



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# 并行程序设计 with 算法

## Pthreads 程序设计

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

中山大学 计算机学院  
国家超级计算广州中心

- 概述

- Pthreads Hello World
  - 矩阵向量乘法

- 临界区与互斥量

- 生产者-消费者同步与信息量

- 路障与条件变量

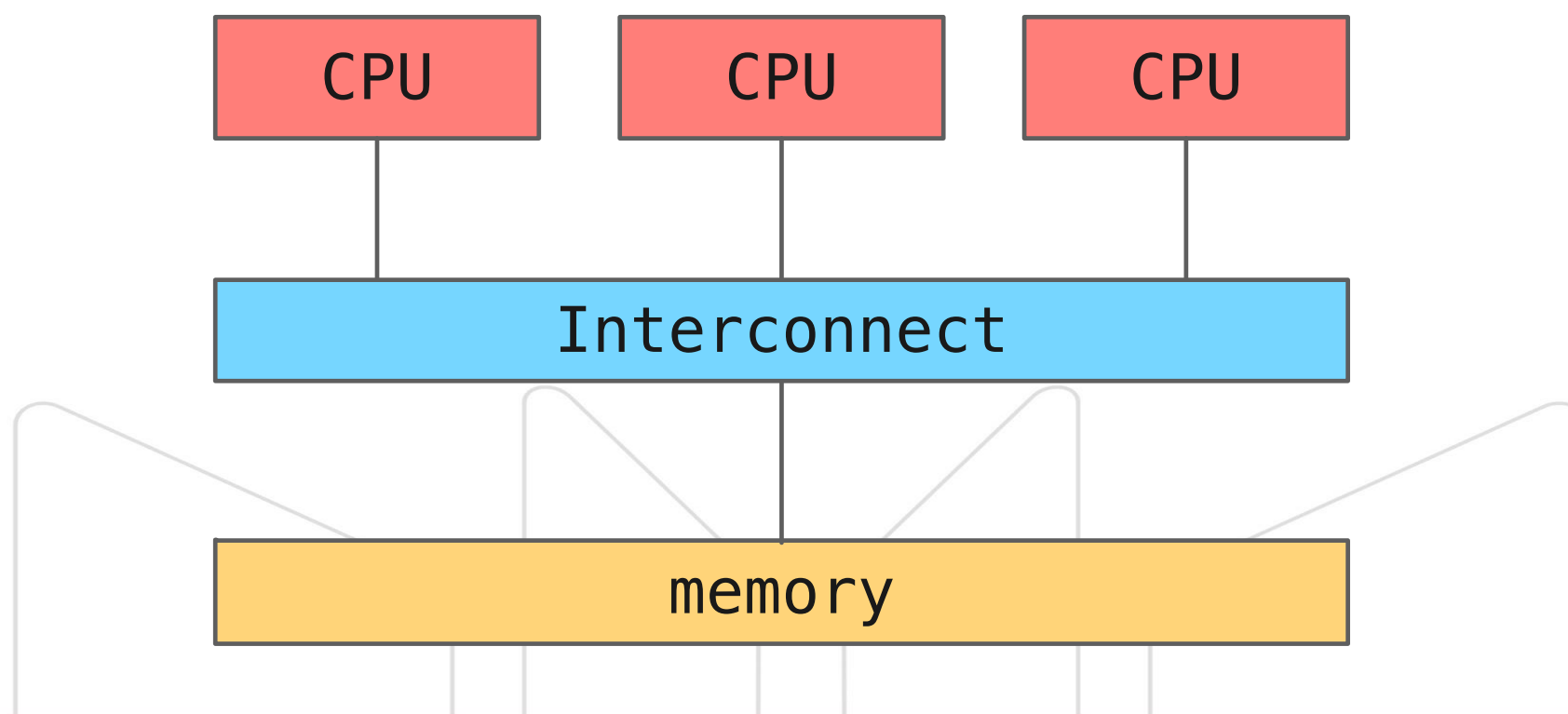
- 读写锁

- 缓存一致性与伪共享

- 线程安全

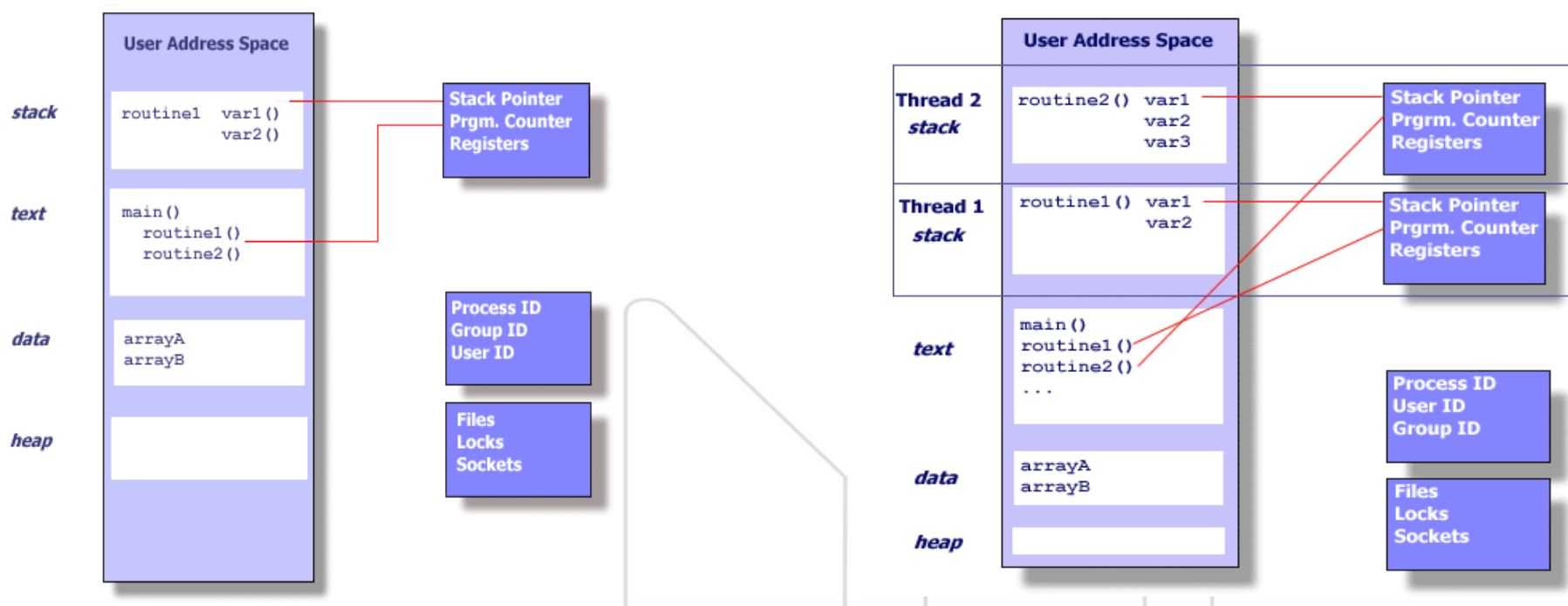
## 共享内存系统

- 如何创建多个线程，使其执行多个任务？
- 线程之间如何交换数据？如何同步？
- 如何保证获得确定性的结果（线程安全）？



## 进程与线程

- 进程：执行中的程序，资源分配的最小单位
  - 独立的栈、堆、数据段、代码段、资源描述符、进程状态等
- 线程：轻量级进程，系统调度的最小单位
  - 共享进程的部分信息
  - 独立的栈、程序计数器等

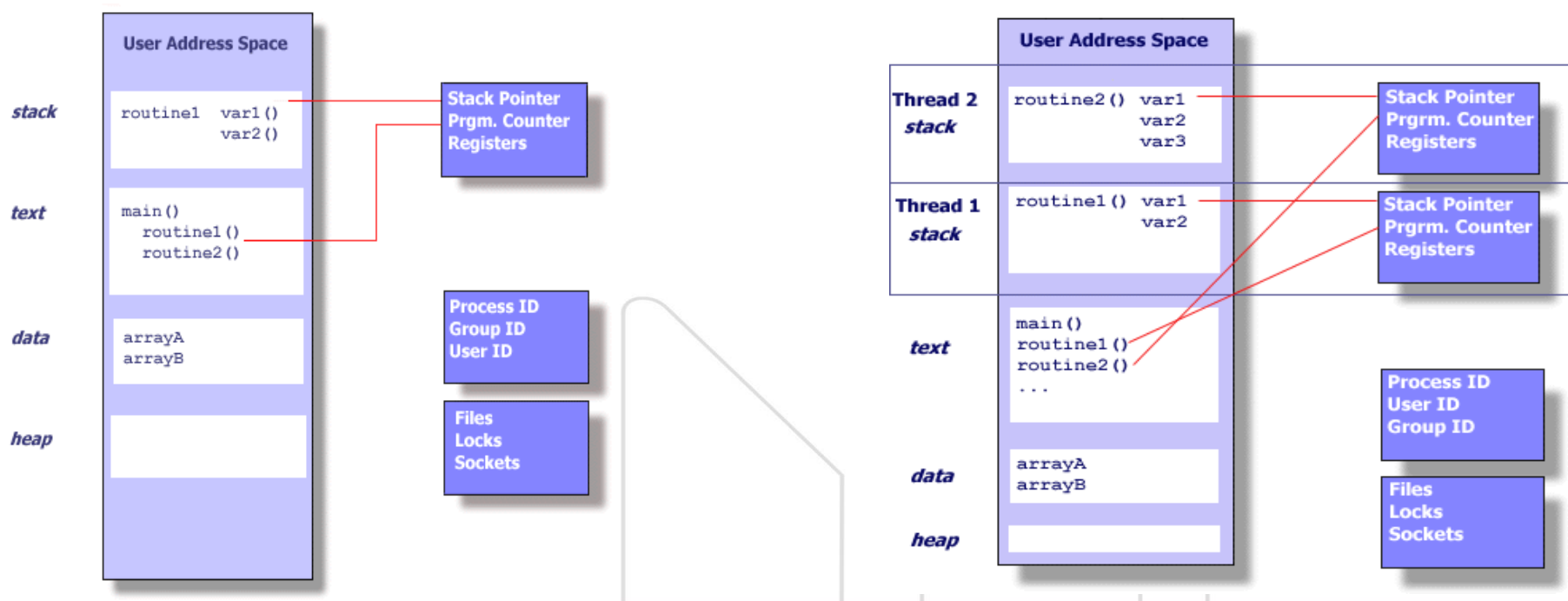


## 多线程的优点

- 轻量级：更低的创建、切换、销毁开销
- 资源共享：更方便数据（文件）共享、更高效的线程通信

## 多线程的缺点

- 容易出错：由于共享地址空间，数据竞争、死锁等问题更严重
- 可靠性差：线程错误可能影响其他线程，导致进程崩溃



## • Pthreads (POSIX线程库)

### – POSIX: Potable Operating System Interface for UNIX

- 定义UNIX操作系统接口的规范
- (类) UNIX操作系统: Linux, MacOS, Solaris, FreeBSD等

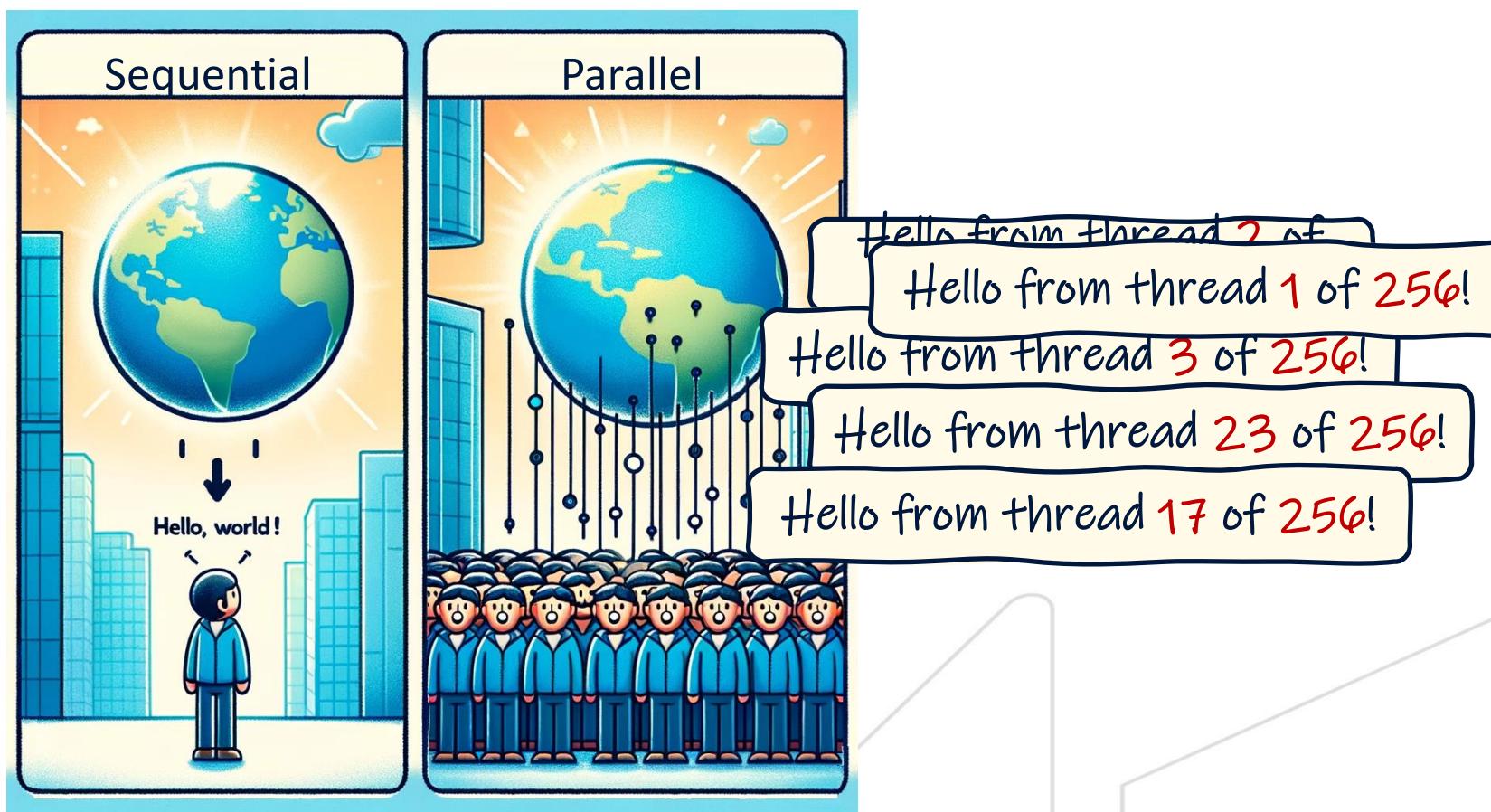
### – 非UNIX操作系统不支持Pthreads, 但通常有类似多线程库, 其基本概念、接口都与Pthreads相似

- 操作系统提供的多线程库: Windows Thread Library
- 编程语言提供的多线程库: Java threads, Python threads
- 第三方提供的类Pthreads库: Pthreads-win32
- 或使用MinGW (Minimalist GNU for Windows) 开发工具集



## 目标

- 每个线程输出 “Hello from thread (编号) of (总线程数)”



## ● 实现

```
#include <pthread.h> Pthreads头文件

/* Global variable: accessible to all threads */
int thread_count;          全局变量，在所有线程共享线程数量
                           （哪些数据应该用作全局变量？）
void *Hello(void* rank);   /* Thread function */

int main(int argc, char* argv[]) {
    thread_count = strtol(argv[1], NULL, 10);  读入线程数量，为线程对象分配空间
    pthread_t* thread_handles = malloc (thread_count*sizeof(pthread_t));

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

    printf("Hello from the main thread\n");    创建线程，指明其执行内容为Hello函数

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

    free(thread_handles);  线程执行完毕，使用汇聚（join）操作合并执行结果，
    return 0;              并释放内存
} /* main */
```



## ● 实现

```
#include <pthread.h>

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

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

int main(int argc, char* argv[]) {
    thread_count = strtol(argv[1], NULL, 10);
    pthread_t* thread_handles = malloc (thread_count*sizeof(pthread_t));

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

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

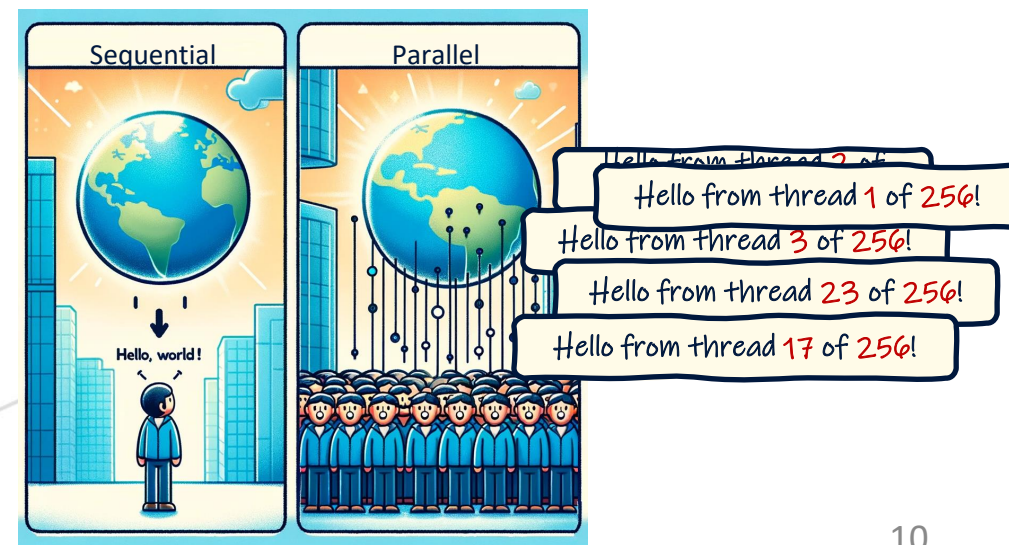
        printf("Hello from thread %ld of %d\n", my_rank, thread_count);

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

## 编译与执行

- 与普通C程序一样，只是使用了Pthreads库
- 编译：`gcc -g -Wall -o pthread_hello pthread_hello.c -lpthread`
  - `-g`, `-Wall`, `-o` 选项与此前MPI程序编译一致
  - `-lpthread`: 手动链接Pthreads库（链接路径下的libpthread.so文件）
- 执行：直接运行可执行文件（与MPI有何不同？）
  - `./pthread_hello <# threads>`
  - 参数由实现决定，非Pthreads要求
  - 输出：`./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
```



## 启动线程

- 每个线程对应一个 `pthread_t` 对象
  - 在创建线程前，必须为 `pthread_t` 对象分配好空间
  - `pthread_create` 不负责分配空间
- `pthread_t` 对象为 **不透明** 对象
  - 存储数据是系统绑定的 (system-specific)
  - 用户级代码无法直接访问里面的数据
  - Pthreads 标准保证对象中必须存有足够多的信息 (足以识别对应线程)

```
int pthread_create(  
pthread_t *thread, ----- 用于返回新创建线程  
const pthread_attr_t *attr, ----- 指定新线程属性  
void *(*start_routine)(void *), ----- 新线程要执行的函数  
void *arg); ----- 传递给要执行函数的参数
```

## 启动线程

- 新线程属性：分离状态，堆栈大小，调度策略，调度优先级等
  - 不指定新线程属性时，`attr`可以为NULL
- 新线程将要执行的函数以函数指针形式传入
  - 其参数为`void*`类型，通过最后一个参数传入
  - 其返回值类型同样为`void*`
  - `void*`：C中万用类型，可转换为任意指针类型
    - 参考MPI（讲义3）中的手动数据打包

```
int pthread_create(  
    pthread_t *thread, ----- 用于返回新创建线程  
    const pthread_attr_t *attr, ----- 指定新线程属性  
    void *(*start_routine)(void *), ----- 新线程要执行的函数  
    void *arg); ----- 传递给要执行函数的参数
```

## 函数指针：指向函数地址，而非数据

```
int add(int a, int b) { return a + b;}  
int subtract(int a, int b) { return a - b;}
```

// 函数指针类型，指向接受两个整数参数并返回整数的函数

```
typedef int (*ArithmeticFunc)(int, int);
```

```
int main() {  
    ArithmeticFunc func_ptr;  
    // 将函数 add 的地址赋值给函数指针变量  
    func_ptr = add;  
    printf("add(3, 4) = %d\n", func_ptr(3, 4));  
    // 将函数 subtract 的地址赋值给函数指针变量  
    func_ptr = subtract;  
    printf("subtract(5, 2) = %d\n", func_ptr(5, 2));  
    return 0;  
}
```

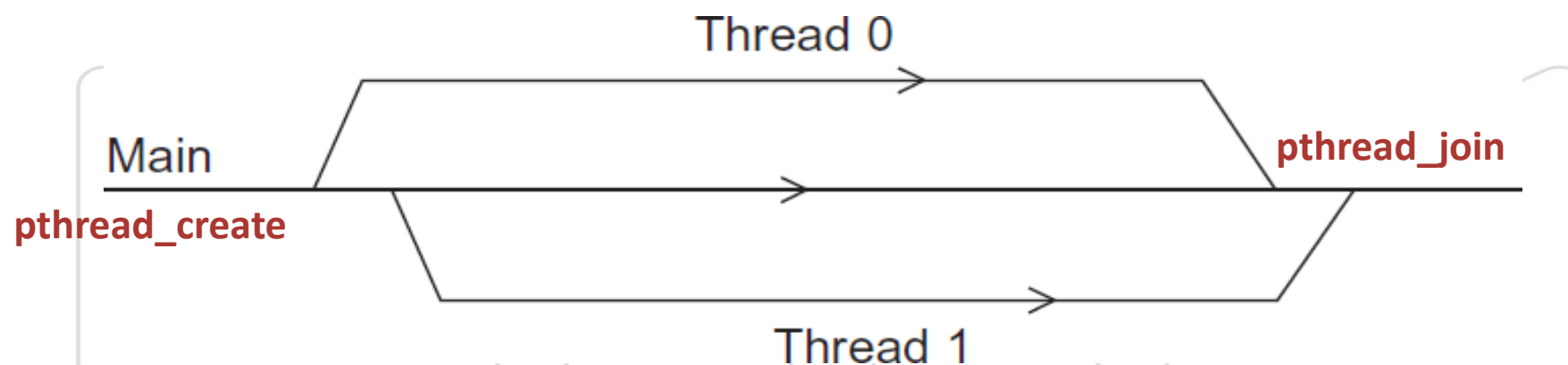
Pthreads 执行函数还需要匹配类型

```
void* subtract(void* ab){  
    int* ab_data = (void*)ab;  
    ab_data[0]-ab_data[1];  
    ...  
}
```

## 停止线程

- 等待指定线程结束执行，并获取返回值
  - 当前线程阻塞，直到线程终止
  - 目标线程终止后，通过`ret_val_p`获取返回值
    - 不需要返回值时，可将`ret_val_p`设置为NULL
  - 确保主线程（或其他线程）继续执行前，目标线程完成工作

```
int pthread_join(  
    pthread_t thread, ----- 线程对象  
    void** ret_val_p); ----- 线程执行的返回值
```





- 矩阵向量乘法:  $y = A \cdot x$

- $y$  的第  $i$  个元素  $y_i = \sum a_{ij}x_j$  ( $A$  的第  $i$  行与向量  $x$  的点积)

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$	$x_0$	$y_0$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$	$x_1$	$y_1$
$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$	$\vdots$	$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$	$\vdots$		$\vdots$	$x_{n-1}$	$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$		$y_{m-1}$

```
/* For each row of A */  
for (i = 0; i < m; ++i){  
    /* i-th row dot x */  
    y[i] = 0.0;  
    for (j = 0; j < n; ++j)  
        y[i] += A[i][j]*x[j];  
}
```

## 矩阵向量乘法：并行设计分析

- 划分：子任务为向量点积  $y_i = \sum a_{ij}x_j$
- 聚合：每个进程处理多行
  - 与MPI对比，并行程序设计上的关注点有何不同？

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

线程0：计算第0行

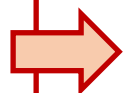
```
y[0] = 0.0;  
for (j = 0; j < n; ++j)  
    y[0] += A[0][j]*x[j];
```

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

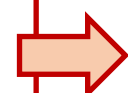
一般情况：计算第*i*行

## • Pthreads实现

根据线程编号  
决定计算任务



执行线程  
计算任务



```
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*n+j]*x[j];  
    }  
  
    return NULL;  
} /* Pth_mat_vect */
```

- 概述
- 临界区与互斥量
- 生产者-消费者同步与信息量
- 路障与条件变量
- 读写锁
- 缓存一致性与伪共享
- 线程安全

## ◉ 例：估算 $\pi$ 值

– 蒙特卡洛方法、无穷级数、割圆法

– 莱布尼茨级数  $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + (-1)^n \cdot \frac{1}{2n+1}$

• 由反正切函数  $\tan^{-1}$  的泰勒展开得到

•  $\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$

– 验证：对  $1 - x^2 + x^4 - x^6 + \cdots = \frac{1}{1+x^2}$  两边同时积分

• 当  $x = 1$ ，有  $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$

– 不同无穷级数收敛速度不一致

## 例：估算 $\pi$ 值

– 不同无穷级数收敛速度不一致

莱布尼茨级数

尼拉卡莎级数

$\pi$ 的无穷级数	第1项	前2项	前3项	前4项	前5项	收敛到:
$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} \dots$	4.0000	2.6666...	3.4666...	2.8952...	3.3396...	$\pi = 3.1415\dots$
$\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} \dots$	3.0000	3.1666...	3.1333...	3.1452...	3.1396...	

• 拉马努金（1910）：
$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum \frac{(4n)!}{(n!)^4} \cdot \frac{26390n+1103}{396^{4n}}$$

– 1项： $\frac{9801}{4412} \sqrt{2} = 3.141592730 \dots$

– 2项： $\frac{9801\sqrt{2}}{4412 + \frac{27493}{256158936}} = 3.141592653589793878 \dots$

– 该公式直到1987年才被证明



## ◉ 例：估算 $\pi$ 值

– 莱布尼茨级数  $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \dots + (-1)^n \cdot \frac{1}{2n+1}$

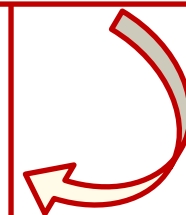
- 求和问题

```
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;
```

```
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) {  
        sum += factor/(2*i+1);  
    }  
  
    return NULL;  
} /* Thread_sum */
```

更新全局变量sum

```
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;
```



- 莱布尼茨公式求 $\pi$ : 单线程（串行） vs 2线程（并行）
  - 单线程估算结果随着 $n$ 增加而越来越精确
  - 双线程误差没有收敛，而是波动

	$n$			
	$10^5$	$10^6$	$10^7$	$10^8$
$\pi$	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

## 竞争条件 (race condition)

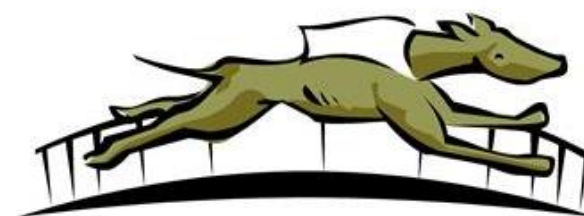
- 对共享变量的更新可能产生冲突
- 忙等待: 不断地执行循环进行检测, 直到条件被满足
  - 只有一个线程能执行实际计算, 其他线程消耗资源进行检查
  - 可能导致死锁

```
y = Compute()  
x = x + y;
```

使用忙等待  
消除竞争条件

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

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



## ● 莱布尼茨公式求 $\pi$ ：并行版（忙等待-v1）

```
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) {  
        sum += factor/(2*i+1);  
    }
```

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

```
    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;  
    }
```

使用忙等待限制对全局  
变量sum的访问

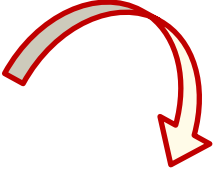
## ● 莱布尼茨公式求 $\pi$ ：并行版（忙等待-v2）

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor, my_sum = 0.0; 增加局部和my_sum  
    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 */
```



```
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;
```

减少对全局变量的访问次数  
每个线程只需一次忙等待



## • 互斥量（mutex, mutual exclusion）

- 确保任意时刻，**只有一个线程**可以访问共享资源/临界区
- 两个基本操作：**加锁（lock）**和**解锁（unlock）**
  - 访问临界区前：尝试加锁
    - `int pthread_mutex_lock(pthread_mutex_t* mutex_p);`
    - 成功获得锁，则可以进入临界区
    - 没有获得锁（有其他线程在临界区中），线程被阻塞
  - 访问临界区后：释放锁，系统会唤醒其他线程进入临界区
    - `int pthread_mutex_unlock(pthread_mutex_t* mutex_p);`
    - 其他线程被阻塞期间，无需通过忙等待检查条件是否满足



## 互斥量 (mutex, mutual exclusion)

- 在使用mutex前，必须对其进行初始化
  - 属性指定了mutex的具体行为，一般通过NULL使用默认属性

```
int pthread_mutex_init(  
    pthread_mutex_t*          mutex_p, ----- 需要初始化的mutex指针  
    const pthread_mutexattr_t* attr_p); ----- mutex属性
```

- 使用完毕后，可销毁mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex_p);
```

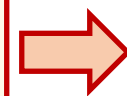


- 莱布尼茨公式求 $\pi$ ：并行版（忙等待 vs 互斥量）
  - 使用两个4核处理器，执行  $n = 10^8$

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



$$\frac{T_1}{T_p} \approx \# \text{ threads}$$



线程数量超过计算核心数目

## ● 莱布尼茨公式求 $\pi$ ：并行版（忙等待 vs 互斥量）

- 使用两个4核处理器，执行  $n = 10^8$
- 线程越多，忙等待上消耗的计算时间就可能越多

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

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

$$\frac{T_1}{T_p} \approx \# \text{ threads}$$

线程数量超过计算核心数目

- 概述
- 临界区与互斥量
- 生产者-消费者同步与信息量
- 路障与条件变量
- 读写锁
- 缓存一致性与伪共享
- 线程安全

## 互斥量的问题

- 忙等待虽然引入额外开销，但能保证线程按顺序访问
- 互斥量**无法保证访问顺序**（随机或由系统调度决定）
- 然而，某些应用控制保证访问临界区的顺序（如，生产者-消费者）

```
void* Thread_work(void* rank){  
    long my_rank = (long) rank;  
    matrix_t my_mat = Allocate_matrix(n);  
    Generate_matrix(my_mat);
```

⇒ 生产：生成矩阵

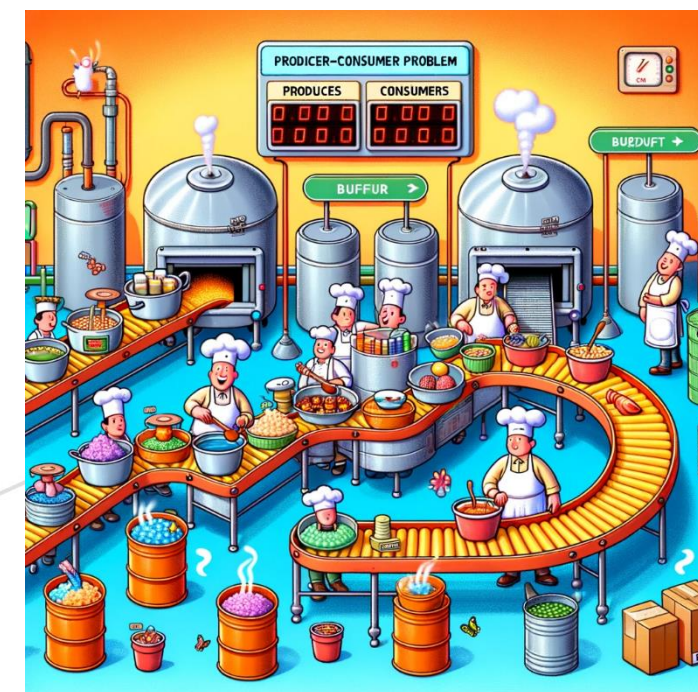
```
    pthread_mutex_lock(&mutex);  
    Multiply_matrix(product_mat, my_mat);  
    pthread_mutex_unlock(&mutex);
```

消费：使用矩阵  
进行矩阵乘法

```
    Free_matrix(&my_mat);  
    return NULL;
```

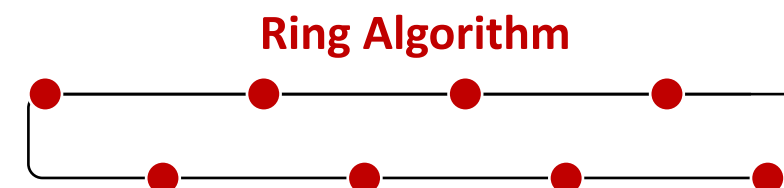
```
} /* Thread_work */
```

然而，矩阵乘法  
不满足交换律！

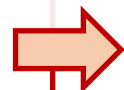




- 生产者-消费者举例：环状消息发送
  - 线程1向线程2发送，线程2向线程3发送...



向全局变量写入消息  
传递给其他线程

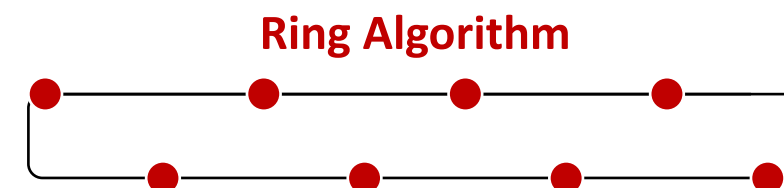


```
void *Send_msg(void* rank) {  
    long my_rank = (long) rank;  
    long dest = (my_rank + 1) % thread_count;  
    long source = (my_rank + thread_count - 1) % thread_count;  
    char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));  
  
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);  
    messages[dest] = my_msg;  
  
    if (messages[my_rank] != NULL) 很可能有线程未收到消息...  
        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 */
```

v1: 使用if语句判断

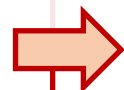
很可能有线程未收到消息...

- 生产者-消费者举例：环状消息发送
  - 线程1向线程2发送，线程2向线程3发送...



```
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 = (char*) malloc(MSG_MAX*sizeof(char));
```

向全局变量写入消息  
传递给其他线程



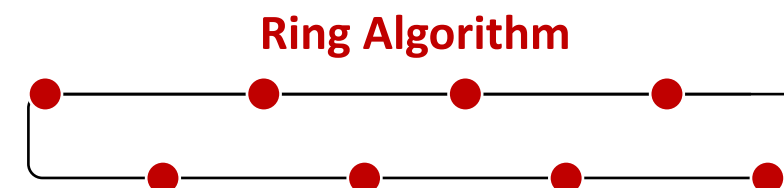
```
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);  
    messages[dest] = my_msg;
```

v2: 使用while循环，忙等待生产者完成生产

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

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

- 生产者-消费者举例：环状消息发送
  - 线程1向线程2发送，线程2向线程3发送...



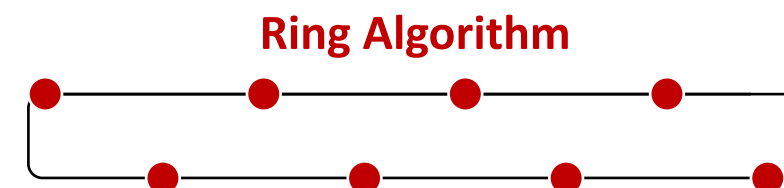
```
void *Send_msg(void* rank) {  
    long my_rank = (long) rank;  
    long dest = (my_rank + 1) % thread_count;  
    long source = (my_rank + thread_count - 1) % thread_count;  
    char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));  
  
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);  
    messages[dest] = my_msg;  
  
    while (messages[my_rank] == NULL);  
  
    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);  
  
    return NULL;  
} /* Send_msg */
```

//通知线程 $dest$ 继续执行

...

//等待线程 $source$ 的通知

- 生产者-消费者举例：环状消息发送
  - 线程1向线程2发送，线程2向线程3发送...



```
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 = (char*) malloc(MSG_MAX*sizeof(char));  
  
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);  
    pthread_mutex_lock(mutex[dest]);  
    message[dest] = my_msg;  
    pthread_mutex_unlock(mutex[dest]);  
  
    pthread_mutex_lock(mutex[myrank]);  
    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);  
  
    return NULL;  
} /* Send_msg */
```

使用mutex保护  
全局变量 message

这个方案是否可行？

## ◉ 信号量 (Semaphore)

- 信号量主要用于控制对共享资源的访问
  - 信号量对应一个计数值，反映了可用资源的数量
  - 常用于生产者-消费者问题
  - 计数值是任意 ( $< n$ ) 整数时，称其为计数信号量
  - 计数值只能取0/1时，称其为二元信号量（可用于实现互斥量）
- 1965年由荷兰计算机科学家Edsger Dijkstra 提出
- **等待操作** (wait/P)：获取资源前调用，确认其可用
  - 如果信号量  $> 0$ ，则信号量-1（表示资源被占用）
  - 如果信号量 = 0，则线程或进程将被阻塞，直到信号量  $> 0$
- **发信号操作** (signal/post/V)：生产或释放资源
  - 将信号量的值+1，表示资源已释放

## ◉ 信号量的使用

– 添加头文件 `#include <semaphore.h>`

– 对信号量进行初始化

- `initial_val` 代表初始可用资源数（使用信号量实现互斥量时用“1”）
- `shared` 表示该信号量是否会被其他进程所使用（线程间共享依然为0）

```
int sem_init(  
    sem_t* semaphore_p, ----- 返回初始化的信号量  
    int shared, ----- 指示信号量是否在进程间共享  
    unsigned initial_val); ----- 初始值
```

– 销毁信号量

```
int sem_destroy(sem_t* semaphore);
```

– 等待

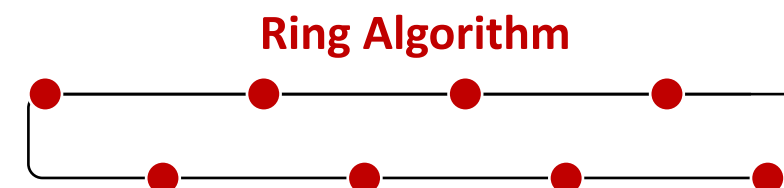
```
int sem_wait(sem_t* semaphore);
```

– 发送信号

```
int sem_post(sem_t* semaphore);
```



- 生产者-消费者举例：环状消息发送
  - 线程1向线程2发送，线程2向线程3发送...



```
void *Send_  
long my_  
long des  
long sou  
char* my  
  
semaphores = malloc(thread_count*sizeof(sem_t));  
for (thread = 0; thread < thread_count; thread++) {  
    sem_init(&semaphores[thread], 0, 0);  
}
```

使用前需要对信号量进行初始化

输出不一定严格有序，但消息发送的顺序是有序的

```
sprintf(  
message[  
sem_post  
  
sem_wait  
printf("  
  
return N  
} /* Send_  
  
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
```

- 概述
- 临界区与互斥量
- 生产者-消费者同步与信息量
- 路障与条件变量
- 读写锁
- 缓存一致性与伪共享
- 线程安全

## 路障 (Barrier)

– 在特定位置**同步**所有线程

- 到达该路障的线程都将被阻塞
- 直到所有线程都到达该路障后，线程才能继续运行

– 路障应用：给线程设置相同起点；确保所有线程已完成前序任务

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
/* Synchronize all processes/threads */
Barrier();
my_start = get_current_time();
/* Code that we want to time */
...
my_finish = get_current_time();
my_elapsed = my finish - my_start;
/* Find the max across all processes/threads */
global_elapsed = global_max(my_elapsed);
```

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

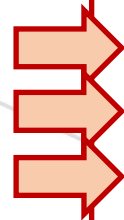


## • 如何实现路障? : Mutex

- 使用一个共享计数器变量，统计到达路障的线程数量
  - 使用Mutex保护共享变量的更新
  - 计数器达到总线程数时，线程离开临界区
  - 需要使用忙等待检查状态

```
/* shared and initialized by the main thread */  
int counter; /* initialized to 0 */  
int thread_count;  
pthread_mutex_t barrier_mutex;  
  
void *Thread_work(...) {  
    ...  
    pthread_mutex_lock(&barrier_mutex);  
    counter++;  
    pthread_mutex_unlock(&barrier_mutex);  
    while (counter < thread_count);  
    ...  
} /* Thread_work */
```

使用mutex  
保证计数准确



在路障处忙等待



## • 如何实现路障? : Semaphore

- 使用两个信号量
  - 一个保护计数器
  - 一个充当路障，阻塞线程
- 最后到达路障的线程
  - 重置计数器为0
  - 发送thread\_count-1次信号，使阻塞在路障的线程恢复执行
- 其他线程
  - 增加计数器
  - 在路障信号量处等待

```
/* shared variables */
int thread_count;
int counter;          /* initialize to 0 */
sem_t count_sem;      /* initialize to 1 */
sem_t barrier_sems;   /* initialize to 0 */

void *Thread_work(...) {
    ...
    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_sems);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sems);
    }
    ...
} /* Thread_work */
```

## • 如何实现路障？条件变量（condition variables）

- 数据对象，允许特定条件或事件发生前挂起线程
  - 条件或事件发生时，由另一个线程唤醒被阻塞的线程
- 总是与互斥量（mutex）配合使用
- 目的：在条件满足时，执行临界区内的代码
  - 避免忙等待

```
lock mutex;
```

```
if condition has occurred
```

```
    signal thread(s)
```

```
else {
```

```
    unlock the mutex and block;
```

```
    /* when thread is unblocked, mutex is relocked */
```

```
}    唤醒后重新尝试获取mutex锁
```

```
unlock mutex;
```

条件满足：唤醒被挂起线程

条件不满足：释放mutex  
并挂起，等待被唤醒



## • 如何实现路障？条件变量（condition variables）

### – 创建条件变量

```
int pthread_cond_init(  
    pthread_cond_t* cond_p, ----- 用于返回条件变量  
    const pthread_condattr_t* cond_attr_p); ----- 设置条件变量属性
```

### – 销毁条件变量

- int **pthread\_cond\_destroy**(pthread\_cond\_t\* cond\_p);

### – 唤醒条件变量

- int **pthread\_cond\_signal**(pthread\_cond\_t\* cond\_p);
- int **pthread\_cond\_broadcast**(pthread\_cond\_t\* cond\_p);

### – 等待条件变量

```
int pthread_cond_wait(  
    pthread_cond_t* cond_p, ----- 条件变量  
    pthread_mutex_t* mutex_p); ----- 条件变量配套的锁
```

## ● 如何实现路障？条件变量（condition variables）

```
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;

void *Thread_work(void* rank) {
    ...
    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);
    ...
} /* Thread_work */
```

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

## 路障 (barrier)

- Pthreads已提供路障实现，并且可以控制等待线程数量
- 创建路障
  - 等待线程达到count后，线程可恢复执行

```
int pthread_barrier_init(  
    pthread_barrier_t*      barrier_p, ----- 用于返回路障的指针  
    const pthread_barrierattr_t* barrier_attr_p, ----- 路障属性指针  
    unsigned                count); ----- 等待线程数量
```

## – 等待路障

- int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier);

## – 销毁路障

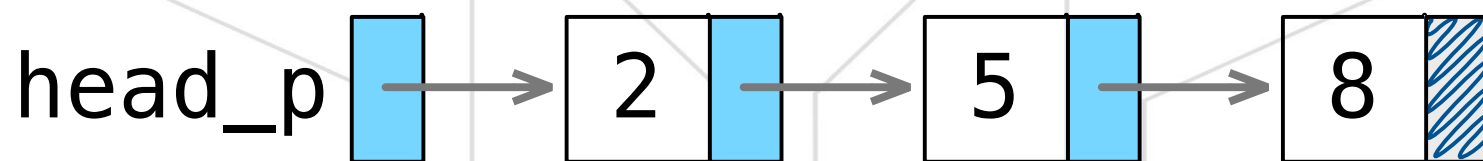
- int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);

- 概述
- 临界区与互斥量
- 生产者-消费者同步与信息量
- 路障与条件变量
- 读写锁
- 缓存一致性与伪共享
- 线程安全

## 如何实现线程间共享的数据结构？并行链表

- 一组结点，每个结点包含一个数据和一个指向下一个结点的指针
  - 内存中不必连续
  - 支持成员查找、插入、删除
  - 查找为线性时间 $O(n)$ ，但插入、删除本身的时间为 $O(1)$

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

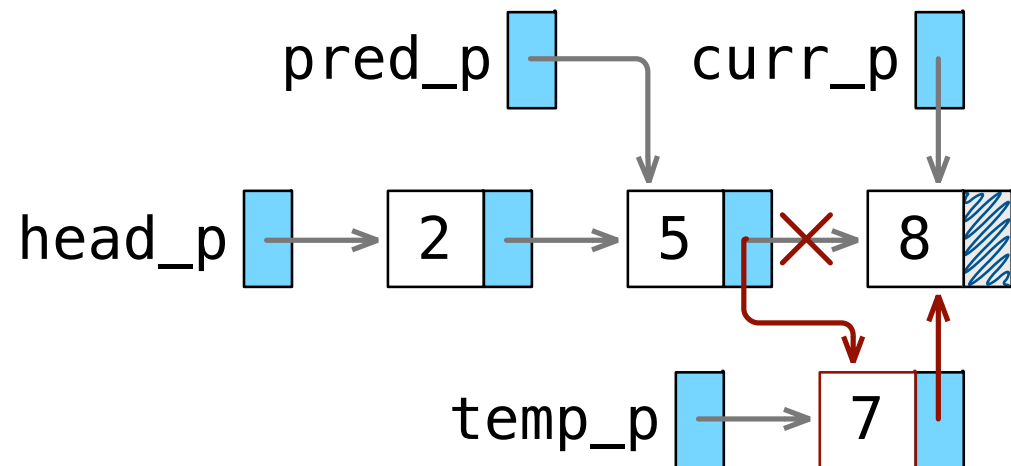


## ◉ 链表：成员查找Member

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



## 链表：插入元素Insert

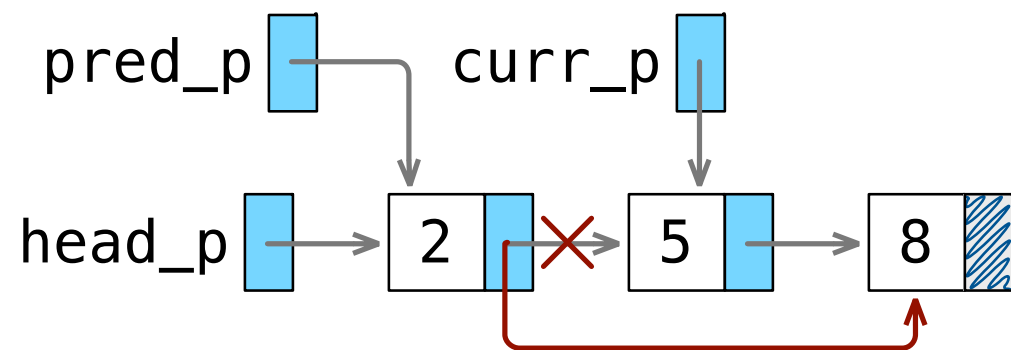


```
int Insert(int value, struct list_node_s** head_pp){
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;
    struct list_node_s* temp_p;

    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)
            *head_pp = temp_p;
        else
            pred_p->next = temp_p;
        return 1;
    } else { /* value in list */
        return 0;
    }
} /* Insert */
```

## 链表：删除元素Delete



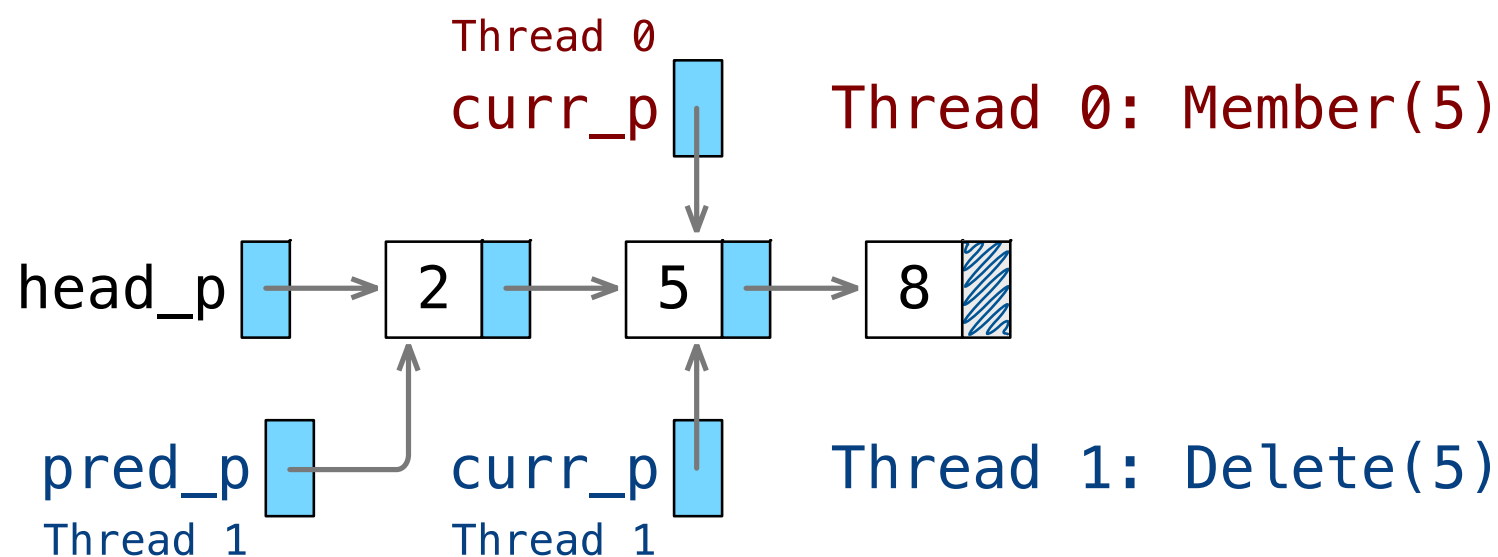
```
int Delete(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;

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

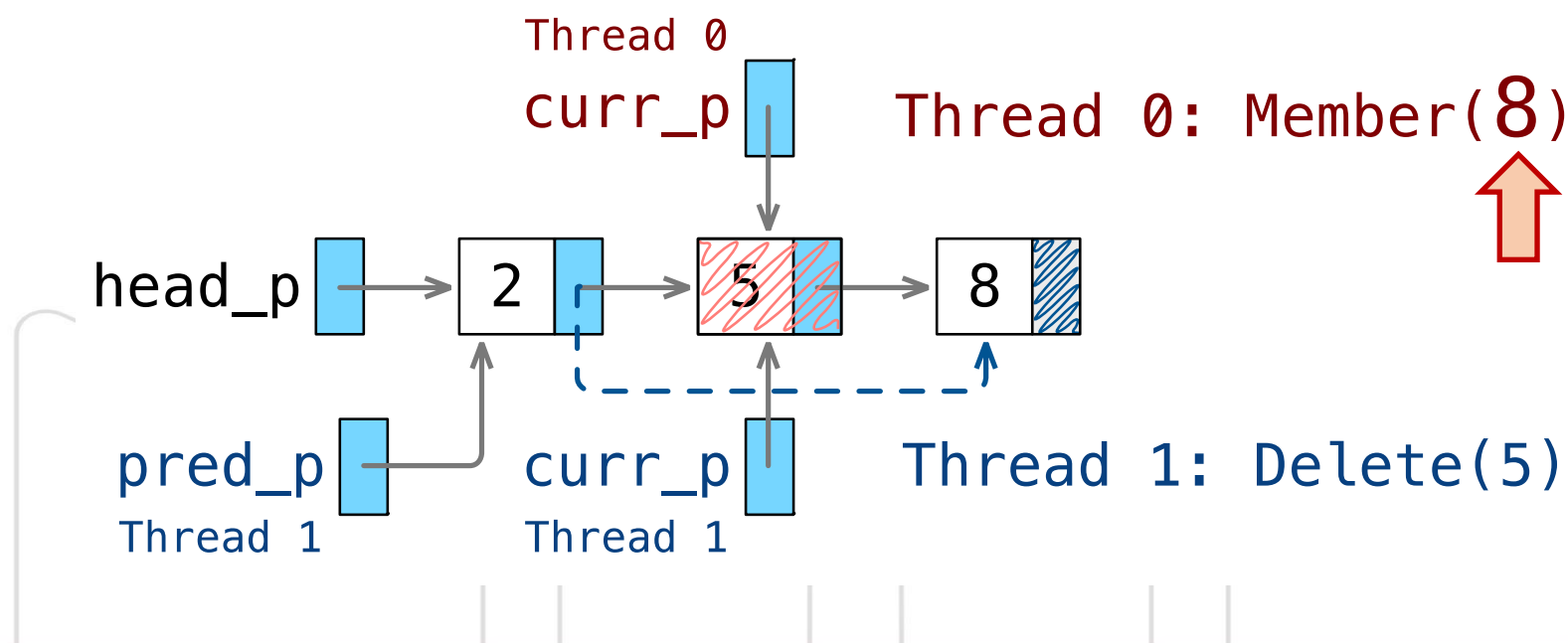
    if (curr_p != NULL && curr_p->data == value) {
        if (pred_p == NULL) { /* first element in list */
            *head_pp = curr_p->next;
            free(curr_p);
        } else {
            pred_p->next = curr_p->next;
            free(curr_p);
        }
        return 1;
    } else {
        return 0;
    }
} /* Delete */
```

## • Pthreads多线程链表：如何使所有线程都能访问链表？

- 增加head\_p全局变量：所有操作都需要head\_p
  - 在调用Member, Insert, Delete时无需再传递head\_p
- 多线程并行操作链表可能出现冲突
  - 在Thread 0返回前，结点5可能已经被删除



- Pthreads多线程链表：如何使所有线程都能访问链表？
  - 增加head\_p全局变量：所有操作都需要head\_p
    - 在调用Member, Insert, Delete时无需再传递head\_p
  - 多线程并行操作链表可能出现冲突
    - 在Thread 0指向结点5，但还没有移动到结点8时，结点5可能被释放
    - Thread 0通过curr\_p访问被释放的结点5中的指针（segmentation fault）



## • Pthreads多线程链表：如何解决冲突？

- 保护共享变量（head\_p及链表中的所有结点）
- 方案1：对任意操作，都使用mutex保护其不受其他线程操作干扰

```
pthread_mutex_lock(&list_mutex);  
Member(value);  
pthread_mutex_lock(&list_mutex);
```

- 优点：无需更改链表实现（数据结构/操作），仅需在调用时加锁
- 缺点：所有链表操作都需串行进行
  - 同一时间，只有一个线程进入临界区（只能执行一个操作）
- 观察1：多个Member调用间不冲突，仅调用Insert与Delete时会产生冲突
  - 对于Member调用较多的应用而言，当前实现无法充分发挥并行性能
  - 但对于Insert与Delete调用较多的应用而言，当前实现可能已经接近最优
- 观察2：只有访问同一结点的调用会产生冲突

## • Pthreads多线程链表：如何解决冲突？

– 方案2：对每个结点分别加锁，而非整个链表加锁

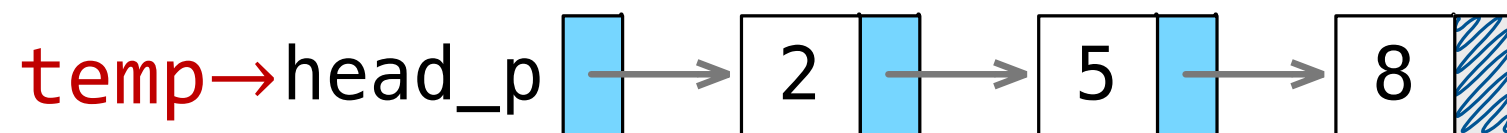
- 修改数据结构，给每个结点增加一个mutex成员变量
- 修改Member，Insert，Delete函数，在访问结点时加锁

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



## • Pthreads多线程链表：如何解决冲突？

- 方案2：对每个结点分别加锁，而非整个链表加锁
  - 修改Member, Insert, Delete函数，在访问结点时加锁



版本1：是否正确？

```
int Member(int value) {
    struct list_node_s *temp;

    pthread_mutex_lock(&head_mutex);
    temp = head;
    while (temp != NULL && temp->data < value) {
        if (temp->next != NULL)
            pthread_mutex_lock(&(temp->next->mutex));
        if (temp == head)
            pthread_mutex_unlock(&head_mutex);
        pthread_mutex_unlock(&(temp->mutex));
        temp = temp->next;
    }
}
```

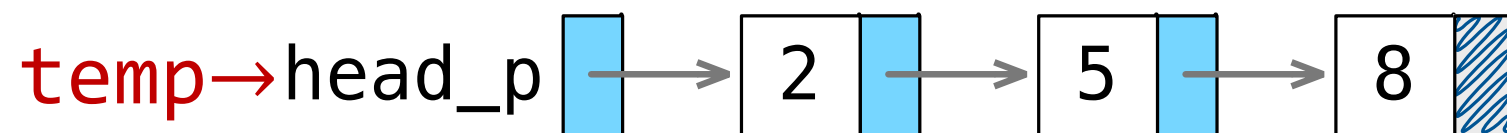
查找

```
if (temp == NULL || temp->data > value) {
    if (temp == head)
        pthread_mutex_unlock(&head_mutex);
    if (temp != NULL)
        pthread_mutex_unlock(&(temp->mutex));
    return 0;
} else { /* temp != NULL && temp->data <= value */
    if (temp == head)
        pthread_mutex_unlock(&head_mutex);
    pthread_mutex_unlock(&(temp->mutex));
    return 1;
}
} /* Member */
```

返回

## • Pthreads多线程链表：如何解决冲突？

- 方案2：对每个结点分别加锁，而非整个链表加锁
  - 修改Member, Insert, Delete函数，在访问结点时加锁



版本2

```
int Member(int value) {  
    struct list_node_s *temp, *old_temp;  
  
    pthread_mutex_lock(&head_mutex);  
    temp = head;  
    if (temp != NULL) pthread_mutex_lock(&(temp->mutex));  
    pthread_mutex_unlock(&head_mutex);  
    while (temp != NULL && temp->data < value) {  
        if (temp->next != NULL)  
            pthread_mutex_lock(&(temp->next->mutex));  
        old_temp = temp;  
        temp = temp->next;  
        pthread_mutex_unlock(&(old_temp->mutex));  
    }  
}
```

查找

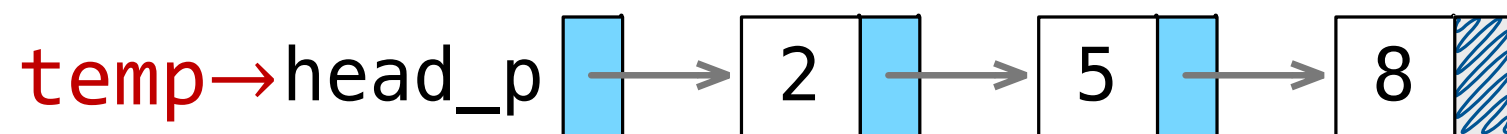
```
    if (temp == NULL || temp->data > value) {  
        if (temp != NULL)  
            pthread_mutex_unlock(&(temp->mutex));  
        return 0;  
    } else { /*temp != NULL && temp->data <= value*/  
        pthread_mutex_unlock(&(temp->mutex));  
        return 1;  
    }  
} /* Member */
```

返回

## • Pthreads多线程链表：如何解决冲突？

– 方案2：对每个结点分别加锁，而非整个链表加锁

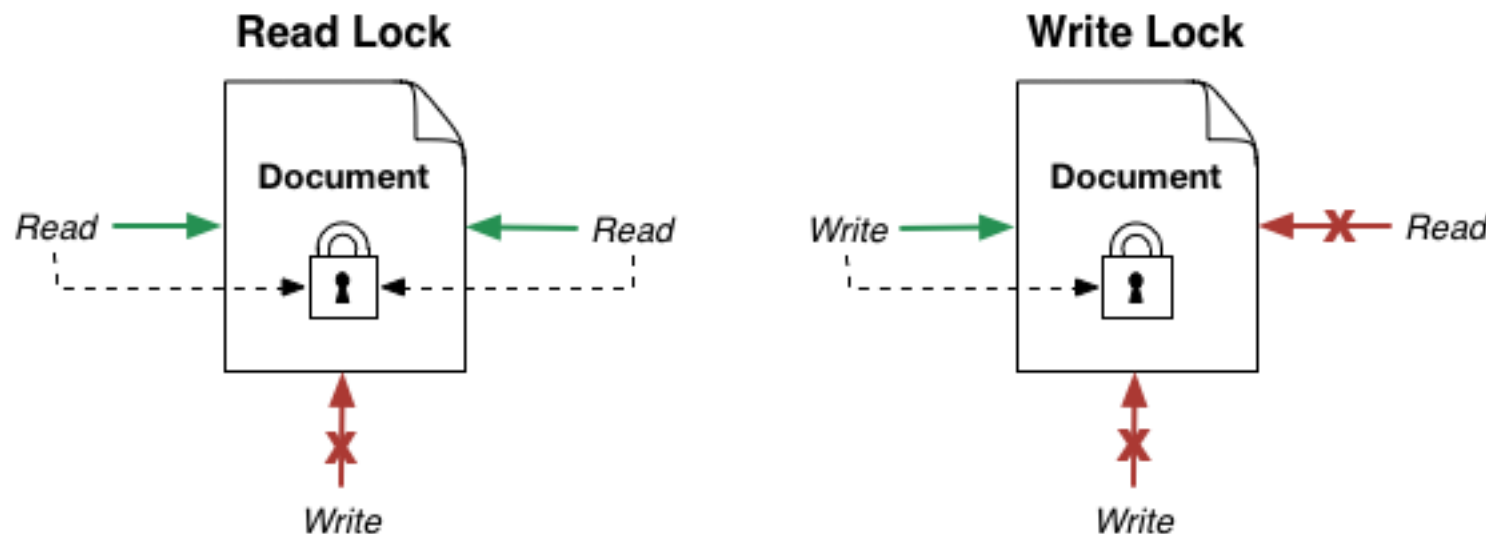
- 修改数据结构，给每个结点增加一个mutex成员变量
- 修改Member, Insert, Delete函数，在访问结点时加锁



- 缺点1：比方案1更为复杂，需要修改数据结构及相关操作函数的实现
  - 复杂就容易出错
- 缺点2：慢：每次访问结点都需要加锁及解锁
- 缺点3：空间开销增加：每个结点都需要多分配一个mutex

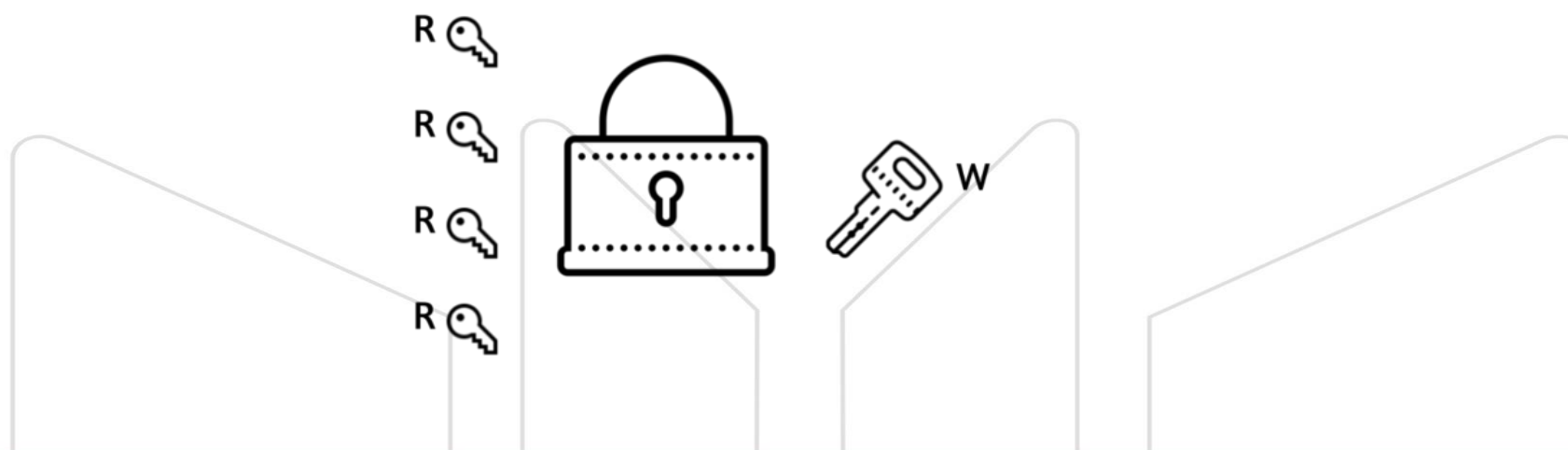
## • Pthreads多线程链表：如何解决冲突？

- 两个使用mutex的解决方案都不理想
  - 方案1只允许一个线程访问链表，方案2只允许一个线程访问结点
- 回顾：多个Member调用不冲突，仅调用Insert与Delete会产生冲突
  - 启示：对读数据与写数据分别进行控制



## ◉ 读写锁（Read-Write Lock）

- 允许多个线程同时对共享资源进行读操作
  - 在写操作期间需要互斥访问，以确保数据的一致性
- 与互斥锁相似，但包含两种锁状态：读锁和写锁
  - 一个线程持有读锁时，其他线程仍然可以获取读锁（允许并发的读取）
  - 一个线程持有写锁时，其他线程必须等待该写锁的释放
  - 任意线程持有读锁时，其他线程无法获得写锁



## 读写锁（Read-Write Lock）的基本操作

- 获取读锁：允许多个线程同时获取读锁，用于并发的读取操作
- 获取写锁：独占地获取写锁，防止其他线程同时进行读或写操作
- 释放锁：释放对共享资源的控制（并发读入/独占写出）

```
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);
```



## 读写锁的性能

### – Pthreads多线程链表的时间对比

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

读操作频繁、写操作相对较少  
允许并发的读操作可提高系统性能

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

写操作频繁时，依然可能造成长期阻塞  
读写锁与单个互斥锁性能差别不大



## • 如何实现读写锁？分析

### – 需要维护的状态

- 读者数量，写者数量
- 需要同步机制控制访问

// 互斥锁

```
pthread_mutex_t mutex;
```

// 条件变量

```
pthread_cond_t read_cond;
```

```
pthread_cond_t write_cond;
```

// 读者数量和写者数量

```
int readers_count = 0;
```

```
int writers_count = 0;
```

### – 需要执行的操作

- 获取读锁时检查是否有写者
- 获取写锁时检查是否读者或其他写者
- 释放读锁时通知所有等待写者
- 释放写锁时通知所有等待读者和写者

// 读锁

```
void read_lock() {
```

```
    pthread_mutex_lock(&mutex);
```

```
    while (writers_count > 0) {
```

```
        pthread_cond_wait(&read_cond, &mutex);
```

```
    }
```

```
    readers_count++;
```

```
    pthread_mutex_unlock(&mutex);
```

```
}
```

- 如何实现读写锁？实现
  - 以下代码是否正确？存在什么问题？

```
// 读锁
void read_lock() {
    pthread_mutex_lock(&mutex);
    while (writers_count > 0) {
        pthread_cond_wait(&read_cond, &mutex);
    }
    readers_count++;
    pthread_mutex_unlock(&mutex);
}

// 读解锁
void read_unlock() {
    pthread_mutex_lock(&mutex);
    readers_count--;
    if (readers_count == 0) {
        pthread_cond_signal(&write_cond);
    }
    pthread_mutex_unlock(&mutex);
}
```

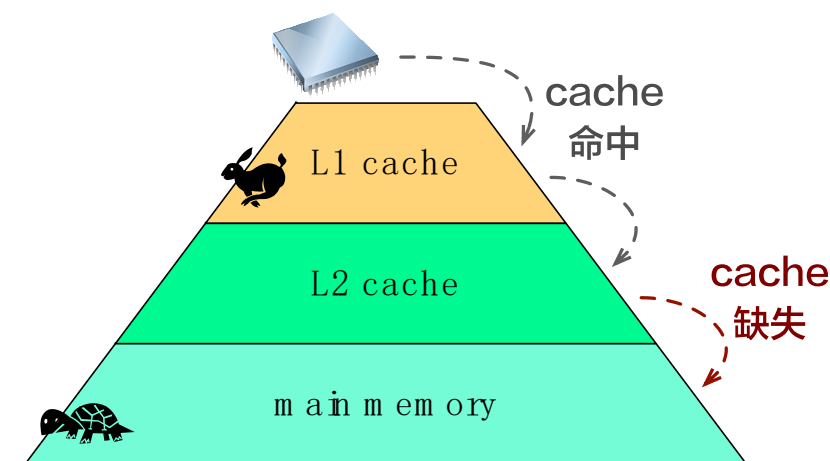
```
// 写锁
void write_lock() {
    pthread_mutex_lock(&mutex);
    while (readers_count > 0 || writers_count > 0) {
        pthread_cond_wait(&write_cond, &mutex);
    }
    writers_count++;
    pthread_mutex_unlock(&mutex);
}

// 写解锁
void write_unlock() {
    pthread_mutex_lock(&mutex);
    writers_count--;
    pthread_cond_broadcast(&read_cond);
    pthread_cond_signal(&write_cond);
    pthread_mutex_unlock(&mutex);
}
```

- 概述
- 临界区与互斥量
- 生产者-消费者同步与信息量
- 路障与条件变量
- 读写锁
- 缓存一致性与伪共享
- 线程安全

## • CPU对数据的访问通过缓存进行

- 缓存：访问速度快，但容量小
- 时间和空间局部性
- 每次读入或写出以**缓存行/缓存块**为单位
  - 需访问变量在缓存中时，缓存命中，直接在缓存中操作
  - 需访问的变量不在缓存中时，产生**缓存缺失**，需要从内存中读取数据



## • 当多个线程并行访问数据时

- **缓存一致性**：缓存中的数据需要保持一致（将产生额外开销）
  - 处理器/核心更改缓存中的数据时，其他处理器/核心能够看到更新的数据
- **伪共享**（False Sharing）：多线程同时访问**同一缓存行的不同部分**
  - 导致额外的缓存一致性操作（即使不在同一个变量上进行操作）
  - 即使线程之间没有真正的共享数据，它们也会因为缓存一致性而产生竞争<sup>57</sup>

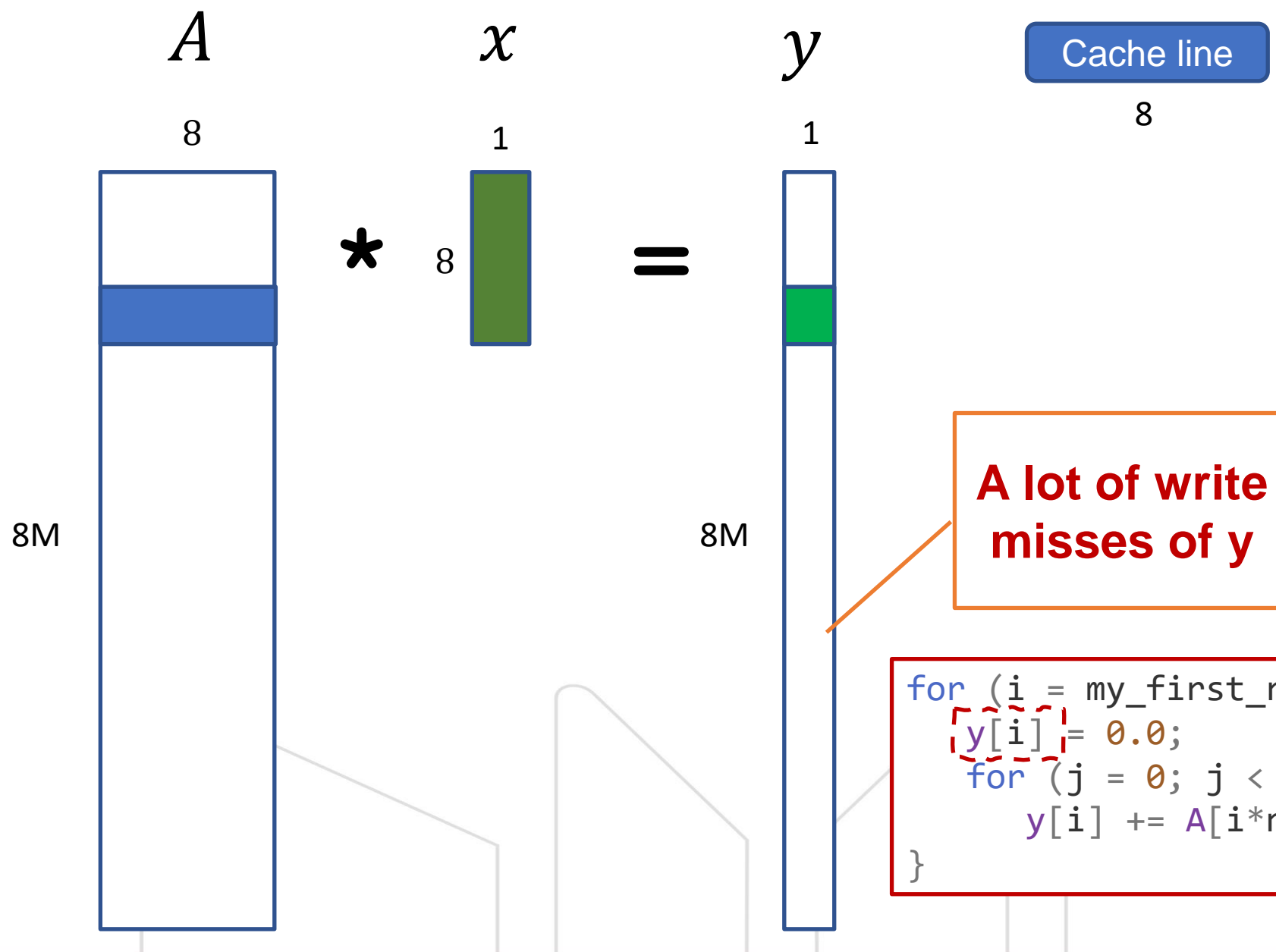
## • Pthreads矩阵向量乘法中，多个线程对数据的并发访问

```
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*n+j]*x[j];  
    }  
  
    return NULL;  
} /* Pth_mat_vect */
```

- Pthreads矩阵向量乘法中，多个线程对数据的并发访问
  - 访问模式： $y[i] += A[i*n+j]*x[j]$ （ $i$ 外层， $j$ 内层）
  - 矩阵维度对性能是否有影响？对比  $8M \times 8$ ,  $8K \times 8K$ ,  $8 \times 8M$ 
    - 分析：运算次数完全相同
    - 单线程性能： $8M \times 8$ 比 $8K \times 8K$ 慢14%， $8 \times 8M$ 比 $8K \times 8K$ 慢28%
      - 受缓存影响
    - 多线程性能差距更大

Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 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

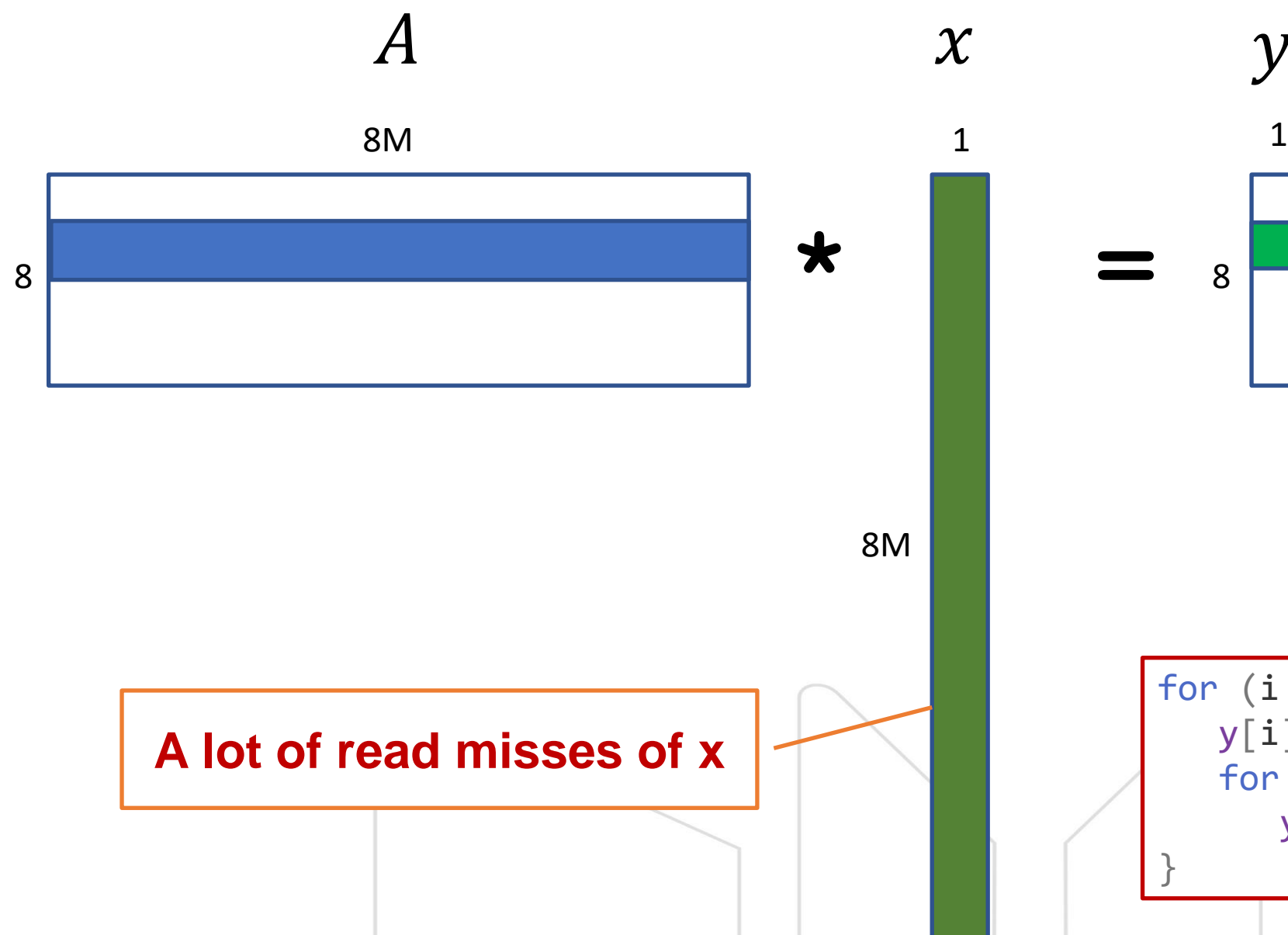
## • Pthreads矩阵向量乘法缓存分析



```
for (i = my_first_row; i <= my_last_row; i++) {  
    y[i] = 0.0; Valgrind缓存分析工具结果  
    for (j = 0; j < n; j++)  
        y[i] += A[i*n+j]*x[j];  
}
```

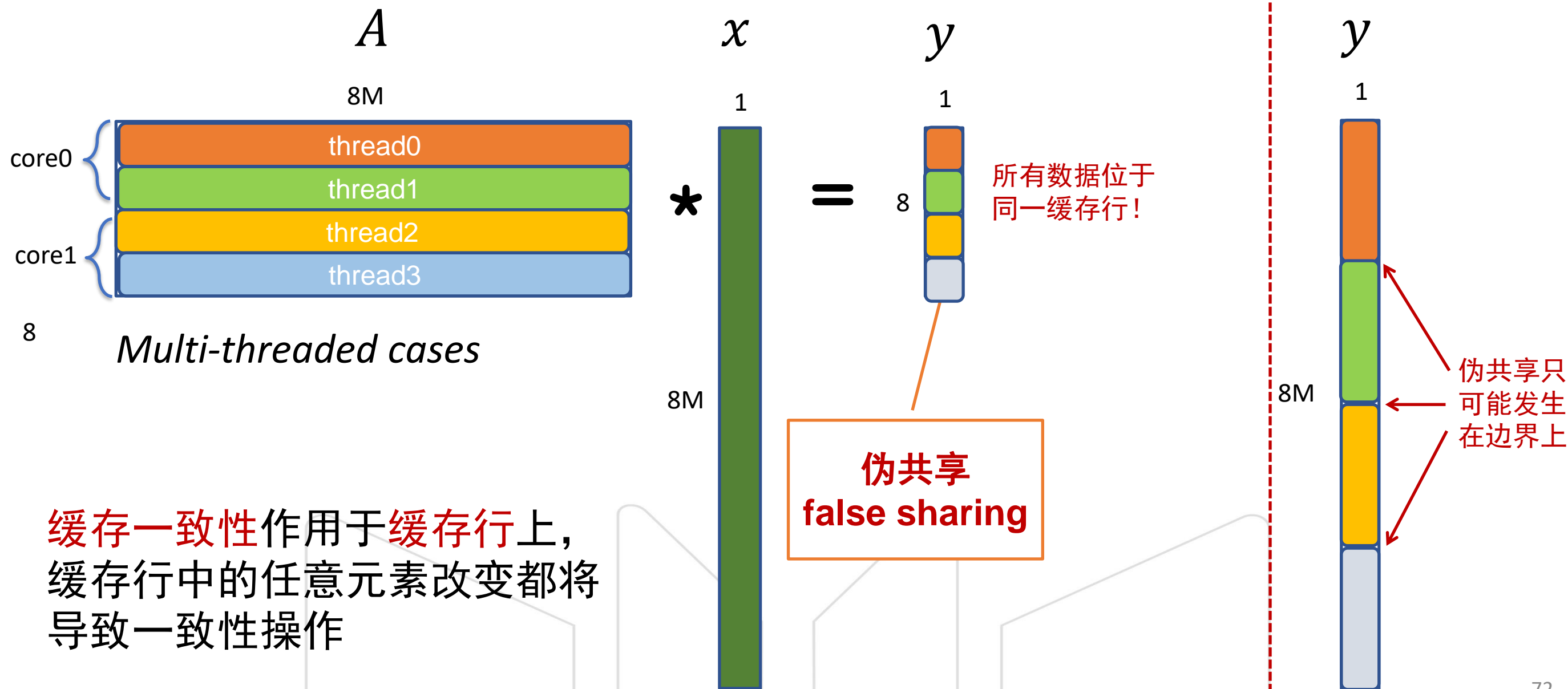


## • Pthreads矩阵向量乘法缓存分析



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

## • Pthreads矩阵向量乘法缓存分析



- 概述
- 临界区与互斥量
- 生产者-消费者同步与信息量
- 路障与条件变量
- 读写锁
- 缓存一致性与伪共享
- 线程安全

## ◉ 线程安全性 (Thread Safety)

- 多线程环境中，程序仍能够正确地执行并产生正确的结果
- 不安全操作常源于对共享资源（数据）的访问导致的数据竞争、损坏
  - 全局变量、静态变量
- C 标准库中的不安全函数
  - `strcpy()`、`strcat()`、`strtok()` 等字符串操作函数
  - `malloc()`、`free()`、`realloc()` 等内存分配和释放函数（某些实现中）
  - `rand()` 和 `srand()` 产生随机数的函数（某些实现中）
- 标准 I/O 库中的不安全函数
  - `gets()` 和 `fgets()` 读取字符串的函数
  - `scanf()` 和 `printf()` 格式化输入输出函数

## ◉ 非线程安全函数举例：**strtok**（字符串分词）

- `char* strtok(char* str, const char* separators)`
- 根据给定的分隔符，将字符串分割成一系列“标记（token）”
- 注意**strtok**每次返回一个token，而不是一系列token
  - 使用**静态存储**，对输入字符串进行缓存

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

## ◉ 简易并行字符串分词

- 每个线程处理一行字符串

## ◉ 简易并行字符串分词

### — 使用信号量制造临界区

```
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;  
  
    /* Sequential reading of the input */  
    sem_wait(&sems[my_rank]);  
    fg_rv = fgets(my_line, MAX, stdin);  
    sem_post(&sems[next]);  
    while (fg_rv != NULL) {  
        printf(...);  
    }
```

```
        count = 0;  
        my_string = strtok(my_line, " \t\n");  
        while ( my_string != NULL ) {  
            count++;  
            printf(...);  
            my_string = strtok(NULL, " \t\n");  
        }  
        if (my_line != NULL)  
            printf(...);  
  
        sem_wait(&sems[my_rank]);  
        fg_rv = fgets(my_line, MAX, stdin);  
        sem_post(&sems[next]);  
    }  
  
    return NULL;  
} /* Tokenize */
```

## ◉ 简易并行字符串分词

### — 多线程时，出现漏词的情况

```
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...



## ◉ 简易并行字符串分词

- 多线程时，出现漏词的情况，为什么？
- **strtok**使用静态存储，对输入字符串进行缓存
  - 当传入**NULL**时，直接使用之前静态存储中的缓存内容
    - `my_string = strtok(NULL, “\t\n”);`
  - 然而，该静态存储为所有线程**共享**的
  - 因此**strtok**并非线程安全
    - 但不正确的并行程序并非必然产生不正确的输出（需要多次实验，仔细分析逻辑）
- **strtok\_r**线程安全的分词函数
  - `char *strtok_r(char *str, const char *delim, char **saveptr)`
  - 使用saveptr返回当前处理位置

## ◉ 线程与进程

- 进程：分配资源的最小单位，可包含多个并行执行的线程
- 线程：调度的最小单位，“轻量级”进程
  - 共享进程资源、地址空间
  - 创建、切换、销毁开销都较小

## ◉ 同步与线程安全

- 多线程访问共享资源时，可能出现竞争条件
- 忙等待：使用while循环不断检查是否满足执行条件
  - 对计算资源的消耗大，使用编译器优化时不可靠
- 临界区：同一时刻只能由一个线程执行的代码段
  - 临界区代码可视为串行执行

## ◉ 同步与线程安全

- 互斥量（**mutex**）：用于避免多个线程同时访问临界区
  - 访问前加锁，访问后释放锁
- 信号量（**semaphore**）：带计数器的“锁”
  - 使用前等待资源，使用中减少资源，使用后增加资源
  - 常用于生产者-消费者问题
- 障碍（**barrier**）：阻塞线程，直到所有线程都到达障碍点
- 读写锁（**read-write lock**）：多个读者或一个写者可同时执行
- C中使用静态变量的库函数是非线程安全的

# Questions?

