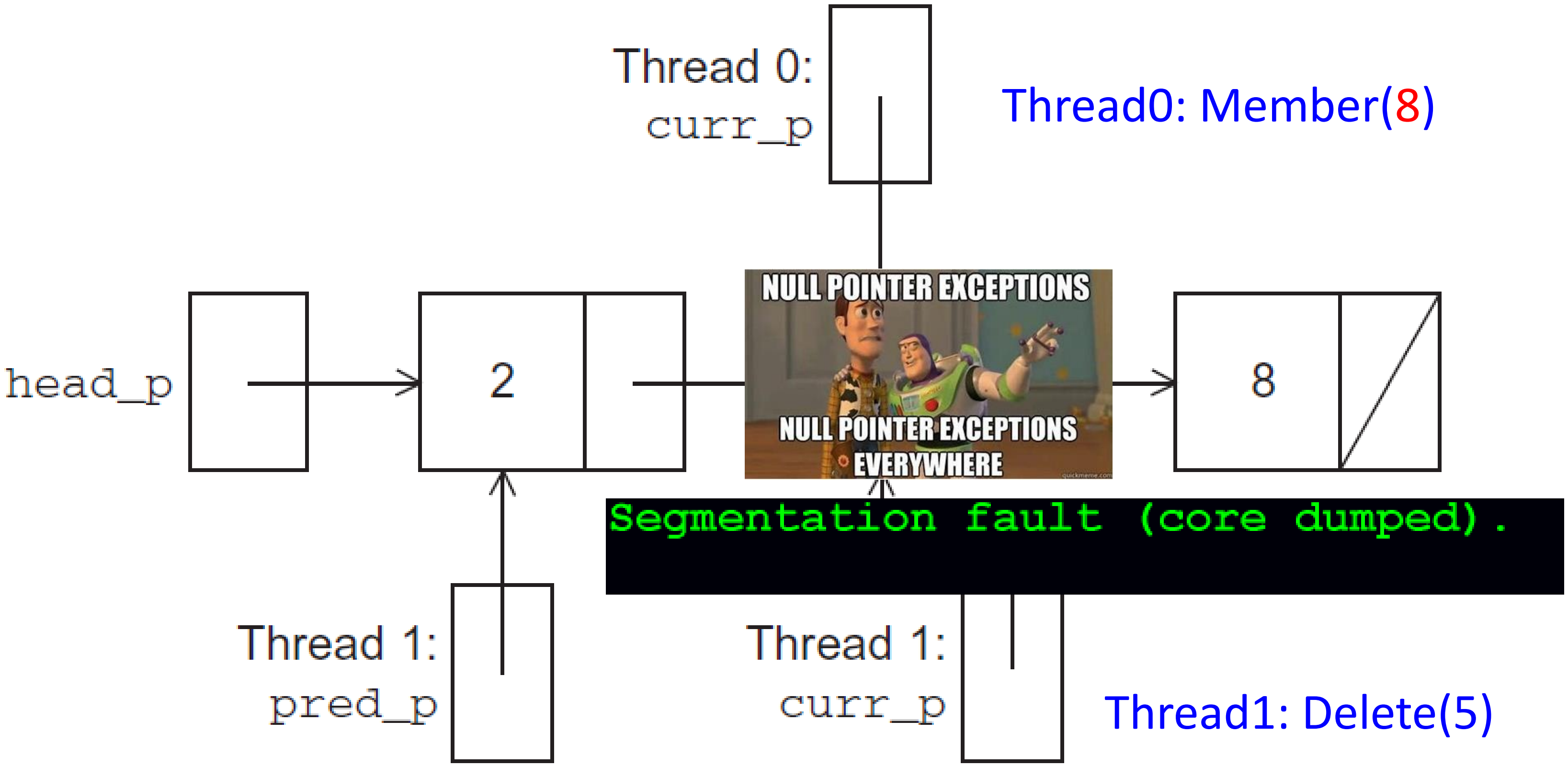
- Let’s try to use these functions in a Pthreads program.
- In order to share access to the list, we can define head_p to be a global variable.
- This will simplify the function headers for Member, Insert, and Delete
  - since we won’t need to pass in either head_p or a pointer to head_p
  - we’ll only need to pass in the value of interest.

# Simultaneous access by two threads




**Problem**: Element 5 may be deleted even before thread 0 returns.

# Simultaneous access by two threads




**Problem**: thread1 may free the memory used for the node storing 5 before thread0 can advance to the node storing 8.

# Solution #1

- An obvious solution is to simply lock the list any time that a thread attempts to access it.
- A call to each of the three functions can be protected by a mutex.

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

In place of calling Member(value).

# Issues

- We're serializing access to the list.
- If the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete, then this may be the best solution
  - since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

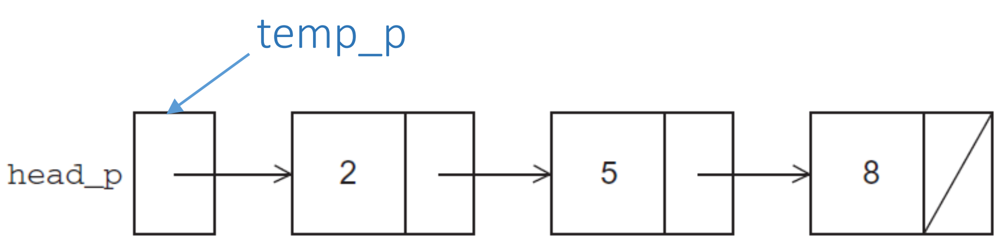
# Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A "finer-grained" approach.

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

# Implementation of Member with one mutex per list node (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;
   }
```



# Implementation of Member with one mutex per list node (2)

```
   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 */
```

# Issues

- This is much more complex than the original Member function.
- It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked.
- The addition of a mutex field to each node will substantially increase the amount of storage needed for the list.

# Pthreads Read-Write Locks

- Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
- The first solution only allows one thread to access the entire list at any instant.
- The second only allows one thread to access any given node at any instant.

# Pthreads Read-Write Locks

- A read-write lock is somewhat like a mutex except that it provides two lock functions.

- The first lock function locks the read-write lock for reading, while the second locks it for writing.

# Pthreads Read-Write Locks

- So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.
- Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.

# Pthreads Read-Write Locks

- If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

# Protecting our linked list functions

```
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);
```

# Linked List Performance

| 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

If there is very few Inserts/Deletes, the RW locks do a very good job of allowing concurrent access to the list.

# Linked List Performance

| 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

If there are a relatively large # of Inserts/Deletes, there is very little difference between RW lock and single mutex.

# Caches, Cache-Coherence, and False Sharing

- Recall that chip designers have added blocks of relatively fast memory to processors called cache memory.

- The use of cache memory can have a huge impact on shared-memory.

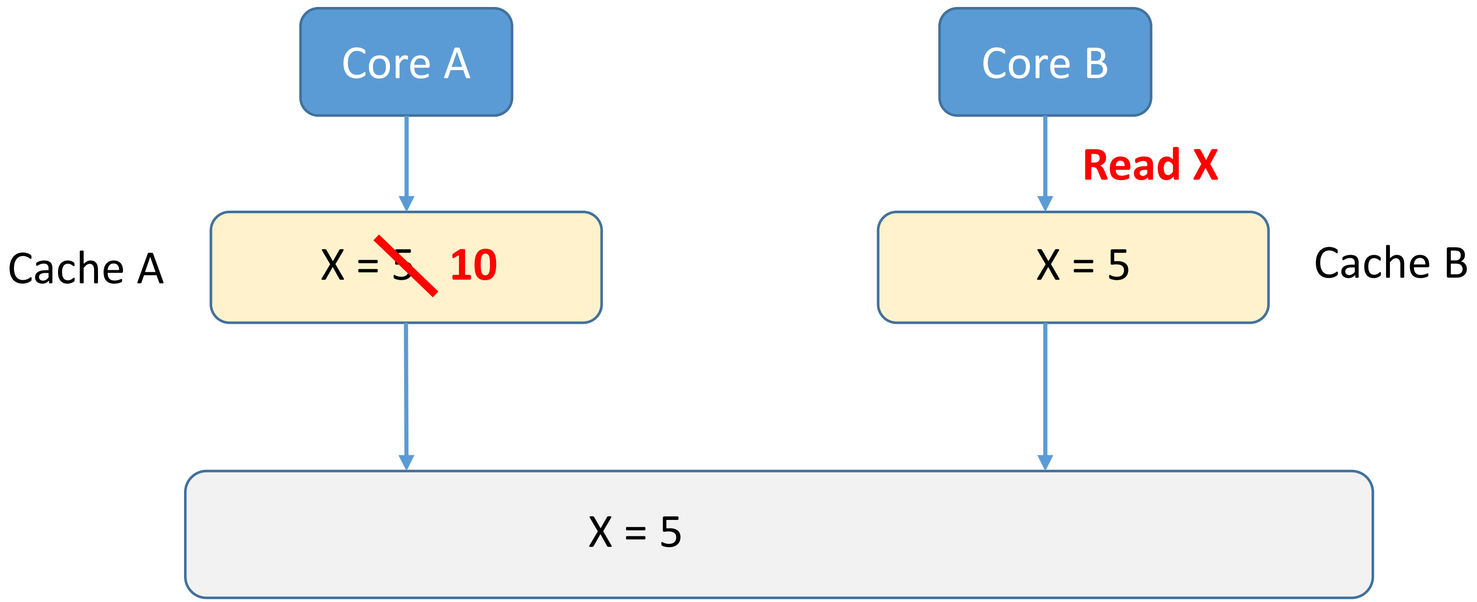
- A write-miss occurs when a core tries to update a variable that’s not in cache, and it has to access main memory.

# Cache coherence（缓存一致性）

- **Cache coherence（缓存一致性）问题是在多处理器系统中出现的一个问题**
  - 当多个核心同时访问同一数据时，就会产生缓存一致性问题。
- **缓存一致性问题的本质是由于每个核心的缓存是独立的**
  - 如果一个核心修改了某些数据，并将其写入其自己的缓存行，那么其他核心的缓存行可能仍然包含旧的数据。
  - 如果这些其他核心继续基于这个旧数据执行操作，就会导致错误的结果。

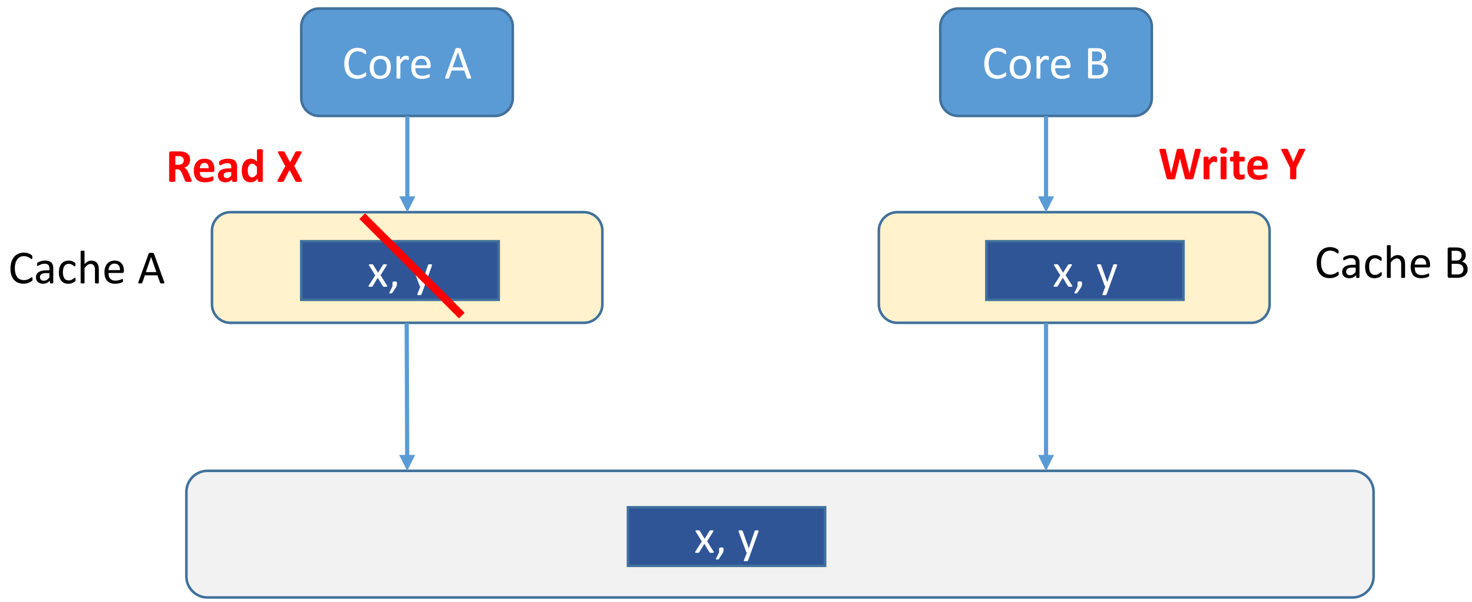
# Cache coherence 例子

- 假设有一个多核处理器的系统，其中有两个核心，A和B，它们共享一个内存块。这个内存块包含一个变量 x。



# False Sharing （伪共享）

- **False-sharing是一种在多核处理器系统中出现的性能问题**
  - 它发生在多个线程访问看似不同的变量，但实际上这些变量位于同一个缓存行时。
  - 在多核处理器中，每个核心都有自己的缓存，但是，当一个线程更新了一个变量，它可能会导致整个缓存行被刷新到主内存，这会影响到访问该缓存行中其他变量的其他线程



# False Sharing Example

```java
public class FalseSharingExample {
    private final int[] array = new int[1024];

    public void thread1() {
        for (int i = 0; i < 1024; i++) {
            array[i] = i;
        }
    }

    public void thread2() {
        for (int i = 0; i < 1024; i++) {
            array[i] = array[i] + 1;
        }
    }
}
```

- 在上面的代码中，两个线程 thread1 和 thread2 都在更新同一个数组的不同元素。
- 尽管它们看似在更新不同的变量，但由于数组元素位于同一个缓存行，线程 thread2 的更新可能会导致线程 thread1 的缓存行被刷新到主内存，这会导致性能下降。

# Pthreads matrix-vector multiplication

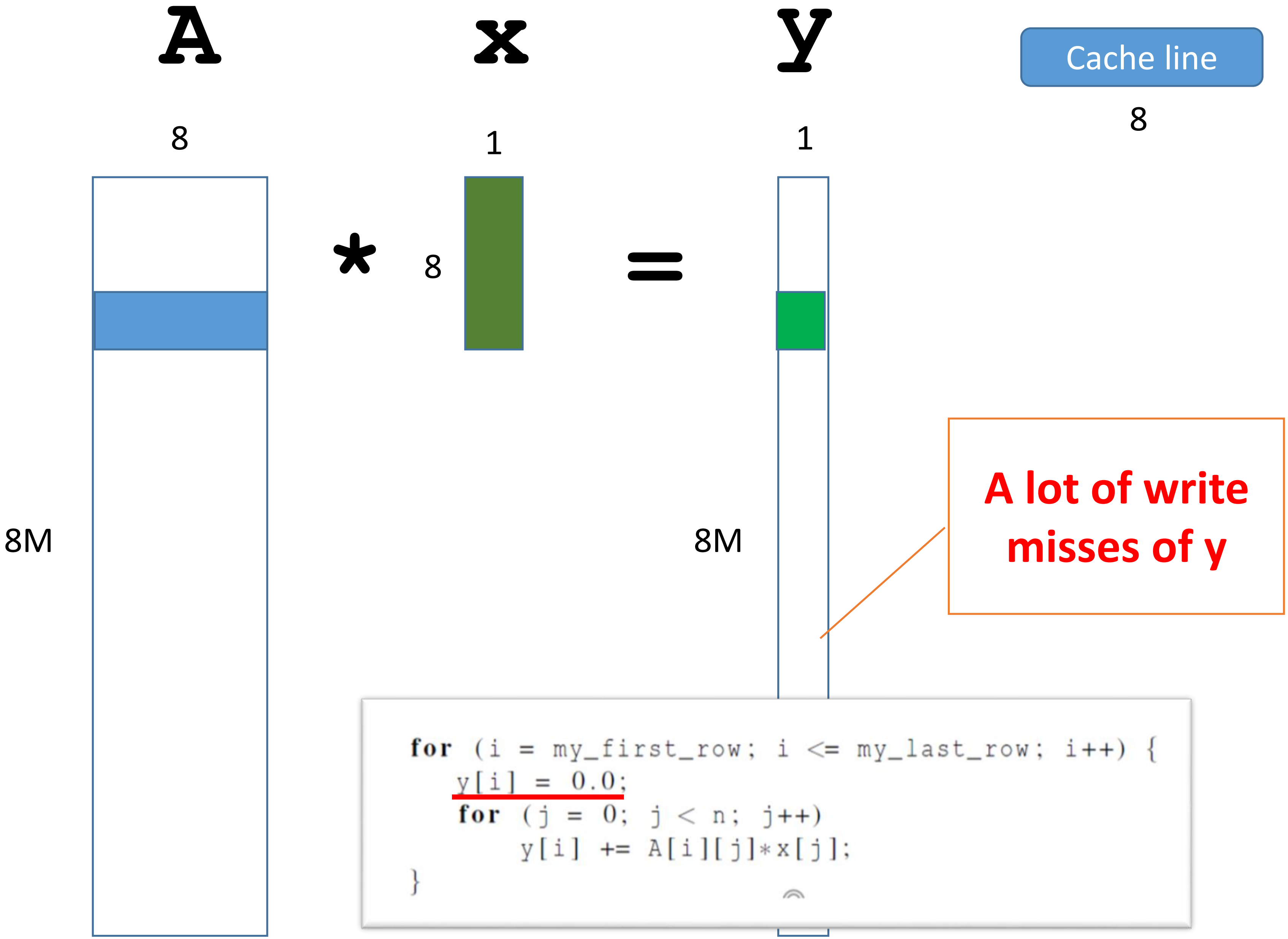
```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   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 */
```

Q: 为什么8M*8（或8*8M）的矩阵计算时间要大于8K*8K的时间?

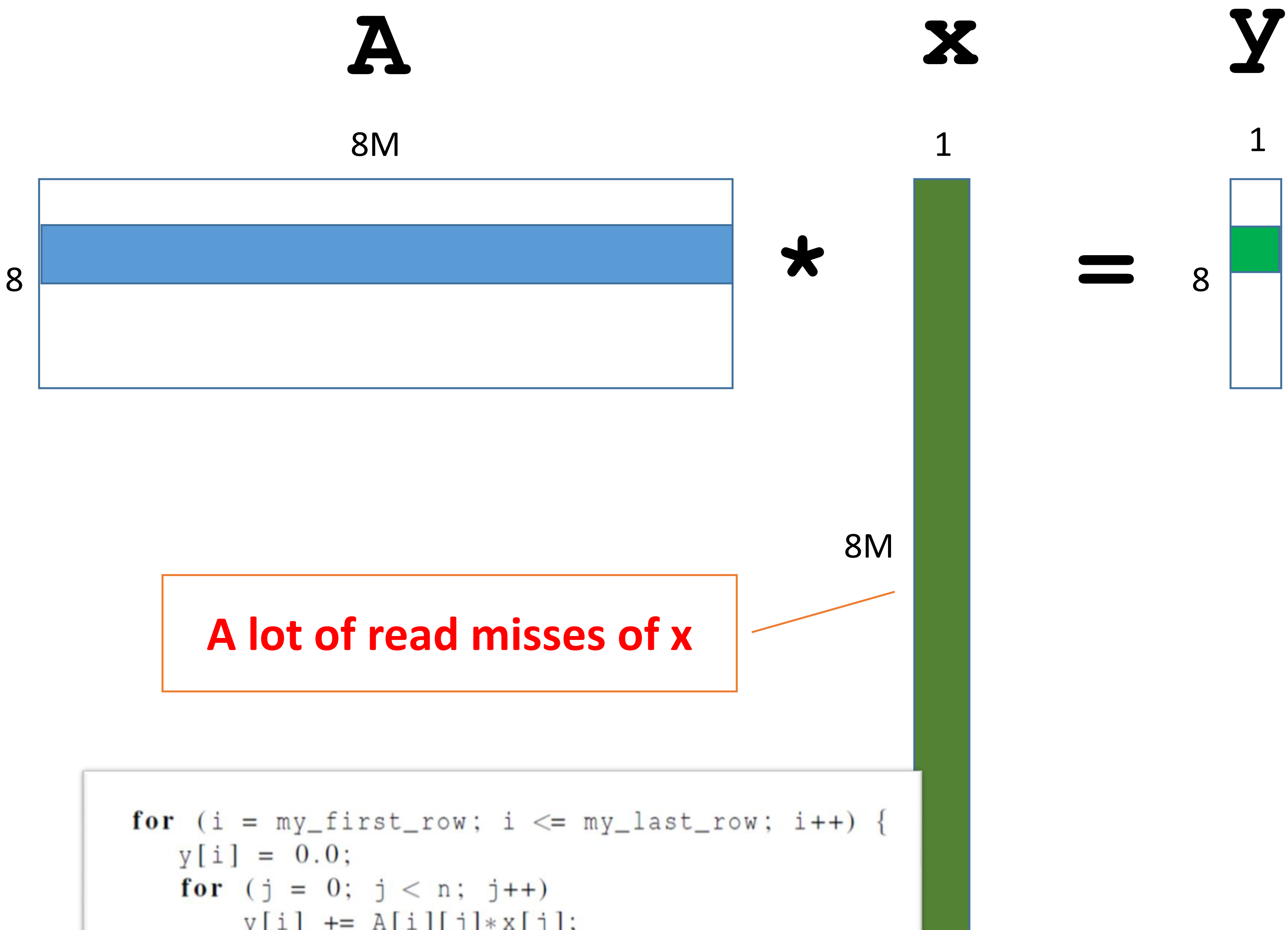
```
y = A*x
```

| Threads | Matrix Dimension | | | | | |
|---|---|---|---|---|---|---|
| | $8,000,000 \times 8$ | | $8000 \times 8000$ | | $8 \times 8,000,000$ | |
| | 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 |

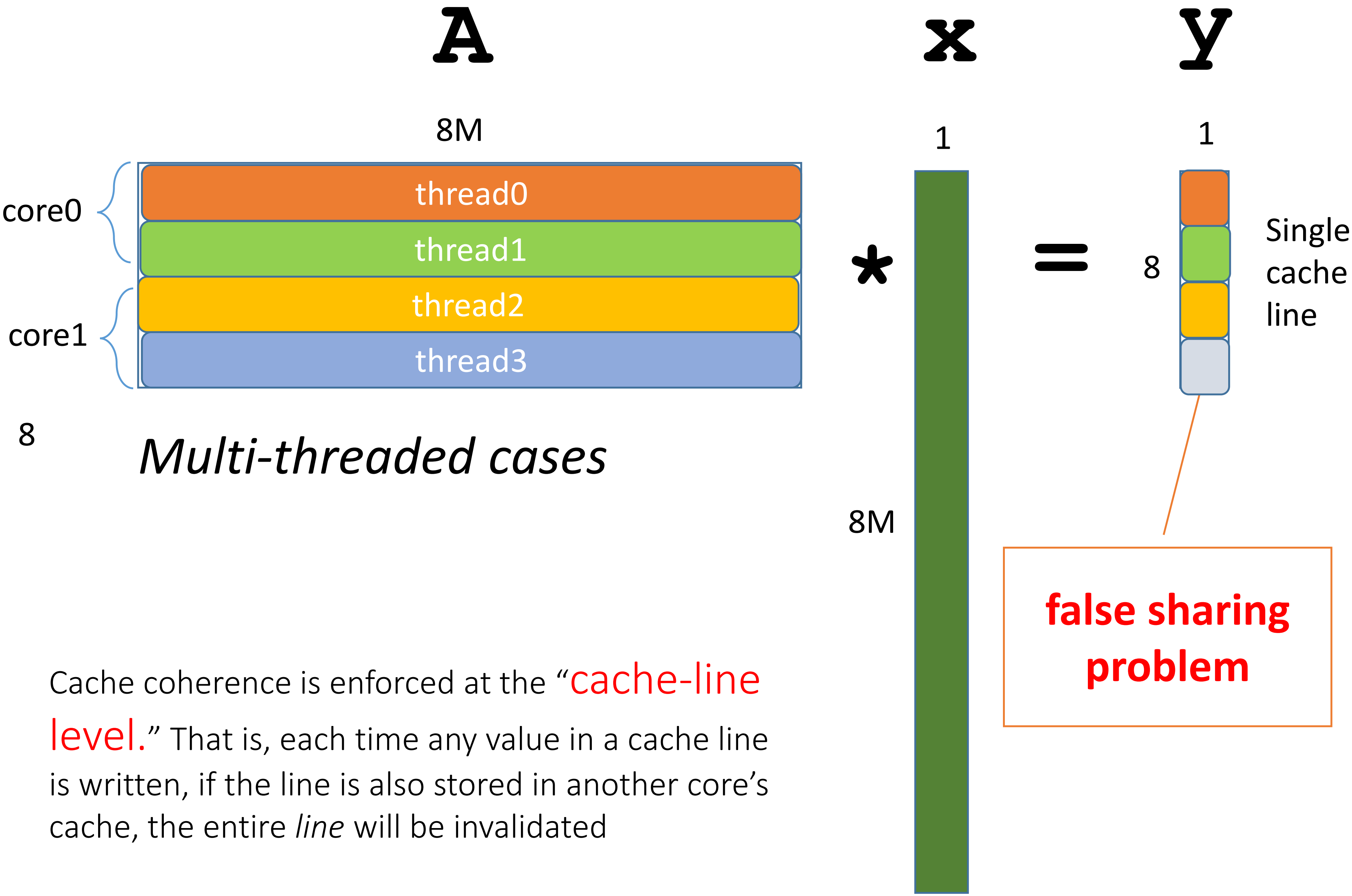
- With 1 thread, 8M*8 requires 14% more time than 8K*8K, and 8*8M requires **28% more time than** 8K*8K.
- The above differences are partially attributed to **cache performance**.
- Efficiency of multi-thread version is even **worse** for 8M*8 and 8*8M.




```
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];
}
```




```
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];
}
```




*Multi-threaded cases*

Cache coherence is enforced at the "cache-line level." That is, each time any value in a cache line is written, if the line is also stored in another core's cache, the entire *line* will be invalidated

# Thread-Safety

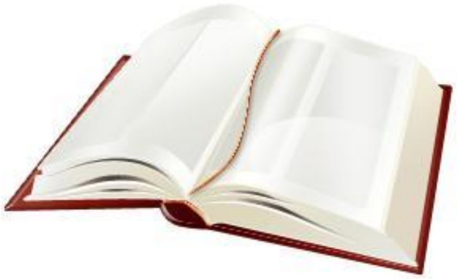
# Thread-Safety

- A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

# Example

- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.

- Suppose we want to use multiple threads to "tokenize" a file that consists of ordinary English text.



# Simple approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.

- The first line goes to thread 0,
- the second goes to thread 1,
- . . . ,
- the t-th goes to thread t,
- the t +1st goes to thread 0,
- etc.

# Simple approach

- We can serialize access to the lines of input using semaphores.

- After a thread has read a single line of input, it can tokenize the line using the strtok function.

# The strtok function

- The first time it's called the string argument should be the text to be tokenized.
  - Our line of input.

- For subsequent calls, the first argument should be NULL.

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

# The strtok function

- The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens taken from the cached copy.

```
my_string = strtok(my_line, "\t\n");
……
my_string = strtok(NULL, "\t\n");
```

# 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);
```

Thread 0's semaphore is initialized to 1, and others are initialized to 0

# 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 */
```

# Running with one thread

- It correctly tokenizes the input stream.

Pease porridge hot.

Pease porridge cold.

Pease porridge in the pot

Nine days old.

# Running with two 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!

Missing...

# What happened?

- strtok caches the input line by declaring a variable to have static storage class.
- This causes the value stored in this variable to persist from one call to the next.
- Unfortunately for us, this cached string is shared, not private.

# What happened?

- Thus, thread 0's call to strtok with the third line of the input has apparently overwritten the contents of thread 1's call with the second line.

- So the strtok function is not thread-safe.

- If multiple threads call it simultaneously, the output may not be correct.

# Other unsafe C library functions

- Regrettably, it's not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.
- The time conversion function localtime in time.h.

# “re-entrant” (thread safe) functions

- In some cases, the C standard specifies an alternate, thread-safe, version of a function.

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

saveptr_p keeps track of where the func is in the input string.

```
my_string = strtok(my_line, “\t\n”);
……
my_string = strtok(NULL, “\t\n”);
```

```
my_string = strtok_r(my_line, “\t\n”, &saveptr);
……
my_string = strtok_r(NULL, “\t\n”, &saveptr);
```

# Concluding Remarks (1)

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
- However, a thread is often lighter-weight than a full-fledged process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.

# Concluding Remarks (2)

- When indeterminacy results from multiple threads attempting to access a shared resource, the accesses can result in an error, and we have a race condition.

# Concluding Remarks (3)

- A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time.

- So the execution of code in a critical section should, effectively, be executed as serial code.

# Concluding Remarks (4)

- Busy-waiting can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.

- It can be very wasteful of CPU cycles.

- It can also be unreliable if compiler optimization is turned on.

# Concluding Remarks (5)

- A mutex can be used to avoid conflicting access to critical sections as well.

- Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.

# Concluding Remarks (6)

- A semaphore is the third way to avoid conflicting access to critical sections.
- It is an unsigned int together with two operations: sem_wait and sem_post.
- Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.

# Concluding Remarks (7)

- A barrier is a point in a program at which the threads block until all of the threads have reached it.

- A read-write lock is used when it’s safe for multiple threads to simultaneously read a data structure,

- but if a thread needs to write to the data structure, then only that thread can access the data structure during the modification.

# Concluding Remarks (8)

- Some C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.

- This type of function is not thread-safe.