第五章：多线程并发编程

随着多核处理器的普及和大规模并发计算的需求增加，多线程并发编程已经成为提高应用程序性能的核心技术。但多线程并发编程在带来性能提升的同时，也增加了编程的复杂性，因此，系统理解和掌握多线程并发编程的技巧变得尤为重要。本章将从多线程编程的角度，讨论并发编程的重要技术。本章首先从线程的基本概念和执行模型入手，在此基础上展开讨论线程的生命周期；而后，本章讨论线程同步与互斥的各种机制，如信号灯、互斥量、自旋锁和条件变量等；接着，本章讨论原子变量，并讨论如何通过原子变量来实现线程同步互斥的几种机制；最后，本章讨论并发和无锁数据结构，这既是并发编程的重要应用和重要发展方向。

5.1 线程的概念

5.1.1 概念

线程是进程内的一个独立执行流，即进程内的多个线程既有独立的执行代码、又共享内容和磁盘等计算资源，以共同协同的方式，完成特定计算任务。

和进程执行模型比较，线程具有多方面独特的优势。首先，线程模型比进程模型更容易表达特定的计算过程。例如，一个游戏软件可能同时要处理用户输入和显示动画效果，则可以利用两个线程，并行完成这两个计算任务。其次，线程可以更高效的利用计算资源。尤其是在现代的多核处理器上，多个核可以同时执行多个线程；最后，线程间可以更高效的通信和同步，由于线程间共享内存，可以直接进行线程通信，而不用操作系统内核的特殊支持。

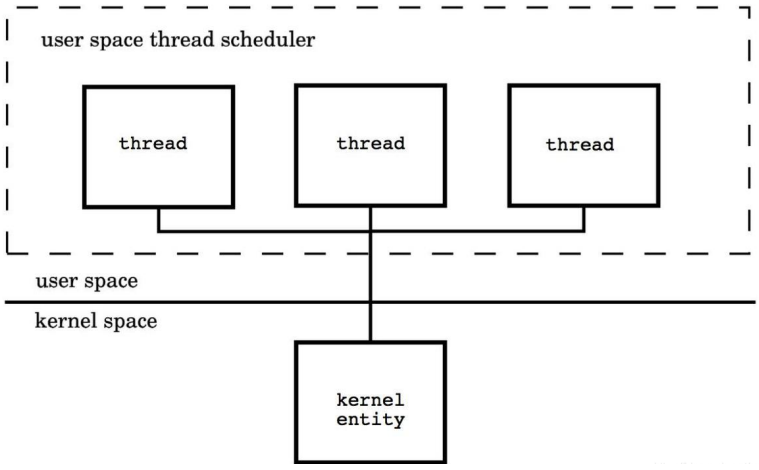
但是，由于线程共享内存等计算资源，因此，和进程相比，线程也带来了更大的编程挑战。首先，线程编程模型比进程更加复杂，对线程间共享变量的处理需要特别谨慎，否则很容易引发竞态条件的问题。其次，线程的内存模型也更加复杂，带来了原子、顺序、和可见等新的问题；最后，一个进程内的线程之间缺少隔离，如果一个线程异常退出，则所有线程以及该进程都将执行退出。

5.1.2 执行模型

线程作为一个独立的执行流，一个进程内的线程既拥有自己独立的资源，也共享部分计算资源。在线程的私有资源中，线程号是线程的唯一标识符，执行优先级决定其与其他线程的执行顺序，errno 用于记录系统调用发生的错误代码，寄存器状态保存线程上下文切换时的信息，私有存储存放线程特定的数据，函数调用栈保存每个线程的执行信息。在线程的共享资源中，全局变量是在程序的全局范围内定义的变量，堆用来存储线程创建的对象等信息，除此之外，所有线程共享同一块代码段。

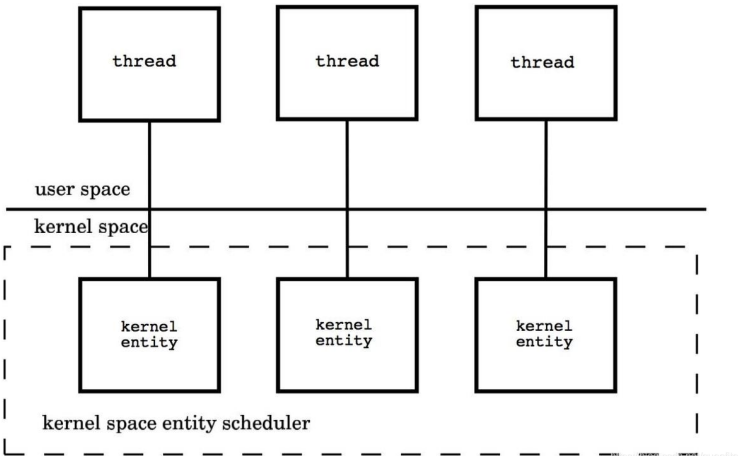
线程私有	线程共享
线程号 执行优先级 errno 调用栈 寄存器状态 私有存储	全局变量 堆 代码

线程模型分，按照线程和具体执行实体（一般是处理器）之间的对应关系，可分为三种，分别是多对一的用户级线程模型，一对一的内核级线程模型，多对多的两级线程模型。

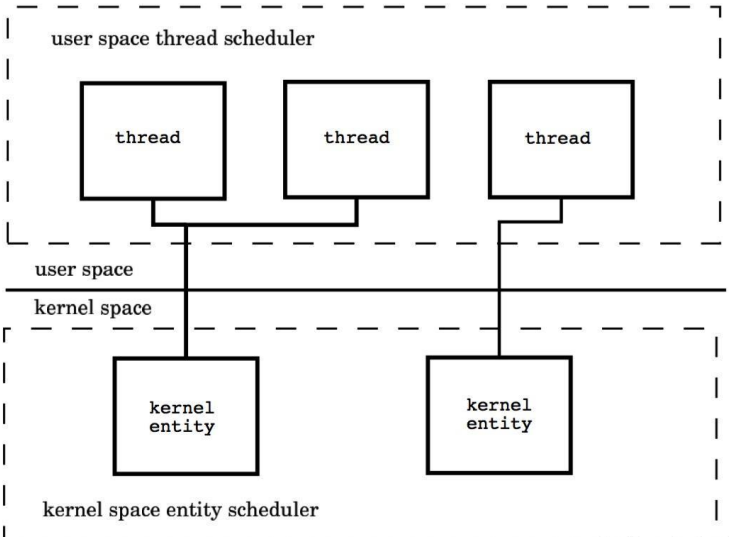


首先，在多对一的用户级线程模型中，线程的创建/调度/同步等操作由进程的用户空间线程库来处理，用户级线程模型的很多操作对内核来讲，都是透明的，不需要内核参与，因此对于线程的处理速度非常的快。但当多线程并发执行时，只会有一个线程在真正执行，其他线程处于就绪态等待。早期 Java 虚拟机实现的蓝线程属于多对一

的线程模型。



其次，在一对一的内核级线程模型中，每个用户线程都对应各自的内核调度实体，内核对每个线程进行调度，多个线程可以真正并发执行；但线程操作会在用户态和内核态直接进行切换，消耗一定的资源。Posix 库中的 pthread 线程属于一对一的线程模型。



最后，在多对多的两级线程模型中，多个线程对应多个执行核心。该执行模型综合上述两种线程模型的优点，多个线程可以拥有多个调度实体，同时，也可以多个线程对应一个调度实体。Go 语言的协程就属于这种线程模型。

为篇幅起见，本章的讨论集中在一对一的线程模型，即我们主要讨论 Posix 中定义的 pthread 线程，这也是 Linux 系统上实现的标准线程模型。

5.2 线程生命周期

线程的生命周期涵盖了从其创建、执行、到终止的所有阶段。下列是常见的函数原型：

```
#include <pthread.h>
int pthread_create(pthread_t *thd, const pthread_attr_t *attr,
    void *(*start)(void *), void *arg);
void pthread_exit(void *);
int pthread_join(pthread_t thd, void **ptr);
int pthread_detach(pthread_t thd);
pthread_t pthread_self(void);
```

其中，pthread_create()用于创建一个新的线程，thd 指向的变量会被设置为新线程的线程编号 ID，attr 指向 pthread_attr_t 的指针，用于设置线程属性，start 是函数指针，指向线程开始执行的函数入口，arg 是传递给 start 函数的参数；pthread_exit()用于终止调用它的线程；pthread_join()阻塞当前线程，直到指定的线程退出，thd 是要等待的线程的线程 ID，ptr 指针用来接收结束线程的返回值；pthread_detach()使线程 thd 进入分离状态，当线程结束时，它的资源会自动被系统回收，不需要其他线程调用程调用 pthread_join()来进行清理；pthread_self()返回当前调用该函数的线程的线程号 ID。

5.2.1 创建

线程的创建是多线程程序的起点，在 POSIX 线程库中由 pthread_create()函数完成。我们使用一个简单的例子来介绍线程的创建，在示例中新创建的线程打印它自身的线程 ID 和主线程传递给它的参数：

```
void *start(void *arg){
    printf("thread ID = %d\n", (unsigned long)pthread_self());
    printf("thread argument = %ld\n", (long)arg);
    pthread_exit("thread return");
}
int main(void){
    pthread_t tid;
    void *ret;
    pthread_create(&tid, NULL, start, (void *)10);
    pthread_join(tid, &ret);
    return 0;
}
```

该示例首先使用 pthread_create 创建线程（第 9 行），指定线程创建后的执行入口函数为 start，传递给线程的参数为 10；然后，函数 start 调用 pthread_self 打印自身线程的 ID（第 2 行），并打印传递给线程的参数（第 3 行），最后调用 pthread_exit 退出线程（第 4 行）。主函数 main 调用 pthread_join 等待线程 tid 执行完，并用 ret 来存放其返回值。

5.2.2 执行

线程的执行是自动的，一旦线程被 pthread_create 函数创建，它就进入执行阶段，一直到调用 pthread_exit 操作退出执行。

在下面的例子中，我们创建了 1000 个线程，这 1000 个线程分别计算 0~999 的平方，主线程将结果进行求和：

```
void *start(void *a){
    long n = (long)a;
    printf("this is thread: %ld\n", n);
    pthread_exit((void *)(n * n));
}
int main(){
    int N = 1000;
    pthread_t tids[N];
    for (long i = 0; i < N; i++) {
        pthread_create(&tids[i], 0, start, (void *)i);
    }
    void *res;
    long sum = 0;
    for (long i = 0; i < N; i++) {
        pthread_join(tids[i], &res);
        sum += (long)res;
    }
    return 0;
}
```

其中，主线程使用 pthread_join 阻塞等待创建的每个线程返回结果（第 15 行），然后在主线程中收集每个线程的计算结果并进行累加（第 16 行）。

5.2.3 终止

线程的执行可以通过多种方式终止，我们将结合实例，详细探讨每种方法。

当线程的启动函数执行完毕并返回时，对应的线程会自动终止。在下面的例子中，当 start 函数执行完毕并返回时，线程执行结束。

```
void* start(void* arg){
    printf("Thread finishing...\n");
    return NULL;
}
```

线程在其执行过程中的任何位置可以调用 pthread_exit()来终止执行。在下面的例子中，线程打印一条消息后立即调用 pthread_exit 结束执行。

```
void* start(void* arg){
    printf("Thread finishing using pthread_exit...\n");
    pthread_exit(NULL);
}
```

线程间可以互相取消，主要通过 pthread_cancel()实现。在下面的例子中，新创建的线程是一个无限循环的线程，主线程在 3 秒后请求取消这个线程（第 12 行）。

```
void* start(void* arg) {
    while(1) {
        printf("Thread running...\n");
        sleep(1);
    }
}
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, start, NULL);
    sleep(3);
    pthread_cancel(tid);
    pthread_join(tid, NULL);
    return 0;
}
```

如果任何线程（包括主线程）在其执行过程中调用 exit()，或者主线程从 main()函数返回，进程将结束执行，则该进程内的所有线程都会终止执行。在下面的例子中，尽管新创建的线程是一个无限循环的线程，但主线程在 3 秒后调用 exit(0)，导致整个进程和它的所有线程都立刻结束。

```
void* start(void* arg){
    while(1) {
        printf("Thread running...\n");
        sleep(1);
    }
}
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, start, NULL);
    sleep(3);
    exit(0);
}
```

5.3 同步与互斥

同步和互斥是并发编程中两个核心概念，用于确保多个线程协同工作时，以正确和可预测的方式访问共享资源，从而保证执行结果的正确性。同步关注于使多个线程或进程按照特定的顺序或条件执行，确保某些操作在其他操作之前或之后完成，从而达到协调各个线程的目的。互斥则是确保在同一时间内，只有一个线程或进程可以访问或修改特定的资源或代码段（称为临界区），以避免出现数据竞态条件和不一致性。

5.3.1 竞态条件与临界区

竞态条件（Race Condition）指的是多个线程或进程同时访问和修改同一数据（至少有一方是写操作），从而导致最终执行结果依赖于执行顺序的现象。由于多线程执行时，其执行顺序是不确定的，导致最终结果也是不确定的。

在下面的例子中，两个线程都试图调用 inc 函数，增加一个共享变量 count 的值：

```
#define NUM_ITERATIONS 10000
int count = 0;
void *inc(void *arg){
    for (int i = 0; i < NUM_ITERATIONS; i++)
        count++;
    pthread_exit(0);
}
int main(){
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, inc, NULL);
    pthread_create(&tid2, NULL, inc, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Expected count: %d\n", 2 * NUM_ITERATIONS);
    printf("Actual count: %d\n", count);
    pthread_exit(0);
}
```

程序执行后，输出的共享全局变量 count 的值一般是小于 20000 的某个值，但一般不会严格等于 20000。

导致这个问题的根本原因在于：在源语言层面看起来像是原子的操作 count++，实际上被编译器编译为汇编程序后，一般会包含类似如下的三个执行步骤：

```
ld %r, count      # 加载变量 count 的值到寄存器 r
inc %r            # 对寄存器 r 的值自增
st %r, count      # 将寄存器 r 的值写回 count
```

即读取 count 的当前值到寄存器 r，对寄存器 r 加 1，将自增后的值写回 count。这样，当一个线程读取 count 的值并尝试写回一个增加的值时（第 9 行），其他线程可能也已经读取了相同的值，并试图将其增加，从而导致结果小于预期的正确结果。

为了解决这个问题，我们引入“临界区”的概念。临界区是程序中访问共享资源的一段代码，要确保在任何时刻只有一个线程进入，从而有效避免竞态条件。从这个角度

看，临界区提供了一种机制，确保对共享资源的序列化访问，但也降低了系统的并发执行程度。

为了确保临界区代码的正确执行，我们需要确保同一时刻只有一个线程进入临界区。这可以通过以下同步机制来实现：

同步机制	说明
信号灯（Semaphore）	一个用于临界区保护的同步原语，用于控制多个线程或进程对共享资源的并发访问数量。也可以用于实现其他复杂的同步策略，如生产者-消费者问题。
互斥量（Mutex）	一个同步原语，用于确保同一时刻只有一个线程可以进入临界区。线程必须首先获取互斥量才能进入临界区，如果其他线程已经获取了互斥量，那么尝试获取互斥量的线程会被阻塞，直到互斥量被释放。
自旋锁（Spinlocks）	当线程尝试获取自旋锁而锁已被其他线程占用时，线程不会被阻塞，而是持续检查锁是否可用。这适用于临界区执行时间非常短的情况。
条件变量（Condition Variables）	允许线程在特定条件下暂停执行并等待，直至该条件满足时被唤醒继续执行。经常与互斥量结合使用。

在接下来的几个小节，我们深入探讨以上四个同步机制。

5.3.2 信号灯（Semaphore）

信号灯（Semaphore）可以在多线程环境下用于协调各个线程，以保证它们能够正确、合理的使用公共资源。信号灯维护了一个许可集，我们在初始化信号灯时需要为这个许可集设置一个数量值，该数量值代表同一时间能访问共享资源的线程数量。

信号灯按实现原理分两种，一种是有名信号灯，一种是基于内存的信号灯，也叫无名信号灯。有名信号灯，是根据外部名字标识，通常指代文件系统中的某个文件。

而无名信号灯，它可以把信号灯放入内存。

Linux 提供了一组函数调用，来实现有名信号灯的功能，其头文件和主要函数的原型为：

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, ...  /* mode_t mode,
unsigned int value */ );
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

其中，sem_open 用于创建一个信号灯并设置其初值，或者打开一个已经存在的信号灯。线程需要访问公共资源或进入临界区时需要调用 sem_wait，此时若信号灯的数值为 0，线程便进入阻塞态，否则信号灯的值减 1（对应 PV 操作中的 P 操作）。线程释放公共资源后，就需要调用 sem_post 增加信号灯的值（V 操作）。

下面的程序创建了两个线程，它们都等待一个初始值为 0 的信号灯，直到主线程增加该信号灯的值后，线程才能继续执行。

```
#define SEM_NAME "/my_semaphore"
sem_t *sem;
void *thread_entry1(void *arg){
    sem_wait(sem);
    pthread_exit(0);
}
void *thread_func2(void *arg){
    sem_wait(sem);
    pthread_exit(0);
}
int main() {
    pthread_t t1, t2;
    sem = sem_open(SEM_NAME, O_CREAT, 0666, 0);
    pthread_create(&t1, NULL, thread_entry1, NULL);
    pthread_create(&t2, NULL, thread_entry2, NULL);
    sem_post(sem);
    sem_post(sem);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sem_close(sem);
    sem_unlink(SEM_NAME);
    pthread_exit(0);
}
```

程序首先初始化一个值为 0 的信号灯 sem（第 13 行）。随后，程序创建两个线程 t1 和 t2（第 14、15 行），这两个线程在开始时尝试获取信号灯（第 4、8 行）。因为信号灯的初始值为 0，所以这两个线程会进入等待状态。主线程在创建完这两个线程后，连续执行两次 sem_post()（第 16 行、第 17 行），从而递增信号灯的值，放行那两个等待中的线程；但是，需要注意的是，这两个线程 t1 和 t2 的放行顺序是不确定的。最后，主线程回收子线程资源并销毁信号灯（第 20、21 行）。

无名信号灯和有名信号灯的差异主要体现在创建和销毁的方式上，但是其他工作都是一样的。无名信号灯只能存在于内存中，要求使用信号灯的进程必须能访问信号

灯所在的内存，所以无名信号灯只能应用在同一进程内的线程之间（共享进程的内存），或者不同进程中已经映射相同内存内容到它们的地址空间中的线程（即信号灯所在内存被通信的进程共享）。相反，有名信号灯可以通过名字访问，因此可以被任何知道它们名字的线程使用。

5.3.3 互斥量（Mutex）

互斥量（Mutex）是并发编程中的一种同步原语，设计用于保护共享资源不被多个线程同时访问，从而避免竞态条件。当一个线程成功获取互斥量后，其他尝试获取该互斥量的线程会被阻塞，直至该互斥量被释放。其核心原则是确保在任何时刻，只有一个线程可以持有互斥量，这为资源的互斥访问提供了保证。

pthread 库主要提供了以下几个关键函数，来实现互斥量的操作：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

其中，pthread_mutex_init 函数用于初始化互斥量，可以选择传入特定的属性或使用默认属性。当线程希望访问受互斥量保护的资源或代码段时，它应调用 pthread_mutex_lock 来尝试获取互斥量。如果互斥量已被其他线程持有，该线程将被阻塞，直到互斥量被释放。对于那些不希望被长时间阻塞的线程，可以使用 pthread_mutex_trylock 尝试获取互斥量，这个函数在互斥量已被其他线程持有时会立即返回一个非零值，而不是等待。最后，当线程完成对受互斥量保护的资源或代码段的访问后，它应该调用 pthread_mutex_unlock 释放互斥量。

下面的例子使用互斥量，来确保两个线程不会同时访问并修改同一个共享变量。

```
#define N 10000
struct counter_t {
    pthread_mutex_t mu;
    int counter;
};
struct counter_t shared_counter;
void *inc(void *arg) {
    for (int i = 0; i < N; i++) {
        pthread_mutex_lock(&shared_counter.mu);
        shared_counter.counter++;
        pthread_mutex_unlock(&shared_counter.mu);
    }
    pthread_exit(0);
}
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&shared_counter.mu, 0);
    pthread_create(&t1, NULL, inc, NULL);
    pthread_create(&t2, NULL, inc, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter: %d\n", shared_counter.counter);
    pthread_mutex_destroy(&shared_counter.counter_mutex);
    pthread_exit(0);
}
```

我们使用了互斥量 mu 对共享变量 counter 进行了保护（第 3 行），因此，线程需要在访问被保护的共享变量前（第 10 行），先获得这个互斥量（第 9 行），在访问结束后，释放这个互斥量（第 11 行）。第 9 行和第 11 行之间就构成了一个临界区，保证不会有两个或更多线程同时进入临界区访问共享变量 counter，从而确保了其正确性。

特别需要注意的问题是临界区的设计与系统并行度的辩证关系，即需要把程序代码中的哪些部分设计成临界区（从而是互斥的），才能既能保证系统的执行正确，又能

最大程度保证系统的并发度。例如，实现上述 inc 函数的另一个可能的方法是：

```
struct counter_t shared_counter;
void *inc(void *arg){
    pthread_mutex_lock(&shared_counter.mu);
    for (int i = 0; i < N; i++) {
        shared_counter.counter++;
    }
    pthread_mutex_unlock(&shared_counter.mu);
    pthread_exit(0);
}
```

即整个循环体都被放在临界区中（第 3 行和第 7 行之间）。尽管这种临界区的设计方式能够保证程序不会出现竞态条件，从而保证执行结果的正确性；但是，这种解决解决方案本质上将两个线程进行了整体排序，从而降低了系统执行的并行度。总的说，一般原则是在保证资源不出现竞态条件的前提下，临界区尽可能小。

5.3.4 自旋锁（SpinLock）

自旋锁（SpinLock）是一种低开销的同步机制，用于保护访问时间较短的共享资源。与传统的互斥量不同，当线程尝试获取一个已被占用的自旋锁时，它不会进入休眠状态，而是持续检查锁的状态，直到锁被释放，这种现象称为争用（contention）。由于自旋锁避免了线程上下文切换的开销，对于预期竞争时间短的场景，通常能提供更高的效率；但在高竞争场景下（即多线程对资源的争抢特别激烈），也可能导致 CPU 资源的浪费。

在 pthread 库中，自旋锁的操作主要涉及以下几个关键函数。

```
#include <pthread.h>
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock (pthread_spinlock_t *lock);
int pthread_spin_trylock (pthread_spinlock_t *lock);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

其中，pthread_spin_init 函数用于初始化一个自旋锁，接受一个锁的指针和一个标志来决定这个锁是否应该在进程间共享。为了获取自旋锁，线程使用 pthread_spin_lock 函数，如果锁已经被其他线程持有，它将一直尝试获取直到成功。pthread_spin_trylock 函数尝试获取锁，但如果锁已被其他线程持有，它将立即返回，而不会自旋。当线程完成其对受保护资源的访问后，它使用 pthread_spin_unlock 来释放锁，从而允许其他线程获取锁。最后，函数 pthread_spin_unlock 释放一个自旋锁。

下面的例子使用自旋锁，来保护对共享变量 counter 的访问。

```
#define N 10000
struct counter_t{
    pthread_spinlock_t spinlock;
    int counter;
};
struct counter_t shared_data;
void *inc(void *arg) {
    for (int i = 0; i < N; i++) {
        pthread_spin_lock(&shared_data.spinlock);
        shared_data.counter++;
        pthread_spin_unlock(&shared_data.spinlock);
    }
    pthread_exit(0);
}
int main() {
    pthread_t t1, t2;
    pthread_spin_init(&counter->spinlock,
PTHREAD_PROCESS_PRIVATE);
    pthread_create(&t1, NULL, inc, NULL);
    pthread_create(&t2, NULL, inc, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter: %d\n", shared_data.counter);
    pthread_spin_destroy(&shared_data.spinlock);
    return 0;
}
```

此例子和之前的互斥量例子类似，只是换成了自旋锁（第 3 行）。我们不再赘述。

5.3.5 条件变量（Conditional variable）

条件变量（Conditional variable）是一种同步机制，用于在特定条件不满足时使线程进入休眠，并在该条件满足时将该线程唤醒继续。它通常与互斥量等锁机制结合使

用，避免了忙等（busy-waiting）导致的资源浪费，提供了一种高效的方式来在多线程环境中进行条件等待与通知，实现更好的线程间交互。

在 pthread 库中，条件变量的操作主要涉及以下几个关键函数。

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const
pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
*mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

其中，pthread_cond_init 函数根据所提供的属性 attr（或默认属性）来初始化条件变量 cond。当线程需要等待某个条件 cond 成真时，它会调用 pthread_cond_wait 函数，休眠在该条件变量 cond 上；此函数同时需要一个互斥量 mutex，并且该互斥量 mutex 在调用此函数之前应该被锁定，在调用该函数后被释放。当另一个线程进行操作，使条件 cond 成真后，则可以调用 pthread_cond_signal，唤醒等待的某一个线程，或者也可以调用 pthread_cond_broadcast，来唤醒等待该条件 cond 的所有线程。一旦不再需要某条件变量 cond，使用 pthread_cond_destroy 将其销毁，以释放关联的资源。

利用条件变量，我们可以优雅的解决许多需要线程进行协作来完成特定任务的问题。下面，我们先讨论生产者-消费者问题，在最简单的单生产者-单消费者问题中，有一个生产者负责生成数据并将其放入缓冲区，而另一个消费者线程，负责从缓冲区中取出数据并处理它。我们用两个线程来分别模拟生产者和消费者，并使用条件变量和互斥量来同步这两个线程。

```
#define N 10
struct buffer_t {
    pthread_mutex_t mutex;
    pthread_cond_t cond_full;
    pthread_cond_t cond_empty;
    int data[N];
    int items;    // how many items reside in this buffer
    int write_index;
    int read_index;
};
struct buffer_t buffer;
void init_buffer(struct buffer_t *buffer) {
    pthread_mutex_init(&buffer->mutex, NULL);
    pthread_cond_init(&buffer->cond_full, NULL);
    pthread_cond_init(&buffer->cond_empty, NULL);
}
void *producer(void *arg) {
    while(1) {
        pthread_mutex_lock(&buffer.mutex);
        while(buffer.items == N){
            pthread_cond_wait(&buffer.cond_full, &buffer.mutex);
        }
        buffer.data[buffer.write_index] = i;
        buffer.items++;
        printf("Produced %d\n", i);
        buffer.write_index = (buffer.write_index + 1) % N;
        pthread_cond_signal(&buffer.cond_empty);
        pthread_mutex_unlock(&buffer.mutex);
    }
    pthread_exit(0);
}
void *consumer(void *arg) {
    while(1) {
        pthread_mutex_lock(&buffer.mutex);
        while (buffer.items == 0) {
            pthread_cond_wait(&buffer.cond_empty, &buffer.mutex);
        }
```

这个程序用一个缓冲区结构体 buffer_t（第 2 行）来实现共享的读-写数据区，它利用互斥量和条件变量来同步和通信，其中互斥量 mutex（第 3 行）保护整个缓冲区，条件变量 cond_full（第 4 行）标识缓冲区已满，而条件变量 cond_empty 标识缓冲区为空；则可知生产者当缓冲区为满时需要等待，而消费者当缓冲区为空时需要等待。主函数分别生成两个线程（第 49、50 行）prod 和 cons，分别向缓冲区中添加和取走数据。

生产者线程 producer（第 17 行）在缓冲区不满时向其添加数据（第 23 到 26 行），这些操作都在处于临界区中，处于互斥量 mutex 的保护中（第 19 行）；而当缓冲区已满时（第 20 行），生产者需要调用 pthread_cond_wait 函数，等待在条件变量 cond_full 上，同时释放已经持有的信号量 mutex；需要特别注意的是，当条件变量 cond_full 不满足时，该线程会被唤醒，已进行接下来的执行，但是它会循环检查所等等的条件是否真的成立（第 20 行），这主要是因为 Linux 系统中存在虚假唤醒（supuirous wakeup）的问题，即线程在条件变量不满足时，也可能会被唤醒；如果遇到虚假唤醒，则循环条件判断仍然成立（第 20 行），线程会继续调用 pthread_cond_wait 函数进行等待（第 21 行）。

类似的，消费者线程 consumer（第 32 行）从缓冲区中读取数据，当缓冲区为空时，需要进行等待（第 36 行）。其处理逻辑与生产者线程的处理逻辑类似，我们将其分析作为练习，留给读者完成。

5.4 原子变量及其应用

原子变量是指在多线程环境下，能够保证特定的变量操作能够原子性执行的变量。由于高级语言层面的一个操作，往往对应于底层语言中的一个操作序列，因此原子变量能够在高级语言层面而不是底层，就提供操作的强原子性保证。

本小节，我们重点讨论原子变量的概念，以及其在实现其它同步原语中的作用。在后续小节，我们还将讨论原子变量在无锁数据结构中的作用。

5.4.1 原子变量

C 语言并没有提供对原子变量类型及其操作的内置语言支持，而是在 C11 及其后的版本中，通过库函数的形式（头文件 stdatomic.h），提供了一系列的原子数据类型及

其上的原子操作。C 语言的原子变量头文件，包括若干个典型的原子类型及其操作：

```
#include <stdatomic.h>
typedef _Atomic _Bool atomic_bool;
typedef _Atomic char atomic_char;
typedef _Atomic int atomic_int;
void atomic_init(volatile A *obj, C desired);
void atomic_store(volatile A *obj , C desired);
C atomic_load(const volatile A *obj);
_Bool atomic_compare_exchange_strong(volatile A *obj, C
*expected, C desired);
C atomic_fetch_add(volatile A *obj, C operand);
```

其中，atomic_bool、atomic_char、atomic_int 是原子数据类型，分别表示存储布尔值、字符、整型的原子变量。函数 atomic_init 用于初始化原子变量 obj，将其设置为 desired；函数 atomic_store 将 desired 的值存储到原子变量 obj 中；函数 atomic_load 从原子变量 obj 中读取值并返回；函数 atomic_compare_exchange_strong 比较原子变量 obj 的值和 expected 的值：如果两个值相等，则将 obj 的值设置为 desired 并返回 true；否则如果不相等，则将 obj 的当前值存储到 expected 中并返回 false。函数 atomic_fetch_add 将 operand 的值加到原子变量 obj 的值上，并返回操作之前的值。

有了原子变量，我们可以在不使用锁的前提下（即无锁），实现线程间的同步操作，从而保证结果的正确性。例如，我们可以用一个原子变量，重新实现实现前面讨论过的自增计数器：

```
#define N 10000
atomic_int acnt;
int cnt;
void *start(void *arg) {
    for (int n = 0; n < N; n++) {
        cnt++;
        atomic_fetch_add(&acnt, 1); // Use atomic functions
    }
    pthread_exit(0);
}
int main(void) {
    pthread_t t1, t2;
    atomic_init(&acnt, 0);  // Initialize atomic variable
    pthread_create(&t1, NULL, start, NULL);
    pthread_create(&t2, NULL, start, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    int atomic_cnt = atomic_load(&acnt);  // Load atomic variable
    printf("The atomic counter is: %d\n", atomic_cnt);
    printf("The non-atomic counter is: %d\n", cnt);
    return 0;
}
```

其中，我们声明了一个 atomic_int 类型的原子变量 acnt（第 2 行）。主线程将该原子变量初始化为 0（第 13 行），然后启用了两个线程执行线程函数 start（第 4 行），每个线程执行 10000 次原子变量加法（第 7 行）。最后，主线程中等待两个线程的结束后，输出原子变量的值（第 19 行）。程序的执行结果如下：

```
The atomic counter is: 20000
The non-atomic counter is: 16727
```

可以看到，原子变量 acnt 生成了正确的结果；而非原子变量 cnt 由于竞态条件的

影响，导致了错误的执行结果。

用原子变量可以实现其它同步机制。接下来，我们以自旋锁和信号灯两个机制为例，讨论用原子变量来实现这些同步机制的基本技术。

5.4.2 自旋锁的实现

自旋锁（Spin Lock）是一种低开销的同步机制，用于短时间内保护共享资源，我们在上一小节已经讨论过自旋锁，下面使用原子变量和原子操作实现自旋锁：

```
struct Spinlock_t{
    atomic_int locked;
};
typedef struct Spinlock_t spinlock_t;
void spinlock_init(spinlock_t *lock){
    atomic_init(&lock->locked, 0);
}
void spinlock_lock(spinlock_t *lock){
    int locked = 0;
    while(!atomic_compare_exchange_strong(&lock->locked, &locked,
1)){
        // busy-wait
    }
}
void spinlock_unlock(spinlock_t *lock){
    atomic_store(&lock->locked, 0);
}
```

首先我们声明了自旋锁的结构体，其中包括一个原子整型 atomic_int 类型的原子变量 locked（第 2 行），其值为 0 的时候，代表自旋锁可用；否则，当其值为 1 的时候，代表自旋锁已被占用。spinlock_init 函数将原子变量 locked 初始化为 0。函数 spinlock_lock 实现加锁，该函数使用原子的比较并交换操作 atomic_compare_exchange_strong（第 10 行），尝试将原子变量 locked 的值置为 1，并返回尝试的结果：如果返回 false，则说明自旋锁正在被占用，需要继续循环，重新尝试加锁；否则，该函数已经将该自旋锁锁定。函数 spinlock_unlock 实现解锁，即将

原子变量 locked 的值置为 0。

我们自己实现的自旋锁，和系统引入的自旋锁的使用方式相同。例如，对上一小节计数器的例子，使用我们实现的自旋锁，代码如下：

```
typedef struct {
    spinlock_t lock;
    int count;
} counter_t;
counter_t counter;
void *start(){
    for(int i = 0; i < 10000; i++){
        spinlock_lock(&counter.lock);
        counter.num++;
        spinlock_unlock(&counter.lock);
    }
    pthread_exit(0);
}
int main(){
    pthread_t tid1, tid2;
    spinlock_init(&counter.lock);
    pthread_create(&tid1, NULL, start, NULL);
    pthread_create(&tid2, NULL, start, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Counter= %d\n", counter.num);
    return 0;
}
```

上述代码除了使用了新的自旋锁实现外，其它并没有本质的区别；我们把对该代码的分析留给读者完成。

5.4.3 信号灯的实现

信号灯通常维护一个整数值用来表示可用的资源数量，在本节中，我们采用原子变

量来实现信号灯，代码如下所示：

```
typedef struct {
    atomic_int value;
} sema_t;
void sema_init(semaphore_t *sem, int init_val) {
    atomic_init(&sem->value, init_val);
}
void sema_wait(semaphore_t *sem){
    while (1) {
        int val = atomic_load(&sem->value);
        if (val > 0 &&
atomic_compare_exchange_strong(&sem->value, &val, val - 1))
            break;
    }
}
void sema_post(sema_t *sem){
    atomic_fetch_add(&sem->value, 1);
}
```

我们首先声明了信号灯数据结构 sema_t（第 1 行），其中声明了原子整型变量 value（第 2 行），表示可用资源的数目。函数 sema_init 对信号灯的初始化，将 value 值设为初始值 init_val（第 5 行）。函数 wait 使用一个 while 循环和原子操作 atomic_compare_exchange_strong，来确保只有当信号灯的值大于 0 时，线程才会尝试对其减 1，如果成功则跳出循环并返回（第 11 行）。最后，post 方法中，使用原子操作 atomic_fetch_add 将资源数目 value 加 1。

我们自己实现的信号灯 sema_t，和系统中的信号灯 sem_t，具有类似的操作接口。因此，实际代码可利用其完成同步功能。例如，对于上述计数器程序，其利用信号灯的代码实现如下：

```
typedef struct {
    sema_t sem;
    int num;
} counter_t;
counter_t counter;
void *start(void *arg) {
    for (int i = 0; i < 10000; i++) {
        wait(&counter.sem);
        counter.num++;
        post(&counter.sem);
    }
    pthread_exit(0);
}
int main() {
    pthread_t tid1, tid2;
    sema_init(&counter.sem, 1);
    pthread_create(&tid1, NULL, start, NULL);
    pthread_create(&tid2, NULL, start, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("Counter= %d\n", counter.num);
    return 0;
}
```

上述代码声明了被信号灯保护的变量 counter（第 5 行），且信号灯的初始值被置为 1（第 16 行）。每个线程在执行时，都是先通过调用函数 sema_wait 获取信号灯（第 8 行），等临界区中的操作完成后，再通过调用函数 sema_post 释放信号灯（第 10 行）；这样就保证了线程执行的同步。

5.5 并发数据结构

并发数据结构能够支持多个并发执行线程对数据的访问和修改，而不会导致出现

竞争条件等并发问题。并发数据结构的设计和实现需要综合考虑执行效率、数据结构的特性、线程安全等问题，因此比单线程数据结构的设计和实现更有挑战性。在本小节，我们将结合并发栈和并发哈希两种数据结构的实现，讨论并发数据结构设计的一般规律和原则。

5.5.1 并发栈

栈是一种支持后入先出操作的常用数据结构，支持入栈 push、出栈 pop、栈的大小 size 等典型操作。并发栈能够支持多线程并发访问，即并发执行上述栈操作。

为了支持并发栈，我们引入如下的并发栈数据结构 con_stack_t:

```
typedef struct con_stack_t{
    pthread_mutex_t mu;
    int buf[N];
    int top;
} con_stack_t;

void con_stack_init(cons_stack_t *stack);
void con_stack_push(con_stack_t *stack, int value);
int con_stack_pop(con_stack_t *stack);
int con_stack_size(con_stack_t *stack);
```

在这个并发栈数据结构 con_stack_t 中（第 1 行），我们设置了一个互斥量 mu 来保护栈的数据（第 2 行），我们用数组 buf 表示一个顺序栈，栈顶下标 top 代表下一个可以插入元素的位置，其初始值为 0。

函数 con_stack_init 对给定的栈进行初始化:

```
void con_stack_init(con_stack_t *stack){
    pthread_mutex_init(&stack->mu, 0);
}
```

函数 push 将给定的元素入栈:

```
void con_stack_push(con_stack_t *stack, int value){
    pthread_mutex_lock(&stack->mu);
    stack->buf[stack->top++] = value; // 简单起见，未考虑溢出
    pthread_mutex_unlock(&stack->mu);
}
```

只有对元素的入栈操作处于临界区内（第 3 行）；为简单起见，这里省略了栈的溢出情况。感兴趣的读者，可以利用条件变量，对溢出的边界情况进行处理。

类似的，函数 pop 从栈顶将元素弹出：

```
int con_stack_pop(con_stack_t *stack){
    int r = 0;
    pthread_mutex_lock(&stack->mu);
    r = (stack->top == 0)? -1: stack->buf[--stack->top];
    pthread_mutex_unlock(&stack->mu);
    return r;
}
```

同样，简单起见，这里对栈为空的情况，直接返回了一个平凡的值；在更复杂的实现中，可以利用条件变量进行等待。

最后，函数 size 计算栈中当前元素的数量：

```
int con_stack_size(con_stack_t *stack){
    int r = 0;
    pthread_mutex_lock(&stack->mu);
    r = stack->top;
    pthread_mutex_unlock(&stack->mu);
    return r;
}
```

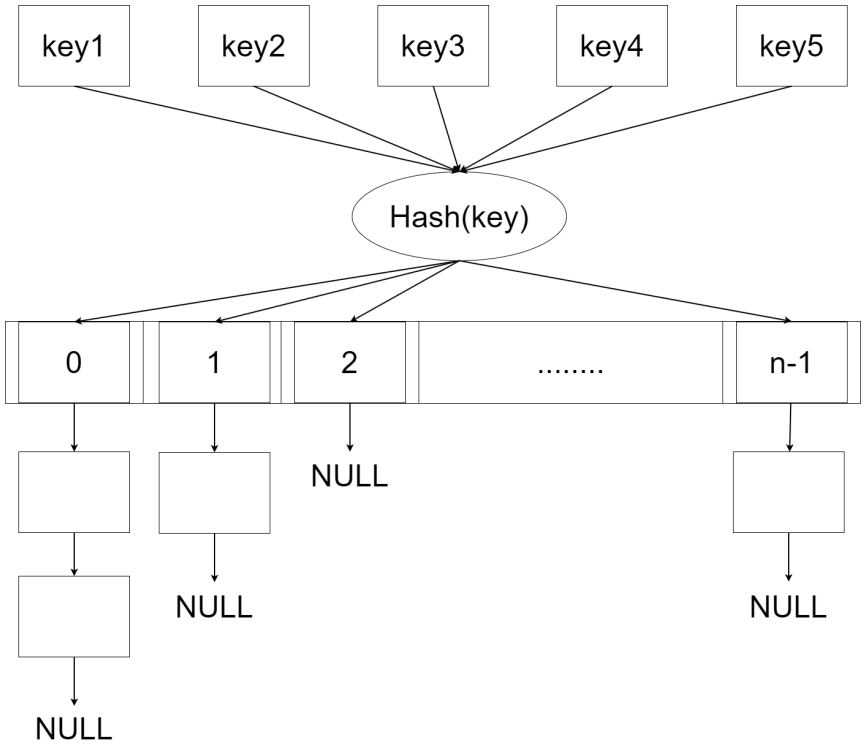
并发数据结构和传统的单线程数据结构相比，有很多相似点，但又有很多本质的区别。一个重要的区别是：对数据结构状态的观察，不是太有意义。以上面计算栈元素数量的函数 size 为例，由于多线程并发的存在，但一个线程计算栈的元素数量并返回

后，可能马上就进行了大量的入栈、出栈操作，栈的状态已经发生了剧烈的变化。

5.5.2 并发哈希

哈希表也叫散列表，是一种支持快速查找、插入和删除的数据结构。哈希的关键思想为通过哈希函数，将键值 key 映射到查找表的特定区间中，从而实现快速查找。

当两个键值 k1 和 k2 被哈希函数映射到同一个哈希值 h 时，我们称其产生了哈希冲突，解决哈希冲突一个常用的方法是拉链法。如下图所示，若干个哈希键值被哈希函数 Hash 映射到哈希值，我们采用数组表示哈希值落入的区间，每个数组元素成为桶，每个桶中放入一个单链表，存储具有相同哈希值的元素：



这样，查找元素分成两个步骤：首先，需要先用哈希函数 Hash 得到元素 key 对应的桶号；其次，再在桶的链表中遍历，直到找到相应元素为止。

与栈类似，传统哈希表在多线程并发的场景下也有竞态条件的风险。我们同样可以通过加锁的方式将传统哈希表改造为并发哈希表。但由于哈希表数据结构的复杂性，对其改造不止一种方案。最简单的一种方案是粗粒度哈希，即将整个哈希表视为一个临界资源，线程每次对哈希表进行操作视为进入一次临界区。因此，可以为哈希

表设置一个互斥量，每个线程在对哈希表操作对需要获得该互斥量，避免竞态条件的产生。另一种可能的方案是细粒度哈希，即对哈希表中的每个桶，我们都设置一把锁，从而可以在保证线程安全的前提下，实现更细粒度的并发。

首先，我们先讨论粗粒度并发哈希。为此，我们给出如下的并发哈希表数据结构和函数接口：

```
typedef struct Node_t{
    char *key;
    char *value;
    struct Node_t *next;
} node_t;
typedef struct Hash_t{
    pthread_mutex_t mu;
    node_t *table[NUM_BUCKETS];
} hash_t;
void hash_init(hash_t *h);
void hash_insert(hash_t *h, char *key, char *value);
int hash_lookup(hash_t *h, char *key);
int hash_delete(hash_t *h, char *key);
int hash_items(hash_t *h);
int hash_to_hash(char *key);
```

不失一般性，我们假设哈希表中的关键字和值都是字符串类型。数据结构 hash_t 编码了哈希表（第 6 行），哈希表被互斥量 mu 保护（第 7 行），哈希表中包括长度为 NUM_BUCKETS 的桶 table，每个元素都是一个单链表类型 node_t。

函数 hash_init 对传入的哈希表 h 进行初始化；函数 hash_insert 将给定的键值 key 和值 value 插入哈希表 h；函数 hash_lookup 在哈希表中查找给定的键值 key；函数 hash_delete 将哈希表中给定的键值 key 及其对应的值删除；函数 hash_items 计算哈希表中元素的数量；而函数 hash_to_hash 计算给定的键值 key，所对应的哈希值。

并发哈希表的操作，需要在临界区代码上合理的使用锁。例如，考虑并发哈希表的插入函数 hash_insert 的实现：

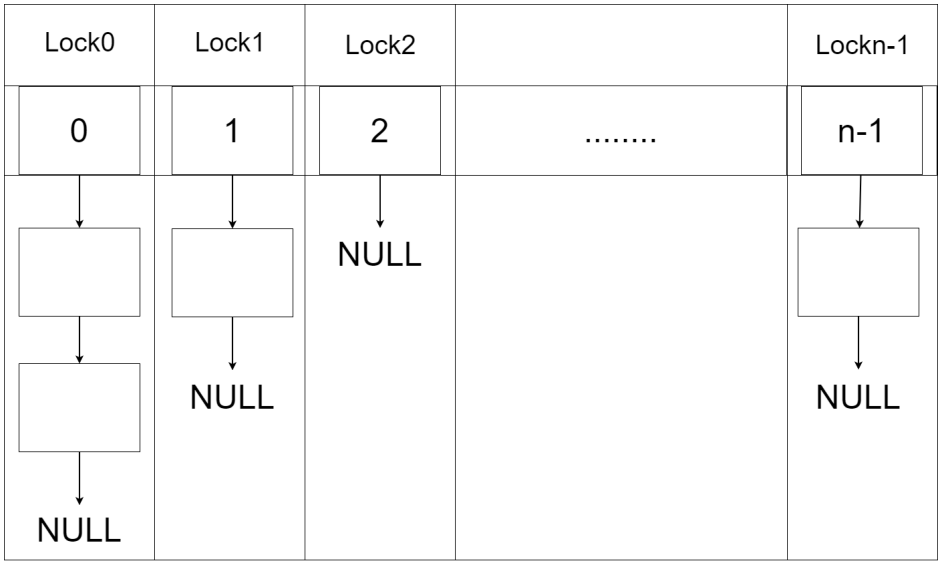
```
void hash_insert(hash_t *h, char *key, char *value){
    int hv = hash_to_hash(key); // the hash value
    int n = hv % NUM_BUCKETS;   // the bucket number
    pthread_mutex_lock(&h->mu);
    node_t *tmp = calloc(1, sizeof(*tmp));
    tmp->key = key;
    tmp->value = value;
    tmp->next = h->table[n];
    h->table[n] = tmp;    // insert into the bucket
    pthread_mutex_unlock(&h->mu);
}
```

首先，函数 hash_insert 为键值 key 计算得到其所在桶的位置 n（第 3 行）；接着，函数分配链表节点 tmp 的内存、并将键值 key、值 value 和 next 指针域设置好（第 5 到 8 行）；最后，函数将新分配的节点 tmp 存入哈希表中（第 9 行）。需要特别注意的是，即便键值原来就在该哈希表中存在，上述函数仍然正确工作，请读者自行分析原因。

并发哈希其它函数的实现与此类似，我们把相关函数的实现，作为练习留给读者完成。

虽然粗粒度并发哈希表通过设置锁保证了线程安全，但也降低了并发执行的能力。例如，假定某时刻有两个线程要对哈希表做插入操作，但桶的位置不同，则这两个线程操作的是不同的链表，因此不会有竞态条件，可以并发执行。但粗粒度哈希表只有一个全局锁，所以每次只可能有一个线程进入插入操作，而另一个线程必须等前一个线程接受并释放锁后，才能继续操作。

为此，我们引入细粒度并发哈希表，将每一个桶都视作一个共享资源，并单独设置一个锁，其示意状态如下：



我们可以定义如下并发哈希表的数据结构（函数接口保持不变）：

```
typedef struct Node_t {
    char *key;
    char *value;
    struct Node_t *next;
} node_t;
typedef struct Hash_t{
    pthread_mutex_t mu[NUM_BUCKETS];
    node_t *table[NUM_BUCKETS];
} hash_t;
```

哈希表中包括同等数量的锁 mu（第 7 行）和桶的元素（第 8 行）。

对该并发哈希表操作时，同样需要获得获取是否目标桶元素对应的锁。仍以哈希表的插入操作为例，其代码如下：

```
void hash_insert(hash_t *h, char *key, int *value) {
    int hv = hash_to_hash(key);
    int n = hv % NUM_BUCKETS;
    pthread_mutex_lock(&h->mu[n]);
    node_t *tmp = calloc(1, sizeof(*tmp));
    tmp->key = key;
    tmp->value = value;
    tmp->next = h->table[n];
    h->table[n] = tmp;    // insert into the bucket
    pthread_mutex_unlock(&h->mu[n]);
}
```

尽管这个插入函数的实现和粗粒度哈希的插入函数类似，但是由于其加锁解锁都是在桶级别进行（第 4 和第 10 行），因此具有更好的并发度。

哈希表其它函数的实现与此类似，我们将其做为练习，留给读者完成。

5.6 无锁数据结构

无锁数据结构（Lock-free data structures）的设计意图，是在不使用显式锁（例如互斥量）的情况下，实现线程安全的并发操作。无锁数据结构由于不需要阻塞或等待其他线程释放锁，从而能进一步提高多线程程序的并发度；但是，无锁数据的设计和实现，也具有更高的技术难度和挑战。在本小节，我们以无锁栈和无锁队列两个数据结构为例，讨论无锁数据结构设计实现的一般思路。

5.6.1 无锁栈

在多线程场景中，并发栈操作的竞争主要出现在栈顶节点处，我们可以用原子操作来解决这个问题。为此，我们首先给出如下无锁栈的数据结构和函数接口：

```
#include <stdatomic.h>
typede struct Node_t{
    int data;
    struct Node_t *next;
} node_t;
typedef struct Lf_stack_t{
    node_t *node;
    atomic_intptr_t top;
}lf_stack_t;

void lf_stack_init(lf_stack_t *stack);
void lf_stack_push(lf_stack_t *stack, int value);
int lf_stack_pop (lf_stack_t *stack);
int lf_stack_size(lf_stack_t *stack);
```

数据结构 lf_stack_t 中的栈顶指针 top，具有原子指针类型 atomic_intptr_t（第 8 行）。

无锁数据结构的操作，往往以对原子变量的操作进行实现。接下来，我们先以无锁并发栈的入栈操作 push 为例，讨论无锁栈的实现：

```
void lf_stack_push(lf_stack_t *stack, int data){
    node_t *new = calloc(1, sizeof(*new));
    new->data = data;
    while(1){
        void *old_top = atomic_load(&stack->top);
        new->next = old_top;
        if(atomic_compare_exchange_strong(&(stack->top),
                                          &old_top, new))
            break;
    }
    return;
}
```

首先，该函数为待插入的数据分配新的栈节点 new（第 2 到 3 行）；然后，该函数取得

栈当前的栈顶指针 old_top（第 5 行），并将新节点 new 的尾指针指向该结点（第 6 行），本质上，这个步骤完成了新节点 new 的部分入栈；接下来，函数使用原子操作 atomic_compare_exchange_strong，对当前栈顶指针栈顶 stack->top 和老的栈顶指针 old_top 进行比较：如果二者相等，则将新的栈顶指针赋值给 stack->top，从而完成了数据的完全入栈；否则，如果这两个值不相等，则说明栈已经被修改过，则函数重新进行下一轮循环，尝试对数据进行入栈。

类似的，对栈的弹出操作 pop，同样需要用到原子操作：

```
int lf_stack_pop(lf_stack_t *stack){
   while(1){
        node_t *old_top = atomic_load(&stack->top);
        if(old_top == 0) // empty stack
            return -1;
        node_t *old_next = old_top->next;
        if(atomic_compare_exchange_strong(&(stack->top),
                                          &old_top, old_next))
            break;
    }
    return old_top->data;
}
```

该出栈函数的实现和入栈函数类似，对其具体分析，我们做为练习留给读者完成。

最后，我们把剩余的无锁栈操作的实现，也作为练习留给读者。

5.6.2 无锁队列

队列是一种先入先出的数据结构，允许在一端插入数据，在另一端读取数据。与无锁栈一样，我们也可以通过原子操作实现无锁队列。首先，给出无锁队列的数据结构和函数接口：

```
#include <stdatomic.h>
typedef struct Node_t {
    int value;
    struct Node_t *next;
} node_t;

typedef struct Lf_queue_t {
    atomic_intptr_t front;
    atomic_intptr_t rear;
} lf_queue_t;

void lf_queue_init(lf_queue_t *queue);
void lf_queue_enq(lf_queue_t *queue, int value);
int lf_queue_deq(lf_queue_t *queue);
int lf_queue_size(lf_queue_t *queue);
```

队列数据结构 Lf_queue_t 中的 front 和 rear 指针，分别都是原子指针类型，并且分别指向队列的队首和队尾位置（第 7 到 10 行）。

无锁队列的函数 lf_queue_enq 完成元素的入队；而函数 lf_queue_deq 完成函数的出队；函数 lf_queue_size 返回队列的元素个数。

初始化队列时，可以先创建一个单独的节点，这样有助于方便处理判断队列空等各种边界条件。为此，我们给出如下的队列初始化函数 lf_queue_init：

```
void lf_queue_init(lf_queue_t *queue){
    node_t *p = calloc(1, sizeof(*p));
    atomic_store(&queue->front, p);
    atomic_store(&queue->rear, p);
    return;
}
```

无锁队列的入队操作 lf_queue_enq，会在队尾增加一个新节点，因此需要用原子操作，保证新节点的插入和 rear 指针的后移：

```
void lf_queue_enq(lf_queue_t *queue, int value) {
    node_t *new = calloc(1, sizeof(*new));
    new->value = value;
    while(1){
        node_t *old_rear = atomic_load(&queue->rear);
        new->next = old_rear;
        if(atomic_compare_exchange_strong(&queue->rear,
                                          &old_rear, new);
            break;
    }
    return;
}
```

操作的过程同样是用原子的方式，插入新节点 new。

无锁队列其它操作的实现方式类似，我们将其做为练习，留给读者自行完成。

5.7 本章小结

本章深入讨论了基于线程的无锁并发编程。首先，我们讨论了线程的基本概念和执行模型。接着，我们讨论了多线程编程中的竞态条件和临界区，并讨论了线程同步与互斥机制。接下来，我们讨论原子变量并给出了其在同步原语实现中的应用。最后，我们讨论了基于锁的和无锁数据结构。

5.8 深入阅读

想要进一步深入了解多线程相关知识的读者，可以继续阅读《深入了解计算机系统》一书的相关章节;《现代多核处理器编程艺术》一书大量讨论了并发数据结构。