第12章 并发编程

同步：高级/Synchronization: Advanced

100076202：计算机系统导论

任课教师：

宿红毅　张艳　黎有琦　李秀星

原作者：

Randal E. **Bryant and** David R. O’Hallaron

议题 Today

- **回顾：信号量、互斥和生产者-消费者 Review: Semaphores, mutexes, producer-consumer**
- **使用信号量调度共享资源 Using semaphores to schedule shared resources**
  - **读者-写者问题 Readers-writers problem**
- **其它并发问题 Other concurrency issues**
  - 线程安全 Thread safety
  - 竞争 Races
  - 死锁 Deadlocks
  - 线程和信号处理之间交互 Interactions between threads and signal handling

提醒：信号量
Reminder: Semaphores

- *Semaphore:* **non-negative global integer synchronization variable**

- **Manipulated by *P* and *V* operations:**
  - *P(s):* [`while (s == 0); s--;`]
    - Dutch for "Proberen" (test)
  - *V(s):* [`s++;`]
    - Dutch for "Verhogen" (increment)

- **OS kernel guarantees that operations between brackets [ ] are executed atomically**
  - Only one *P* or *V* operation at a time can modify s.
  - When `while` loop in *P* terminates, only that *P* can decrement `s`

- **Semaphore invariant: *(s >= 0)***

回顾：使用信号量通过互斥保护共享资源 Review: Using semaphores to protect shared resources via mutual exclusion

- **基本思想： Basic idea:**
  - 将一个唯一的信号量互斥锁（mutex）（最初为1）与每个共享变量（或相关的共享变量集）相关联 Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables)
  - 用P(mutex)和V(mutext)操作包围对共享变量的每次访问 Surround each access to the shared variable(s) with *P(mutex)* and *V(mutex)* operations

```
mutex = 1

P(mutex)
cnt++
V(mutex)
```

回顾：使用锁进行互斥
Review: Using Lock for Mutual Exclusion

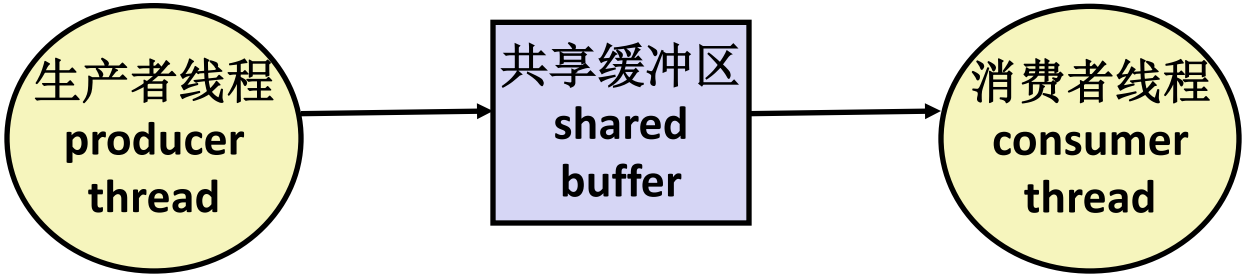
- 基本思想： **Basic idea:**
  - 互斥锁Mutex是只有值0（锁定）或1（解锁）的信号量的特殊情况 Mutex is special case of semaphore that only has value 0 (locked) or 1 (unlocked)
  - *Lock(m):* [`while (m == 0); m=0;`]
  - *Unlock(m):* [`m=1`]
- **比使用信号量快约2倍 ~2x faster than using semaphore for this purpose**
  - 而且，更清楚地表明程序员的意图 And, more clearly indicates programmer's intention

```
mutex = 1

lock(mutex)
cnt++
unlock(mutex)
```

关于示例的注释 Note about Examples

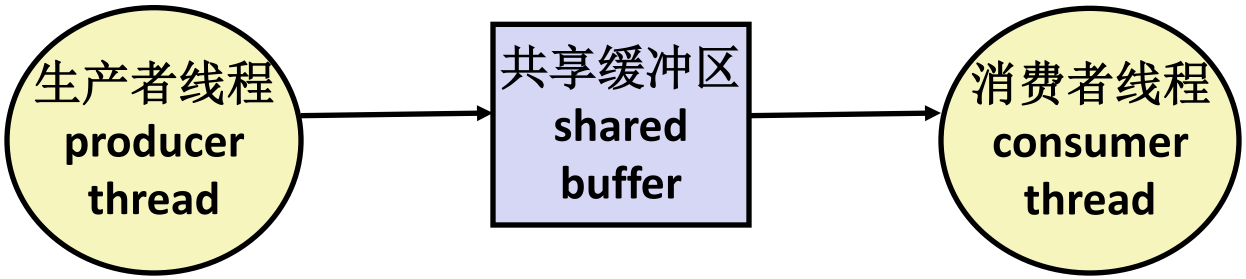
- 课程示例将使用信号量进行计数和互斥 **Lecture examples will use semaphores for both counting and mutual exclusion**
  - 代码比使用pthread_mutex短得多 Code is much shorter than using pthread_mutex

回顾：生产者-消费者问题
Review: Producer-Consumer Problem



- 通用同步模式：**Common synchronization pattern:**
  - 生产者等待空**槽位**，将项目插入缓冲区，并通知消费者 Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
  - 消费者等待**项目**，将其从缓冲区中删除，并通知生产者 Consumer waits for ***item***, removes it from buffer, and notifies producer

回顾：生产者-消费者问题
Review: Producer-Consumer Problem



- 示例 **Examples**
  - 多媒体处理： Multimedia processing:
    - 生产者创建视频帧，消费者对其进行渲染 Producer creates video frames, consumer renders them
  - 事件驱动的图形用户界面 Event-driven graphical user interfaces
    - 生产者检测鼠标点击、鼠标移动和键盘点击，并在缓冲区中插入相应的事件 Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - 消费者从缓冲区检索事件并绘制显示 Consumer retrieves events from buffer and paints the display

回顾：使用信号量协调共享资源的访问 Review: Using Semaphores to Coordinate Access to Shared Resources

- 基本思想：线程使用信号量操作通知另一个线程某些条件已变为真 **Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true**
  - 使用计数信号量来跟踪资源状态 Use counting semaphores to keep track of resource state.
  - 使用二元信号量通知其他线程 Use binary semaphores to notify other threads.

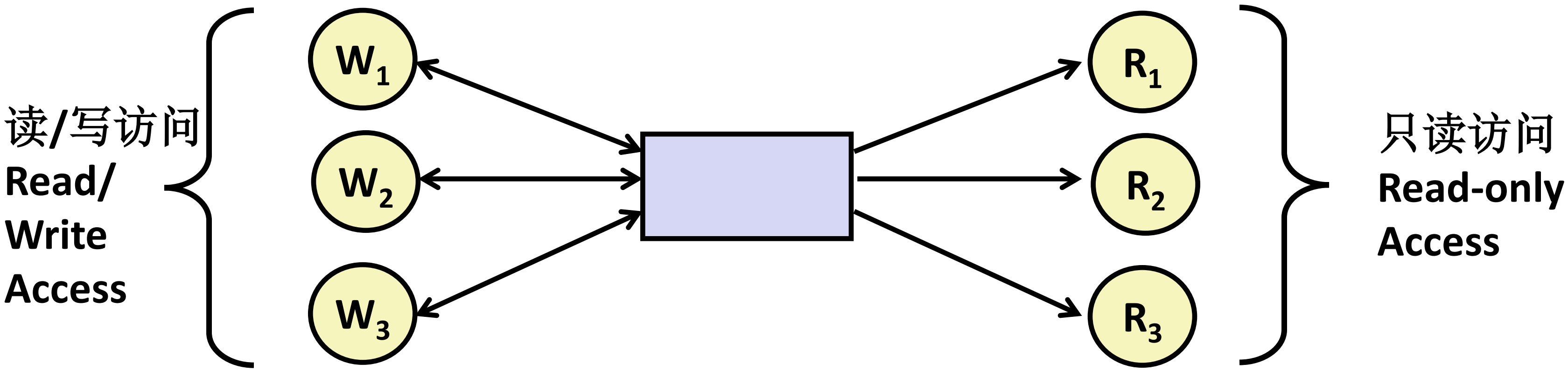
回顾：使用信号量协调共享资源的访问 Review: Using Semaphores to Coordinate Access to Shared Resources

- 生产者-消费者问题 **The Producer-Consumer Problem**
  - 对进程之间的交互活动进行协调，一个进程产生信息，另一个进使用该信息 Mediating interactions between processes that generate information and that then make use of that information
  - 用两个二元信号量实现单条目缓冲区 Single entry buffer implemented with two binary semaphores
    - 一个用于控制生产者的访问 One to control access by producer(s)
    - 一个用于控制消费者的访问 One to control access by consumer(s)
  - 使用信号量+环形缓冲区实现N个条目缓冲区 N-entry implemented with semaphores + circular buffer

议题 Today

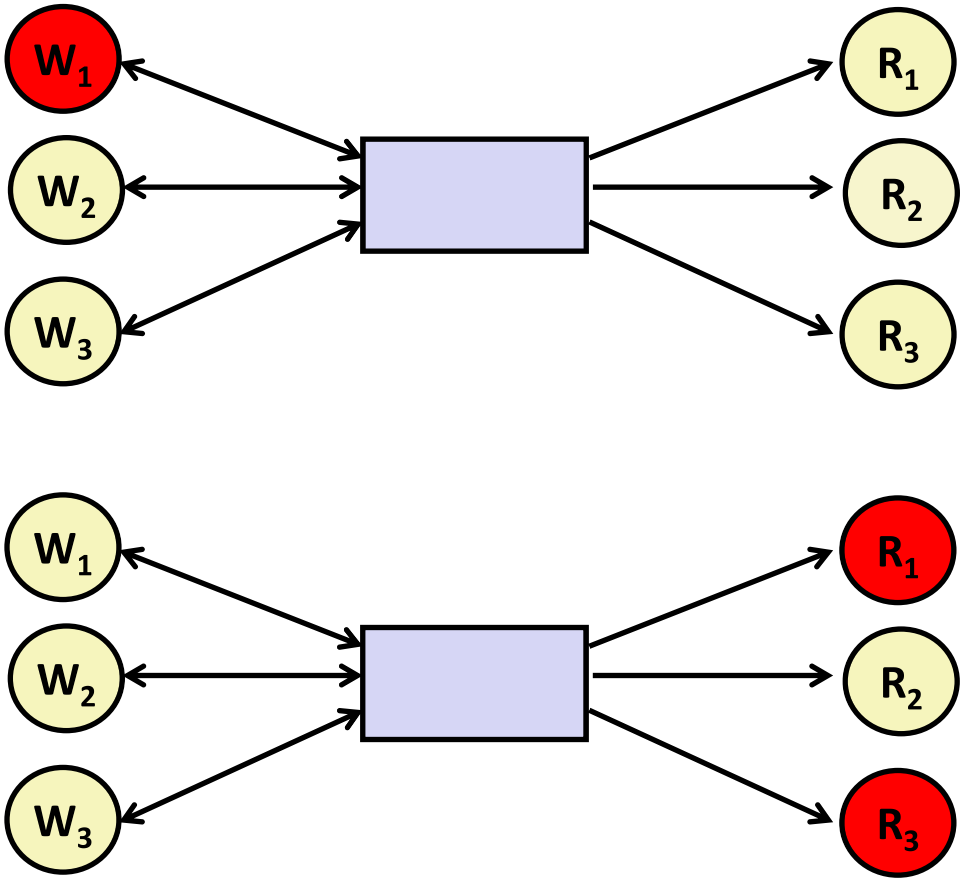
- 回顾：信号量、互斥和生产者-消费者 **Review: Semaphores, mutexes, producer-consumer**
- **使用信号量调度共享资源 Using semaphores to schedule shared resources**
  - **读者-写者问题 Readers-writers problem**
- 其它并发问题 **Other concurrency issues**
  - 线程安全 Thread safety
  - 竞争 Races
  - 死锁 Deadlocks
  - 线程和信号处理交互 Interactions between threads and signal handling

读者和写者问题
Readers-Writers Problem



- 问题陈述： **Problem statement:**
  - *读者*线程仅读取对象 *Reader* threads only read the object
  - *写者*线程修改对象（读/写访问） *Writer* threads modify the object (read/write access)
  - 写者必须具有对对象的独占访问权限 Writers must have exclusive access to the object
  - 无限数量的读者可以访问该对象 Unlimited number of readers can access the object
- 在真实系统中频繁发生，例如 **Occurs frequently in real systems, e.g.,**
  - 在线航空预订系统 Online airline reservation system
  - 多线程缓存Web代理 Multithreaded caching Web proxy

读者/写者示例
Readers/Writers Examples



读者和写者的变体 Variants of Readers-Writers

- 第一类读者-写者问题（有利于读者-读者优先）***First readers-writers problem*** **(favors readers)**
  - 除非已授予写者使用对象的权限，否则不应让任何读者等待 No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - 等待写者之后到达的读者比写者优先 A reader that arrives after a waiting writer gets priority over the writer.
- 第二类读者-写者问题（有利于写者-写者优先） ***Second readers-writers problem*** **(favors writers)**
  - 一旦写者准备好写入，它将尽快执行写入 Once a writer is ready to write, it performs its write as soon as possible
  - 在写者之后到达的读者必须等待，即使写者也要等待 A reader that arrives after a writer must wait, even if the writer is also waiting.
- 在这两种情况下都有可能出现饿死情况（线程无限期等待）***Starvation*** **(where a thread waits indefinitely) is possible in both cases.**

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

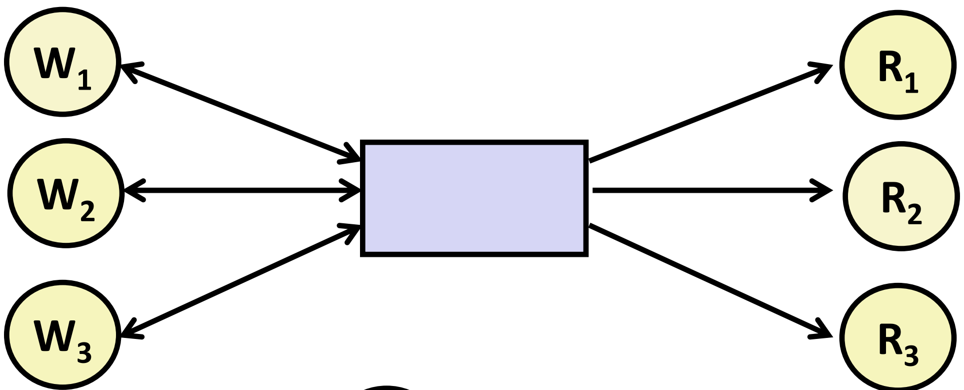
```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

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);
  }
}
```

写者 Writers:

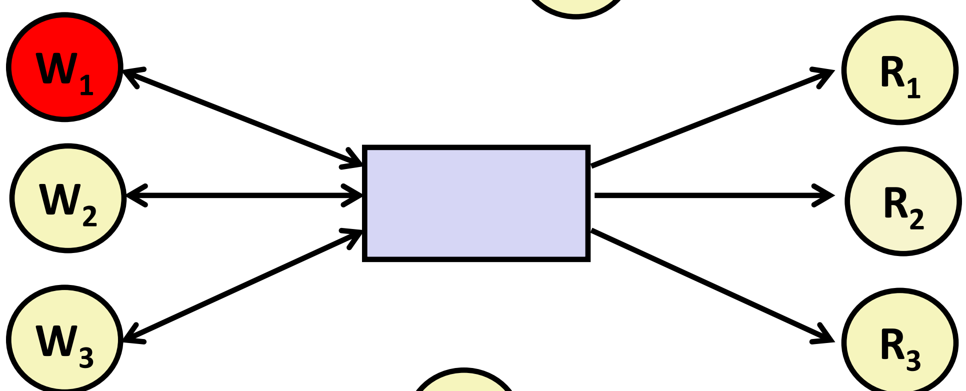
```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

读者/写者示例
Readers/Writers Examples

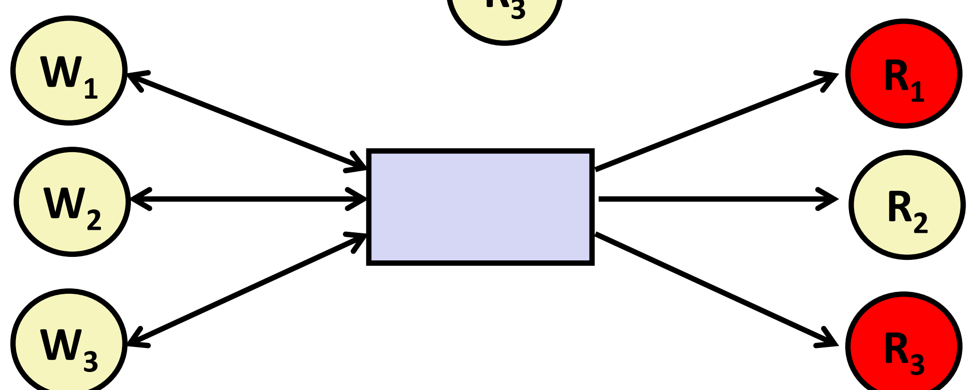


w = 1
readcnt = 0



w = 0
readcnt = 0

w = 0
readcnt = 2



第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

```
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);
  }
}
```

写者 Writers:

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

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： Arrivals: R1 R2 W1 R3

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

```
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);
  }
}
```

R1

写者 Writers:

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

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

第一类读者-写者问题的解决方案

Solution to First Readers-Writers Problem

读者 Readers:

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

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
R2  if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

R1  /* Reading happens here */

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

写者 Writers:

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

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 2

W == 0

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

```
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);
  }
}
```

R2

R1

写者 Writers:

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

    /* Writing here */

    V(&w);
  }
}
```

W1

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 2

W == 0

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

```
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);

R2 → /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
R1 → }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {   ← W1
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

第一类读者-写者问题的解决方案

Solution to First Readers-Writers Problem

读者 Readers:

```
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);
  }
}
```

R3

R2

R1

写者 Writers:

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

    /* Writing here */

    V(&w);
  }
}
```

W1

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 2

W == 0

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 **Readers:**

```
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);

R3 →
    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
R2 →V(&mutex);
  }
}
```

写者 **Writers:**

```
void writer(void)
{
  while (1) {
    P(&w);      ← W1

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： **Arrivals: R1 R2 W1 R3**

Readcnt == 1
W == 0

第一类读者-写者问题的解决方案

Solution to First Readers-Writers Problem

读者 Readers:

```
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);
R3 →V(&mutex);
  }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {
    P(&w);        ← W1

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达: Arrivals: R1 R2 W1 R3

Readcnt == 0

W == 1

其它读者-写者版本
Other Versions of Readers-Writers

- 第一类解决方案的不足 **Shortcoming of first solution**
  - 源源不断的读者将无限期地阻止写者 Continuous stream of readers will block writers indefinitely
- 第二个版本 **Second version**
  - 一旦写者出现，就会阻止以后的读者访问 Once writer comes along, blocks access to later readers
  - 一系列写入可能会阻止所有读取 Series of writes could block all reads
- 先进先出实现 **FIFO implementation**
  - 请参阅code目录中的rwqueue代码 See rwqueue code in code directory
  - 按顺序接收服务请求 Service requests in order received
  - 保存线程在先进先出队列中 Threads kept in FIFO
  - 每一个都有信号量，可以访问临界区 Each has semaphore that enables its access to critical section

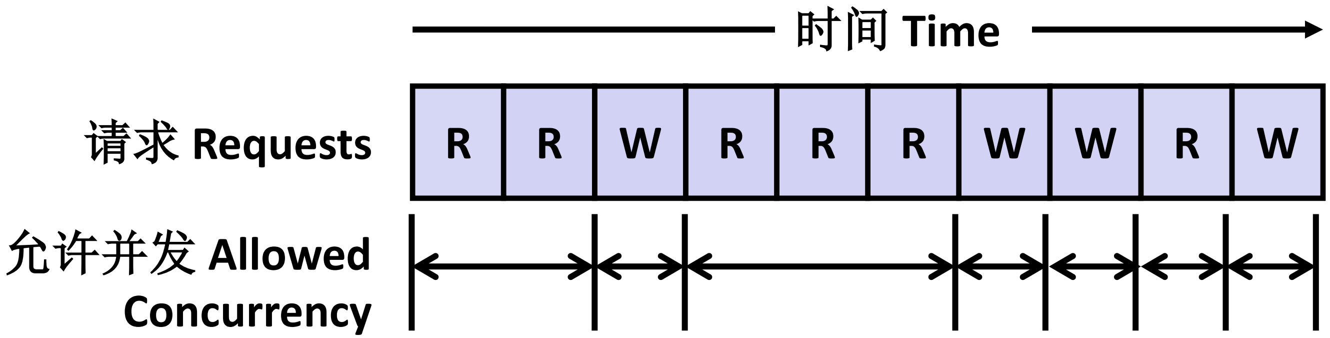
第二类读者-写者问题解决方案
Solution to Second Readers-Writers Problem

```
int readcnt, writecnt;        // Initially 0
sem_t rmutex, wmutex, r, w; // Initially 1
void reader(void)
{
  while (1) {
    P(&r);
    P(&rmutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&rmutex);
    V(&r)

    /* Reading happens here */

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

第二类读者-写者问题解决方案
Solution to Second Readers-Writers Problem

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

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

    P(&wmutex);
    writecnt--;
    if (writecnt == 0);
        V(&r);
    V(&wmutex);
  }
}
```

用先进先出队列管理读者/写者
Managing Readers/Writers with FIFO



- **思想 Idea**
  - 读/写请求插入先进先出队列 Read & Write requests are inserted into FIFO
  - 请求在从队列删除时进行处理 Requests handled as remove from FIFO
    - 如果当前空闲或正在处理读取，则允许继续读取 Read allowed to proceed if currently idle or processing read
    - 仅允许在空闲时继续写入请求 Write allowed to proceed only when idle
  - 请求完成后通知控制器 Requests inform controller when they have completed
- **公平 Fairness**
  - 保证最终会处理每个请求 Guarantee every request is eventually handled

读者写者先进先出实现
Readers Writers FIFO Implementation

- 完整代码见rwqueue.h和rwqueue.c   Full code in rwqueue.{h,c}

```
/* Queue data structure */
typedef struct {
    sem_t mutex;  // Mutual exclusion
    int reading_count;     // Number of active readers
    int writing_count;     // Number of active writers
    // FIFO queue implemented as linked list with tail
    rw_token_t *head;
    rw_token_t *tail;
} rw_queue_t;
```

```
/* Represents individual thread's position in queue */
typedef struct TOK {
    bool is_reader;
    sem_t enable;          // Enables access
    struct TOK *next;    // Allows chaining as linked list
} rw_token_t;
```

读者写者先进先出队列使用
Readers Writers FIFO Use

- 在rwqueue-test.c程序中 In rwqueue-test.c

```
/* Get write access to data and write */
void iwriter(int *buf, int v)
{
    rw_token_t tok;
    rw_queue_request_write(&q, &tok);
    /* Critical section */
    *buf = v;
    /* End of Critical Section */
    rw_queue_release(&q);
}
```

```
/* Get read access to data and read */
int ireader(int *buf)
{
    rw_token_t tok;
    rw_queue_request_read(&q, &tok);
    /* Critical section */
    int v = *buf;
    /* End of Critical section */
    rw_queue_release(&q);
    return v;
}
```

读者/写者锁的库函数
Library Reader/Writer Lock

- 数据类型 **Data type `pthread_rwlock_t`**
- 操作 **Operations**
  - 获得读锁 Acquire read lock

```
Pthread_rwlock_rdlock(pthread_rw_lock_t *rwlock)
```

  - 获得写锁 Acquire write lock

```
Pthread_rwlock_wrlock(pthread_rw_lock_t *rwlock)
```

  - 释放（其中一个）锁 Release (either) lock

```
Pthread_rwlock_unlock(pthread_rw_lock_t *rwlock)
```

- 观察 **Observation**
  - 必须正确使用库函数 Library must be used correctly!
    - 由程序员决定哪些需要读访问，哪些需要写访问 Up to programmer to decide what requires read access and what requires write access

议题 Today

- 回顾：信号量、互斥和生产者-消费者 **Review: Semaphores, mutexes, producer-consumer**
- 使用信号量调度共享资源 **Using semaphores to schedule shared resources**
  - 读者-写者问题 Readers-writers problem
- 其它并发问题 **Other concurrency issues**
  - **竞争 Races**
  - 死锁 Deadlocks
  - 线程安全 Thread safety
  - 线程和信号处理之间交互 Interactions between threads and signal handling

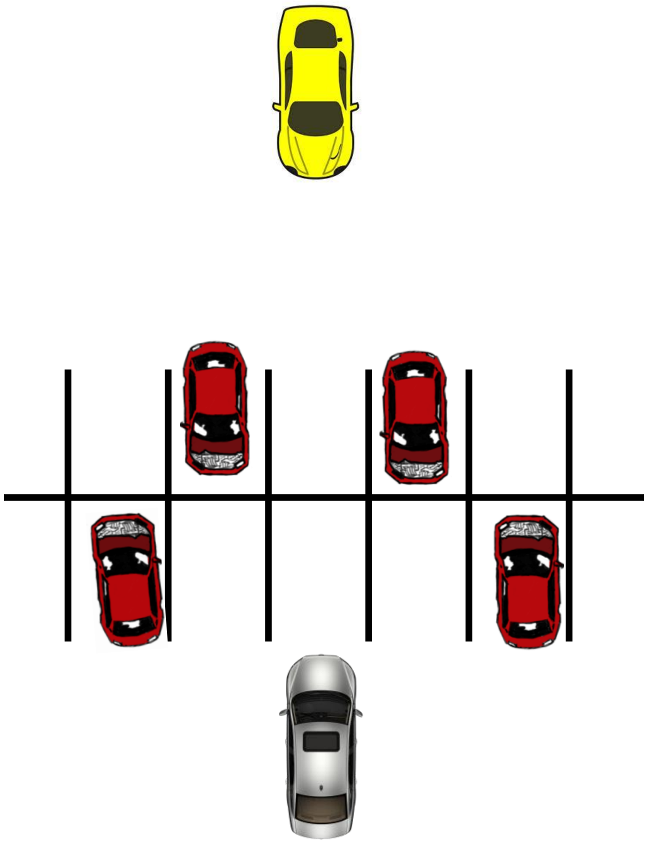
一个担忧：竞争 One Worry: Races

- 当程序的正确性取决于一个线程在另一个线程到达点y之前到达点x时，就会发生*竞争*   **A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y**

```
/* a threaded program with a race */
int main(int argc, char** argv) {
    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);
    return 0;
}

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

race.c

数据竞争 Data Race



消除竞争 Race Elimination

- 不要共享状态 **Don’t share state**
  - 例如，使用malloc为每个线程生成单独的参数拷贝 E.g., use malloc to generate separate copy of argument for each thread

- 使用同步原语控制对共享状态的访问 **Use synchronization primitives to control access to shared state**
  - 不同的共享状态可能使用不同的原语 Different shared state can use different primitives

议题 Today

- 回顾：信号量、互斥和生产者-消费者 **Semaphores, mutexes, producer-consumer**
- 使用信号量调度共享资源 **Using semaphores to schedule shared resources**
  - 生产者-消费者问题 Producer-consumer problem
- **其它并发问题 Other concurrency issues**
  - **竞争 Races**
  - **死锁 Deadlocks**
  - 线程安全 Thread safety
  - 线程和信号处理之间交互 Interactions between threads and signal handling

一个担忧：死锁 A Worry: Deadlock

- 定义：当且仅当一个进程正在等待一个永远不会为真的条件，那么它就会*死锁* **Def: A process is *deadlocked* iff it is waiting for a condition that will never be true.**

- 典型场景 **Typical Scenario**
  - 进程1和进程2需要两个资源（A和B）才能继续 Processes 1 and 2 needs two resources (A and B) to proceed
  - 进程1获取A，等待B Process 1 acquires A, waits for B
  - 进程2获取B，等待A Process 2 acquires B, waits for A
  - 两个进程都将永远等待！ Both will wait forever!

一个担忧：死锁 A Worry: Deadlock

- 定义：当且仅当一个进程正在等待一个永远不会为真的条件，那么它就会*死锁* **Def: A process is *deadlocked* iff it is waiting for a condition that will never be true.**

- 更全面的知识（超出了**213**课程的范围），死锁有四个要求 **More fully (and beyond the scope of 213), a deadlock has four requirements**
  - 互斥 Mutual exclusion
  - 循环等待 Circular waiting
  - 保持和等待 Hold and wait
  - 非抢占式 No pre-emption

信号量死锁 Deadlocking With Semaphores

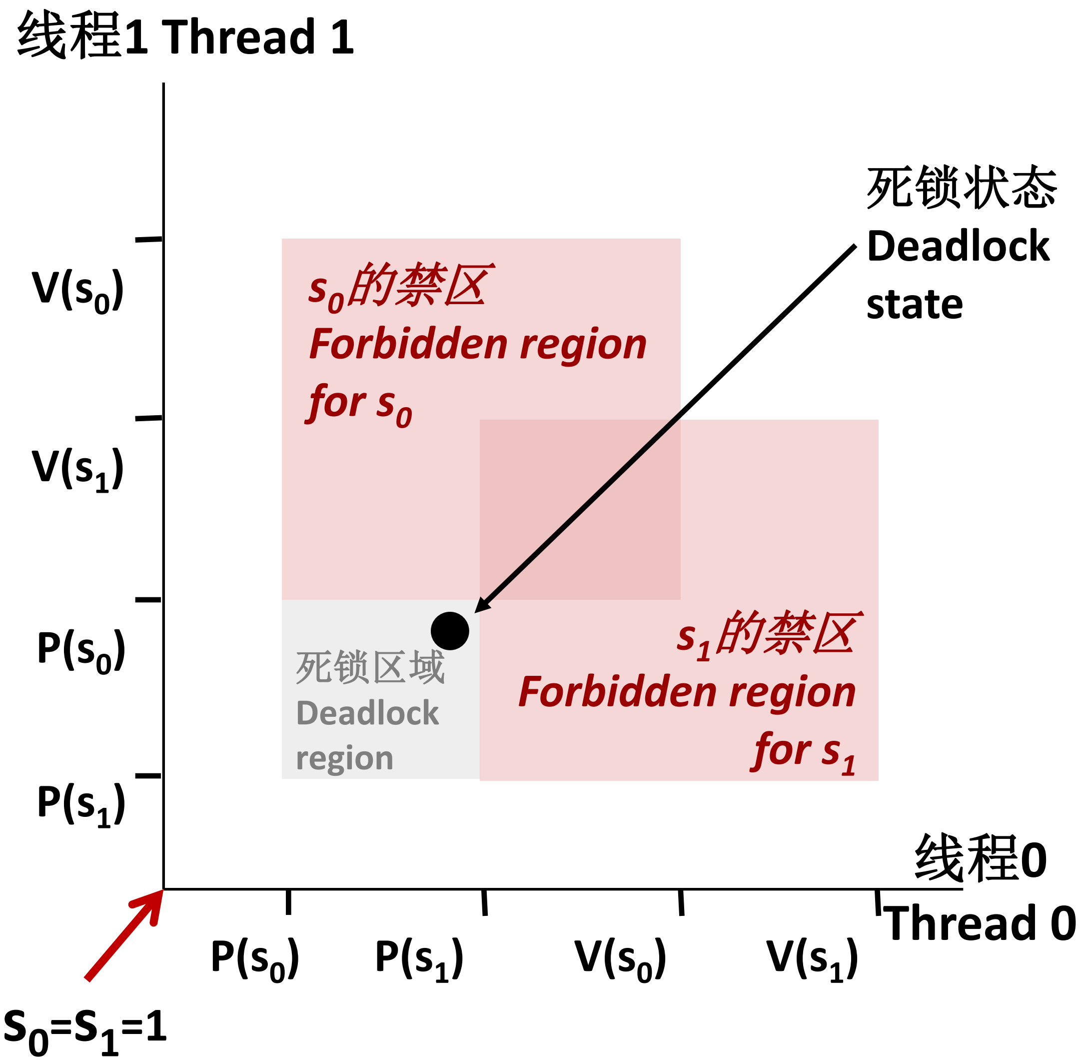
```
int main(int argc, char** argv)
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
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

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

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

进度图中显示的死锁
Deadlock Visualized in Progress Graph



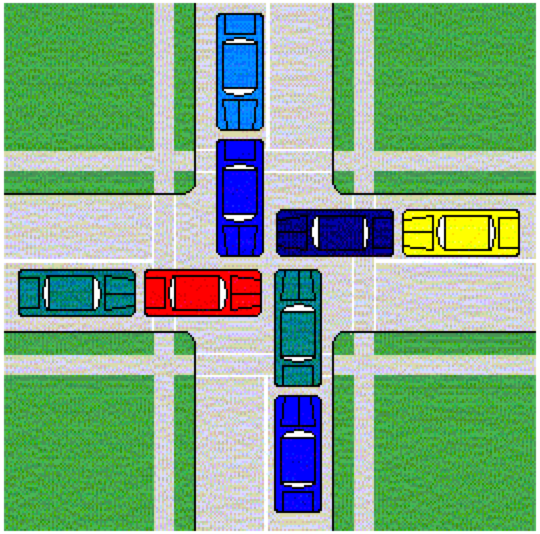
锁定引入了*死锁*的可能性：等待一个永远不会成真的条件 Locking introduces the potential for ***deadlock:*** waiting for a condition that will never be true

任何进入*死锁区域*的轨迹将最终到达*死锁状态*，等待s0或s1变为非零 Any trajectory that enters the ***deadlock region*** will eventually reach the ***deadlock state***, waiting for either S_0 or S_1 to become nonzero

其他轨迹幸运地避开了死锁区域 Other trajectories luck out and skirt the deadlock region

不幸的事实：死锁往往是不确定的（竞争） Unfortunate fact: deadlock is often nondeterministic (race)

死锁 Deadlock



避免死锁

Avoiding Deadlock

以相同的顺序获取共享资源

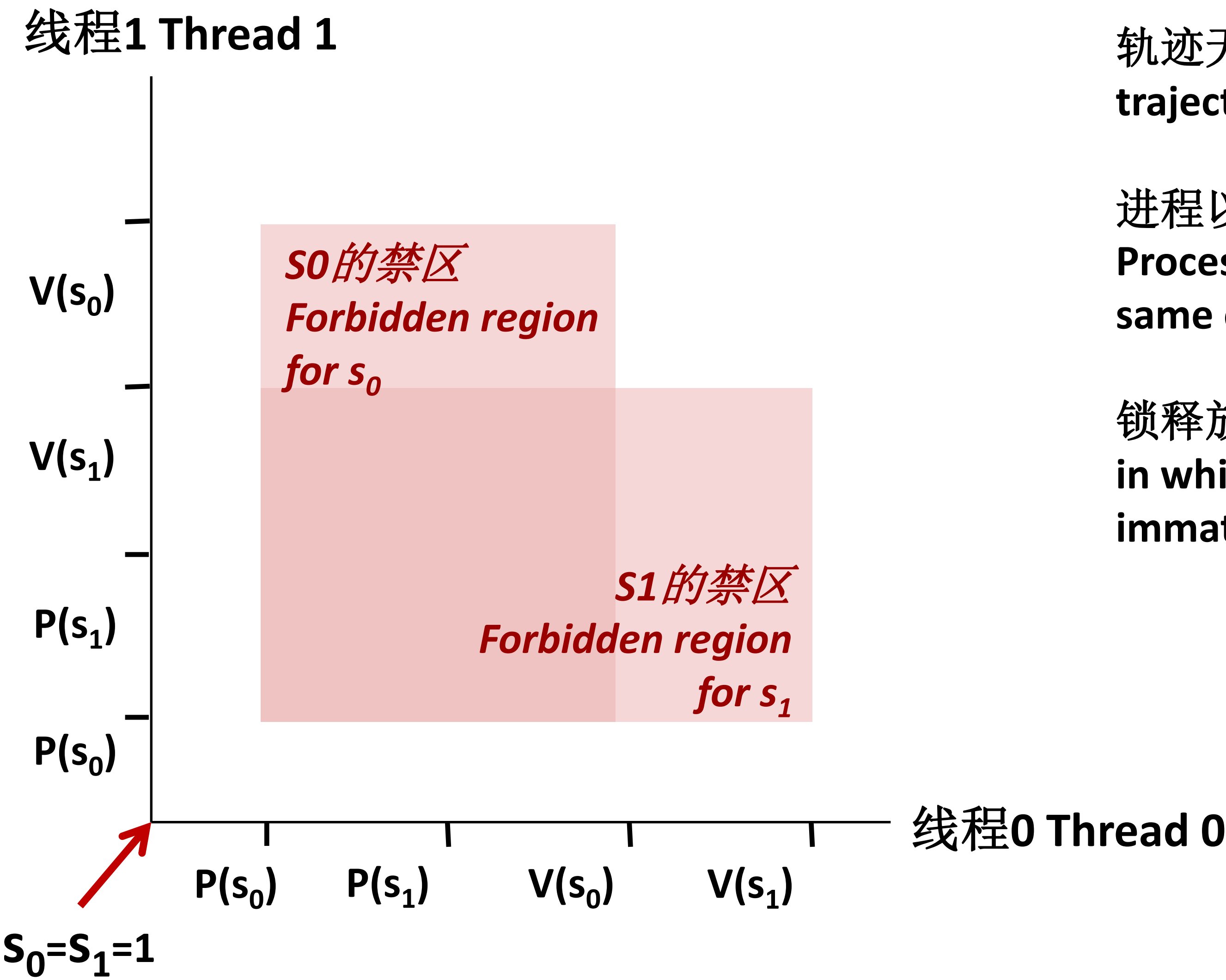
Acquire shared resources in same order

```
int main(int argc, char** argv)
{
    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);
    return 0;
}
```

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

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

在进度图中避免死锁
Avoided Deadlock in Progress Graph



轨迹无法卡住 No way for trajectory to get stuck

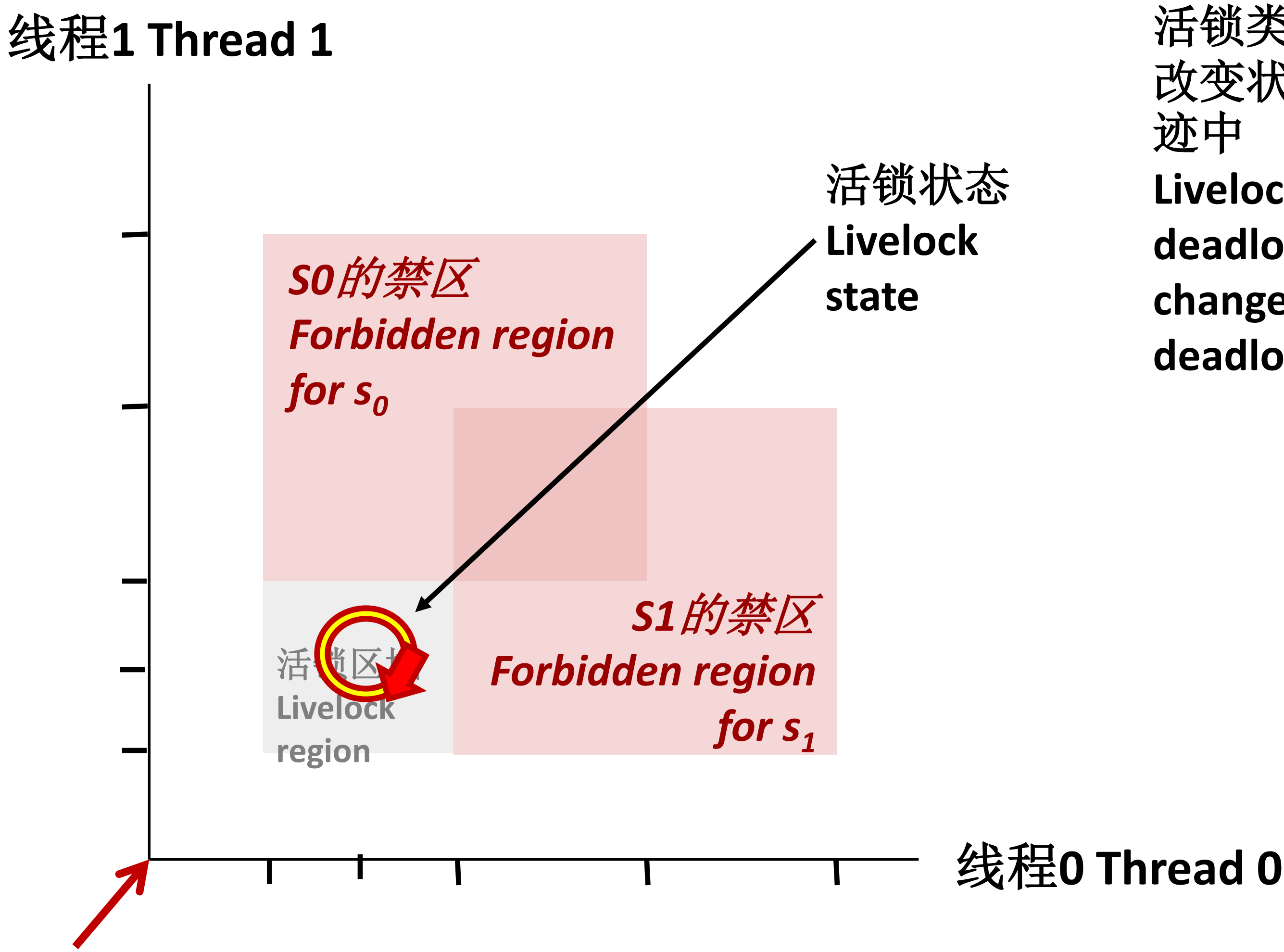
进程以相同的顺序获取锁 Processes acquire locks in same order

锁释放的顺序无关紧要 Order in which locks released immaterial

演示 Demonstration

- 参见程序**deadlock.c See program deadlock.c**
- **100个线程，每个线程获得同样的两个锁 100 threads, each acquiring same two locks**
- 风险模式 **Risky mode**
  - 偶数线程请求锁的顺序与奇数线程相反 Even numbered threads request locks in opposite order of odd-numbered ones
- 安全模式 **Safe mode**
  - 所有线程以相同的顺序获取锁 All threads acquire locks in same order

在进度图中显示活锁
Livelock Visualized in Progress Graph



活锁类似于死锁，只是线程改变状态，但仍处于死锁轨迹中

Livelock is similar to a deadlock, except the threads change state, but remain in a deadlock trajectory.

死锁、活锁、饿死
Deadlock, Livelock, Starvation

- 死锁 **Deadlock**
  - 一个或多个线程正在等待一个永远不会为真的条件 One or more threads is waiting on a condition that will never be true

- 活锁 **Livelock**
  - 一个或多个线程正在更改状态，但永远不会离开死锁/活锁轨迹 One or more threads is changing state, but will never leave a deadlock / livelock trajectory

- 饿死 **Starvation**
  - 一个或多个线程暂时无法取得进展 One or more threads is temporarily unable to make progress

议题 Today

- 回顾：信号量、互斥和生产者-消费者 Review: Semaphores, mutexes, producer-consumer
- 使用信号量调度共享资源 Using semaphores to schedule shared resources
  - 读者-写者问题 Readers-writers problem
- **其它并发问题 Other concurrency issues**
  - 竞争 Races
  - 死锁 Deadlocks
  - **线程安全 Thread safety**
  - 线程和信号处理之间交互 Interactions between threads and signal handling

关键概念：线程安全
Crucial concept: Thread Safety

- 从线程调用的函数必须是线程安全的 **Functions called from a thread must be *thread-safe***
- *定义*：函数是线程安全的，只要它在从多个线程同时调用时总是产生正确的结果 ***Def:* A function is *thread-safe* iff it will always produce correct results when called simultaneously from multiple threads.**
- 线程不安全函数的分类： **Classes of thread-unsafe functions:**
  - 类1：不保护共享变量的函数 Class 1: Functions that do not protect shared variables
  - 类2：跨多个调用保持状态的函数 Class 2: Functions that keep state across multiple invocations
  - 类3：返回指向静态变量的指针的函数 Class 3: Functions that return a pointer to a static variable
  - 类4：调用线程不安全函数的函数 Class 4: Functions that call thread-unsafe functions

线程不安全函数（类1）
Thread-Unsafe Functions (Class 1)

- 未能保护共享变量 **Failing to protect shared variables**
  - 修复：使用P和V信号量操作（或互斥锁） Fix: Use *P* and *V* semaphore operations (or mutex)
  - 示例：goodcnt.c Example: **`goodcnt.c`**
  - 问题：同步操作会降低代码速度 Issue: Synchronization operations will slow down code

线程不安全函数（类2）
Thread-Unsafe Functions (Class 2)

- 跨多个函数调用依赖持久状态 **Relying on persistent state across multiple function invocations**
  - 示例：依赖于静态状态的随机数生成器 Example: Random number generator that relies on static state

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

线程安全随机数生成器
Thread-Safe Random Number Generator

- 传递状态作为参数的一部分 **Pass state as part of argument**
  - 从而消除静态状态 and, thereby, eliminate static state

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- 结论：使用rand_r的程序员必须保持种子 **Consequence: programmer using `rand_r` must maintain seed**

线程不安全函数（类3）
Thread-Unsafe Functions (Class 3)

- 返回指向静态变量的指针 **Returning a pointer to a static variable**
- 修复：重写函数，以便调用方传递变量地址以存储结果 **Fix: Rewrite function so caller passes address of variable to store result**
  - 需要更改调用者和被调用者 Requires changes in caller and callee
- 修复2：用互斥锁包装函数 **Fix 2: Wrap function with mutex**
  - 调用方仍需更改 Caller still has to be changed
  - 可以保留旧函数 Can preserve old function
  - 函数可能成为瓶颈 Function may become a bottleneck

```
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    snprintf(buf, sizeof buf,
             "%d", x);
    return buf;
}
```

```
void fix_itoa(int x, char *dst,
              size_t dstsz)
{
    snprintf(dst, dstsz, "%d", x);
}
```

```
void wrap_itoa(int x, char *dst,
               size_t dstsz)
{
    static sem_t mutex;
    P(mutex);
    strncpy(dst, itoa(x), dstsz);
    V(mutex);
}
```

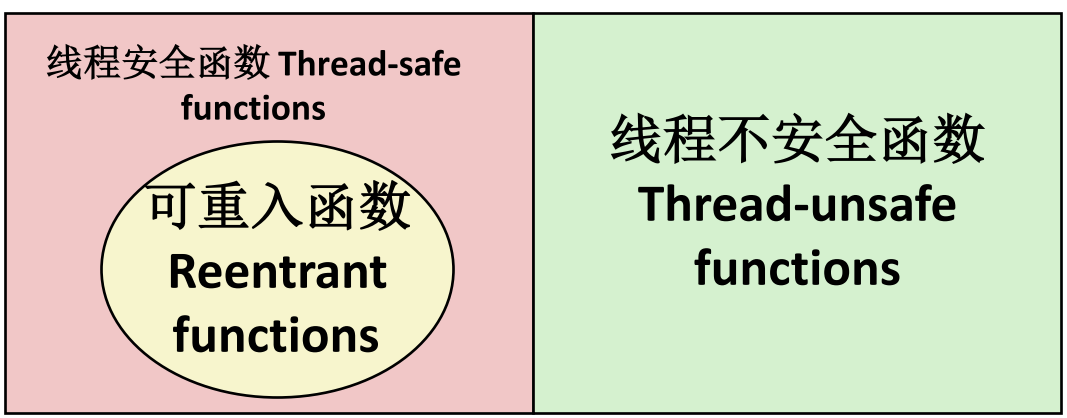
线程不安全函数（类4）
Thread-Unsafe Functions (Class 4)

- 调用线程不安全函数 **Calling thread-unsafe functions**
  - 调用一个线程不安全函数会使调用它的整个函数不安全 Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - 修复：修改函数，使其仅调用线程安全函数 Fix: Modify the function so it calls only thread-safe functions ☺

可重入函数 Reentrant Functions

- 定义：当且仅当函数被多个线程调用时不访问共享变量，则该函数是*可重入*的 **Def: A function is *reentrant* iff it accesses no shared variables when called by multiple threads.**
  - 线程安全函数的重要子集 Important subset of thread-safe functions
    - 不需要同步操作 Require no synchronization operations
    - 使类2函数线程安全的唯一方法是使其可重入（例如rand_r）Only way to make a Class 2 function thread-safe is to make it reentrant (e.g., `rand_r`)

全部函数 **All functions**



线程安全的库函数
Thread-Safe Library Functions

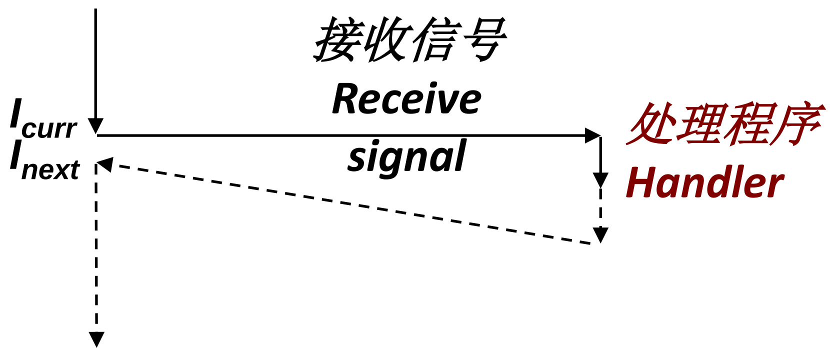
- 标准C语言库（K&R教材后面）中的大多数函数都是线程安全的 Most functions in the Standard C Library (at the back of your K&R text) are thread-safe
  - 示例：malloc、free、printf、scanf　Examples: **`malloc`**, **`free`**, **`printf`**, **`scanf`**
  - 例外：strtok、rand、ctime　Exceptions: **`strtok`**, **`rand`**, **`ctime`**
- POSIX添加了更多的异常，但也添加了不安全函数的可重入版本 POSIX adds more exceptions, but also reentrant versions of unsafe functions

线程不安全函数 Thread-unsafe function	Class	可重入版 Reentrant version
`asctime`	3	`strftime`
`ctime`	3	`strftime`
`gethostbyaddr`	3	`getnameinfo`
`gethostbyname`	3	`getaddrinfo`
`inet_ntoa`	3	`getnameinfo`
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

议题 Today

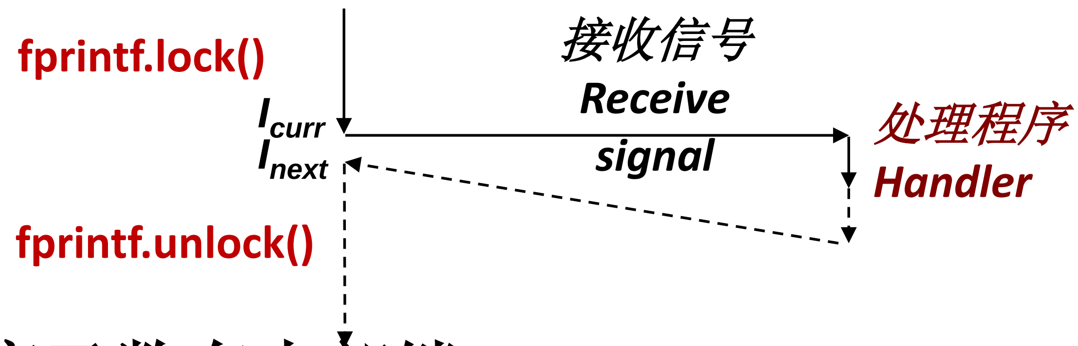
- 回顾：信号量、互斥和生产者-消费者 Review: Semaphores, mutexes, producer-consumer
- 使用信号量调度共享资源 Using semaphores to schedule shared resources
  - 读者-写者问题 Readers-writers problem
- **其它并发问题 Other concurrency issues**
  - 竞争 Races
  - 死锁 Deadlocks
  - 线程安全 Thread safety
  - **线程和信号处理之间交互 Interactions between threads and signal handling**

信号处理回顾 Signal Handling Review



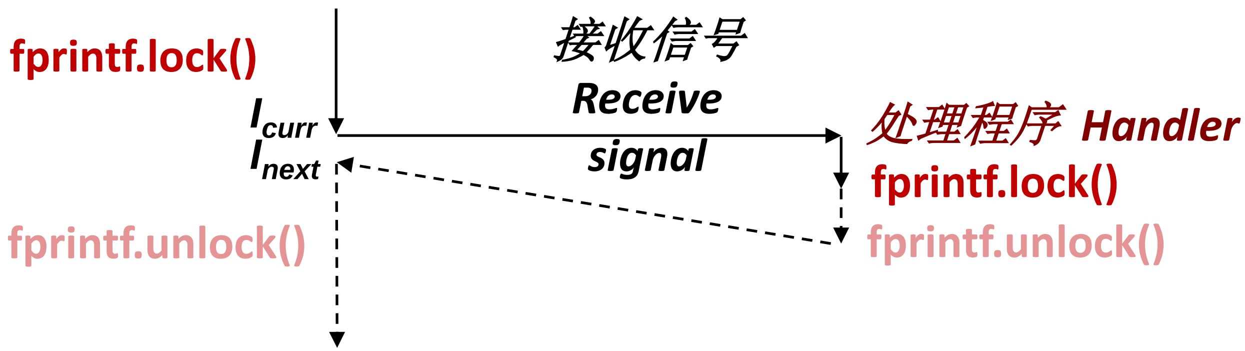
- 动作 Action
  - 信号可以发生在程序执行的任何点 Signal can occur at any point in program execution
    - 除非信号被阻塞 Unless signal is blocked
  - 信号处理程序在同一个线程内运行 Signal handler runs within same thread
  - 必须运行到完成，然后返回到正常的程序执行 Must run to completion and then return to regular program execution

线程/信号交互
Threads / Signals Interactions



- 很多库函数有内部锁 **Many library functions have internal locking**
  - 为了保护隐藏状态（避免第一类线程不安全） To protect hidden state (avoid being class 1 thread-unsafe)
  - malloc
    - 释放列表 Free lists
  - fputs, fprintf, snprintf
    - 以便从多个线程的输出不会交错 So that outputs from multiple threads don’t interleave
    - 内部使用malloc Internal use of malloc
- 不使用这些库函数的处理程序没有问题 **OK for handler that doesn’t use these library functions**

有问题的线程/信号交互
Bad Thread / Signal Interactions



- 如果以下情况会怎样：**What if:**
  - 当库函数保持加锁时接收信号 Signal received while library function holds lock
  - 处理程序调用同样（或相关）库函数 Handler calls same (or related) library function
- 死锁！ **Deadlock!**
  - 信号处理程序不能继续直到获得锁 Signal handler cannot proceed until it gets lock
  - 主程序不能继续直到处理程序完成 Main program cannot proceed until handler completes
- 关键点 **Key Point**
  - 线程采用对称并发 Threads employ symmetric concurrency
  - 信号处理是异步的 Signal handling is asymmetric

处理线程/信号交互
Handling Thread/Signal Interactions

- 临界区周围阻塞信号 **Block signals around critical sections**
  - pthread_sigmask函数类似sigprocmask，但是仅影响正调用的线程 pthread_sigmask – like sigprocmask, but only affects calling thread
- 专用于信号处理的线程 **Dedicate a thread to signal handling**
  - 循环调用sigsuspend()或sigwaitinfo() Loop calling sigsuspend() or sigwaitinfo()
  - 所有其他线程阻塞所有信号 All other threads block all signals
  - 信号处理线程可以使用异步信号不安全函数 Signal handling thread can use async-signal-unsafe functions
    - 因为我们知道信号只能在sigsuspend()期间传递 Because we know signals will only be delivered during sigsuspend()

线程小结 Threads Summary

- 线程为编写并发程序提供了另一种机制 **Threads provide another mechanism for writing concurrent programs**
- 线程越来越受欢迎 **Threads are growing in popularity**
  - 比进程开销小 Somewhat cheaper than processes
  - 易于在线程之间共享数据 Easy to share data between threads
- 然而，共享的便捷性有代价： **However, the ease of sharing has a cost:**
  - 易于引入细微的同步错误 Easy to introduce subtle synchronization errors
  - 小心对待线程！ Tread carefully with threads!

- 有关详细信息： **For more info:**
  - “Posix线程编程” D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997