```c
#include <pthread.h>
int
pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void*),
        void *arg);

int
pthread_join(
        pthread_t thread,
        void **value_ptr
);
```

1. The second argument, *attr*, is used to specify any attributes this thread might have. Some examples include setting the stack size or perhaps information about the scheduling priority of the thread.
2. The second argument is a pointer to the return value you expect to get back.
3. Never return a pointer which refers to something allocated on the thread's call stack

———————————————————————

Xchg

```c
int TestAndSet(int *old_ptr, int new) {
        int old = *old_ptr;
        *old_ptr = new;
        return old;
}
```

Spin Lock Using Test-and-set

```c
typedef struct __lock_t {
        int flag;
} lock_t;
void init(lock_t *lock) {
// 0: lock is available, 1: lock is held
        lock->flag = 0;
}
void lock(lock_t *lock) {
        while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}
void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

<u>On multiple CPUs, spin locks work reasonably well.</u>

Compare-and-swap

```c
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected) {
          *addr = new;
  }
  return actual;
}
void lock(lock_t *lock) {
        while (CompareAndSwap(&lock->flag, 0, 1) == 1)
                ;
}
```

Load-Linked and Store-Conditional

```c
int LoadLinked(int *ptr) {
   return *ptr;
}
int StoreConditional(int *ptr, int value) {
   if (no one has updated *ptr since the LoadLinked to this address)
{
      *ptr = value;
      return 1; // success
   } else {
      return 0; // failed to update
   }
}
void lock(lock_t *lock) {
   while (1) {
      while (LoadLinked(&lock->flag) == 1) {
         // spin until it's zero
      }
      if (StoreConditional(&lock->flag, 1) == 1) {
         return; // if set-it-to-1 was a success: all done
         // otherwise: try it all over again
      }
   }
}
void unlock(lock_t *lock) {
   lock->flag = 0;
}
```

Load-Linked 在hardware establish a link with lock->flag监视

Fetch-And-Add

```c
int FetchAndAdd(int *ptr) {
        int old = *ptr;
        *ptr = old + 1;
        return old;
}
```

Ticket lock

```c
typedef struct __lock_t {
        int ticket; //thread's ticket number int turn; //whose turn it is
} lock;

void lock_init(lock_t *lock) {
  lock->ticket = 0;
  lock->turn = 0;
}

void lock(lock_it *lock) {
        //first, reserve this thread's turn
        int myturn = FetchAndAdd(&lock->ticket);
        while (lock->turn != myturn)
                ; //spin until thread's turn
}

void unlock(lock_t *lock) {
  lock->turn = lock->turn+1;
}
```

保障了公平性,未来肯定会轮到

A Simple Approach to spin waiting

```c
void init() {
        flag = 0;
}

void lock() {
        while (TestAndSet(&flag, 1) == 1)
                yield(); // give up the CPU
}

void unlock() {
        flag = 0;
}
```

this approach 没有解决 starvation.

Two-Phase Locks

```c
typedef struct __lock_t {
   int flag;
   int guard;
   queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
   m->flag = 0;
   m->guard = 0;
   queue_init(m->q);
}
void lock(lock_t *m) {
   while (TestAndSet(&m->guard, 1) == 1)
      ; // acquire guard lock by spinning
   if (m->flag == 0) {
      m->flag = 1; // lock is acquired
      m->guard = 0;
   } else {
      queue_add(m->q, gettid());
      m->guard = 0;
      park();
   }
}
void unlock(lock_t *m) {
   while (TestAndSet(&m->guard, 1) == 1)
      ; // acquire guard lock by spinning
   if (queue_empty(m->q))
      m->flag = 0; // let go of lock; no one wants it
   else
      unpark(queue_remove(m->q)); // hold lock
                                  // (for next thread!)
   m->guard = 0;
}
```

刚要park的时候context switch会出现wakeup/waiting race.

Spin-Waiting vs. Blocking

Each approach is better under different circumstances:
- Uniprocessor
  - Waiting process is scheduled → Process holding lock is not
  - Waiting process should always relinquish processor

  Associate queue of waiters with each lock (as in previous implementation)
- Multiprocessor
  - Waiting process is scheduled → process holding lock might be
  - spin or block depends on _t_ time before lock released vs context-switch cost _C_:
  - Lock released quickly (_t_ « _C_) → Spin-wait
  - Lock released slowly (_t_ ≥ _C_) → Block

为每个锁关联等待队列意味着在锁上建立一个队列结构，以便在锁被占用时，有序管理等待获取锁的进程

——————————————————————————————

Approximate Counter

```c
typedef struct __counter_t {
   int global;                 // 全局计数器
   pthread_mutex_t glock;      // 全局计数器的锁
   int local[NUMCPUS];         // 每个 CPU 的局部计数器
   pthread_mutex_t llock[NUMCPUS]; // 每个局部计数器的锁
   int threshold;              // 当局部计数器到达阈值时，累加到全局计数器
} counter_t;
void init(counter_t *c, int threshold) {
   c->threshold = threshold;  // 设置阈值
   c->global = 0;             // 初始化全局计数器为 0
   pthread_mutex_init(&c->glock, NULL);  // 初始化全局锁

   int i;
   for(i = 0; i < NUMCPUS; i++) {
      c->local[i] = 0;      // 初始化每个局部计数器为 0
      pthread_mutex_init(&c->llock[i], NULL);  // 初始化每个局部锁
   }
}
void update(counter_t *c, int threadID, int amt) {
   int cpu = threadID % NUMCPUS;
   pthread_mutex_lock(&c->llock[cpu]);
   c->local[cpu] += amt;
   if (c->local[cpu] >= c->threshold) {
      pthread_mutex_lock(&c->glock);
      c->global += c->local[cpu];
      pthread_mutex_unlock(&c->glock);
      c->local[cpu] = 0;
   }
   pthread_mutex_unlock(&c->llock[cpu]);
}
int get(counter_t *c) {
   pthread_mutex_lock(&c->glock);
   int val = c->global;
   pthread_mutex_unlock(&c->glock);
   return val;
}
```

Concurrent linked list

```c
typedef struct __node_t {
   int key;                // 节点保存的值
   struct __node_t *next;  // 指向下一个节点的指针
} node;

typedef struct __list_t {
   node *head;             // 指向链表头节点的指针
   pthread_mutex_t lock;   // 用于保护链表操作的互斥锁
} list;
void List_Init(list *L) {
   L->head = NULL;                      // 初始化链表为空
   pthread_mutex_init(&L->lock, NULL);  // 初始化互斥锁
}
int List_Insert(list *L, int key) {
   pthread_mutex_lock(&L->lock);
   node_t *n = malloc(sizeof(node));
   if (n == NULL) {
      perror("malloc");
      pthread_mutex_unlock(&L->lock);
      return -1; // fail
   }
   n->key = key;
   n->next = L->head;
   L->head = n;
   pthread_mutex_unlock(&L->lock);
   return 0; // success
}
int List_Lookup(list *L, int key) {
   pthread_mutex_lock(&L->lock);
   node *curr = L->head;
   while (curr) {
      if (curr->key == key) {
         pthread_mutex_unlock(&L->lock);
         return 0; // success
      }
      curr = curr->next;
   }
   pthread_mutex_unlock(&L->lock);
   return -1; // failure
```

<u>hand-over-hand locking (a.k.a. lock coupling</u>

Instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next node's lock and then releases the current node's lock

However, in practice, it is hard to make such a structure faster than the simple single lock approach, as the overheads of acquiring and releasing locks for each node of a list traversal is prohibitive.

Concurrent Queue

One standard approach – add a big lock around the entire data structure

Improved approach – add a lock around the head and another around the tail, Allows concurrent enqueue and dequeue operations

Add a dummy node to separate the head from the tail

```c
typedef struct __node_t {
   int value;              // 节点的值
   struct __node_t *next;  // 指向下一个节点的指针
} node;

typedef struct __queue_t {
   node *head;             // 队列的头节点
   node *tail;             // 队列的尾节点
   pthread_mutex_t hLock;  // 用于保护队列头部的锁
   pthread_mutex_t tLock;  // 用于保护队列尾部的锁
} queue;
void Queue_Init(queue *q) {
   node *tmp = malloc(sizeof(node));  // 创建一个哑节点（dummy node）
   tmp->next = NULL;            // 哑节点的 next 指针为空
   q->head = q->tail = tmp;     // 初始化时，头和尾都指向这个哑节点
   pthread_mutex_init(&q->hLock, NULL); // 初始化头部锁
   pthread_mutex_init(&q->tLock, NULL); // 初始化尾部锁
}
void Enqueue(queue *q, int value) {
   node *tmp = malloc(sizeof(node));   // 分配新节点的内存
   assert(tmp != NULL);         // 确保内存分配成功
   tmp->value = value;          // 设置新节点的值
   tmp->next = NULL;            // 新节点的下一个指针为空
   pthread_mutex_lock(&q->tLock);    // 锁定队列的尾部，防止其他线程同时修改
   q->tail->next = tmp;         // 将新节点添加到队列尾部
   q->tail = tmp;               // 更新队列的尾指针
   pthread_mutex_unlock(&q->tLock);  // 解锁，允许其他线程继续访问队列
}
int Dequeue(queue *q, int *value) {
   pthread_mutex_lock(&q->hLock);    // 锁定队列的头部，防止其他线程同时修改
   node *tmp = q->head;         // 临时保存头节点
   node *newHead = tmp->next;   // 获取下一个节点（即新的头节点）
   if (newHead == NULL) {       // 如果队列为空，返回错误
      pthread_mutex_unlock(&q->hLock);
      return -1;                // 队列为空
   }
   *value = newHead->value;     // 获取新的头节点的值
   q->head = newHead;           // 更新头指针，移除旧的头节点
   pthread_mutex_unlock(&q->hLock);  // 解锁，允许其他线程继续访问队列
   free(tmp);                   // 释放旧头节点的内存
   return 0;                    // 成功出队
}
```

Concurrent Hash Table

Utilizes a concurrent list for each hash bucket

Rather than having a single lock, each bucket is locked independently

```c
#define BUCKETS (101)
typedef struct __hash_t {
   list lists[BUCKET];
} hash_t;
void Hash_Init(hash_t *H) {
   int i = 0;
   for(i = 0; i < BUCKETS; i++) {
      List_Init(&H->lists[i]);
   }
}
int Hash_Insert(hash_t *H, int key) {
   int bucket = key % BUCKETS;
   return List_Insert(&H->lists[bucket], key);
}
int Hash_Lookup(hash_t *H, int key) {
   int bucket = key % BUCKETS;
   return List_Lookup(&H->lists[bucket], key);
}
```

————————————————————————————————

Semaphores

<u>used as a lock</u>

```c
int sem_wait(sem_t *s) {
        decrement the value of semaphore s by one
        wait if value of semaphore s is negative
}
int sem_post(sem_t *s) {
        increment the value of semaphore s by one
        if there are one or more threads waiting, wake one
}
```

Binary Semaphore

```c
sem_t m;
sem_init(&m, 0, 1);
sem_wait(&m);
// critical section here
sem_post(&m);
```

Semaphores For Ordering

```c
sem_t s;
void *child(void *arg) {
        printf("child\n");
        sem_post(&s); // signal here: child is done
        return NULL;
}
int main(int argc, char *argv[]) {
        sem_init(&s, 0, 0);
```

```
		printf("parent: begin\n");
		pthread_t c;
		Pthread_create(&c, NULL, child, NULL);
		sem_wait(&s); // wait here for child
		printf("parent: end\n");
		return 0;
}
```

The Producer/Consumer (Bounded Buffer) Problem

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // Line P1
        sem_wait(&mutex);           // Line P1.5 (lock)
        put(i);                     // Line P2
        sem_post(&mutex);           // Line P2.5 (unlock)
        sem_post(&full);            // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);            // Line C1
        sem_wait(&mutex);           // Line C1.5 (lock)
        int tmp = get();            // Line C2
        sem_post(&mutex);           // Line C2.5 (unlock)
        sem_post(&empty);           // Line C3
        printf("%d\n", tmp);
    }
}
```

Reader-Writer Locks

```
typedef struct _rwlock_t {
    sem_t lock;        // binary semaphore (basic lock)
    sem_t writelock;   // allow ONE writer/MANY readers
    int readers;       // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1) // first reader gets writelock
        sem_wait(&rw->writelock);
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0) // last reader lets it go
        sem_post(&rw->writelock);
    sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_post(&rw->writelock);
}
```

Thread Throttling

If all of the threads enter the memory-intensive region at the same time, the sum of all the memory allocation requests will exceed the amount of physical memory on the machine. As a result, the machine will start thrashing (i.e., swapping pages to and from the disk), and the entire computation will slow to a crawl.
A simple semaphore can solve this problem. By initializing the value of the semaphore to the maximum number of threads you wish to enter the memory-intensive region at once, and then putting a sem wait() and sem post() around the region, a semaphore can naturally throttle the number of threads that are ever concurrently in the dangerous region of the code.
We call this *approach throttling*, and consider it a form of *admission control*.

Implementing Zemaphores With Locks And CVs

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

——————————————————————————

Atomicity violation bugs

```
Thread 1::
if (thd->proc_info) {
		fputs(thd->proc_info, ...);
}
Thread 2::
thd->proc_info = NULL;
```

Order violation bugs

```
Thread 1::
void init() {
		mThread = PR_CreateThread(mMain, ...);
}
Thread 2::
void mMain(...) {
		mState = mThread->State;
}
```

If Thread 2 runs immediately once created, the variable mThread has not been initialized. It needs sometime to be initialized.

——————————————————————————

Conditions of Deadlock

1. **Mutual Exclusion** – Threads claim exclusive control of resources that they require (e.g. a thread grabs a lock)
    1. lock-free (and related wait-free) approaches here is simple: using powerful hardware instructions
2. **Hold-and-wait** – Threads hold resources allocated to them while waiting for additional resources
    1. The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once. But it is problematic for a number of reasons.
3. **No preemption** – Resources cannot be forcibly removed from threads
    1. The routine pthread mutex trylock() either grabs the lock (if it is available) and returns success or returns an error code indicating the lock is held. But *livelock*. It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. One solution: one could add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads. But problem that would likely exist again arises due to encapsulation.
4. **Circular wait** – Circular chain of threads hold resources that other threads need
    1. For example, if there are only two locks in the system (L1 and L2), you can prevent deadlock by always acquiring L1 before L2. Such strict ordering ensures that no cyclical wait arises; hence, no deadlock.
    2. *total ordering/partial ordering*

We can use some hardware instructions to avoid using lock:

```
void insert(int value) {
		node_t *n = malloc(sizeof(node_t));
		assert(n != NULL);
		n->value = value;
		do {
				n->next = head;
		} while (CompareAndSwap(&head, n->next, n) == 0);
}
```

——————————————————————————

Event-based concurrency

```
while (1) {
		events = getEvents();
		for (e in events)
				processEvent(e);
}
```

Select

```
int select(
		int nfds,
		fd_set *restrict readfds,
		fd_set *restrict writefds,
		fd_set *restrict errorfds,
		struct timeval *restrict timeout
);
```

- aio_read(struct aiocb *aiocbp);：用于启动异步读操作。该函数会将读取请求提交给操作系统，而不会阻塞当前线程。读操作会从文件描述符中读取数据到指定的缓冲区，直到完成读取任务为止。
- aio_error(const struct aiocb *aiocbp);：用于检查异步操作的状态。该函数不会阻塞，它返回异步操作的错误状态。如果操作仍在进行中，则返回 EINPROGRESS；如果操作已完成，则返回 0 表示成功，或者返回具体错误码表示失败。

如果文件描述符数量较少且跨平台性要求高，可以使用 select；若需监视大量文件描述符，poll 更加合适

difficult with events

1. Specifically, in order to utilize more than one CPU, the event server has to run multiple event handlers in parallel; when doing so, the usual synchronization problems (e.g., critical sections) arise, and the usual solutions (e.g., locks) must be employed.
2. Another problem with the event-based approach is that it does not integrate well with certain kinds of systems activity, such as paging.
3. A third issue is that event-based code can be hard to manage overtime, as the exact semantics of various routines changes. For example, if a routine changes from non-blocking to blocking, the event handler that calls that routine must also change to accommodate its new nature, by ripping itself into two pieces.

——————————————————————————

Rendezvous Problem

- Thread A
    - statement a1
    - statement a2
- Thread B
    - statement b1
    - statement b2

We want to guarantee that a1 happens before b2 and b1 happens before a2.

```C
#include <semaphore.h>
#include <pthread.h>

sem_t semA, semB;

void* threadA(void* arg) {
    // statement a1
    printf("Executing a1\n");

    // 通知 Thread B 的 b2 可以执行
    sem_post(&semB);

    // 等待 Thread B 完成 b1
    sem_wait(&semA);

    // statement a2
    printf("Executing a2\n");

    return NULL;
}

void* threadB(void* arg) {
    // statement b1
    printf("Executing b1\n");

    // 通知 Thread A 的 a2 可以执行
    sem_post(&semA);

    // 等待 Thread A 完成 a1
    sem_wait(&semB);

    // statement b2
    printf("Executing b2\n");

    return NULL;
}

int main() {
    pthread_t tA, tB;

    // 初始化信号量
    sem_init(&semA, 0, 0);
    sem_init(&semB, 0, 0);

    // 创建线程
    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);

    // 等待线程完成
    pthread_join(tA, NULL);
    pthread_join(tB, NULL);

    // 销毁信号量
    sem_destroy(&semA);
    sem_destroy(&semB);

    return 0;
}
```

——————————————————————————

Barrier

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct {
    int count;            // 当前到达屏障的线程数
    int num_threads;      // 需要等待的线程总数
    pthread_mutex_t lock;  // 互斥锁
    pthread_cond_t cond;   // 条件变量
} reusable_barrier_t;

// 初始化屏障
void reusable_barrier_init(reusable_barrier_t *barrier, int num_threads) {
    barrier->count = 0;
    barrier->num_threads = num_threads;
    pthread_mutex_init(&barrier->lock, NULL);
    pthread_cond_init(&barrier->cond, NULL);
}

// 屏障等待函数
void reusable_barrier_wait(reusable_barrier_t *barrier) {
    pthread_mutex_lock(&barrier->lock);

    barrier->count++;

    // 检查是否所有线程都已到达屏障
    if (barrier->count == barrier->num_threads) {
        barrier->count = 0;  // 重置计数器以便重新使用屏障
        pthread_cond_broadcast(&barrier->cond);  // 唤醒所有等待的线程
    } else {
        // 等待其他线程到达屏障
        pthread_cond_wait(&barrier->cond, &barrier->lock);
    }

    pthread_mutex_unlock(&barrier->lock);
}

// 销毁屏障
void reusable_barrier_destroy(reusable_barrier_t *barrier) {
    pthread_mutex_destroy(&barrier->lock);
    pthread_cond_destroy(&barrier->cond);
}

// 测试函数
void* thread_func(void* arg) {
    reusable_barrier_t *barrier = (reusable_barrier_t*)arg;

    printf("Thread %ld before barrier\n", pthread_self());
    reusable_barrier_wait(barrier);  // 等待所有线程到达屏障
    printf("Thread %ld after barrier\n", pthread_self());

    return NULL;
}

int main() {
    int num_threads = 5;
    pthread_t threads[num_threads];
    reusable_barrier_t barrier;

    reusable_barrier_init(&barrier, num_threads);

    // 创建多个线程
    for (int i = 0; i < num_threads; i++) {
        pthread_create(&threads[i], NULL, thread_func, &barrier);
    }

    // 等待所有线程完成
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    reusable_barrier_destroy(&barrier);
    return 0;
}
```