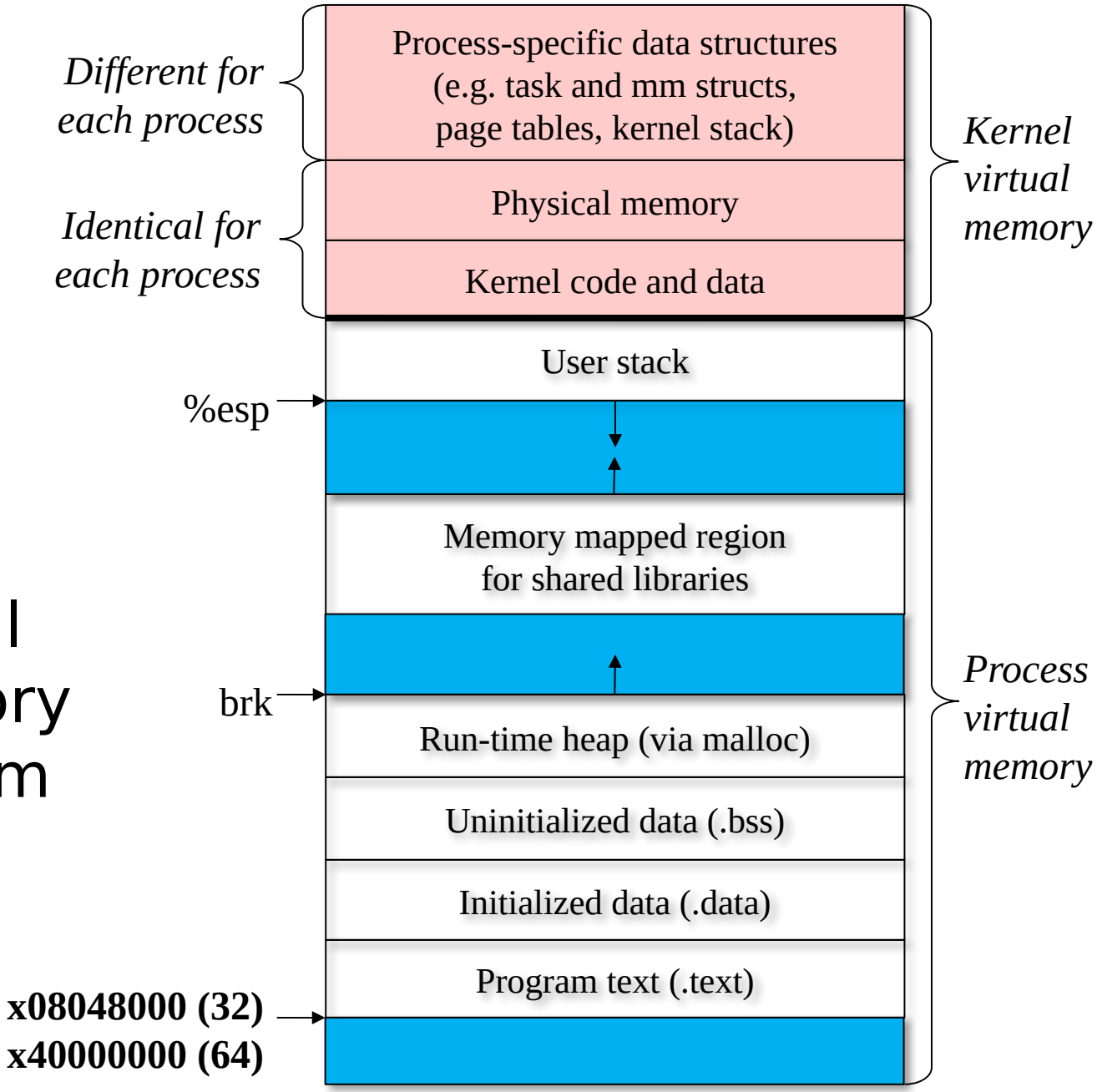
基于多线程的并发编程

线程是什么?
有什么优点?

参考书: CSAPP Section 12

Linux Virtual Memory System



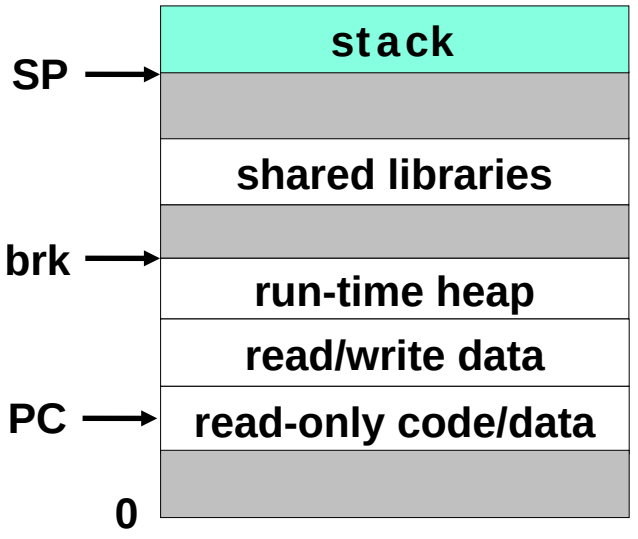
Traditional view of a process

- Process = process context + code, data, heap and stack

Process context

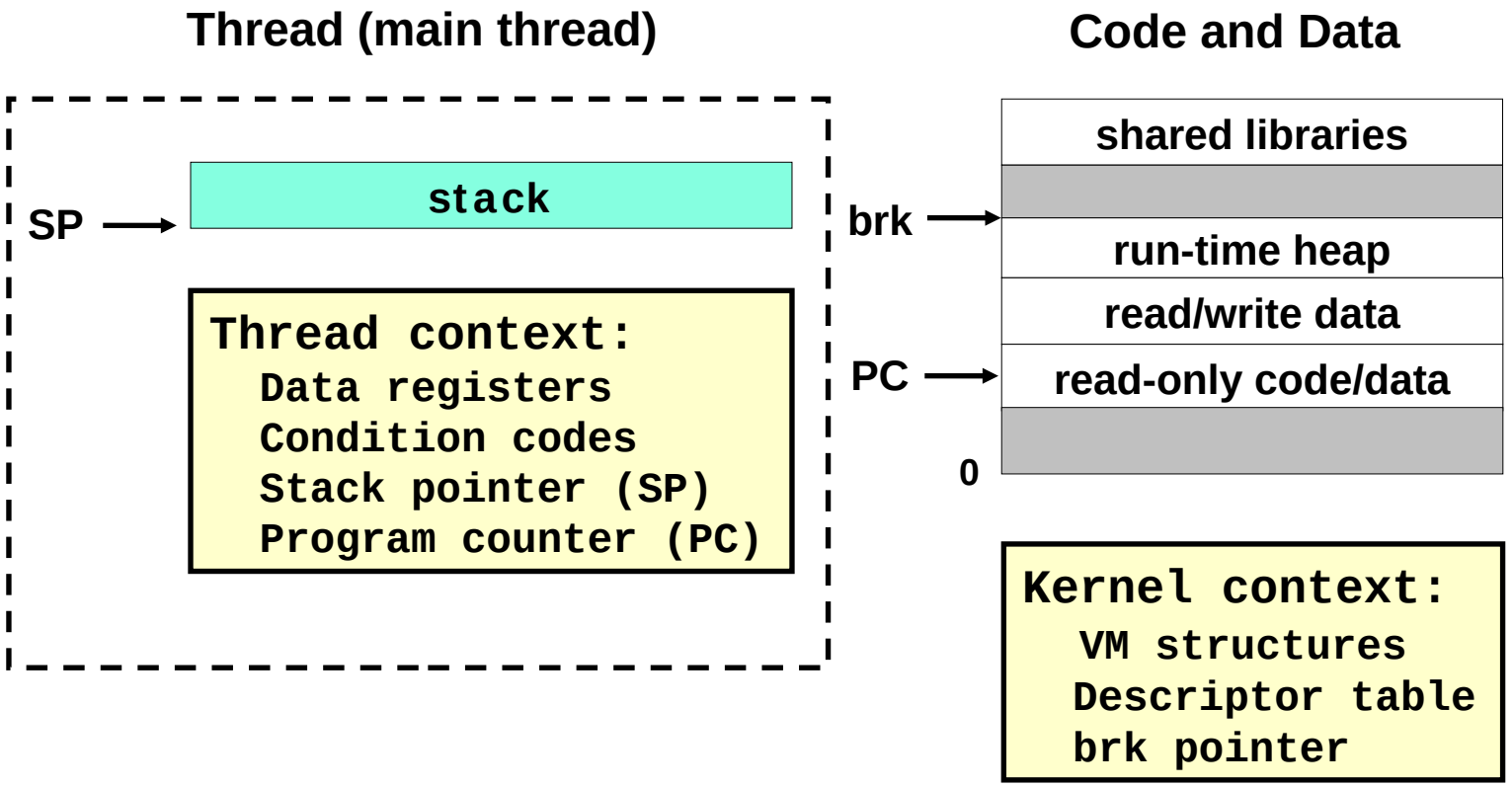
```
Program context:
  Data registers
  Condition codes
  Stack pointer (SP)
  Program counter (PC)

Kernel context:
  VM structures
  Descriptor table
  brk pointer
```

Code, data, and stack



Alternate view of a process

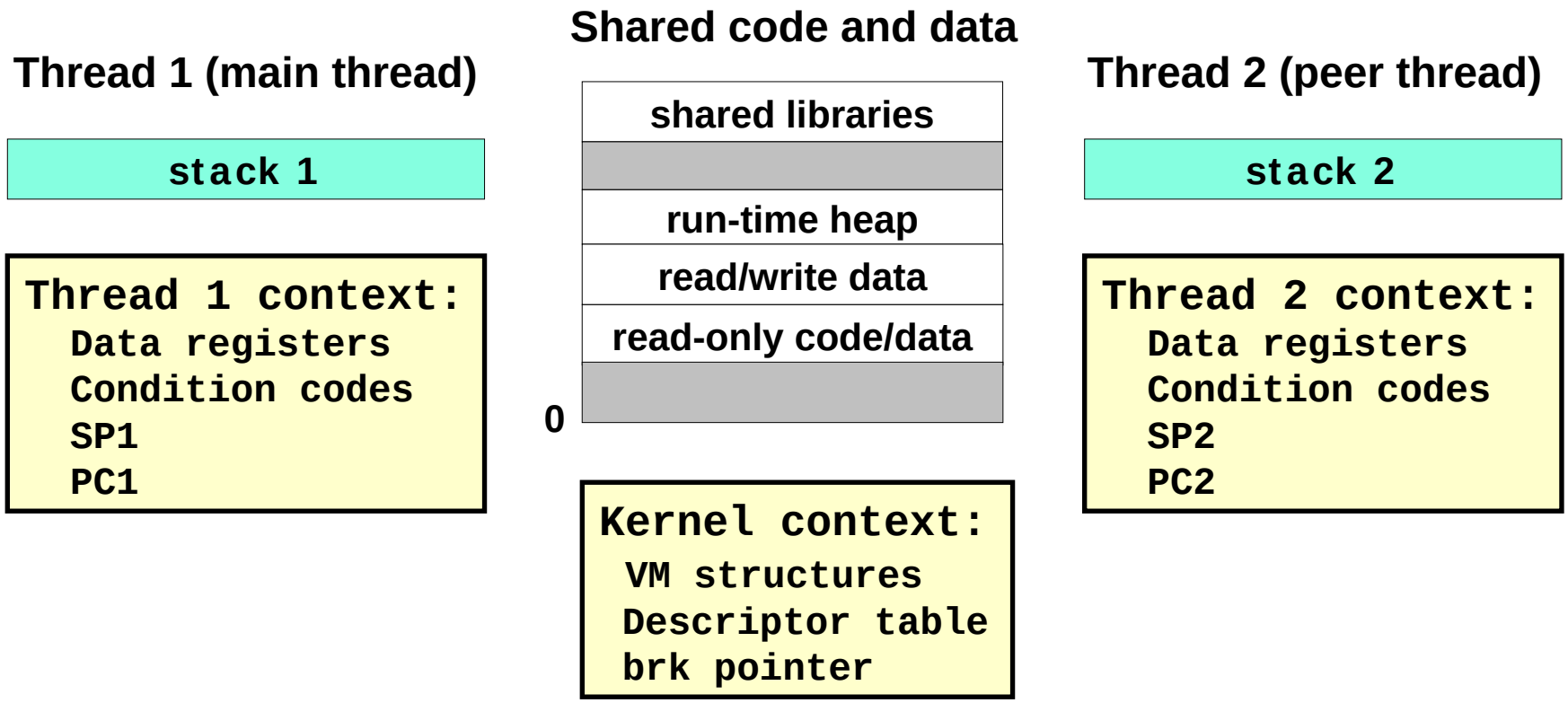
- Process = thread + code, data, heap, and kernel context



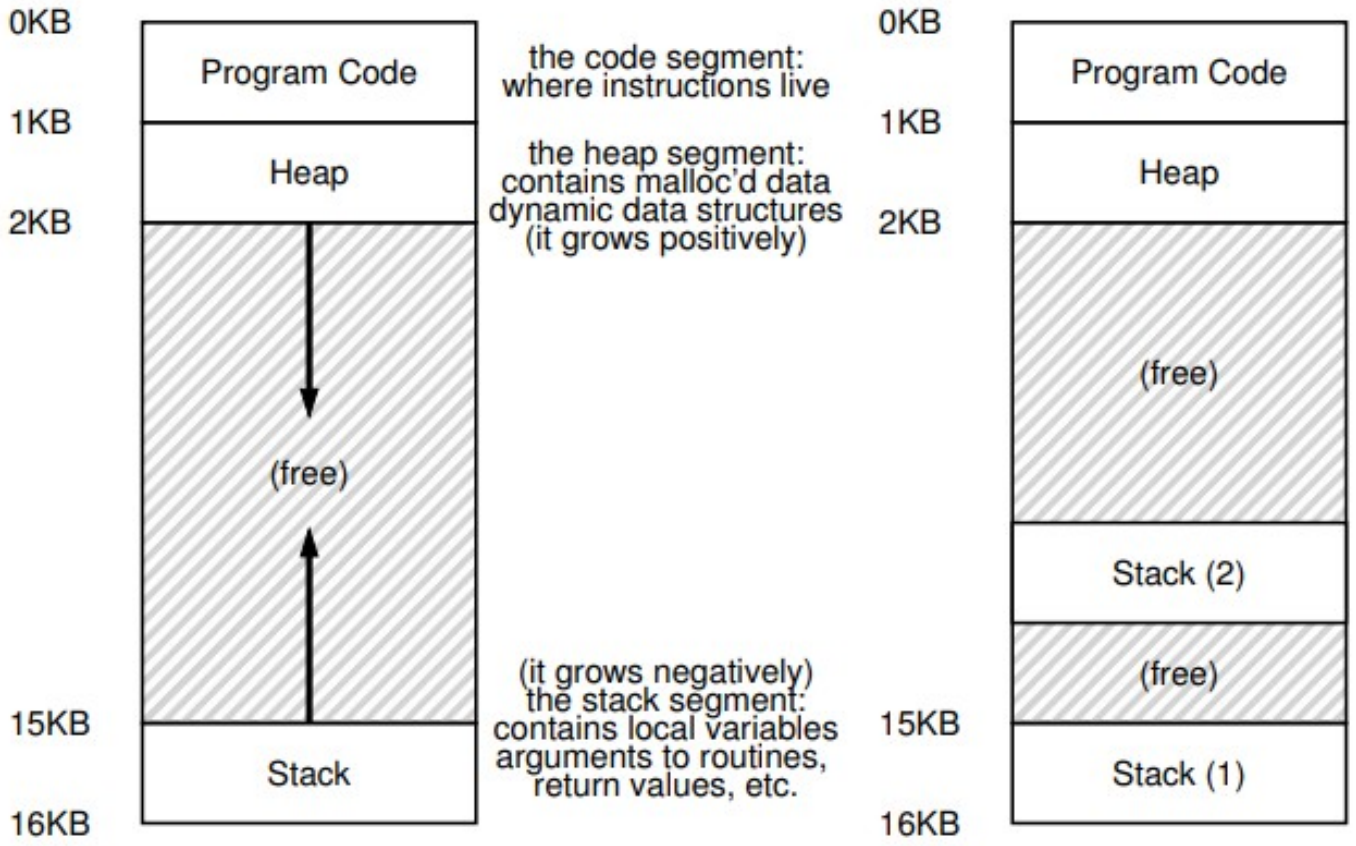
A process with multiple threads

- 多个线程可以共享同一个进程中的部分资源
  - 每个线程有自己的流程控制（PC 的值，寄存器的值，栈）
  - 每个线程共享相同的 code, data, heap 和 kernel context
  - 每个线程共享页表
  - 每个线程有自己的 thread id (TID)

A process with multiple threads



A process with multiple threads



single-thread

multi-thread

线程 vs. 进程（1）

- 进程之间可以并发、共享数据
  - E.g., PostgreSQL：多进程
  - MySQL：多线程
- 但是开销会比较大
  - 每个进程有自己独立的虚拟内存空间，无法按照虚拟地址共享内存；需要用 IPC （进程间通信）机制，但开销更高
- 而在一个进程空间的多线程，天然数据共享
  - Data, heap, stack

线程 vs. 进程（2）

- 线程切换和进程切换开销比较
  - Context switch 需要保存的内容差别?
    - 需要保存内容差别不大
    - 但进程切换，访问新的 kernel context, process memory ，可能 cache 命中率较低

Link

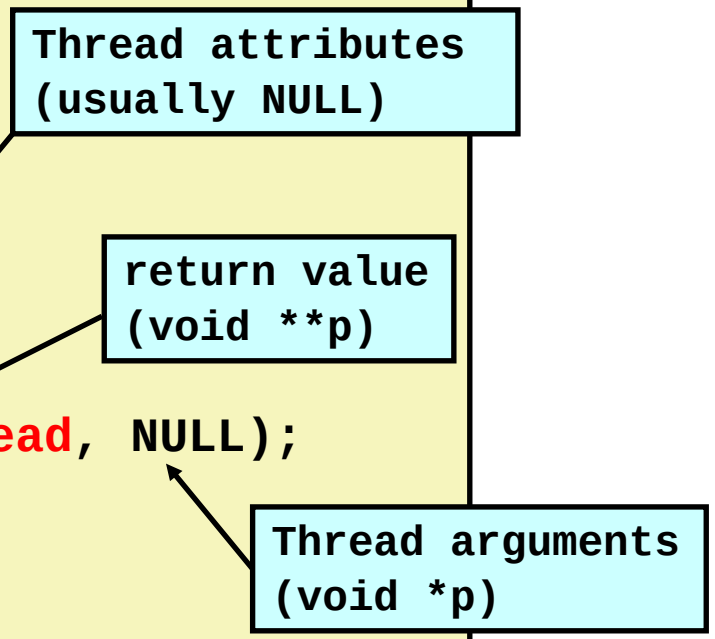
  - 线程切换：不需要切换地址空间
  - 进程切换：切换地址空间
    - 更换页表（辅助寄存器：例如页表首地址）
    - TLB 命中率下降

Posix threads (Pthreads) interface

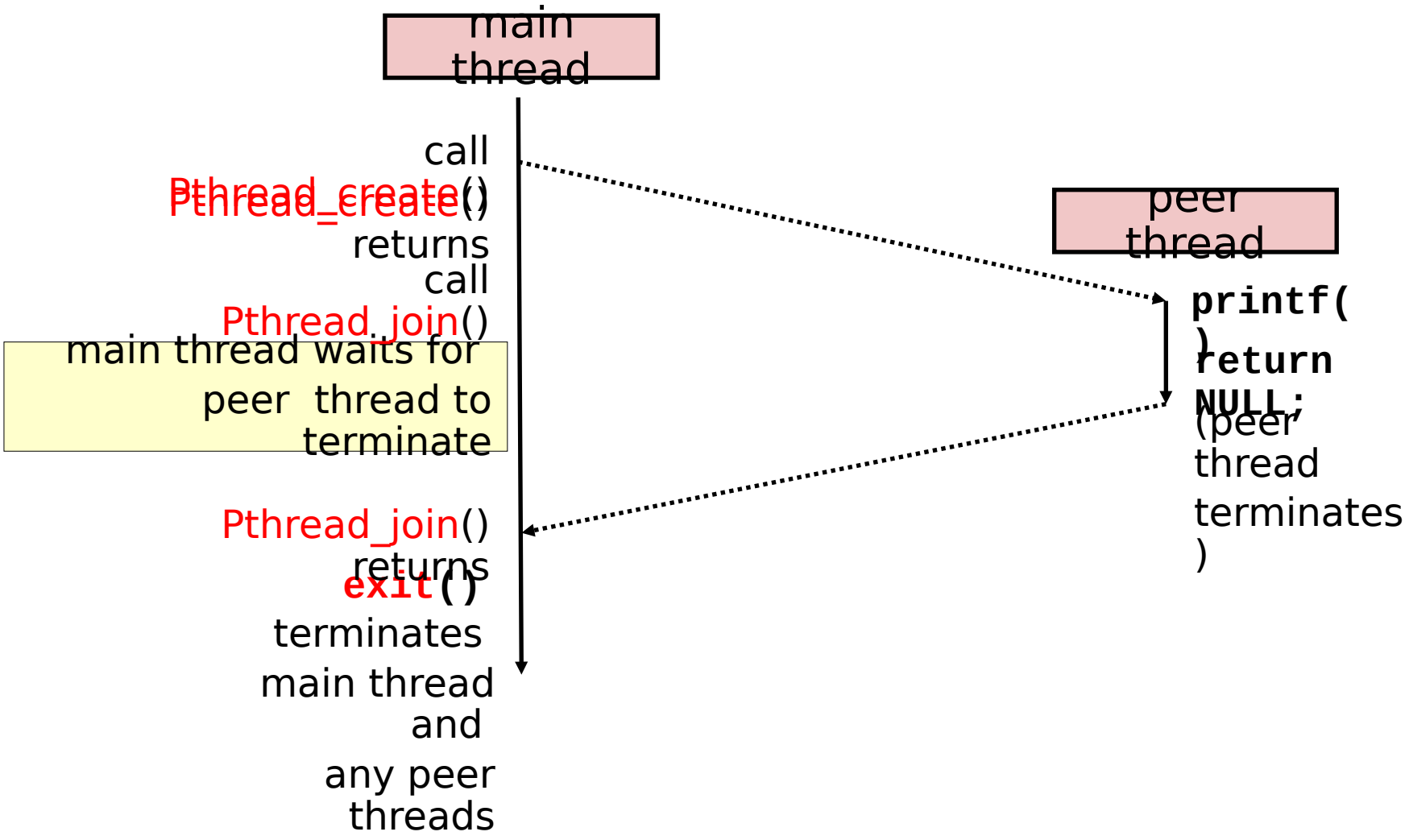
- Pthreads: ~60 个标准的函数接口
  - 支持在 C 语言中使用 thread
  - 创建和收割 (reap)threads
    - **pthread_create**
    - **pthread_join**
  - 确认 thread ID
    - **pthread_self**
  - 终止 thread
    - **pthread_cancel** **发送中止请求给某线程**
    - **pthread_exit** **线程主动退出**
    - **exit** [terminates all threads]
    - **return** [terminates current thread]

The Pthreads "hello, world" Program

```
/* hello.c - Pthreads "hello, world" program */
#include "csapp.h"

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}
```



Execution of Threaded“hello, world”



Issues with threads

- 线程分为 joinable 和 detached 两种模式
  - joinable thread 可以 reap，也可以被其他线程 kill
    - must be reaped (with **`pthread_join`**) to free memory resources (e.g., thread stack)
  - 处于 detached 模式的 thread，执行结束后自动回收资源（不会内存泄漏），无法被其他线程 reap 或 kill

线程默认为 joinable

Issues with threads

- 注意避免内存泄露，不 join 则必须用线程的“detached”模式
  - use int **`pthread_detach(pthread_t tid)`** to make a thread detached

多线程的共享变量

Shared variable analysi

```
1  #include "csapp.h"
2  #define N 2
3  void *thread(void *vargp);
4
5  char **ptr;
6  /* global variable */
```

```
7  int main()
8  {
9          int i;
10         pthread_t tid;
11         char *msgs[N] = {
12           "Hello from foo",
13           "Hello from bar"
14         };
15
16         ptr = msgs;
17         for (i = 0; i < N; i++)
18            Pthread_create(&tid, NULL, thread, (void *)i);
19         Pthread_exit(NULL);
20 }
21
22 void *thread(void *vargp)
23  {
24         int myid = (int)vargp;
25         static int cnt = 0;
26
27         printf("[%d]:%s(cnt=%d)\n", myid, ptr[myid], ++cnt);
28  }
```

Shared variable analysis

- 每个线程在 CPU 中运行时，占有独立的 register
- 每个线程有独立的 stack，但是共享 code (text), data, heap, shared libs
  - 全局 / 静态变量、动态内存中的变量是共享的
  - 栈中的局部变量是私有的（正常情况下）
- 但线程的 stack 并不受 kernel 保护，可以被其他线程读写
  - 例如上一页中 main thd 的 msgs 通过全局指针 ptr 被 peer thds 访问了

Shared variable analysis

- Which variables are shared?

Variable instance	Referenced by main thread?	Referenced by peer thread-0?	Referenced by peer thread-1?
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

Shared variable analys

```
1  #include "csapp.h"
2
3  #define NITERS 100000000
4  void *count(void *arg);
5
6  /* shared variable */
7  unsigned int cnt = 0;
8
```

```
9  int main()
10 {
11      pthread_t tid1, tid2;
12
13      Pthread_create(&tid1, NULL, count, NULL);
14      Pthread_create(&tid2, NULL, count, NULL);
15      Pthread_join(tid1, NULL);
16      Pthread_join(tid2, NULL);
17
18      if (cnt != (unsigned)NITERS*2)
19          printf("BOOM! cnt=%d\n", cnt);
20      else
21          printf("OK cnt=%d\n", cnt);
22      exit(0);
23 }
```

Shared variable analysis

```
24
25 /* thread routine */
26 void *count(void *arg)
27 {
28       int i;
29       for (i=0; i<NITERS; i++)
30           cnt++;
31       return NULL;
32 }
```

Shared variable analysis

```
linux> badcnt
BOOM! cnt=198841183

linux> badcnt
BOOM! cnt=198261801

linux> badcnt
BOOM! cnt=198269672
```

- **cnt** should be equal to 200,000,000.
- What went wrong?!

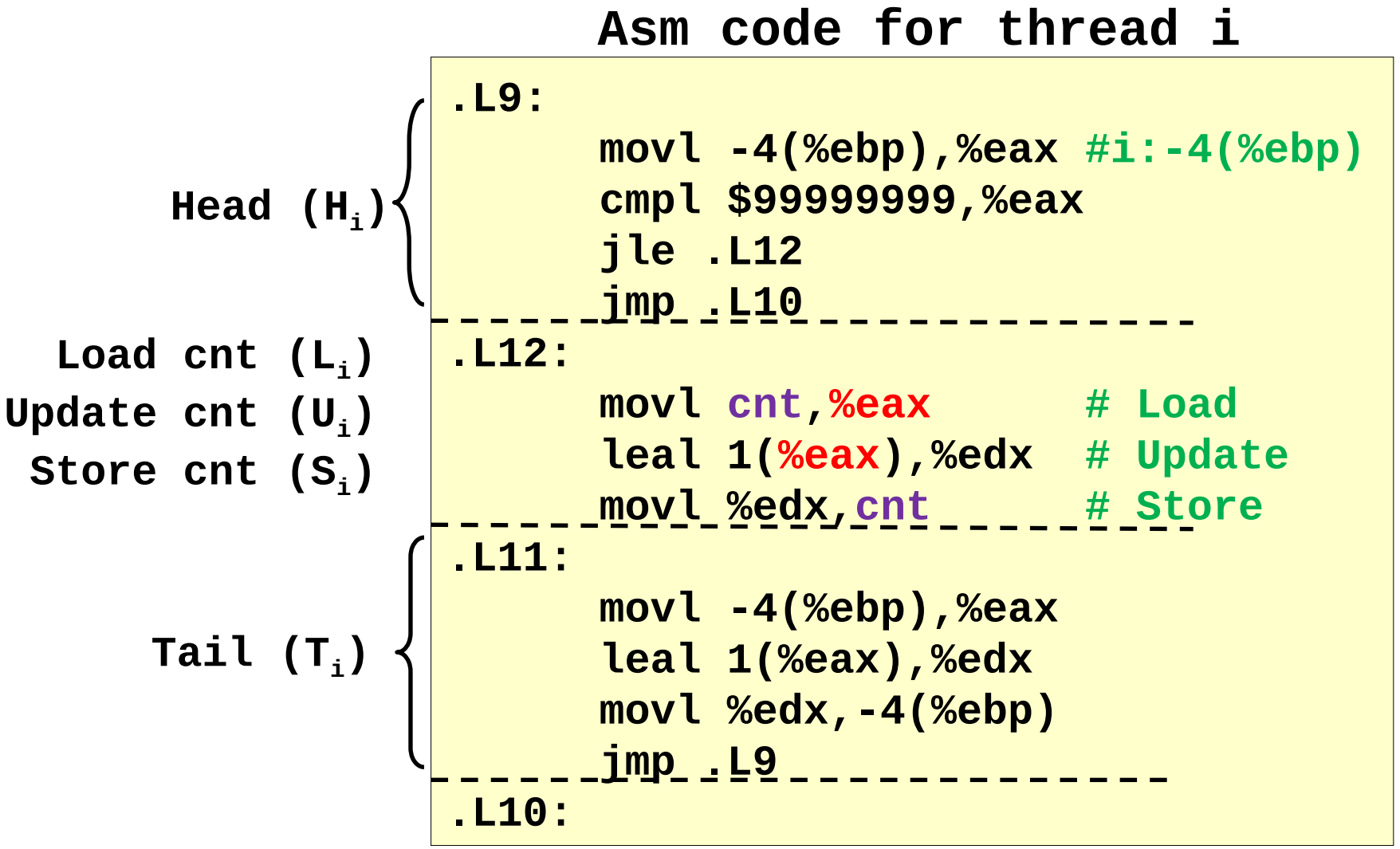
Assembly code for counter loop

C code for thread i

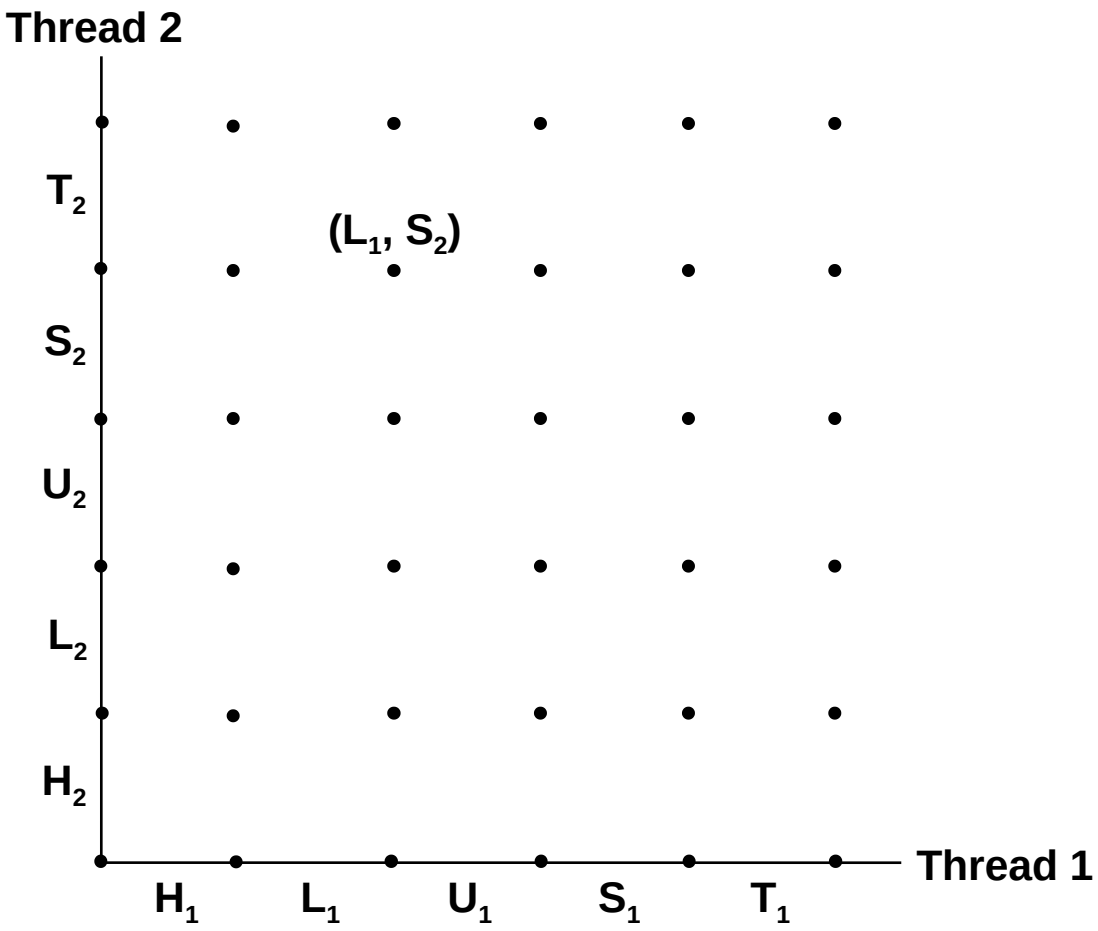
```
for (i=0; i<NITERS; i++)
    cnt++;
```

++ 操作不是原子的!

Assembly code for counter loop



Progress graphs（进展图）



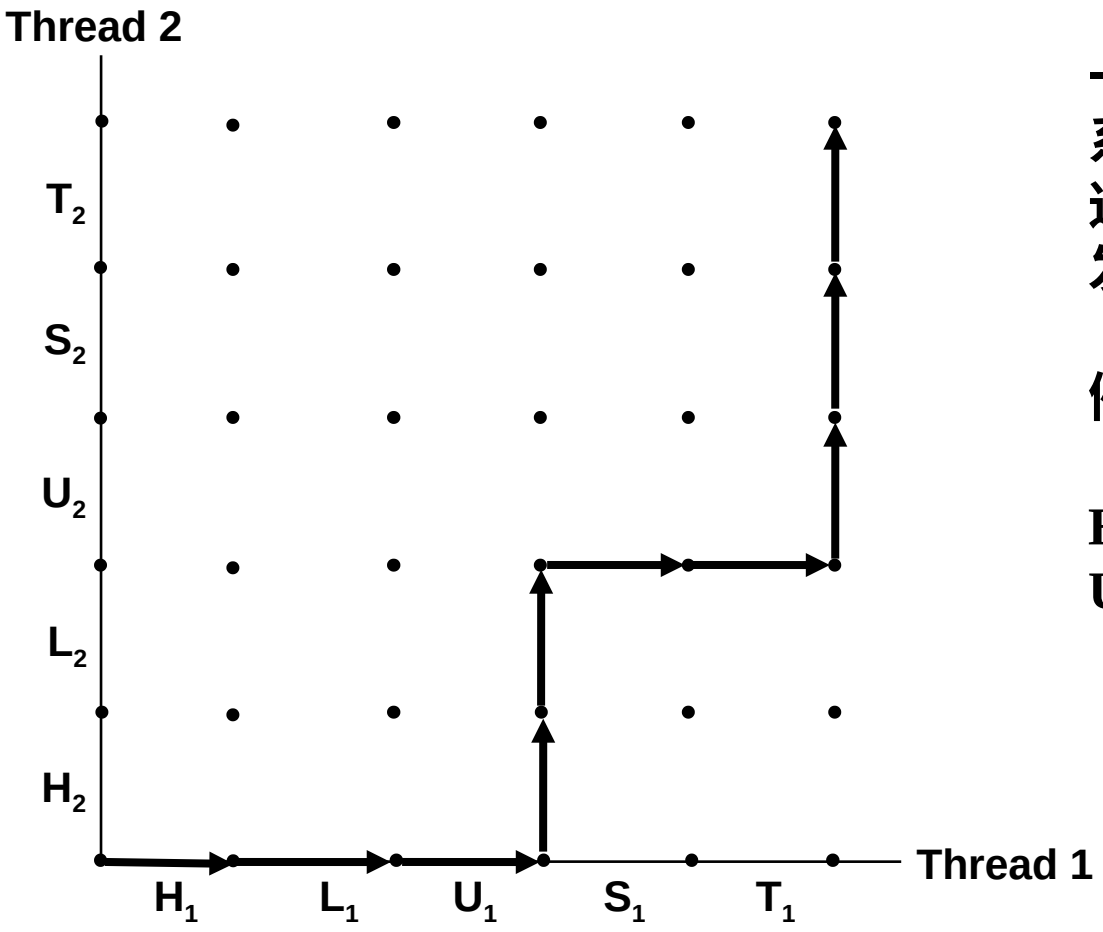
progress graph 将并发线程的离散执行状态空间 (execution state space) 可视化

每个轴对应于线程中指令的先后顺序

每个点对应一个可能的执行状态 $(Inst_1, Inst_2)$

例如 (L_1, S_2) 表示线程 1 完成了 L_1 、线程 2 完成了 S_2

Trajectories in progress graphs

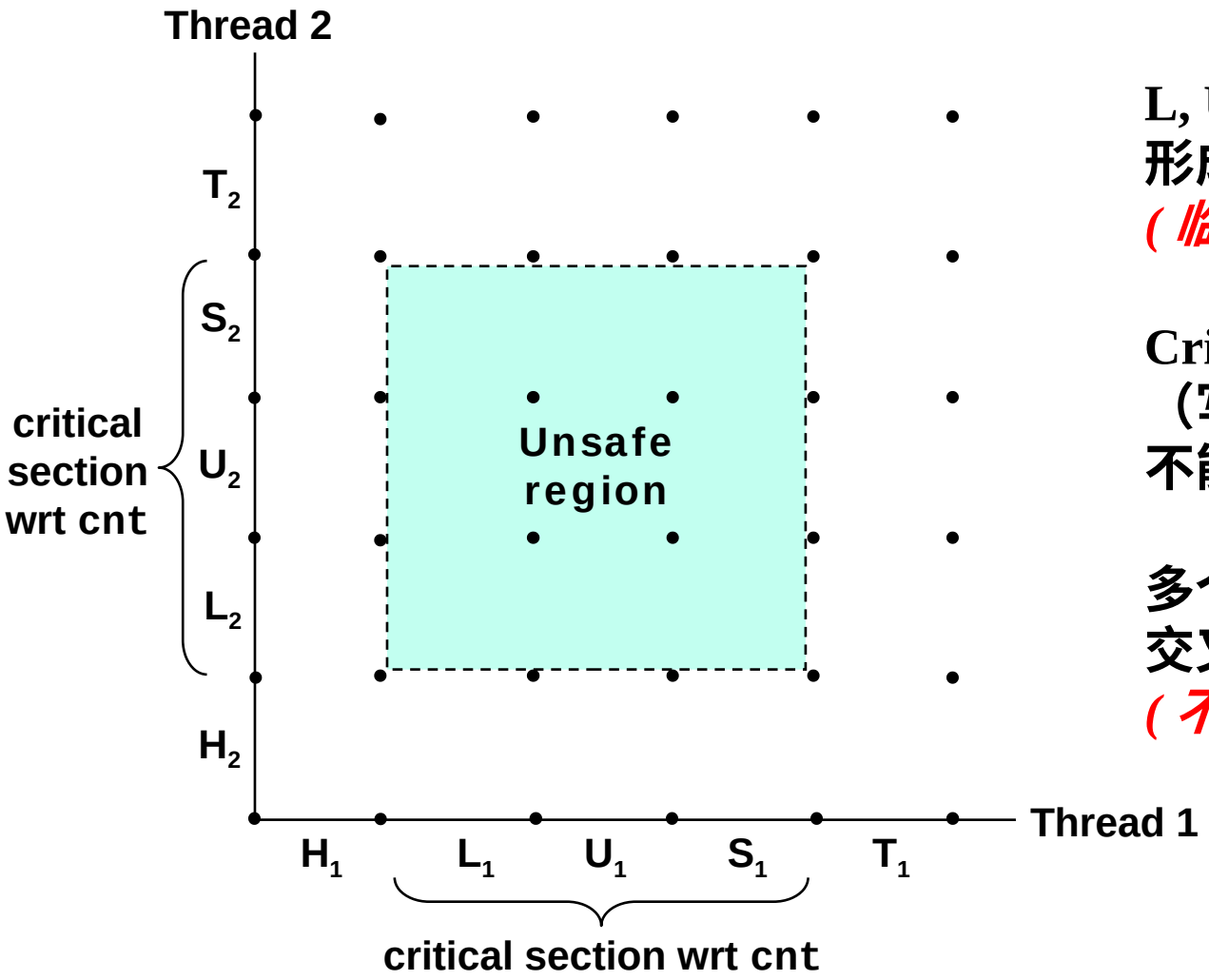


一条路径 (trajectory) 是一系列合法的状态转移，描述了线程的一种可能的并发执行过程

例如：

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical sections and unsafe regions

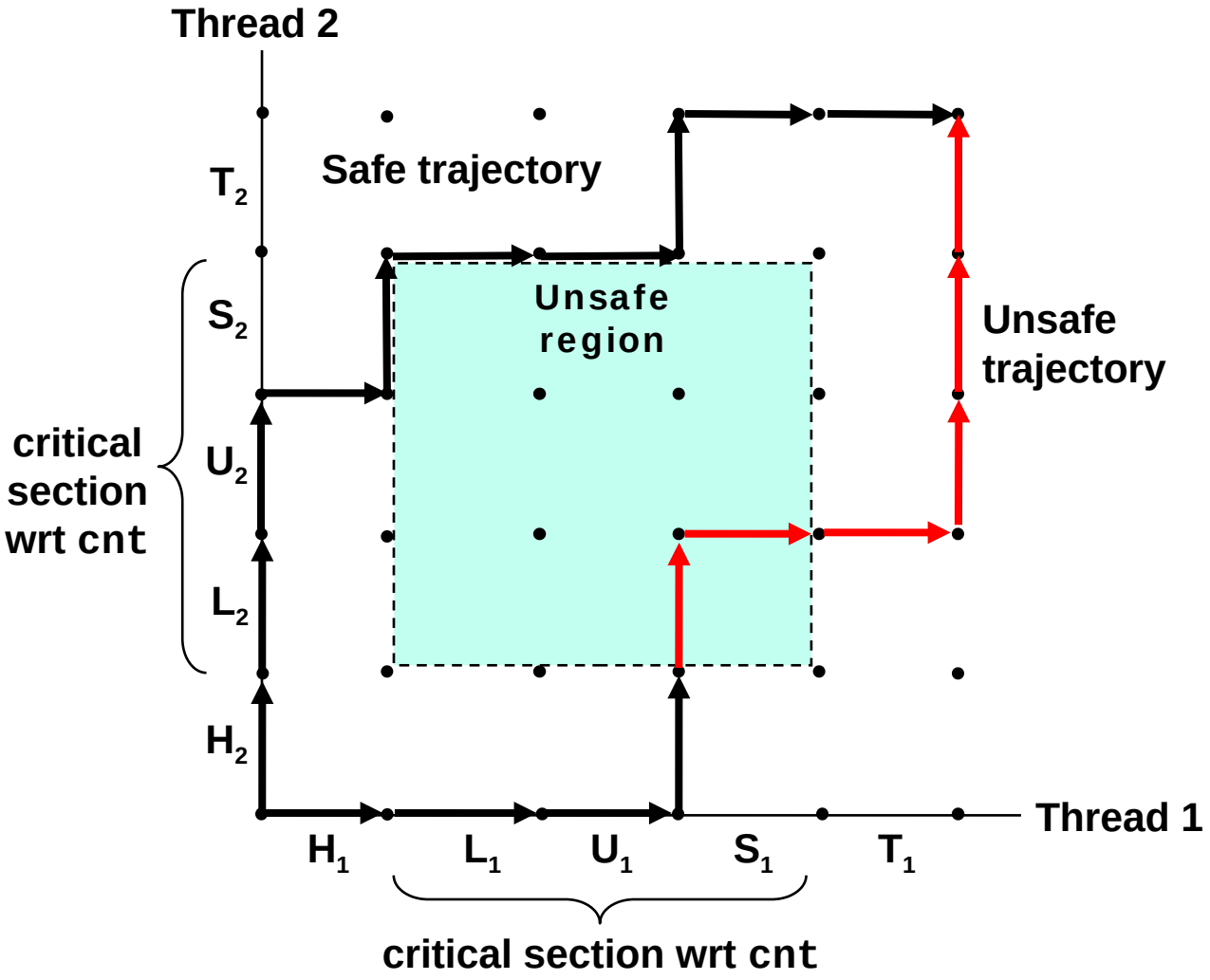


L, U, 和 S 针对共享变量 cnt 形成了一个 *critical section (临界区)*

Critical section 中的指令（写相同的共享变量）不能交错执行 (interleave)

多个线程的 critical section 交叉构成 *unsafe regions (不包含外周)*

Safe and unsafe trajectories



定义：一条路径是安全的
iff 它不触及任何的
unsafe region

断言：一条路径是正确的
iff 它是安全的

我们要编写正确的并发程序，就要保证所有的轨迹都是安全

Semaphores (信号量)

- Dijkstra's P and V operations on semaphores
  - semaphore*:* 非负的整型全局变量
  - 信号量上的原子操作 :
    - **P(s): [while (s == 0) wait(); s--;]**
      - 荷兰语 "Proberen" (test)
    - **V(s): [s++;]**
      - 荷兰语 "Verhogen" (increment)

Dutch Tomatoes

POSIX semaphores

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */

#include “csapp.h”

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_wait */
```

sem_init 的第二个参数为 0 表示信号量在线程间共享（需要放到线程可共享的内存区域），为非 0 表示信号量在进程间共享（需要放到共享内存中）

Sharing with POSIX semaphores

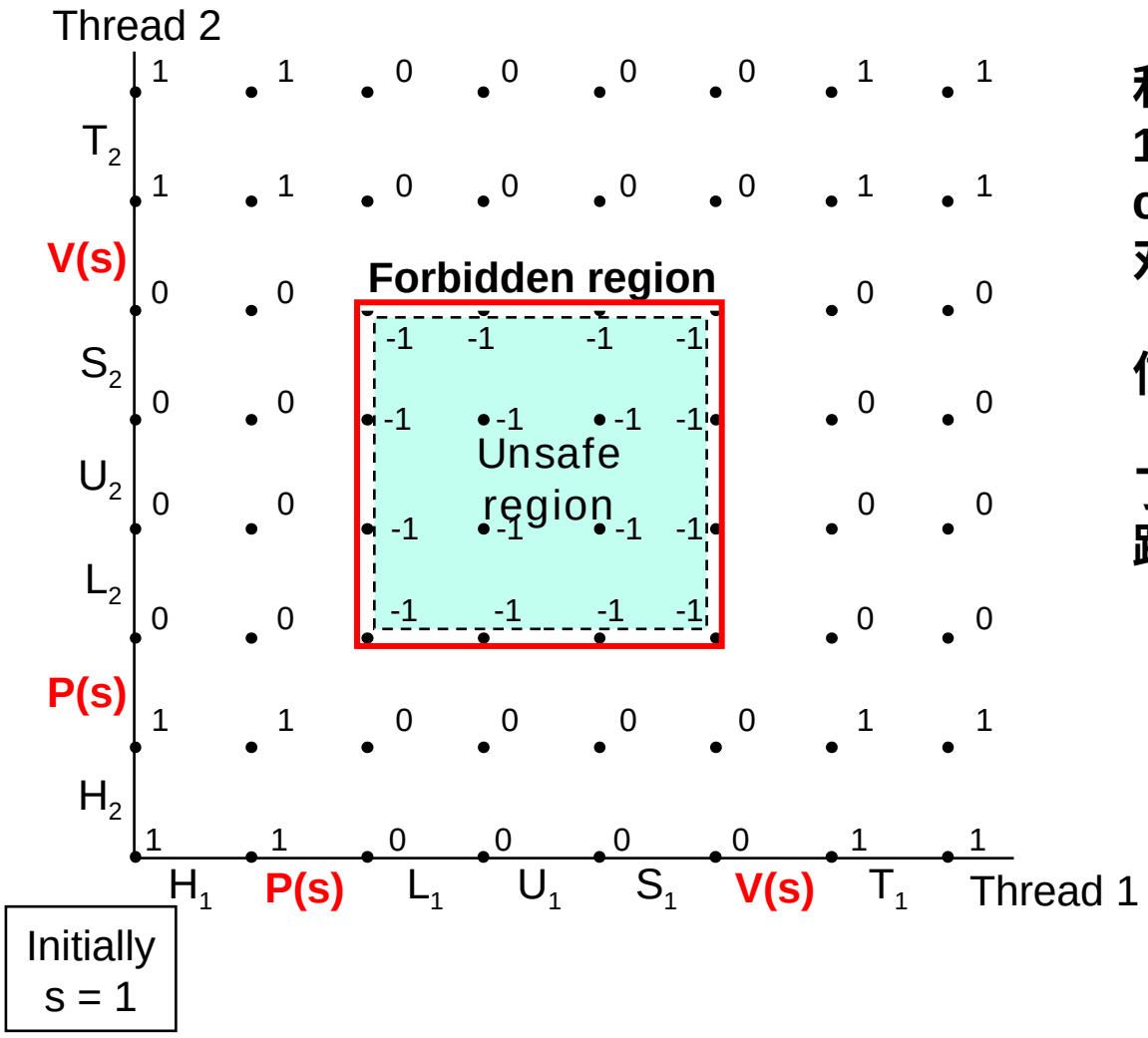
```
#include "csapp.h"
#define NITERS 10000000
unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;
    Sem_init(&sem, 0, 1);
    /* create 2 threads and wait */
    ...
    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}
```

Sharing with POSIX semaphores

```
/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

为每个（组）共享变量创建一个 semaphore s，并用 P(s) 和 V(S) 包围这个（组）变量的 critical section
这种 binary semaphore 也叫 mutex,P=locking, V=unlocking

Safe sharing with semaphores



利用对信号量 s （初始值为 1）的 P 、 V 操作可以保卫 critical section ，这样提供对共享变量的互斥访问

信号量创造了一个禁止区域（ s<0 ），这个禁止区保护了不安全区域，从而让任何路径都不能进入这个区域

Issues: Deadlock

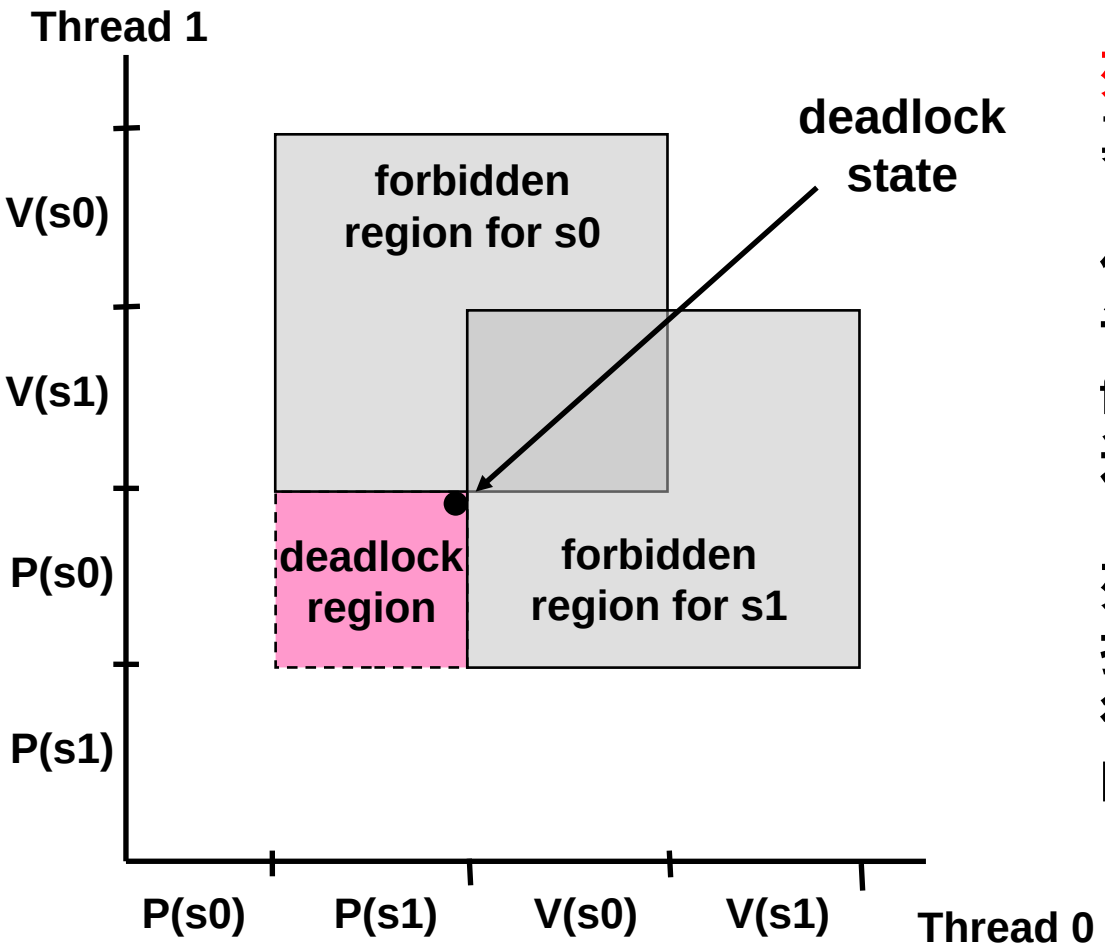
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*)0);
    Pthread_create(&tid[1], NULL, count, (void*)1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

Issues: Deadlock

Thread0	Thread1
Tid[0]:	Tid[1]:
$P(s_0)$;	$P(s_1)$;
$P(s_1)$;	$P(s_0)$;
cnt++;	cnt++;
$V(s_0)$;	$V(s_1)$;
$V(s_1)$;	$V(s_0)$;

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Issues: Deadlock



Initially, s0=s1=1

死锁：等待一个永远不会为真的条件

任何进入**死锁区域**的路径无论怎么走都会撞上forbidden region，一定会达到死锁状态

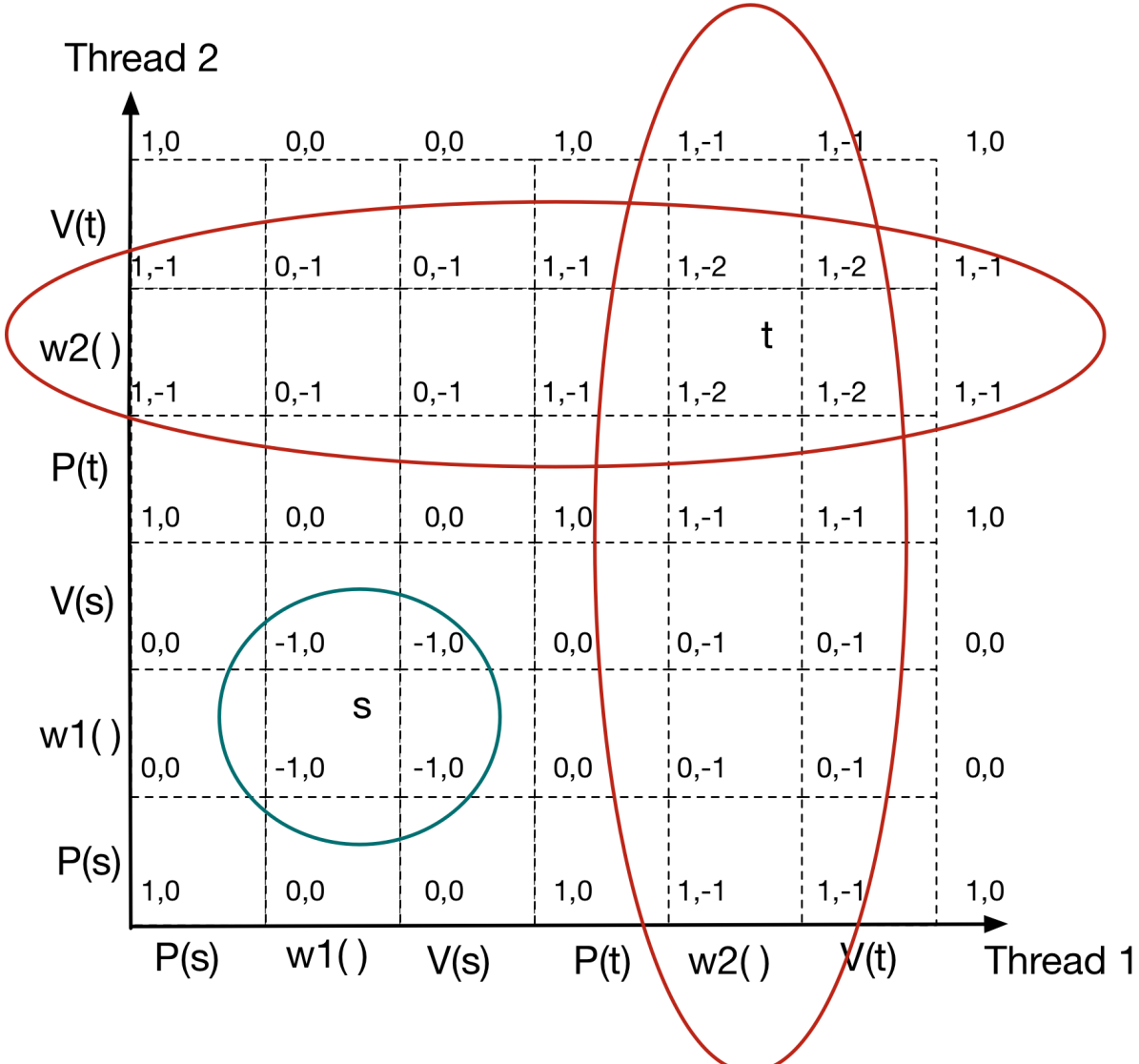
死锁的检测、预防、解决是操作系统、数据库系统、并行和分布式计算系统中重要的研究方向

课堂练习

- 信号量 s 和 t，初始状态 s=1, t=0
- 有两个线程，工作流程都是：

  P(s)， w1()， V(s)， P(t)， w2()， V(t)
- A. 画出这个程序的进展图（标记禁止区）
- B. 它会死锁吗?
- C. 如果是，那么对初始信号量的值做哪些改变就能消除潜在的死锁呢?
- D. 画出无死锁程序的进展图

练习答案


Thread 2
V(t)
w2( )
P(t)
V(s)
w1( )
P(s)
1,0
0,0
0,0
1,0
1,-1
1,-1
1,0
1,-1
0,-1
0,-1
1,-1
1,-2
1,-2
1,-1
t
1,-1
0,-1
0,-1
1,-1
1,-2
1,-2
1,-1
1,0
0,0
0,0
1,0
1,-1
1,-1
1,0
0,0
-1,0
-1,0
0,0
0,-1
0,-1
0,0
s
0,0
-1,0
-1,0
0,0
0,-1
0,-1
0,0
1,0
0,0
0,0
1,0
1,-1
1,-1
1,0
P(s)
w1( )
V(s)
P(t)
w2( )
V(t)
Thread 1

课堂练习

- 为什么没有输出?

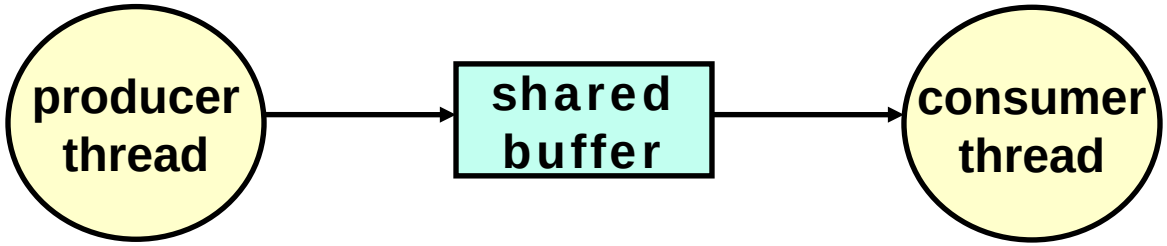
code/conc/hellobug.c

```
1   /* WARNING: This code is buggy! */
2   #include "csapp.h"
3   void *thread(void *vargp);
4
5   int main()
6   {
7       pthread_t tid;
8
9       Pthread_create(&tid, NULL, thread, NULL);
10      exit(0);
11  }
12
13  /* Thread routine */
14  void *thread(void *vargp)
15  {
16      Sleep(1);
17      printf("Hello, world!\n");
18      return NULL;
19  }
```

code/conc/hellobug.c

并发：多线程并发程序实例

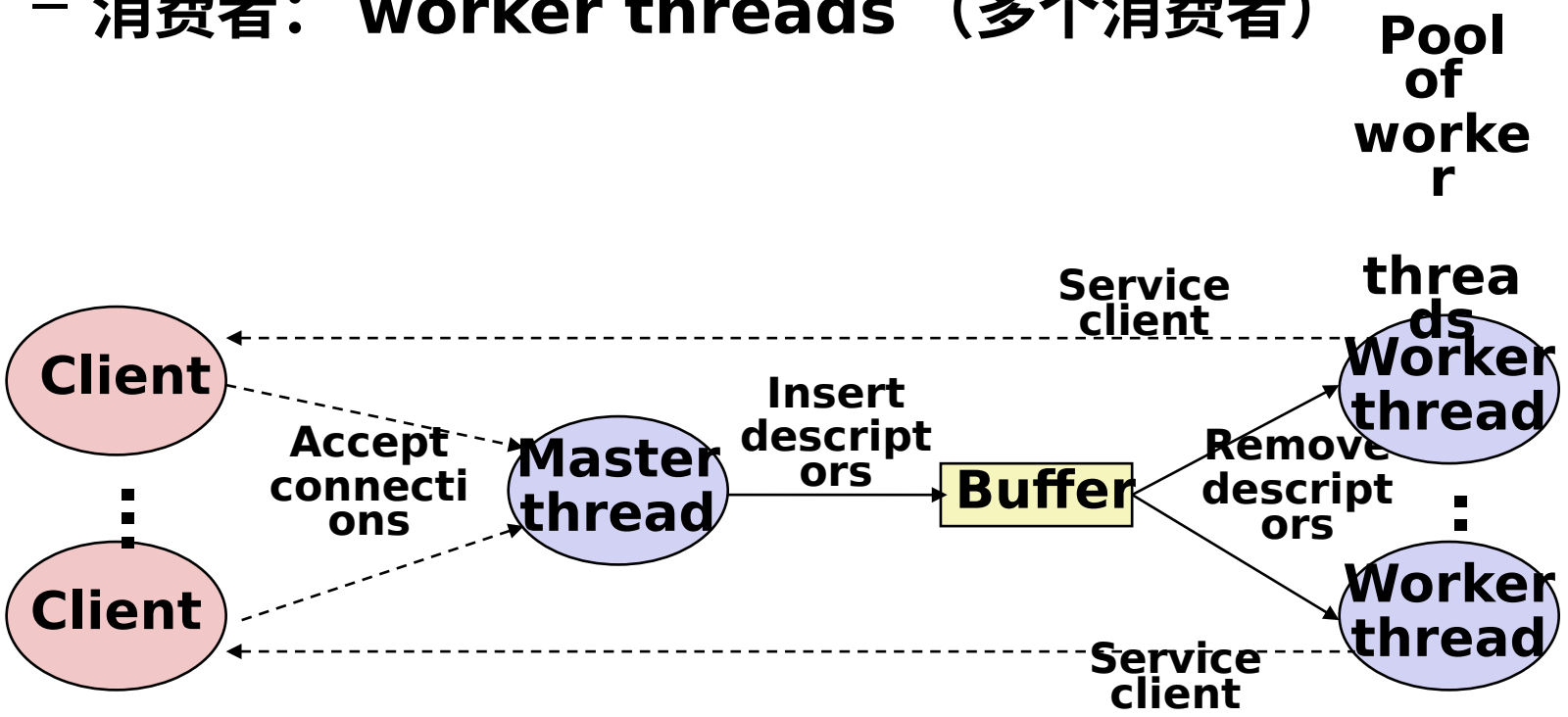
Producer-Consumer Problem



- “生产者 - 消费者”同步模式:
  - 生产者等待空的 slot，插入 shared buffer，并通知消费者
  - 消费者等待 shared buffer 中的 item，从 buffer 中删除，并通知生产者

Case Study: A Concurrent Server

- 生产者： master thread
- 消费者： worker threads （多个消费者）



sbuf Package - Declarations

```
struct {
  int *buf;     /* Buffer array */
  int n;        /* Maximum number of slots */
  int front;    /* buf[(front+1)%n] is first item */
  int rear;     /* buf[rear%n] is last item */
  sem_t mutex;  /* protects accesses to buf */
  sem_t slots;  /* Counts available slots */
  sem_t items;  /* Counts available items */
} sbuf_t;
```

sbuf Package - Implementation

```
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    /* Buffer holds max of n items */
    sp->n = n;
    /* Empty buffer iff front == rear */
    sp->front = sp->rear = 0;
    /* Binary semaphore for locking */
    Sem_init(&sp->mutex, 0, 1);
    /* Initially, buf has n empty slots */
    Sem_init(&sp->slots, 0, n);
    /* Initially, buf has zero data items */
    Sem_init(&sp->items, 0, 0);
}
```

sbuf Package - Implementation

```
void sbuf_insert(sbuf_t *sp, int item)
{
    /* Wait for available slot */
    P(&sp->slots);
    /*Lock the buffer */
    P(&sp->mutex);
    /*Insert the item */
    sp->buf[(++sp->rear)%(sp->n)] = item;
    /* Unlock the buffer */
    V(&sp->mutex);
    /* Announce available items*/
    V(&sp->items);
}
```

sbuf Package - Implementation

```
void sbuf_remove(sbuf_t *sp)
{
    int item;
    /* Wait for available item */
    P(&sp->items);
    /*Lock the buffer */
    P(&sp->mutex);
    /*Remove the item */
    item = sp->buf[(++sp->front)%(sp->n)];
    /* Unlock the buffer */
    V(&sp->mutex);
    /* Announce available slot*/
    V(&sp->slots);
    return item;
}
```

Prethreading

```
#define NTHREADS  4
#define SBUFSIZE  16

/* shared buffer of connected descriptors */
sbuf_t sbuf;

int main(int argc, char **argv)
{
    int i, listenfd, connfd, port;
    int clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    pthread_t tid;
```

Prethreading

```
    port = atoi(argv[0]);
    sbuf_init(&sbuf, SBUFSIZE);
    listenfd = open_listenfd(port);
    /* Create worker threads */
    for (i = 0; i < NTHREADS; i++)
        Pthread_create(&tid, NULL, thread, NULL);

    while (1) {
        connfd = Accept(listenfd,
                (SA *)&clientaddr, &clientlen);
        /* Insert connfd in buffer */
        sbuf_insert(&sbuf, connfd);
    }
}
```

Prethreading

```
void *thread(void *vargp)
{
   Pthread_detach(pthread_self());
   while (1) {
      int connfd = sbuf_remove(&sbuf);
/* Remove connfd from buffer */
      echo_cnt(connfd);
/* Service client */
      Close(connfd);
   }
}
```

生产者消费者模型最常见的应用是消息队列

课堂练习

- 假设 p 表示生产者数量， c 表示消费者数量， n 表示缓冲区大小。对于下面的每个场景，指出 sbuf_insert 和 sbuf_remove 中的互斥锁信号量 mutex 是否为必需的。
  - p=1, c=1, n=1
  - p>1, c>1, n=1
  - p=1, c=1, n>1

Readers-Writers Problem

- 问题描述：
  - *Reader threads* 只是读
  - *Writer threads* 只是写
  - 写操作是排他的（包括读、写）
  - 读操作之间可以并行，而且没有数量限制

- 在实际系统经常发生，例如
  - 航空在线预订系统
  - 多线程缓存的 web 代理

实际上，读者写者问题最常见的应用是数据库系统

Variants of Readers-Writers

- *First readers-writers problem* (favors readers)
  - 除非已经有一个 write 正在写，否则读操作都会被允许
  - 新来的 reader 的优先级比正在等待的 writer 还要高
- *Second readers-writers problem* (favors writers)
  - 与上述读优先完全相反
- *以上都有可能产生 Starvation* (where a thread waits indefinitely)

主要区别：读写都在等待时，优先执行谁

读优先的读者写者问题

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
   while (1) {
            P(&mutex);
            readcnt++;
            if (readcnt == 1) /* First in */
                     P(&w);
            V(&mutex);

            /* Reading happens here */

            P(&mutex);
            readcnt--;
            if (readcnt == 0) /* Last out */
                     V(&w);
            V(&mutex);
   }
}
```

Readers

```
void writer(void)
{
   while (1) {
            P(&w);

            /* Writing here */

            V(&w);
   }
}
```

Writers

rw1.c

读写平衡的读写者问题（最多同时 N 个读者）

Readers

```
sem_t w; /* Initially 1 */
sem_t common_sem; /* Initially N */

void reader(void)
{
   while (1) {
            P(&common_sem);

            /* Reading happens here */

            V(&common_sem);
   }
}
```

Writers

```
void writer(void)
{
   while (1) {
            int i;
            P(&w);
            for (i=0;i<N;i++)
                     P(&common_sem);
            V(&w);

            /* Writing happens here */

            for (i=0;i<N;i++)
                     V(&common_sem);
   }
}
```

写优先的读写者问题

```
int readcnt, writecnt;    /* Initially 0 */
sem_t r, w, rcnt, wcnt; /* All initially 1 */

void reader(void)
{
   while (1) {
            P(&r);
            P(&rcnt);
            readcnt++;
            if (readcnt == 1) /* First in */
                     P(&w);
            V(&rcnt);
            V(&r);

            /* Reading happens here */

            P(&rcnt);
            readcnt--;
            if (readcnt == 0) /* Last out */
                     V(&w);
            V(&rcnt);
   }
}
```

Readers

```
void writer(void)
{
   while (1) {
            P(&wcnt);
            writecnt++;
            if (writecnt == 1)
                     P(&r);
            V(&wcnt);

            P(&w);
            /* Writing happens here */
            V(&w);

            P(&wcnt);
            writecnt--;
            if (writecnt == 0) /* Last out */
                     V(&r);
            V(&wcnt);
   }
}
```

Writers

Starvation

- 在第一个例子中， readers 会阻塞 writer
  - Writer 可能一直得不到响应，发生饥饿
- 我们需要保证 readers 不会长时间一直在执行

线程安全 (Thread-safe) 函数

- 线程中所调用的函数必须是 thread-safe 的
- 定义： *一个函数是* thread-safe 的 iff 它对来自并发线程的反复调用总是产生正确的结果

Thread-safe 函数

- 线程不安全的函数分为四个类型:
  - 类型 1： 不能保护共享变量
  - 类型 2： 多次函数调用之间依赖同样的持久状态
  - 类型 3： 返回一个指向静态变量的指针
  - 类型 4： 调用其他线程不安全的函数

Thread-unsafe functions

- 类型 1：不能保护共享变量
  - 解决方法：使用 Pthreads P/V 操作
    - 不需要修改调用代码（只需要改函数内部代码）
  - 问题：同步操作会使运行变慢

Thread-unsafe functions (case 2)

- 类型 2： 多次函数调用之间依赖同样的持久状态
  - 例如 rand 函数（本质是还是修改共享变量；不同点是实际多线程之间本来不需要交互）

```
unsigned int next = 1;
/* rand – return pseudo-random int on 0..32767 */
int rand(void)
{
    next = next * 110351524 + 12345 ;
    return (unsigned int)((next/65536) % 32768);
}
/* srand – set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Thread-unsafe functions (case 2)

- 把依赖的全局 / 静态变量变为参数不断传递来记录状态
  - 消除了全局 / 静态变量
  - 很类似函数式编程的思路（ Erlang 等）

```
/* rand_r – return pseudo-random int on 0..32767 */
int rand_r(int *nextp)
{
    *nextp = *nextp * 110351524 + 12345;
    return (unsigned int)((*nextp/65536) % 32768);
}
```

Thread-safe functions

- 类型 3： 返回一个指向静态变量的指针

```
struct hostent *gethostbyname(char name)
{
    static struct hostent host;
    <contact DNS and fill in host>
    return &host;
}
```

Thread-safe functions

- 类型 3 解决方法
  - 方法 1： 重写函数，调用者传递存放结果的地址
    - 问题 : caller 和 callee 的代码都需要修改
  - 方法 2： “Lock-and-copy”
    - 问题 : caller 必须记得释放内存（容易忘）

```
hostp = Malloc(...));
gethostbyname_r(name, hostp);
```

```
struct hostent *gethostbyname_ts(char *name)
{
    struct hostent *p,*q = Malloc(...);
    P(&mutex);
    p = gethostbyname(name);
    *q = *p;
    V(&mutex);
    return q;
}
```

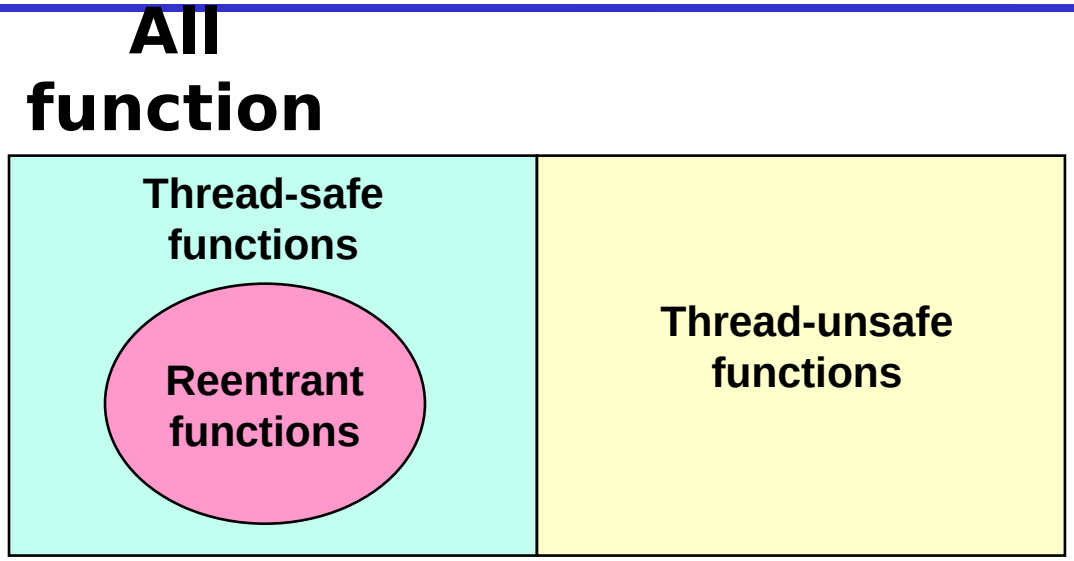
Thread-safe functions

- 类型 4：调用其他线程不安全的函数
  - 解决方法：只调用线程安全函数

可重入 (Reentrant) 函数

- 一个函数是 reentrant 的 iff 它被多线程调用时不访问任何共享变量
  - Reentrant 函数是线程安全函数的一个重要子集
  - 不需要同步操作

Reentrant 函数



NOTE: The fixes to Class 2 thread-unsafe functions require modifying the function to make it reentrant

Thread-safe 库函数

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe.
- Most Unix system calls are thread-safe, with a few exceptions:

Thread-unsafe function	Class	Reentrant version
rand	2	rand_r
Strtok	2	strtok_r
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r

课堂练习

- 说明以下函数是否 reentrant 或 thread-safe.
- int t;
- void swap1(int *x, int *y) {
  t = *x; *x = *y; *y = t; }
- void swap2(int *x, int *y) {
  P(&mutex); t = *x; *x = *y; *y = t; V(&mutex); }
- void swap3(int *x, int *y) {
  int t; t = *x; *x = *y; *y = t; }

Issues: Races（竞争、争用）

如果程序的正确性依赖于 一个线程先到达 x 、另一个线程后到达 y ，两个线程之间发生了竞争 (race)

```
/*thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp) ;
    printf("Hello from th. %d\n", myid);
    return NULL ;
}
```

```
#define N 4
int main()
{
    pthread_t tid[N];
    int i ;
    for ( i=0 ; i<N ; i++ )
       pthread_create(&tid[i], NULL, thread, &i);
    for ( i=0 ; i<N ; i++ )
       pthread_join(tid[i], NULL) ;
    exit(0) ;
}
```

N threads are sharing i

main thd 和 peer thds 之间存在 races

Issues: Races（竞争、争用）

竞争的本质是多个线程之间存在预期之外的变量共享，或者多线程对共享变量的访问顺序在预期之外

```
#define N 4
int main()
{
    pthread_t tid[N];
    int i ;
    for ( i=0 ; i<N ; i++ )
       pthread_create(&tid[i], NULL, thread, &i);
    for ( i=0 ; i<N ; i++ )
       pthread_join(tid[i], NULL) ;
    exit(0) ;
}
```

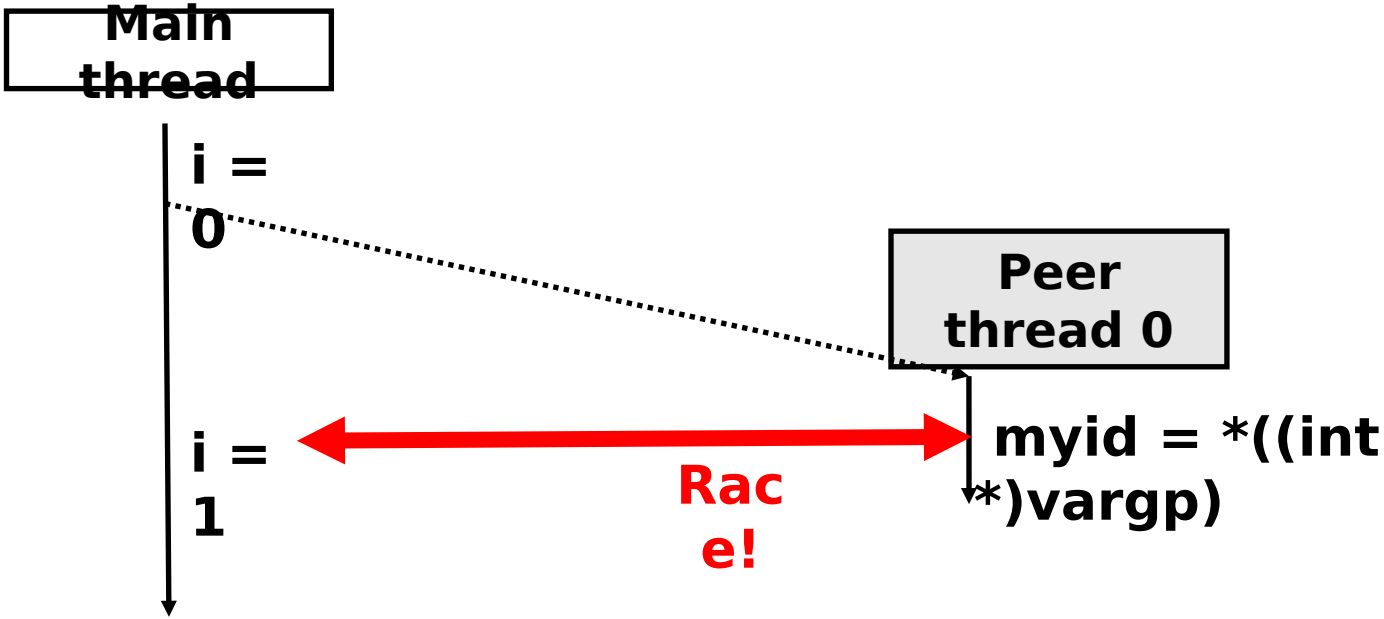
```
/*thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp) ;
    printf("Hello from th. %d\n", myid);
    return NULL ;
}
```

N threads are sharing i

main thd 和 peer thds 之间存在 races

Race Illustration

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, &i);
```



- main thread 中的 i++ 和 peer thread 中的对 vargp 的解引用形成了 race:
  - 如果解引用发生时 i = 0，则 OK
  - 否则，peer thread 会得到错误的 myid

这样的竞争是否会发生？

Main thread

```
int i;
for (i = 0; i < 100; i++) {
    Pthread_create(&tid, NULL,
                   thread,&i);
}
```

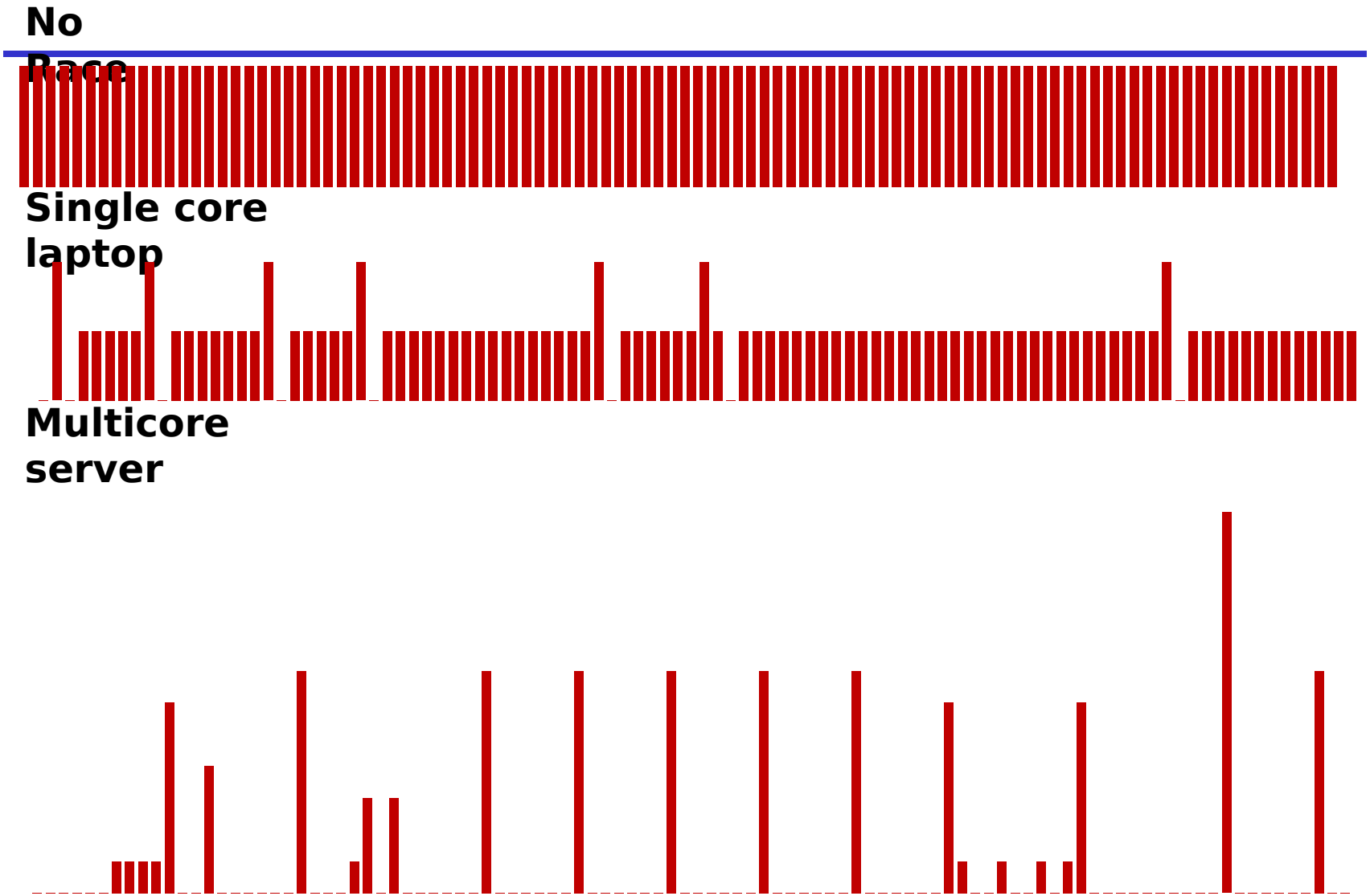
Peer thread

```
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    int i = *((int *)vargp);
    save_value(i);
    return NULL;
}
```

race.c

- 竞争测试
  - 如果没有竞争，则没有线程都会获得不同的 i 值
  - 那么将保存 0-99 之间的所有值，且没有重复

Experimental Results



- The race can really happen!

Race Elimination

- 避免非预期的状态共享

```
int main()
{
    pthread_t tid[N];
    int i, *ptr ;
    for ( i=0 ; i<N ; i++ ) {
       ptr = malloc((sizeof)int); //freed in thread
       *ptr = i ;
       pthread_create(&tid[i], NULL, thread, ptr);
    }
    for ( i=0 ; i<N ; i++ )
       pthread_join(tid[i], NULL) ;
    exit(0) ;
}
```

活锁（Live Lock）

- 类似于死锁问题，两个 thread 各自持有对方想要的锁
- 当一个 thread 意识到它不能获得下一个需要获得的锁时，它会释放已获得的锁，并等待 1ms 再尝试一次
- 但是，如果两个 thread 都这样做，还是无法解决问题
  - 类似于一条路上相遇的两个人都在让路
- 例子
  - Fork 创建进程，但系统进程数量有上限；fork 失败会等待一段时间再重试
    - 假设系统最多 100 个进程， 10 个进程，每个要创建 12 个子进程
    - 每个进程创建到 9 个后失败，都放弃前面创建的进程，等待重试
  - 分布式系统选举 leader
    - 都申请当 leader ，无法获得半数以上选票

课堂练习

A. Does the following program contain a race on the value of `myid`? Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    Free(vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i, *ptr;

    for (i = 0; i < 2; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], 0, foo, ptr);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

课堂练习

B. Does the following program contain a race on the value of `myid`? Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], NULL, foo, &i);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
}
```

C. Does the following program contain a race on the value of `myid`? Yes No

```
void *foo(void *vargp) {
    int myid;
    myid = (int)vargp;
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    for (i = 0; i < 2; i++)
        Pthread_create(&tid[i], 0, foo, i);
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

课堂练习

D. **Does the following program contain a race on the value of `myid`?** Yes No

```
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    P(&s);
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 1); /* S=1 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

课堂练习

E. **Does the following program contain a race on the value of `myid`?** Yes No

```
sem_t s; /* semaphore s */

void *foo(void *vargp) {
    int myid;
    myid = *((int *)vargp);
    V(&s);
    printf("Thread %d\n", myid);
}

int main() {
    pthread_t tid[2];
    int i;

    sem_init(&s, 0, 0); /* S=0 INITIALLY */

    for (i = 0; i < 2; i++) {
        Pthread_create(&tid[i], 0, foo, &i);
        P(&s);
    }
    Pthread_join(tid[0], 0);
    Pthread_join(tid[1], 0);
}
```

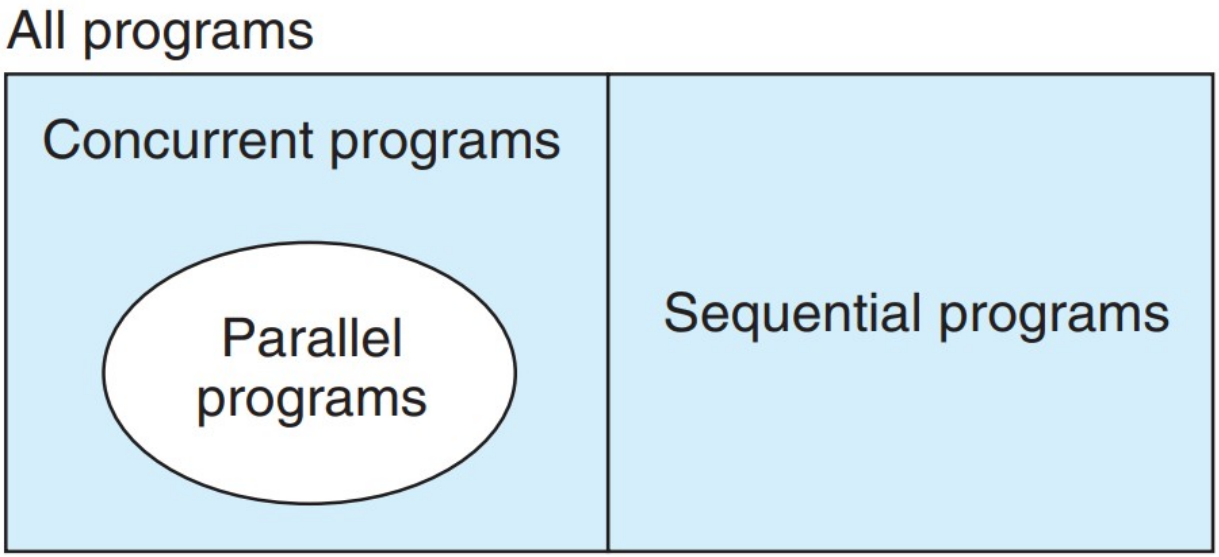
Thread-Level Parallelism

Outline

- Parallel Computing Hardware
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Efficient execution of multiple threads on single core
- Thread-Level Parallelism
  - Splitting program into independent tasks
    - Example 1: Parallel summation
  - Divide-and-conquer parallelism
    - Example 2: Parallel quicksort
- Consistency Models
  - What happens when multiple threads are reading & writing shared state

Concurrent VS. Parallel

- 之前，我们假设多个线程运行在单个 CPU 核心上
  - 这种情况可以加速 I/O 操作
- 但当多线程程序运行在多核计算机上时， OS Kernel 会将不同线程调度到不同的核上执行
  - 这时，多线程可以显著加速计算密集型任务



Example 1: Parallel Summation

- Sum numbers *0, …, n-1*
  - Should add up to *((n-1)*n)/2*
- Partition values *1, …, n-1* into *t* ranges
  - *n/t* values in each range
  - Each of *t* threads processes 1 range
  - For simplicity, assume *n* is a multiple of *t*

- Let's consider different ways that multiple threads might work on their assigned ranges in parallel

First attempt: psum-mutex

- 最简单的方法：所有线程都向一个由 mutex 保护的全局变量上累加

```
void *sum_mutex(void *vargp); /* Thread routine */

/* Global shared variables */
long gsum = 0;            /* Global sum */
long nelems_per_thread;   /* Number of elements to sum per thread*/
sem_t mutex;              /* Mutex to protect global sum */

int main(int argc, char **argv)
{
    long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
    pthread_t tid[MAXTHREADS];

     /* Get input arguments */
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems);
    nelems_per_thread = nelems / nthreads;
    sem_init(&mutex, 0, 1);
```

psum-mutex.c

psum-mutex (cont)

- 最简单的方法：所有线程都向一个由 mutex 保护的全局变量上累加

```
    /* Create peer threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {

        myid[i] = i;
        Pthread_create(&tid[i], NULL, sum_mutex, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
       Pthread_join(tid[i], NULL);

    /* Check final answer */
    if (gsum != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", gsum);

    exit(0);
}
```

psum-mutex.c

psum-mutex Thread Routine

- 最简单的方法：所有线程都向一个由 mutex 保护的全局变量上累加

```
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
    long myid = *((long *)vargp);           /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        P(&mutex);
        gsum += i;
        V(&mutex);
    }
    return NULL;
}
```

psum-mutex.c

`psum-mutex` Performance

- A machine with 8 cores, $n=2^{31}$

Threads (Cores)	1 (1)	2 (2)	4 (4)	8 (8)	16 (8)
psum-mutex (secs)	51	456	790	536	681

- **Nasty surprise:**
  - **Single thread is very slow**
  - **Gets slower as we use more cores**

Next Attempt: psum-array

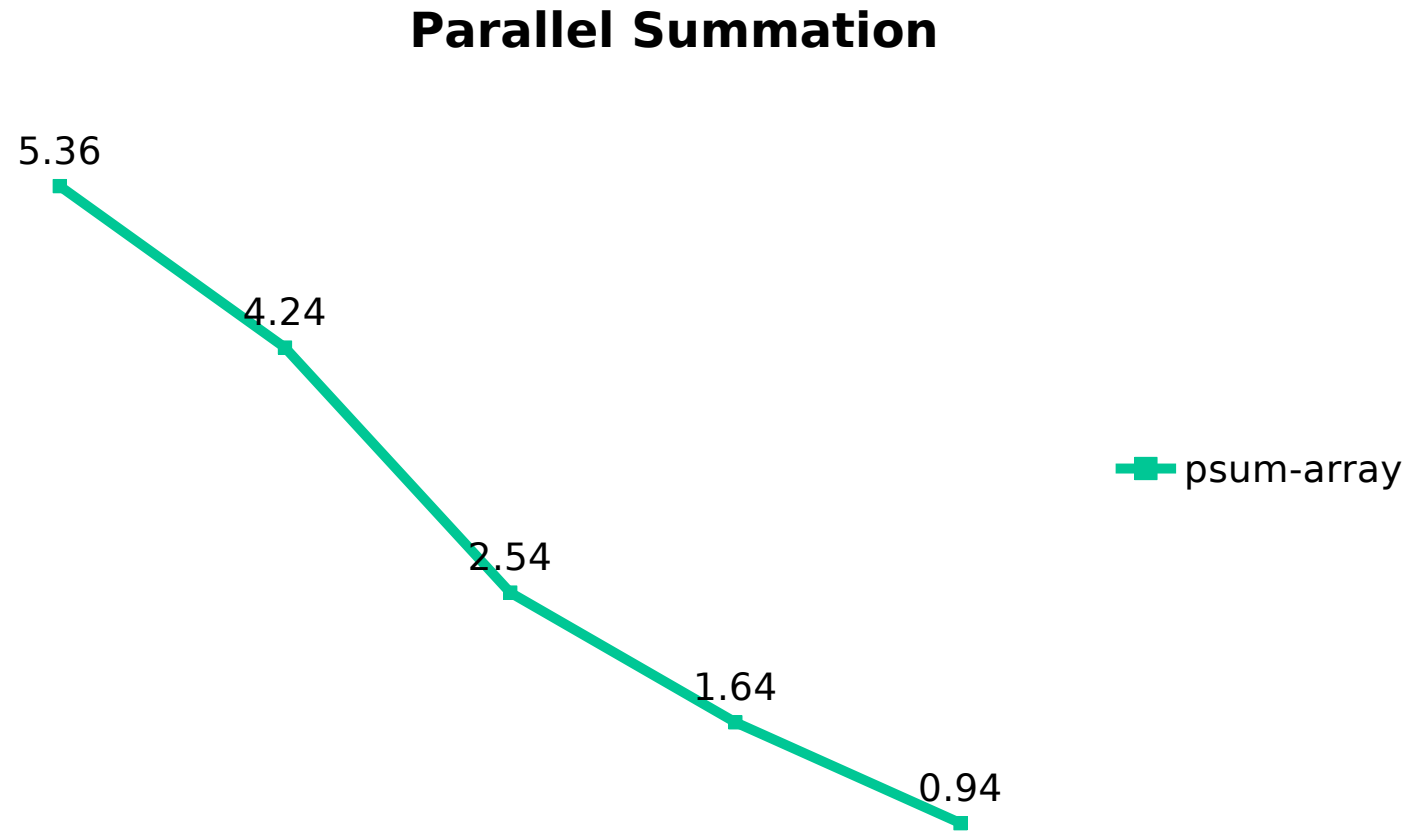
- Peer thread i 向一个全局数组的元素 psum[i] 上累加
- Main thread 等待所有 peer 完成，并对 psum 中的元素求和
- 消除了 mutex

```
/* Thread routine for psum-array.c */
void *sum_array(void *vargp)

{

    long myid = *((long *)vargp);           /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;



    for (i = start; i < end; i++) {
        psum[myid] += i;
    }
    return NULL;
```

psum-array.c

psum-array Performance

- 比 psum-mutex 快几个量级



Next Attempt: psum-local

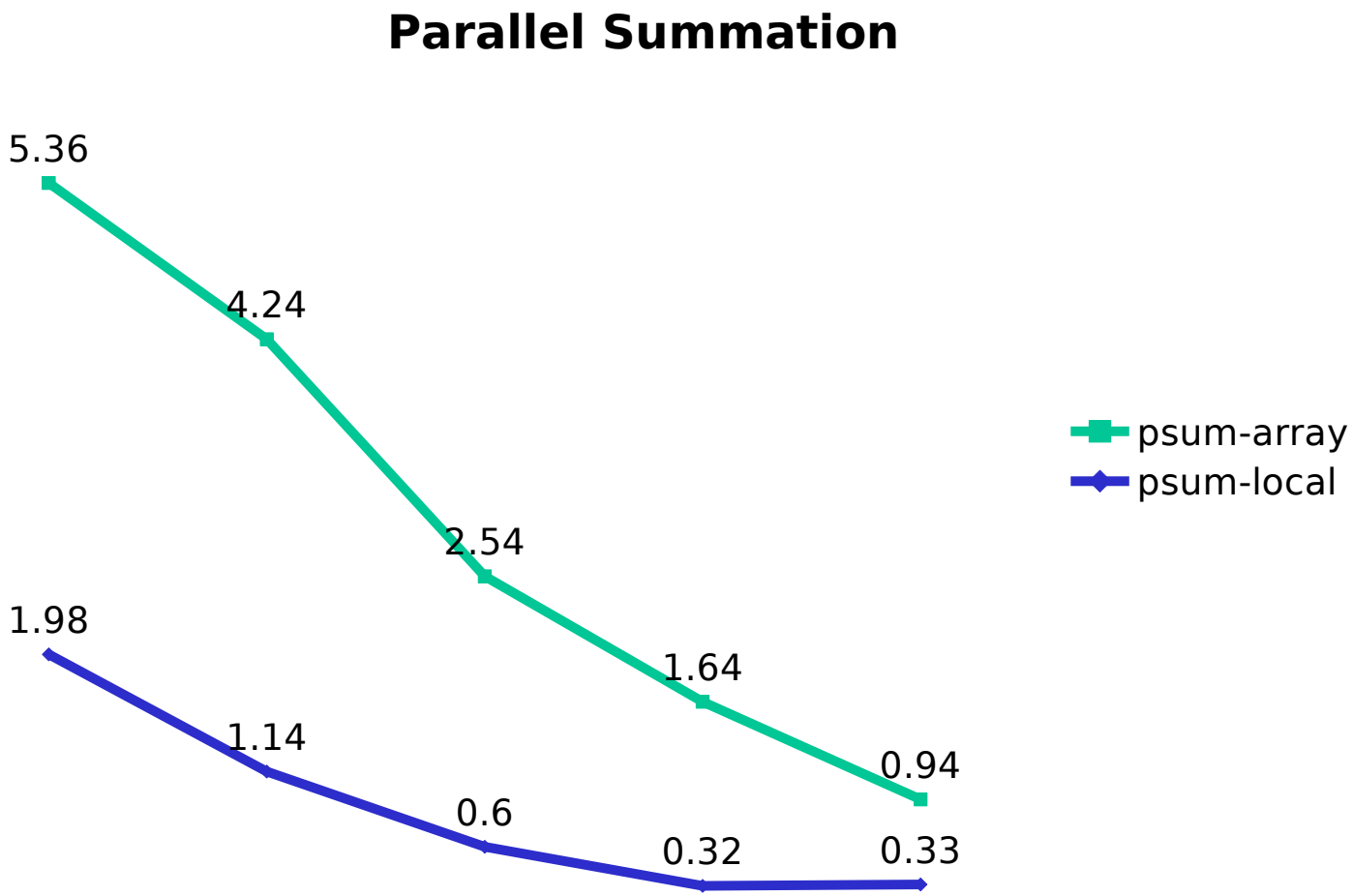
- 让 peer thread i 向一个局部变量进行累加，减少对全局内存的访问。局部变量容易用寄存器加速

```
/* Thread routine for psum-local.c */
void *sum_local(void *vargp)
{
    long myid = *((long *)vargp);           /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;
    return NULL;
}
```

psum-local.c

psum-local Performance

- 显著快于 psum-array



Characterizing Parallel Program Performance

- *总共有* p 个 CPU 核，T_k 是适用 k 个核的运行时间

- *Def.* *加速比 (speedup):* $S_p = T_1 / T_p$
  - S_p *是相对加速比，如果* T_1 是并行版本的程序在 1 个核上的运行时间
  - S_p 是*绝对加速比*，如果 T_1 是串行版本的程序在 1 个核上的运行时间
  - 绝对加速通常对于测量程序的并行加速效果更有意义

- *Def.* *效率 (Efficiency):* $E_p = S_p / p = T_1 / (pT_p)$
  - 一个 (0, 100] 之间的百分数
  - 可以测量并行带来的额外开销

Performance of `psum-local`

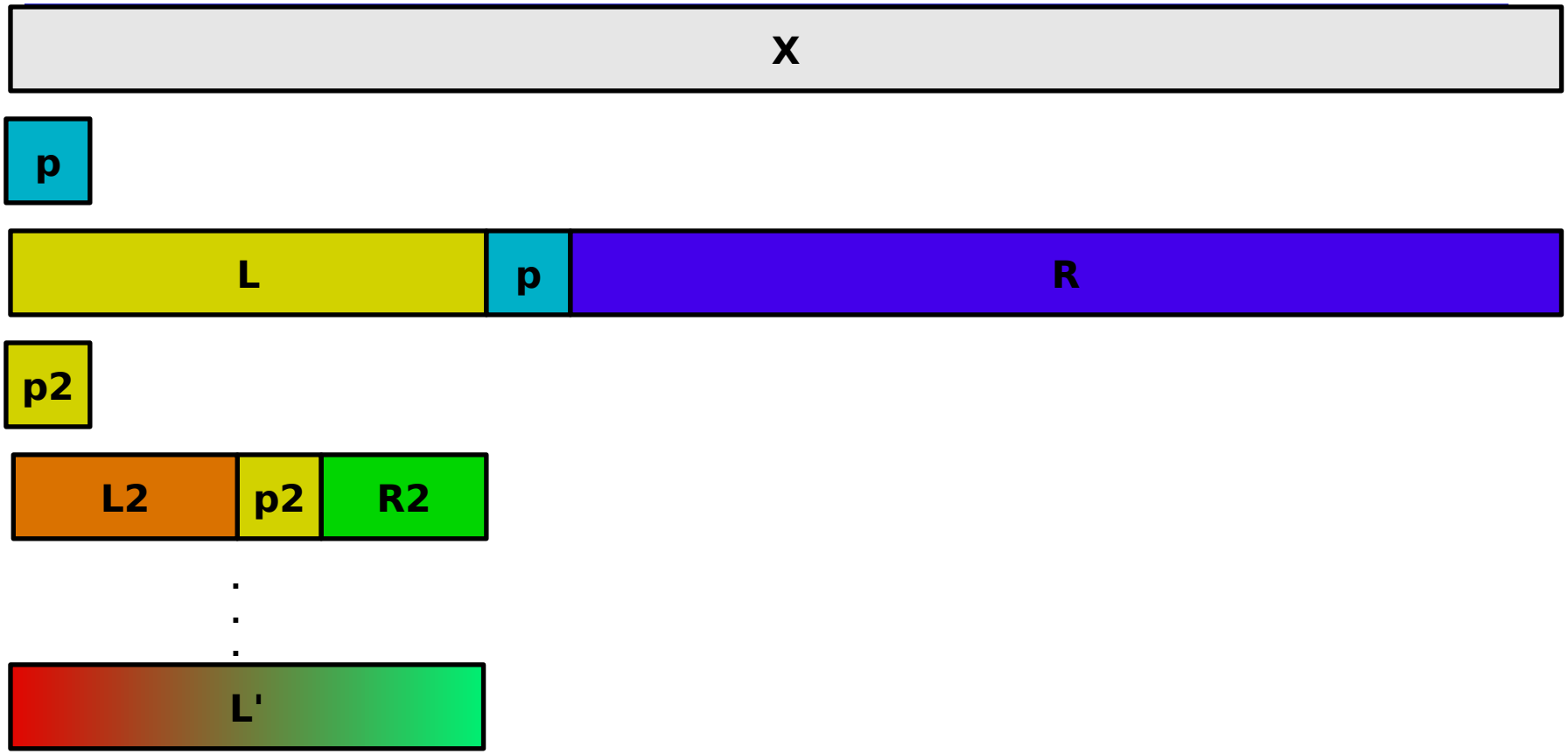
Threads (t)	1	2	4	8	16
Cores (p)	1	2	4	8	8
Running time (T_p)	1.98	1.14	0.60	0.32	0.33
Speedup (S_p)	1	1.74	3.30	6.19	6.00
Efficiency (E_p)	100%	87%	82%	77%	75%

- **Efficiencies OK, not great**
- **Our example is easily parallelizable**
- **Real codes are often much harder to parallelize**
  - **e.g., parallel quicksort later in this lecture**

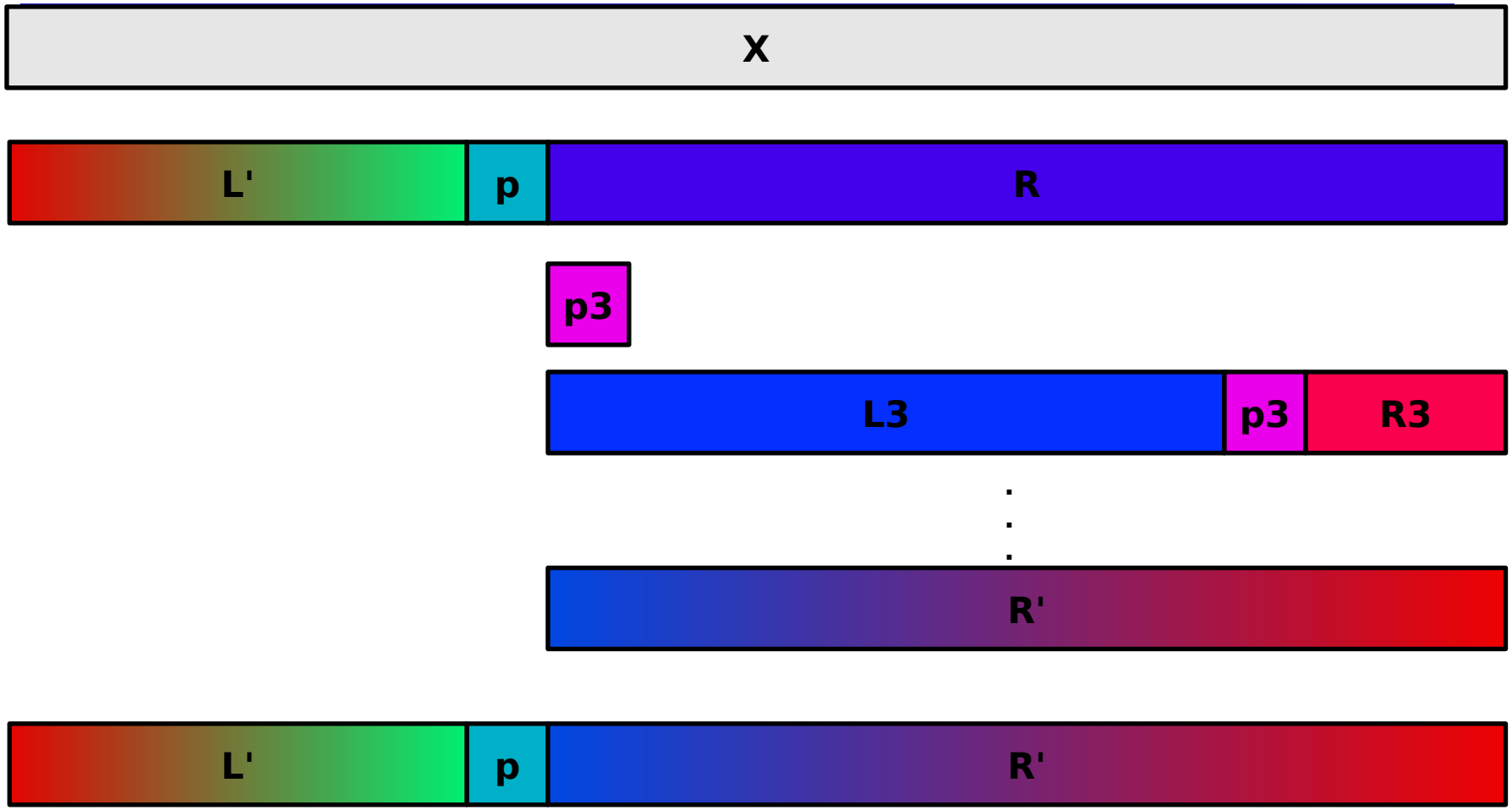
A More Substantial Example: Sort

- Sort set of N random numbers
- Multiple possible algorithms
  - Use parallel version of quicksort
- Sequential quicksort of set of values X
  - Choose “pivot” p from X
  - Rearrange X into
    - L: Values <= p
    - R: Values > p
  - Recursively sort L to get L'
  - Recursively sort R to get R'
  - Return L' : p : R'

Sequential Quicksort Visualized



Sequential Quicksort Visualized



Sequential Quicksort Code

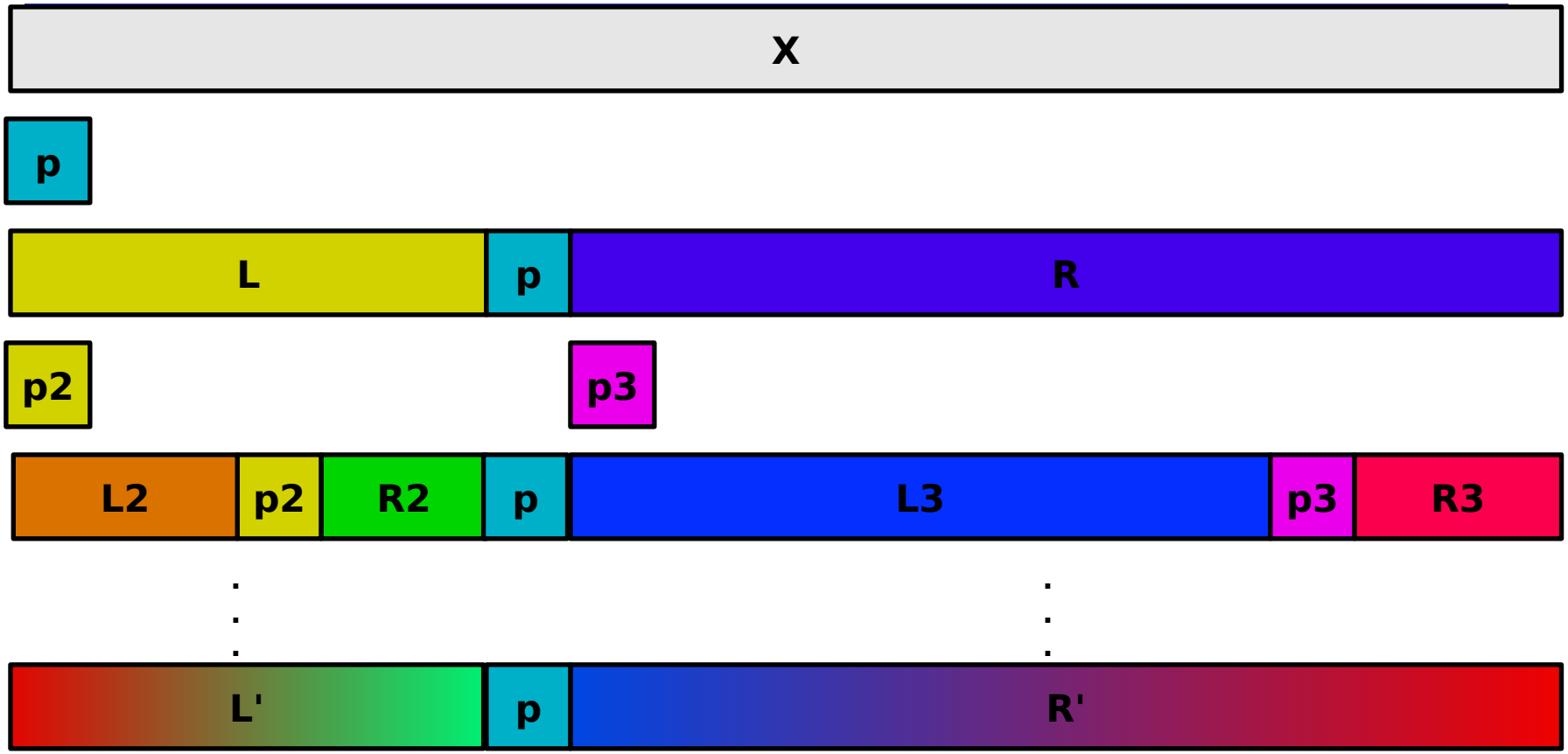
```
void qsort_serial(data_t *base, size_t nele) {
  if (nele <= 1)
    return;
  if (nele == 2) {
    if (base[0] > base[1])
      swap(base, base+1);
    return;
  }

  /* Partition returns index of pivot */
  size_t m = partition(base, nele);
  if (m > 1)
    qsort_serial(base, m);
  if (nele-1 > m+1)
    qsort_serial(base+m+1, nele-m-1);
}
```

- Sort nele elements starting at base
  - Recursively sort L or R if has more than one element

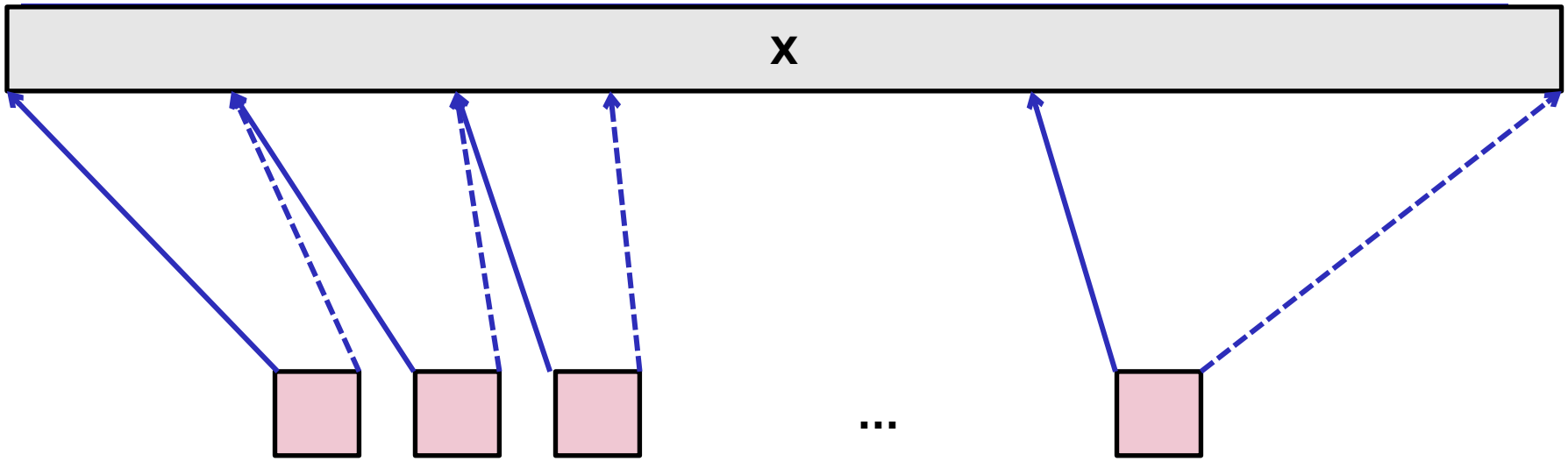
Parallel Quicksort

- Parallel quicksort of set of values X
  - If N < Nthresh, do sequential quicksort
  - Else
    - Choose “pivot” p from X
    - Rearrange X into
      - L: Values <= p
      - R: Values > p
    - Recursively spawn separate threads
      - Sort L to get L'
      - Sort R to get R'
    - Return L': p : R'

Parallel Quicksort Visualized

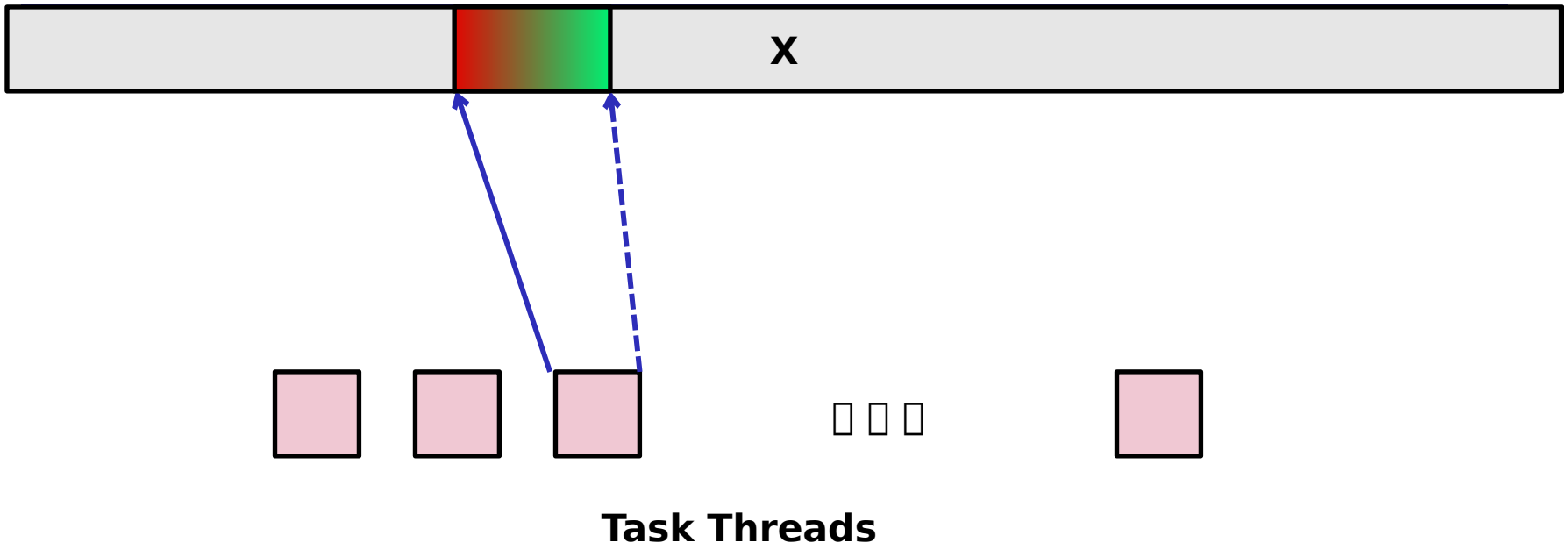


Thread Structure: Sorting Tasks



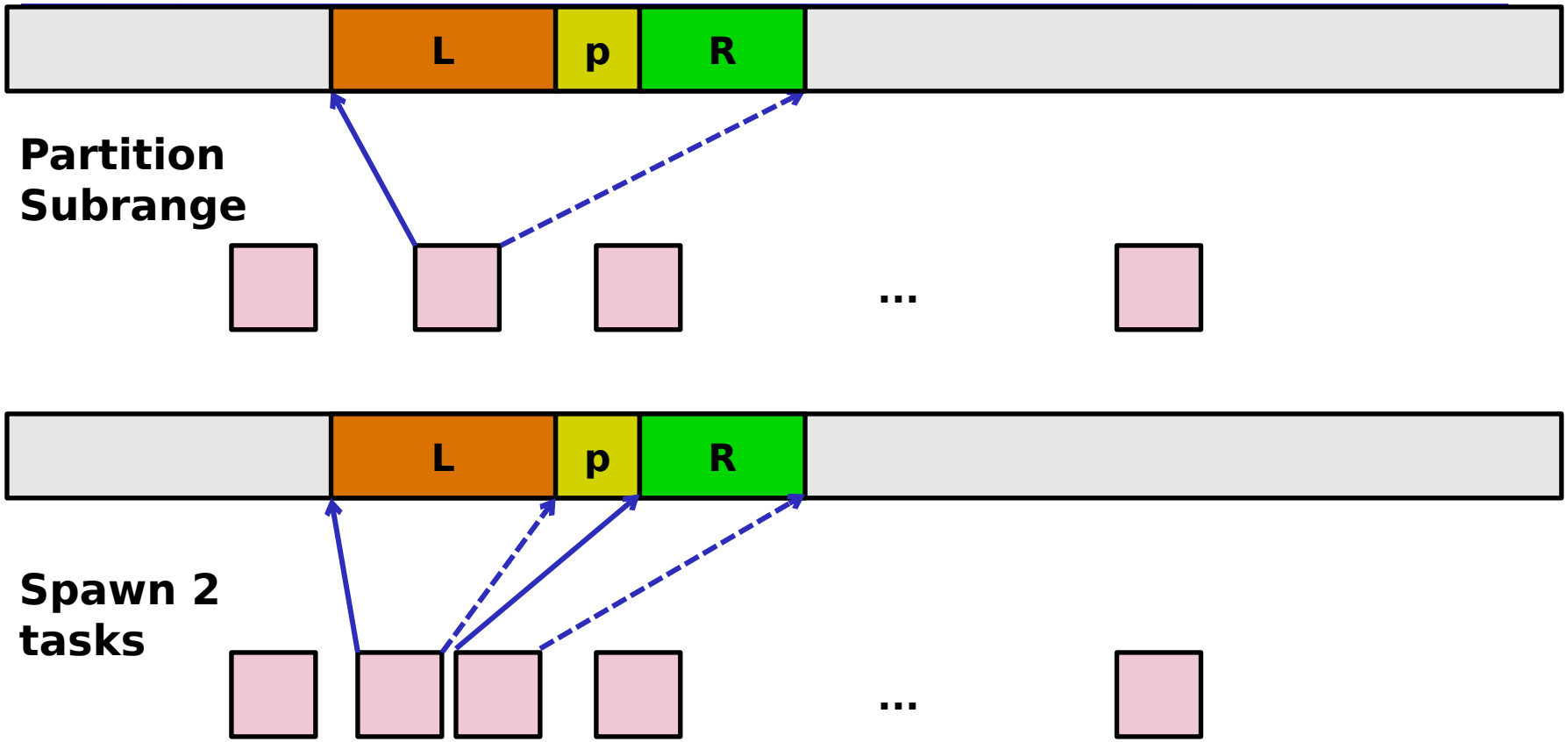
- Task: Sort subrange of data
  - Specify as:
    - **`base`**: Starting address
    - **`nele`**: Number of elements in subrange
- Run as separate thread

Small Sort Task Operation



- Sort subrange using serial quicksort

Large Sort Task Operation



Top-Level Function (Simplified)

```
void tqsort(data_t *base, size_t nele) {
    init_task(nele);
    global_base = base;
    global_end = global_base + nele - 1;
    task_queue_ptr tq = new_task_queue();
    tqsort_helper(base, nele, tq);
    join_tasks(tq);
    free_task_queue(tq);
}
```

- Sets up data structures
- Calls recursive sort routine
- Keeps joining threads until none left
- Frees data structures

Recursive sort routine (Simplified)

```
/* Multi-threaded quicksort */
static void tqsort_helper(data_t *base, size_t nele,
                          task_queue_ptr tq) {
    if (nele <= nele_max_sort_serial) {
        /* Use sequential sort */
        qsort_serial(base, nele);
        return;
    }
    sort_task_t *t = new_task(base, nele, tq);
    spawn_task(tq, sort_thread, (void *) t);
}
```

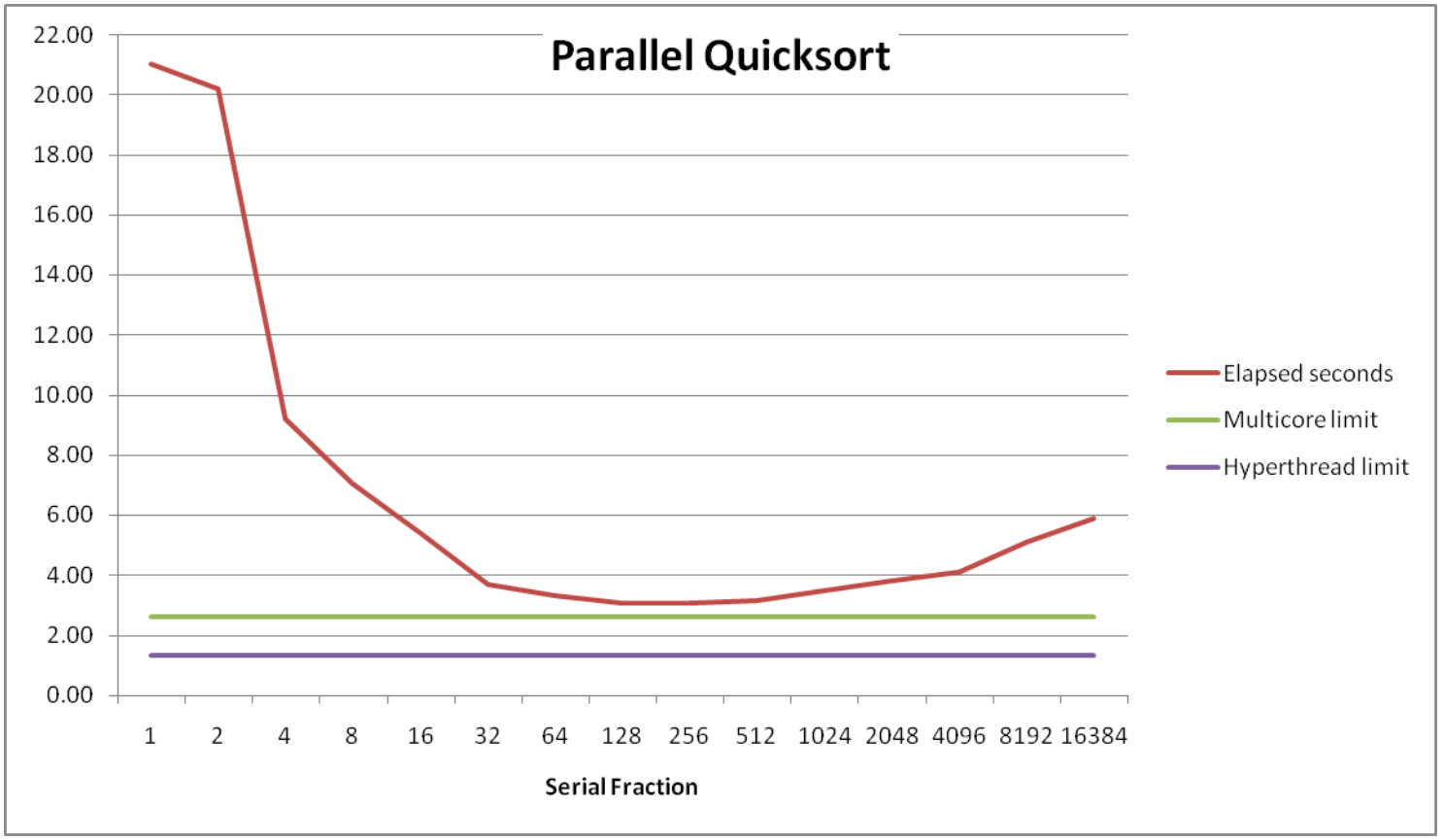
- Small partition: Sort serially
- Large partition: Spawn new sort task

Sort task thread (Simplified)

```
/* Thread routine for many-threaded quicksort */
static void *sort_thread(void *vargp) {
    sort_task_t *t = (sort_task_t *) vargp;
    data_t *base = t->base;
    size_t nele = t->nele;
    task_queue_ptr tq = t->tq;
    free(vargp);
    size_t m = partition(base, nele);
    if (m > 1)
        tqsort_helper(base, m, tq);
    if (nele-1 > m+1)
        tqsort_helper(base+m+1, nele-m-1, tq);
    return NULL;
}
```

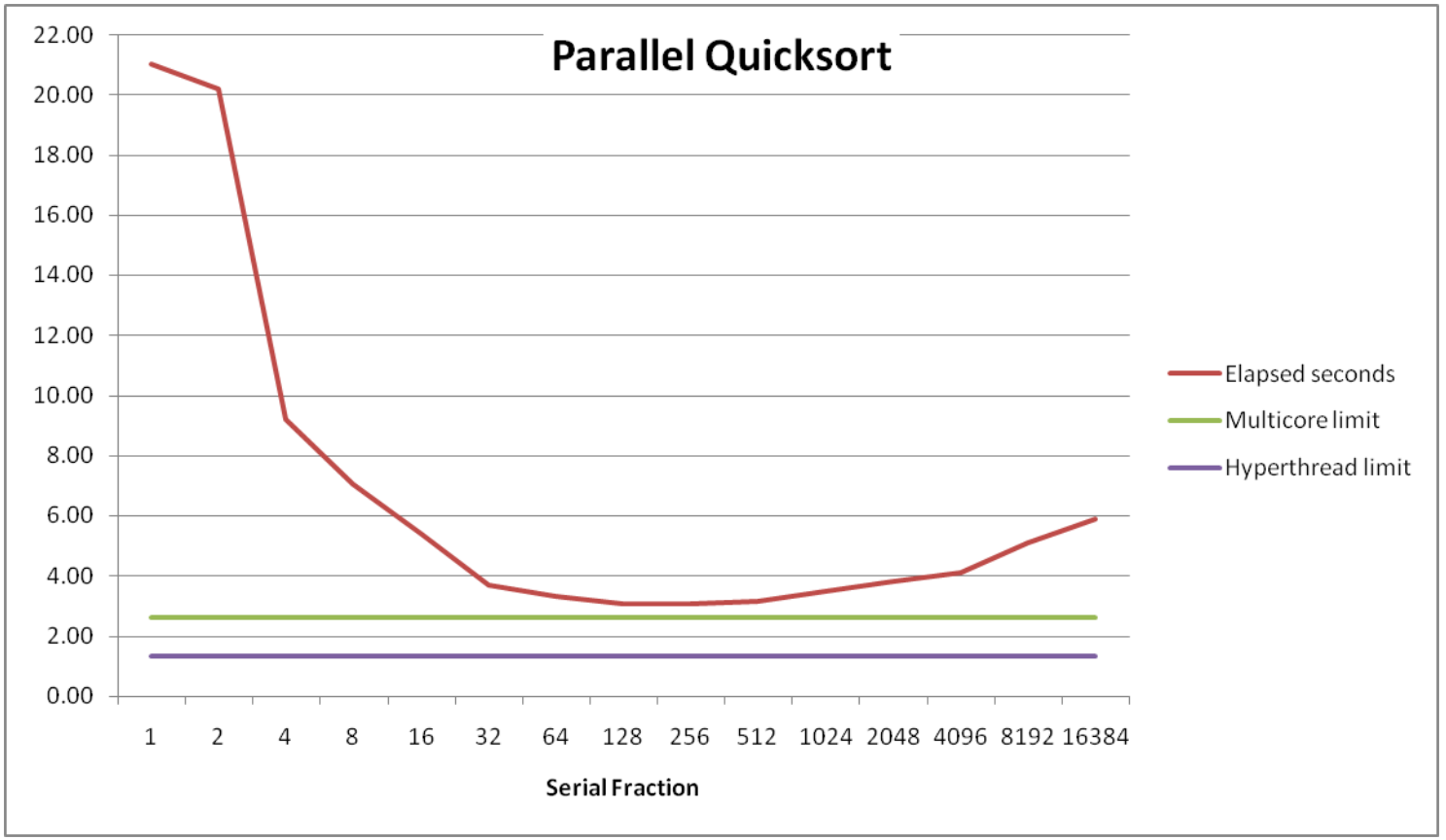
- Get task parameters
- Perform partitioning step
- Call recursive sort routine on each partition

Parallel Quicksort Performance



- Serial fraction: Fraction of input at which do serial sort
- Sort 2^{27} (134,217,728) random values
- Best speedup = 6.84X

Parallel Quicksort Performance



- Good performance over wide range of fraction values
  - F too small: Not enough parallelism

Lessons Learned

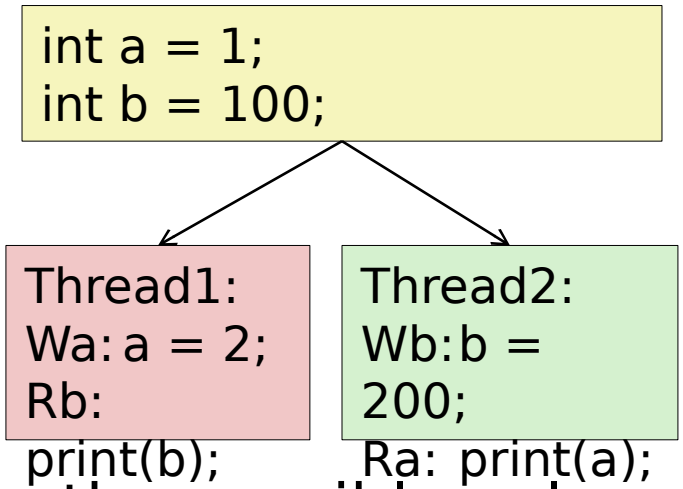
- Must have parallelization strategy
  - Partition into K independent parts
  - Divide-and-conquer
- Inner loops must be synchronization free (e.g., local sum)
  - Synchronization operations very expensive

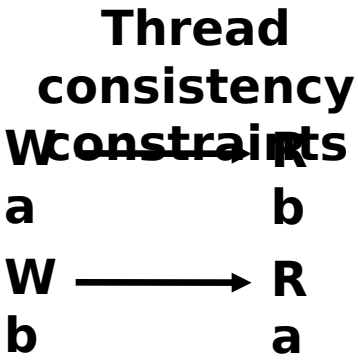
- You can do it!
  - Achieving modest levels of parallelism is not difficult
  - Set up experimental framework and test multiple strategies

课堂练习

线程（t）	1	2	4
核（p）	1	2	4
运行时间（T_p）	12	8	6
加速比（S_p）		1.5	
效率（E_p）	100%		50%

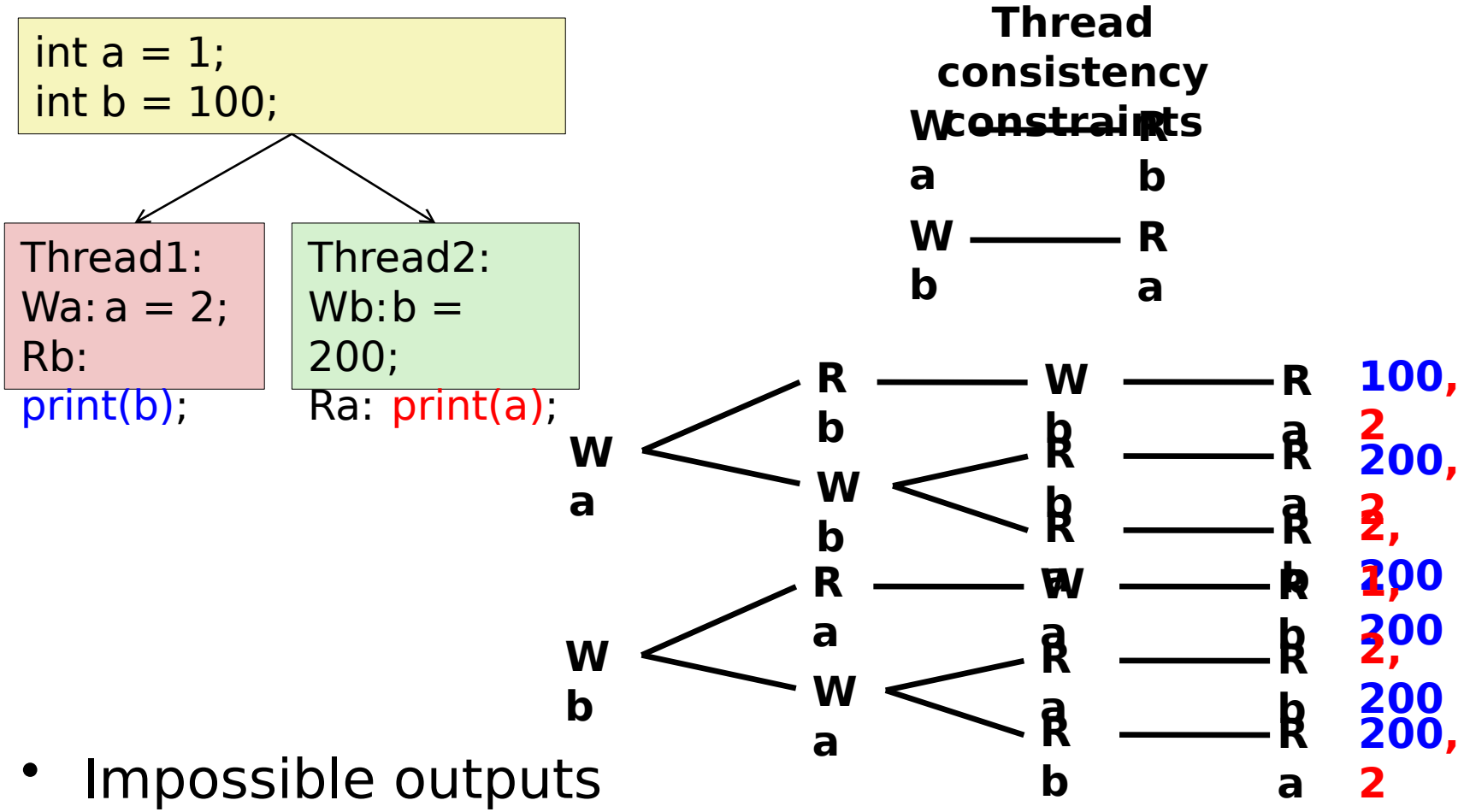
Memory Consistency





- What are the possible values printed?
  - Depends on memory consistency model
  - Abstract model of how hardware handles concurrent accesses
- Sequential consistency
  - Overall effect consistent with each individual thread
  - Otherwise, arbitrary interleaving

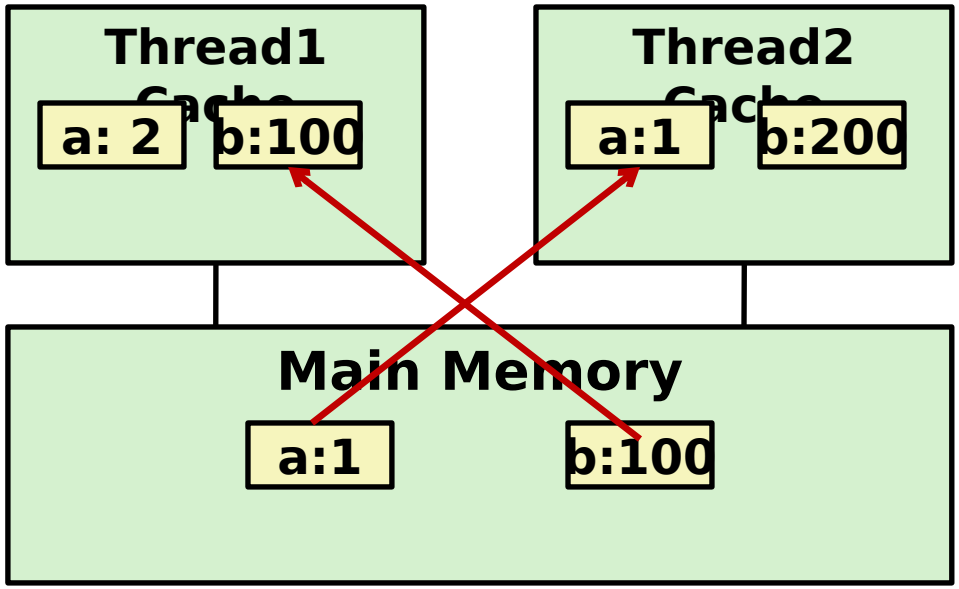
Sequential Consistency Example

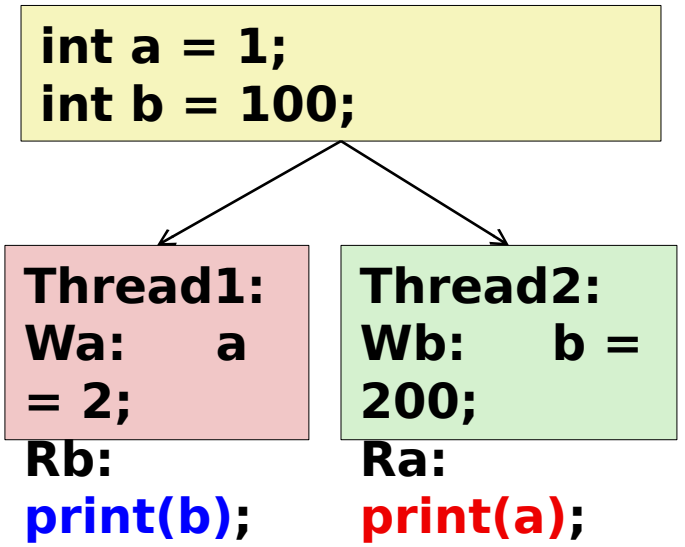


- Impossible outputs
  - 100, 1 1, 100 200, 1
  - Would require reaching both Ra and Rb before Wa and Wb

Non-Coherent Cache Scenario

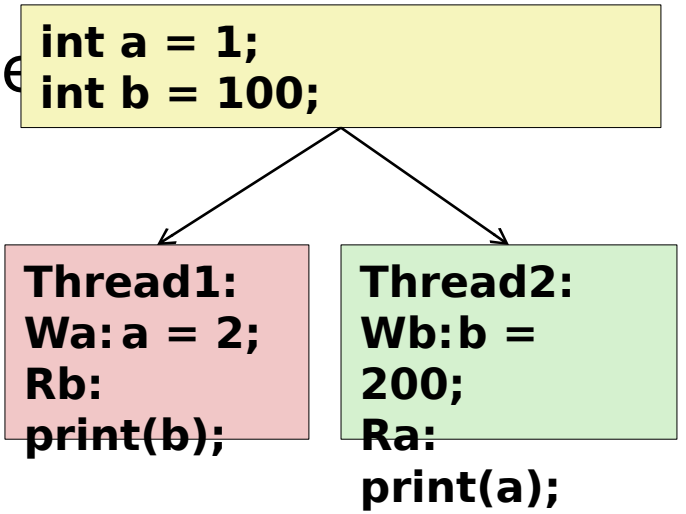
- Write-back caches, without coordination between them

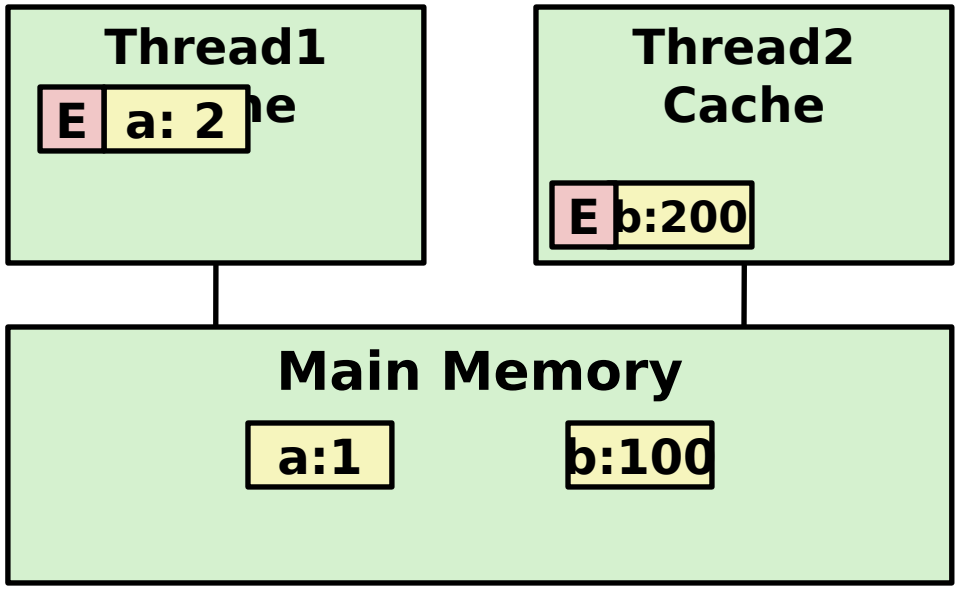




Snoopy Caches

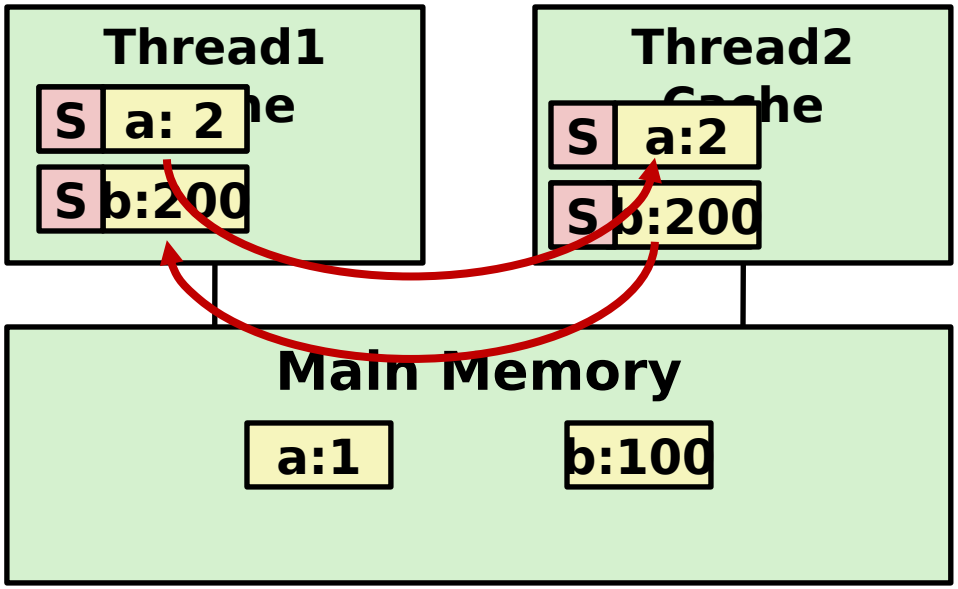
- Tag each cache block with state
  - Invalid Cannot use value
  - Shared Readable copy
  - Exclusive Writeable copy

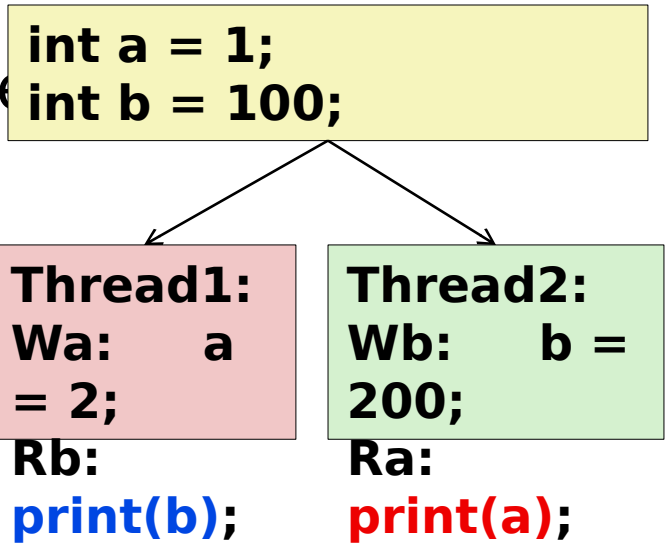




Snoopy Caches

- Tag each cache block with state
  - Invalid Cannot use value
  - Shared Readable copy
  - Exclusive Writeable copy





- When cache sees request for one of its E-tagged blocks
  - Supply value from