UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Shared memory: C threads

Introduction

- C has no native support for parallel programming
- The approach that has to be followed to write multithreaded programs is to use C APIs, such as POSIX C standard for multithreads (Pthreads) or Windows API
- Pthreads are supported on Unix-like systems (OSX, Linux). A Windows version is also available.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Pthreads

Introduction

- Data types, interface definitions, and macros of Pthreads are usually available via the header file `<pthread.h>`
- Pthreads functions are named in the form
  - `pthread[_<object>]_<operation>()`
- where `<operation>` describes the operation to be performed and the optional `<object>` describes the object to which this operation is applied.
  - For example, `pthread_mutex_init()` is a function for the initialization of a mutex variable; thus, the `<object>` is `mutex` and the `<operation>` is `init`.

Data formats

Pthread data types	Meaning
`pthread_t`	Thread ID
`pthread_attr_t`	Thread attributes object
`pthread_mutex_t`	Mutex variable
`pthread_mutexattr_t`	Mutex attributes object
`pthread_cond_t`	Condition variable
`pthread_condatrr_t`	Condition variable attributes object
`pthread_key_t`	Access key
`pthread_once_t`	*One-time initialization* control context

Creation and termination

- `#include <pthread.h>`

- `int pthread_create (pthread_t* thread_handle, const pthread_attr_t* attribute, void* (*thread_function)(void*), void* arg);`

- `int pthread_join ( pthread_t thread, void** status);`

- The function `pthread_create` invokes function `thread_function` as a thread.

- The function `pthread_join` waits for the thread to be finished and the value passed to `pthread_exit` (by the terminating thread) is returned in the location pointer `**ptr`.

Creation and termination

- `#include <pthread.h>`
- `int pthread_create (pthread_t* thread_handle, const pthread_attr_t* attribute, void* (*thread_function)(void*), void* arg);`

- `pthread_t* thread_handle`: handle/ ID (TID) of the successfully created thread. Use it to refer to the thread.
- `const pthread_attr_t* attribute`: attributes of the thread. NULL means standard attributes.
- `void* (*thread_function)(void*)`: function executed by the thread once it is created
- `void* arg`: argument passed to the thread_function. Use a structure to pass multiple arguments.
- Returns `0` if successful `<errno.h>` codes otherwise.

Creation and termination

- `#include <pthread.h>`

- `int pthread_create (pthread_t* thread_handle, const pthread_attr_t* attribute, void* (*thread_function)(void*), void* arg);`

- `int pthread_join ( pthread_t thread, void** status);`

- The function `pthread_create` invokes function `thread_function` as a thread.

- The function `pthread_join` waits for the thread to be finished and the value passed to `pthread_exit` (by the terminating thread) is returned in the location pointer `**ptr`.

- `pthread_t thread`: handle/ID of the thread to wait for
- `void** status`: completion status of exiting thread, copied into `*status` unless `status == NULL` (no copy)
- Returns `0` if successful `<errno.h>` codes otherwise.
- Note: once a thread is joined the thread handle/ID is no longer valid.

- `int pthread_join ( pthread_t thread, void** status);`
- The function `pthread_create` invokes function `thread_function` as a thread.
- The function `pthread_join` waits for the thread to be finished and the value passed to `pthread_exit` (by the terminating thread) is returned in the location pointer `**ptr`.

Creation and termination

- `#include <pthread.h>`

- `int pthread_create (pthread_t* thread_handle, const pthread_attr_t* attribute, void* (*thread_function)(void*), void* arg);`

- `int pthread_join ( pthread_t thread, void** status);`

- The function `pthread_create` invokes function `thread_function` as a thread.

- The function `pthread_join` waits for the thread to be finished and the value passed to `pthread_exit` (by the terminating thread) is returned in the location pointer `**ptr`.

Destroying threads

- There are some methods to destroy a thread:
  - simply return from the thread function
  - use `pthread_exit()` to return a status to `pthread_join`
  - do not return a pointer to a local variable of the thread function: these local variables are stored on the runtime stack and may not exist any longer after the termination of the thread.
  - return a global variable or a variable that has been dynamically allocated.

Destroying threads

- There are some methods to destroy a thread:
  - simply return from the thread function
  - use `pthread_exit()` to return a status to `pthread_join`
    - `void pthread exit (void* status)`

      If a thread exits with a return then the function is implicitly called and the return value is used as status
  - return a global variable or a variable that has been dynamically allocated.

Thread ID (TID)

- It is unique and it should be treated as an opaque type, without trying to access its members.

- A thread may determine its ID by calling:
  ```
  pthread_t pthread_self();
  ```

- To compare two TID use:
  ```
  int pthread_equal(pthread_t t1,
  pthread_t t2);
  ```
  that returns 0 if two threads are different, nonzero otherwise.

Thread status and join

- The runtime system of the Pthreads library allocates for each thread an internal data structure to store information and data needed to control the execution of the thread.

- This internal data structure is preserved by the runtime system also after the termination of the thread to ensure that another thread can later successfully access the return value of the terminated thread using `pthread_join()`.

Detach

- After the call to `pthread_join()`, the internal data structure of the terminated thread is released and can no longer be accessed.
  If there is no `pthread_join()` for a specific thread, its internal data structure is not released after its termination and occupies memory space until the complete process is terminated.

- The preservation of the internal data structure of a thread after its termination can be avoided by calling the function

  - `int pthread_detach (pthread_t thread)`

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Thread synchronization

Mutex

- A mutex variable denotes a data structure of the predefined opaque type `pthread_mutex_t`. Such a mutex variable can be used to ensure mutual exclusion when accessing common data
  - When a thread A tries to lock a mutex variable that is already owned by another thread B, thread A is blocked until thread B unlocks the mutex variable. The Pthreads runtime system ensures that only one thread at a time is the owner of a specific mutex variable.
- It's up to the programmer to appropriately protect with a mutex a data structure, e.g. by creating a new structure that comprised the mutex and the required data structure

Mutex: static and dynamic (de)allocation

- Static mutex can be allocated using a macro:
- `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER`
- Dynamically allocated mutexes are initialized with:
- `int pthread_mutex_init (pthread_mutex_t* mutex, const pthread_mutexattr_t* attr)`
- `attr=NULL` uses default attributes
- Destroy dynamically allocated mutexes with:
- `int pthread_mutex_destroy (pthread_mutex_t* mutex)`
- Remind to destroy only if no thread is waiting for the mutex variable and if there is currently no owner. A mutex that has been destroyed can later be re-used after a new initialization.

Mutex lock

- `int pthread_mutex_lock (pthread_mutex_t* mutex)`
- A thread shouldn't lock if it is already the owner. Depending on Pthreads implementation this results in EDEADLK error.
  - threads waiting for the mutex wait in a queue and are chosen by the scheduler
- `int pthread_mutex_unlock (pthread_mutex_t* mutex)`
- `int pthread_mutex_trylock (pthread_mutex_t* mutex)`
- attempts to lock without blocking. Returns EBUSY if mutex is already owned.

Read/Write lock

- It' s a sort of mutex but with two lock functions. Multiple threads can obtain read-lock, while only one can get the write-lock. Read and write-lock block each other.

```
int pthread_rwlock_rdlock(pthread_rwlock_t*
                          rwlock_p);
int pthread_rwlock_wrlock(pthread_rwlock_t*
                          rwlock p);
int pthread_rwlock_unlock(pthread_rwlock_t*
                          rwlock_p);
```

Read/Write lock

Usual creation and destruction of read/write locks:

```
int pthread_rwlock_init(
pthread_rwlock_t* rwlock_p, const pthread_rwlockattr_t* attr p);

int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p);
```

```
int pthread_rwlock_rdlock(pthread_rwlock_t*
                          rwlock_p);
int pthread_rwlock_wrlock(pthread_rwlock_t*
                          rwlock p);
int pthread_rwlock_unlock(pthread_rwlock_t*
                          rwlock_p);
```

Deadlock: backoff

- A possible solution to deadlock, without using the ordered mutex acquisition strategy, is to use backoff: a thread locks the first mutex then uses try lock for the subsequent. If try lock returns EBUSY all the locks are unlocked and the whole process is restarted.

- This approach may be slower than the ordered acquisition.

Semaphore

- `#include <semaphore.h>`

```
int sem_init( sem_t* semaphore_p,
              int shared,
              unsigned initial_val );
```

`0` to share the semaphore between the threads

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

- ```
  int sem_post(sem_t* semaphore p );
  int sem_wait(sem_t* semaphore_p );
  ```

- A named semaphore is identified by a name of the form `/somename` (251 chars).
  ```

Semaphore

- `#include <semaphore.h>`

```
sem_t *sem_open(const char *name, int oflag);
```

If `O_CREAT` is specified in oflag, then the semaphore is created if it does not already exist.

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

- ```
  int sem_post(sem_t* semaphore p );
  int sem_wait(sem_t* semaphore_p );
  ```

- A named semaphore is identified by a name of the form `/somename` (251 chars).
  ```

Semaphore

- `#include <semaphore.h>`

```
sem_t *sem_open(const char *name, int oflag);
```

```
int sem_close(sem_t *sem);
```

is used to indicate that the calling process is finished using the named semaphore indicated by sem.

```
int sem_unlink(const char *name);
```

removes the semaphore named by the string name.

- `int sem_post(sem_t* semaphore p );`
  `int sem_wait(sem_t* semaphore_p );`

- A named semaphore is identified by a name of the form `/somename` (251 chars).

Condition variables

- A **condition variable** is an opaque data structure which enables a thread to wait for the occurrence of an arbitrary condition without active waiting.
  A signaling mechanism is provided which blocks the executing thread during the waiting time. The waiting thread is woken up again as soon as the condition is fulfilled.

- To use this mechanism, the executing thread must define a condition variable and a mutex variable. The mutex variable is used to protect the evaluation of the specific condition which is waiting to be fulfilled.
  This is necessary, since the evaluation of a condition usually requires to access shared data which may be modified by other threads concurrently.

Mutex vs. condition variable

- Mutexes allow you to avoid data races, unfortunately while they allow you to protect an operation, they don't permit you to wait until another thread completes an arbitrary activity.

- Condition Variables solve this problem.

Condition variables: static and dynamic (de)allocation

- Static condition variables can be allocated using a macro:
- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER`
- Dynamically allocated condition variables are initialized with:
- `int pthread_cond_init (pthread_cond_t* cond, const pthread_condattr_t* attr)`
- `attr`=NULL uses default attributes
- Destroy dynamically allocated mutexes with:
- `int pthread_cond_destroy (pthread_cond_t* cond)`

Condition variable and mutex

- Each condition variable must be uniquely associated with a specific mutex variable.
  All threads which wait for a condition variable at the same time must use the same associated mutex variable. It is not allowed that different threads associate different mutex variables with a condition variable at the same time. But a mutex variable can be associated with different condition variables.

- A condition variable should only be used for a single condition to avoid deadlocks or race conditions.

Condition variable: wait and signal

- wait for a specific condition to be fulfilled using the function:
- `int pthread_cond_wait (pthread_cond_t* cond, pthread_mutex_t* mutex)`
- if waiting on a condition the mutex is released and the execution of the thread is blocked, waiting for the signal of another thread that alerts about changes in the condition. It is useful to evaluate the condition again after the wake up because there are other threads working concurrently.
- Pthreads provide two functions to wake up (signal) a thread waiting on a condition variable:
- `int pthread_cond_signal(pthread_cond_t* cond)`
- `int pthread_cond_broadcast(pthread_cond_t* cond)`
- A thread should evaluate the condition before signaling. Checking the variable should be done within a mutex lock, signaling should not be protected.

Condition variable: wait and signal

- wait for a specific condition to be fulfilled using the function:
- `int pthread_cond_wait (pthread_cond_t* cond, pthread_mutex_t* mutex)`
- if waiti … ad is blocke … the condit … use there a …
- Pthrea … condit …
- `int p …`
- `int pthread_cond_broadcast(pthread_cond_t* cond)`

Typical usage:

```
pthred_mutex_lock( &mutex );
while ( !isCondition() )
  pthread_cond_wait( &cond, &mutex);
do_something();
pthread_mutex_unlock( &mutex );
```

- A thread should evaluate the condition before signaling. Checking the variable should be done within a mutex lock, signaling should not be protected.

Barrier using condition variable

- A barrier is a synchronization method that makes sure that all threads are at the same point. It's not fully standard in Posix, but can be implemented as:

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex; pthread_cond_t cond_var; ...
void* Thread work(. . .) {
...
/* Barrier */
pthread mutex lock(&mutex);
counter++;
if (counter == thread count) {
    counter = 0;
    pthread_cond_broadcast(&cond_var);
} else {
    while (pthread_cond_wait(&cond_var, &mutex) != 0);
}
pthread mutex unlock(&mutex);
...
}
```

Barrier using condition variable

- A barrier is a synchronization method that makes sure that all threads are at the same point. It's not fully standard in Posix, but can be implemented as:

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex; pthread_cond_t cond_var; ...
void* Thread work(. . .) {
...
/* Barrier */
pthread mutex lock(&mutex);
counter++;
if (counter == thread count) {
    counter = 0;
    pthread_cond_broadcast(&cond_var);
} else {
    while (pthread_cond_wait(&cond_var, &mutex) != 0);
}
pthread mutex unlock(&mutex);
...
}
```

Typical: Pthreads may be awaken also without a broadcast, this ensures that only a signal on the condition awakes the thread

t

- Timed wait is possible using:

- `int pthread_cond_timedwait(pthread_cond_t* cond, pthread_mutex_t* mutex, const struct timespec* time)`

- with:

```
struct timespec {
   time t tv sec; // seconds
   long tv nsec; // nanoseconds
}
```

- timeout is returned as `ETIMEDOUT`

One-Time Initialization

- A one-time initialization of a variable/operation can be achieved using a boolean variable initialized to 0 and protected by a mutex variable.
- Pthreads provide another solution for one-time operations by using a control variable of the predefined type `pthread_once_t`.
- The variable must be initialized with static initialization:
  - `pthread_once_t once_control = PTHREAD_ONCE_INIT`
  - the code that performs the operation is associated with the control and executed using:
  - `pthread_once(pthread_once_t* once_control, void (*once_routine)(void))`
  - the function `once_routine()` can be executed by different threads with the same `once_control` but it's executed only once by the first thread that calls `pthread_once`.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Threads setup and cleanup

Thread attributes

- Characteristics of a thread are specified with the `pthread_attr_t*` attribute in `pthread_create()`. To use non-default attributes we need to initialize a variable of this type, then modify its values:

- `int pthread_attr_init (pthread_attr_t* attr)`

- Each characteristic is modified with an appropriate function call

Return value

- By default, the runtime system assumes that the return value of a thread may be used by another thread after its termination.
  - This is the motivation to keep the thread data structure after its termination , until a `pthread_join()` is issued.
- If we are sure we are not going to return a value then set the thread as detached:
  - `int pthread_attr_getdetachstate (const pthread_attr_t* attr, int* detachstate)`
  - `int pthread_attr_setdetachstate (pthread_attr_t* attr, int detachstate)`
    - `detachstate=PTHREAD_CREATE_DETACHED`
    - `detachstate=PTHREAD_CREATE_JOINABLE`
  - When a thread is created detached (PTHREAD_CREATE_DETACHED), its thread ID and other resources can be reused as soon as the thread terminates.

Cancellation

- In some situations, it is useful to stop the execution of a thread from outside. In Pthreads, a thread can send a cancellation request to another thread by calling the function

- `int pthread_cancel (pthread_t thread)`

- where thread is the thread ID of the thread to be terminated. A call of this function does not necessarily lead to an immediate termination of the specified target thread: it depends on the cancellation type of this thread.

- In any case, control immediately returns to the thread issuing the cancellation request, that does not wait for the cancelled thread to be terminated.

- `int pthread_setcanceltype (int type, int *oldtype)`

- `type=PTHREAD_CANCEL_ASYNCHRONOUS` - can stop everywhere… BAD

- `type = PTHREAD_CANCEL_DEFERRED` - can stop at specified cancellation points

Cancellation

- In some situations, it is useful to stop the execution of a thread from outside. In Pthreads, a thread can send a cancellation request to another thread by calling the function

- `int pthread_cancel (pthread_t thread)`

- where thread is the thread ID of the thread to be terminated. A call of this function does not necessarily lead to an immediate termination of the specified target thread: it depends on the cancellation type of this thread.

- In any case, control immediately returns to the thread issuing the cancellation request, that does not wait for the cancelled thread to be terminated.

cancellation points typically include all functions at which the executing thread may be blocked for a substantial amount of time. Examples are `pthread_cond_wait()`, `pthread_cond_ timedwait()`, `open()`, `read()`, `wait()`, or `pthread_join()`. The programmer can insert additional cancellation points into the program by calling the function:

`void pthread_testcancel()`

Cancellation

- A thread can set its cancellation type by calling the function
- `int pthread_setcancelstate (int state, int *oldstate)`
- A call with `state = PTHREAD_CANCEL_DISABLE` / `PTHREAD_CANCEL_ENABLE` disables/enables the cancellability of the calling thread. The previous cancellation type is stored in `*oldstate`.
- If the cancellability of a thread is disabled, it does not check for cancellation requests when reaching a cancellation point or when calling `pthread_testcancel()`, and the thread cannot be cancelled from outside.

Cleanup functions

- A thread may need to restore some state when it is cancelled. For example, a thread may have to release a mutex variable when it is the owner before being cancelled.

- It is possible to associate function that perform housekeeping, by setting them on a LIFO stack of cleanup functions, with

- `void pthread_cleanup_push (void (*routine) (void *), void *arg)`

- To eliminate functions from the stack call:

- `void pthread_cleanup_pop (int execute)`

- where execute=0 means to just remove the last handler and !=0 means to execute the handler before removing it.

Cleanup and cancellation: example

- A counting semaphore is an example of code that requires cleanup to handle possible cancellations:

```
typedef struct Sema {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int count;
} semaphore_t;

void cleanupHandler(void* arg) {
pthread_mutex_unlock( (pthread_mutex_t*)
arg);
}

void acquireSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps->mutex));
  pthread_cleanup_push( cleanupHandler &(ps-
>mutex));
  while( ps->count == 0)
    pthread_cond_wait( &(ps->cond), &(ps-
>mutex));
  --ps->count;
  pthread_cleanup_pop(1);
}
```

```
void relaeaseSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps_>mutex));
  ptrhead_cleanup_push(cleanupHandler, &(ps-
>mutex));
  ++ps->count;
  pthread_cond_signal( &(ps->cond));
  pthread_cleanup_pop(1);
}
```

Cleanup and cancellation: example

- A counting semaphore is an example of code that requires cleanup to handle possible cancellations:

```
typedef struct Sema {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int count;
} semaphore_t;

void cleanupHandler(void* arg) {
pthread_mutex_unlock( (pthread_mutex_t*)
arg);
}

void acquireSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps->mutex));
  pthread_cleanup_push( cleanupHandler &(ps-
>mutex));
  while( ps->count == 0)
    pthread_cond_wait( &(ps->cond), &(ps-
>mutex));
  --ps->count;
  pthread_cleanup_pop(1);
}
```

```
void relaeaseSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps_>mutex));
  ptrhead_cleanup_push(cleanupHandler, &(ps-
>mutex));
  ++ps->count;
  pthread_cond_signal( &(ps->cond));
  pthread_cleanup_pop(1);
}
```

This is a cancellation point. We risk to hold the mutex…

Cleanup and cancellation: example

- A counting semaphore is an example of code that requires cleanup to handle possible cancellations:

```
typedef struct Sema {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int count;
} semaphore_t;

void cleanupHandler(void* arg) {
pthread_mutex_unlock( (pthread_mutex_t*)
arg);
}

void acquireSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps->mutex));
  pthread_cleanup_push( cleanupHandler &(ps-
>mutex));
  while( ps->count == 0)
    pthread_cond_wait( &(ps->cond), &(ps-
>mutex));
  --ps->count;
  pthread_cleanup_pop(1);
}
```

```
void relaeaseSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps_>mutex));
  ptrhead_cleanup_push(cleanupHandler, &(ps-
>mutex));
  ++ps->count;
  pthread_cond_signal( &(ps->cond));
  pthread_cleanup_pop(1);
}
```

Cleanup and cancellation: example

- A counting semaphore is an example of code that requires cleanup to handle possible cancellations:

```
typedef struct Sema {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int count;
} semaphore_t;

void cleanupHandler(void* arg) {
pthread_mutex_unlock( (pthread_mutex_t*)
arg);
}

void acquireSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps->mutex));
  pthread_cleanup_push( cleanupHandler &(ps-
>mutex));
  while( ps->count == 0)
    pthread_cond_wait( &(ps->cond), &(ps-
>mutex));
  --ps->count;
  pthread_cleanup_pop(1);
}
```

```
void relaeaseSemaphore(sema_t* ps) {
  pthread_mutex_lock(&(ps_>mutex));
  ptrhead_cleanup_push(cleanupHandler, &(ps-
>mutex));
  ++ps->count;
  pthread_cond_signal( &(ps->cond));
  pthread_cleanup_pop(1);
}
```

This executes the registered cleanup that unlocks the mutex

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Thread shared data

Thread specific data

- Standard rules of process/thread variable access are used, due to sharing a common address space:
  - global and dynamically allocated variables can be accessed by a thread
  - local thread variables are accessible only by a thread and have same lifetime of thread
- To avoid passing too many parameters between the functions of a thread it is better to use thread-local storage (TSL)

key/value creation

- Thread specific data is implemented using key/value pairs.
- Each key/value can be accessed by all the threads
- If a key is duplicated each thread that created it sees only his original version
- `int pthread_key_create (pthread_key_t* key, void (*destructor)(void *))`
- `int pthread_key_delete (pthread_key_t key)`
- call a `pthread_key_create` once for each `pthread_key_t`. Ensure this with `pthread_once()`.

key/value access

- Associate/overwrite a value to a key using:

- `int pthread_setspecific (pthread_key_t key, void* value)`
  - typically value is the address of a dynamically allocated variable… avoid to use local variables… this is C!

- Get the value associated to a key with:

- `void* pthread_getspecific (pthread_key_t key)`

TLS

- Thread local storage can be implemented since C99 standard adding the storage class keyword `_thread`
- Each thread will get a separate instance of the variable
- Can be applied to global and static variables
  - no non-static or block-scoped variables

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Lock-based Concurrent Data Structures

Concurrent linked list

```
// basic node structure
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;
```

```
// basic list structure (one used per list)
typedef struct __list_t {
    node_t* head;
    pthread_mutex_t lock;
} list_t;
```

```
void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}
```

```
int List_Insert(list_t *L, int key) {
     pthread_mutex_lock(&L->lock);
     node_t *new = malloc(sizeof(node_t));
     if (new == NULL) {
          pthread_mutex_unlock(&L->lock);
          return -1; // fail
     }
     new->key  = key;
     new->next = L->head;
     L->head   = new;
     pthread_mutex_unlock(&L->lock);
     return 0; // success
}

int List_Lookup(list_t *L, int key) {
     pthread_mutex_lock(&L->lock);
     node_t *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
}
```

Concurrent queue

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;


typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;


void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}


Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent queue

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;
```

```
typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;
```

```
void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}
```

Concurrent enqueue and dequeue

```
Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent queue

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;


typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;


void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}


Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent queue

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;


typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;


void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}
```

```
Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
```

dummy node (allocated in the queue initialization code);
this dummy enables the separation of head and tail operations.

Concurrent queue

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;


typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;


void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;
    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}


Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent hash table

```
#define BUCKETS (101)


typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;
```

```
void Hash_Init(hash_t *H) {
    int i;
    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);
}
```

Books

- Principles of Parallel Programming, Calvin Lyn and Lawrence Snyder, Pearson - Chapt. 6

- An Introduction to Parallel Programming, Peter Pacheco, Morgan Kaufmann - Chapt. 4

- Parallel Programming for Multicore and Cluster Systems, Thomas Dauber and Gudula Rünger, Springer - Chapt. 6