Threads and thread-safe ADTs
Threads and Processes

- Processes have the following components:
  - An address space
  - A collection of operating system state
  - A CPU context – a *thread* of control

- On a multiprocessor system, with several CPUs, it would make sense for a process to have several CPU contexts (threads of control)

- Even if we only have a single CPU, multiple threads of control could run in the same address space
  - The concepts of “thread of control” and “address space” are orthogonal concepts
Threads

- A thread shares a process address space with zero or more other threads
- Each thread has its own
  - Register state (PC, SP, general purpose registers, …)
  - Stack
- A traditional process can be viewed as an address space with a single thread
Single thread within a process

Process Address Space

- stack
  - routine1/var1
  - routine1/var2

- text
  - main()
  - routine1()
  - routine2()

- data
  - arrayA
  - arrayB

- heap

Stack Pointer
Program Counter
Registers

- Process ID
- Group ID
- User ID

- Files
- Locks
- Sockets
Multiple threads in a process

Process Address Space

<table>
<thead>
<tr>
<th>Stack</th>
<th>Text</th>
<th>Data</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>routine1/var1</td>
<td>routine1()</td>
<td>arrayA</td>
<td>routine2/var2</td>
</tr>
<tr>
<td>routine1/var2</td>
<td>routine2()</td>
<td>arrayB</td>
<td></td>
</tr>
</tbody>
</table>

Thread 1
- Stack Pointer
- Program Counter
- Registers

Thread 2
- Stack Pointer
- Program Counter
- Registers

- Process ID
- Group ID
- User ID

- Files
- Locks
- Sockets
What is a thread?

- A thread executes a stream of instructions
  - An abstraction of a control-flow (thread of control)
- Practically, it is a processor context and stack
  - Assigned to a CPU by a scheduler
  - Executes in the context of a process’s memory address space
Private per-thread state

- Stack (local variables, procedure call frames)
- Program counter
- Stack pointer
- General purpose registers
- Scheduling properties (e.g. priority)
- Set of pending and blocked signals
- Other thread-specific data
State that is shared among threads in the same process

- Open files, sockets, locks
- User ID, group ID, process ID
- Address space
  - Text (compiled instructions)
  - Data (off-stack variables [global, static])
  - Heap (memory used for malloc()/free())
- Changes made to shared state by one thread will be visible to the other threads in the process
Independent execution of threads

Each thread has its own stack
How and why program using threads?

- Split program into routines to execute in parallel
- Potentially use multiple CPU’s concurrently
- Low cost communication between threads via shared memory (versus pipes between processes)
- Able to overlap computation and blocking on a single CPU
  - Blocking due to I/O
  - Computation and communication
- Handle asynchronous events
Thread example

A word processor with three threads
Threads in a web server

A multithreaded web server
Dispatched thread

while(1) {
    get_next_request(&buf);
    handoff_work(&buf);
}

Worker thread

while(1) {
    wait_for_work(&buf);
    if (! Obtain_page_from_cache(&buf, &page)) {
        read_page_from_disk(&buf, &page);
        add_page_to_cache(&buf, &page);
    }
    return_page(&buf, &page);
}
Common thread programming models

- **Manager/worker**
  - Manager thread handles I/O and assigns work to worker threads
  - Worker threads may be created dynamically, or allocated from a thread pool
- **Peer**
  - similar to the manager/worker model, but after the main thread creates other threads, it participates in the work
- **Pipeline**
  - Each thread handles a different stage of an assembly line
  - Threads hand work to each other in a producer-consumer relationship
Suitable reasons for threading

- The code in question:
  - Blocks for potentially long periods of time
  - Uses many CPU cycles
  - Must respond to asynchronous events
  - Is of lesser or greater importance than other tasks
  - Can be performed in parallel with other tasks
Thread architectures

- Threads can be implemented in the OS or at user level
- User level thread implementations
  - Thread scheduler runs as user code inside the process
  - Manages thread contexts in user space
  - OS only sees a traditional, single-threaded process
Kernel-level threads

The thread-switching code is in the kernel
User-level threads

The thread-switching code is in user space
Hybrid thread implementations

Multiplexing user-level threads onto kernel-level threads
Assuming that your application is appropriate for multithreading, one must have access to programming facilities that address the following types of behaviour:

- thread lifecycle
- critical regions
- conditional critical regions
Thread lifecycle

- One must be able to:
  - Create a new thread
    - Create a thread context
    - Associate the thread with a particular function/method
    - Pass arguments to the function/method
    - Receive an identifier to be used in other lifecycle calls
  - Start a new thread executing
  - Interrupt another thread
  - Wait for another thread to terminate
Critical Regions

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating threads
- Consider the following race condition that can occur if two threads cannot avail themselves of such mechanisms:
int count = 0;
int in = 0;
int out = 0;
Type buffer[N];

for (;;) {
    // produce an item and put in nextProduced
    while (count == N)
        ; // do nothing
    buffer [in] = nextProduced;
in = (in + 1) % N;
count++;
}

for (;;) {
    // consume the item in nextConsumed
    while (count == 0)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % N;
count--;
Producer-consumer race condition(2)

- `count++` could be implemented as
  
  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- `count--` could be implemented as
  
  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```

- Consider this execution interleaving with “`count = 5`” initially:
  
  S0: producer execute `register1 = count`  {`register1 = 5`}
  S1: producer execute `register1 = register1 + 1`  {`register1 = 6`}
  S2: consumer execute `register2 = count`  {`register2 = 5`}
  S3: consumer execute `register2 = register2 - 1`  {`register2 = 4`}
  S4: producer execute `count = register1`  {`count = 6`}
  S5: consumer execute `count = register2`  {`count = 4`}

Locks – a mechanism for serializing access

- Associate a lock with each collection of shared data for which consistency must be maintained
- Before accessing the data protected by a particular lock, a thread must **acquire** the lock
- After completing the data access, the owning thread must **release** the lock
- The semantics of **acquire** are:
  - if another thread owns the lock, the thread requesting the lock will be blocked until the owning thread releases the lock
  - if a thread requests a lock and it is not owned, that thread will be granted ownership of the lock
  - upon release of lock ownership by a thread, if any other threads are blocked waiting for the lock, one of them will be granted ownership of the lock and allowed to continue execution
Producer-consumer with lock

Lock theLock;
int count = 0;
int in = 0;
int out = 0;
Type buffer[N];

for (;;) {
    // produce an item and put in nextProduced
    acquire(theLock);
    while (count == N)
        do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % N;
    count++;
    release(theLock);
}

for (;;) {
    acquire(theLock);
    while (count == 0)
        do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % N;
    count--;
    release(theLock);
    // consume the item in nextConsumed
}
Do locks completely solve the problem?

- Consider the situation where count == 0 and the Consumer executes the statements:
  
  ```
  while (count == 0)
  ;
  ```

- While executing this loop, the Consumer owns theLock, implying that the Producer cannot acquire theLock in order to place a new item in the buffer, thus incrementing count.

- As a result, we have deadlock; neither thread can make progress.

- There needs to be a way to release ownership of the lock until one knows that the condition is not true.
Conditional Critical Regions

- In order to completely address race conditions as in the Producer-Consumer example, we need an additional synchronization construct called a Condition Variable.
- Condition variables enable threads to wait to be notified by another thread that something has changed.
- Each condition variable is associated with a Lock.
- Condition variables support three calls by threads:
  - wait()
  - signal()
  - broadcast()
Condition variables

- Semantics of wait()
  - Can only be called if the thread owns the associated Lock
  - Causes the thread to release the lock and to be blocked until another thread signals the condition variable
  - When the blocked thread starts executing again (due to another thread signalling the condition variable), it has resumed ownership of the lock

- Semantics of signal()
  - Can only be called if the thread owns the associated Lock
  - If any threads are blocked awaiting the condition, one of them will be unblocked

- Semantics of broadcast()
  - Can only be called if the thread owns the associated Lock
  - If any threads are blocked awaiting the condition, all of them are unblocked
Producer-consumer - lock and condition variable

Lock theLock;
Condition theCondition;
int count = 0;
int in = 0;
int out = 0;
Type buffer[N];

for (;;) {
    // produce an item and put in nextProduced
    acquire(theLock);
    while (count == N)
        wait(theCondition, theLock);
    buffer [in] = nextProduced;
in = (in + 1) % N;
count++;
broadcast(theCondition);
release(theLock);
}

for (;;) {
    acquire(theLock);
    while (count == 0)
        wait(theCondition, theLock);
    nextConsumed = buffer[out];
    out = (out + 1) % N;
count--;
broadcast(theCondition);
release(theLock);
    // consume the item in nextConsumed
}
How do we use threads in a C program?

- IEEE has defined the Pthreads interface (a set of C function prototypes and associated type definitions) for manipulating threads and synchronization mechanisms needed in multi-threaded applications.
- The implementation of the pthreads library on a particular operating system maps onto the thread functionality provided by that operating system.
- If the operating system does NOT provide kernel threads, then the pthreads implementation provides user-level threads.
- Linux supports kernel threads.
- To access the pthreads type definitions and function prototypes, a program needs to include the following file:

  ```
  #include <pthread.h>
  ```
- You may also have to specify `-lpthread` in the link command line.
Creating threads inside a process

- The first thread is created by the operating system when it creates the process; it is this thread that executes `main()`. We will refer to this as the main thread in the following slides.
- After processing arguments, `main()` then creates any other threads required by the application.
- A new thread is created by invoking the function `pthread_create()`.

```c
int pthread_create (pthread_t *thread_id,
                   const pthread_attr_t *attributes,
                   void *(*thread_function)(void *),
                   void *arguments);
```
**pthread_create()**

- `pthread_t *thread_id`
  this argument is the address of a variable into which the thread ID of the created thread will be written
- `pthread_attr_t *attributes`
  this argument allows you to fine tune various parameters; to use the defaults pass `NULL`
- `void *(*thread_function)(void *)`
  this argument is the function to execute in the new thread; the thread will terminate when this function terminates, or if the thread invokes `pthread_exit()`
- `void *arguments`
  this argument is passed as the single parameter to the function; to pass no arguments to the function, specify `NULL`
- On success, the identifier of the newly created thread is stored in the location pointed by the `thread_id` argument, and a 0 is returned. On error, a non-zero value is returned.
Causing a thread to exit

- When the routine invoked upon thread creation returns, the thread is terminated.
- Another way for a routine to terminate the thread in which it is executing is to invoke `pthread_exit()`.
- A routine in a thread should NEVER invoke `exit()`, as this will cause the process to terminate, thus killing all threads in the process.

```c
void pthread_exit (void *retval);
```
**pthread_exit()**

- `void *retval`  
  the return value of the thread; another thread can obtain this return value by **invoking** `pthread_join()`

- this function never returns – the thread that was executing the call to `pthread_exit()` has committed thread suicide
Waiting for another thread to die

- Often, after creating one or more threads to perform different functions, the main thread waits for the others to complete.
- A thread waits for another thread to terminate by invoking `pthread_join()`.
- The execution of the thread that invokes `pthread_join()` is suspended until the thread of interest terminates.

```c
int pthread_join (pthread_t thread,
                 void ** thread_return);
```
**pthread_join()**

- `pthread_t thread`
  this argument is the id of the thread whose termination is of interest
- `void **thread_return`
  If `thread_return` is not NULL, the return value of `thread` is stored in the location pointed to by `thread_return`
- **On success, the return value of `thread` is stored in the location pointed to by `thread_return`, and 0 is the function return. On error, a non-zero value is returned as the value of the function.**
Asking another thread to commit suicide

- Sometimes it is necessary for one thread to terminate another
- It is usually the main thread that does this
- Cancellation is the mechanism by which a thread can terminate the execution of another thread. More precisely, a thread can send a cancellation request to another thread. Depending on the target thread’s settings, it can either ignore the request, honour it immediately, or defer it until it reaches a cancellation point.
- A thread asks another to commit suicide by invoking `pthread_cancel()`

```c
int pthread_cancel (pthread_t thread);
```
pthread_cancel()

- `pthread_t thread`
  this argument is the id of the thread being asked to terminate itself

- On success, 0 is returned. On error, a non-zero value is returned.

- When a thread eventually honours a cancellation request, it performs as if `pthread_exit(PTHREAD_CANCELED)` has been called at that point – i.e. the thread stops executing with the return value `PTHREAD_CANCELED`
Example program

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *fun1(void *args)
{
    int N = *(int *)args;

    while (N-- > 0) {
        usleep(100000); /* sleep for 0.1 s */
        printf("sub1: %d more tenths of a second to go\n", N);
    }
    return NULL;
}

void *fun2(void *args)
{
    int N = *(int *)args;

    while (N-- > 0) {
        usleep(50000); /* sleep for 0.05 s */
        printf("sub2: %d more twentieths of a second to go\n", N);
    }
    return NULL;
}
```
```c
#define N1 100
#define N2 200

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    int n1 = N1, n2 = N2;

    printf("Creating thread 1: prints %d lines at 0.1 sec intervals\n", n1);
    if (pthread_create(&t1, NULL, fun1, (void *)&n1)) {
        fprintf(stderr, "Error creating thread 1\n");
        exit(1);
    }
    printf("Creating thread 2: prints %d lines at 0.05 sec intervals\n", n2);
    if (pthread_create(&t2, NULL, fun2, (void *)&n2)) {
        fprintf(stderr, "Error creating thread 2\n");
        exit(1);
    }
    pthread_join(t1, NULL); /* wait for thread 1 to finish */
    pthread_join(t2, NULL); /* wait for thread 2 to finish */
    return 0;
}
```
Mutexes

- The pthreads library provides a mechanism that provides the lock functionality – a **mutex**
- A mutex is a lock that guarantees three things:
  - **Atomicity** – locking a mutex is an atomic operation, meaning that the threads library assures you that if you locked a mutex, no other thread can succeed in locking that mutex at the same time
  - **Singularity** – when a thread manages to lock a mutex, it is assured that no other thread will be able to lock the mutex until the original thread releases the lock
  - **Non-busy wait** – if a thread (1\textsuperscript{st}) attempts to lock a mutex that was locked by another thread (2\textsuperscript{nd}), the 1\textsuperscript{st} thread will be suspended (and will not consume any CPU resources) until the lock is freed by the 2\textsuperscript{nd} thread. At that time, the 1\textsuperscript{st} thread will be resumed and continue execution, now owning the locked mutex
Creating and initializing a mutex

- In order to create a mutex, we first need to declare a variable of type `pthread_mutex_t` and to initialize it.
- The simplest form of initialization is to assign the value `PTHREAD_MUTEX_INITIALIZER` to the variable, as in:
  ```c
  pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
  ```
- Note that the default type of mutex, known as a “fast” mutex, is created using this initializer; if a thread locks the mutex and then tries to lock it again (e.g. in a recursive call), it will block until it releases the lock, which it never will – this is a deadlock.
- Pthreads supports a “recursive” mutex which does not suffer from this limitation. If a thread has locked a mutex, subsequent attempts to lock the mutex by the owning thread are successful; if this recursive locking has happened N levels deep, the thread must unlock the mutex N times before the mutex is free for another thread to lock it:
  ```c
  pthread_mutex_t b_mutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
  ```
- The “NP” means this is not available on all systems; for all of the systems of interest in this course, recursive mutexes are supported; we will use recursive mutexes when we implement thread-safe abstract data types.
Locking and unlocking a mutex

- To lock a mutex, we use `pthread_mutex_lock()`:
  ```c
  int pthread_mutex_lock(pthread_mutex_t *mutex);
  ```
  The function attempts to lock the mutex; if it is already locked by another thread, the calling thread blocks until the lock is unlocked.

- To unlock a mutex, we use `pthread_mutex_unlock()`:
  ```c
  int pthread_mutex_unlock(pthread_mutex_t *mutex);
  ```
  This causes the mutex to be unlocked; if another thread is blocked on this mutex, it will be unblocked and will have locked the mutex.

```c
if (pthread_mutex_lock(&a_mutex)) {
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
}
```
```c
if (pthread_mutex_unlock(&a_mutex)) {
    perror("pthread_mutex_unlock");
    pthread_exit(NULL);
}
```
pthread_mutex_lock() and pthread_mutex_unlock()

- `pthread_mutex_t *mutex`
  this argument is a pointer to a previously initialized mutex

- On success, a 0 is returned. On error, a non-zero value is returned.

- A mutex can be destroyed by calling

  ```
  int pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```

- This function destroys the specified mutex object, freeing any resources that it might possess; the mutex must be unlocked at the time of this call.

- If successful, returns 0; otherwise (i.e. mutex is currently locked), returns non-zero.
Example program with mutex to guard access to stdout

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

/* this mutex is used to guard access to stdout */
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

void *fun1(void *args)
{
    int N = *(int *)args;

    while (N-- > 0) {
        usleep(100000); /* sleep for 0.1 s */
        pthread_mutex_lock(&my_mutex); /* obtain exclusive access */
        printf("sub1: %d more tenths of a second to go\n", N);
        pthread_mutex_unlock(&my_mutex); /* release exclusive access */
    }
    return NULL;
}

void *fun2(void *args)
{
    int N = *(int *)args;

    while (N-- > 0) {
        usleep(50000); /* sleep for 0.05 s */
        pthread_mutex_lock(&my_mutex); /* obtain exclusive access */
        printf("sub2: %d more twentieths of a second to go\n", N);
        pthread_mutex_unlock(&my_mutex); /* release exclusive access */
    }
    return NULL;
}
```

CIS 415, Spring 2015  Threads and thread-safe ADTs 46
#define N1 100
#define N2 200

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    int n1 = N1, n2 = N2;

    printf("Creating thread 1: prints %d lines at 0.1 sec intervals\n", n1);
    if (pthread_create(&t1, NULL, fun1, (void *)&n1)) {
        fprintf(stderr, "Error creating thread 1\n");
        exit(1);
    }
    printf("Creating thread 2: prints %d lines at 0.05 sec intervals\n", n2);
    if (pthread_create(&t2, NULL, fun2, (void *)&n2)) {
        fprintf(stderr, "Error creating thread 2\n");
        exit(1);
    }
    pthread_join(t1, NULL); /* wait for thread 1 to finish */
    pthread_join(t2, NULL); /* wait for thread 2 to finish */
    return 0;
}
Creating, initializing, and signalling of a condition variable

- Creation of a condition variable requires declaration of a variable of type `pthread_cond_t` and initializing it properly

  ```
  pthread_cond_t nonempty_q = PTHREAD_COND_INITIALIZER;
  ```

- To signal a condition variable, use `pthread_cond_signal()`, which will wake up one thread (of the possibly many threads waiting on the variable)

  ```
  int pthread_cond_signal(&nonempty_q);
  ```

- Can also use `pthread_cond_broadcast()`, which will wake up all threads waiting on the variable

  ```
  int pthread_cond_broadcast(&nonempty_q);
  ```
Waiting on a condition variable

- A thread may wait for a signal on a condition variable using `pthread_cond_wait()`
- The user must supply both a condition variable and a mutex in the call; the mutex **must** have been locked prior to calling the wait function.
- The calling thread will block until the condition has been signalled and it has been permitted to continue; note that the calling thread will continue to own the lock on the mutex when it continues.
- The example on the next page shows how to use `pthread_cond_wait()`; it assumes that `nonempty_q` is a properly initialized condition variable and that `q_mutex` is a properly initialized mutex.
Worker thread with cond variable

/* function prototype for a function that returns the request at the head of a FIFO queue; returns pointer to that request or NULL */
Request *get_request(void);

/* worker thread code to obtain the next request to process WITHOUT busy waiting */

pthread_mutex_lock(&q_mutex); /* first, lock the mutex */

/* obtain request from the queue; if queue is empty, wait for another thread to signal the condition variable; repeat until have a Request */

Request *p;
while ((p = get_request()) == NULL)
    pthread_cond_wait(&nonempty_q, &q_mutex);
pthread_mutex_unlock(&q_mutex); /* release the mutex */

/* now process the request . . . */
Who signals the condition variable?

- Why the “while” test for a non-NULL return from get_request()?
  - The signaller of the condition variable can use either
    `pthread_cond_signal()` or `pthread_cond_broadcast()`. If it
    uses the latter, then you might wake up and another thread has already
    retrieved the request from the nonempty queue.

- Who signals the condition variable? The dispatcher thread signals
  the condition variable every time it adds a Request to the queue.

- The relevant code is:

```c
pthread_mutex_lock(&q_mutex); /* lock the mutex */
add_request(/* the address of a request structure */);
pthread_cond_signal(&nonempty_q); /* send the signal */
pthread_mutex_unlock(&q_mutex); /* release the mutex */
```
Thread-Safe ADTs in C
Thread-safe ADTs

- The non-thread-safe ADTs covered earlier provide us with all the functionality that we require for those particular data types.
- To create thread-safe versions, we need to create appropriate critical sections around calls to the non-thread-safe methods to guarantee the lack of race conditions.
- Rather than create another complete implementation, but this time with appropriate pthread logic to create the critical sections, it is far easier and less error prone to simply encapsulate an instance of the non-thread-safe ADT inside of a thread-safe instance. Each method will then act like a Java synchronized method.
Transactions

- One often wishes to create a transaction across two or more separate calls – e.g. insert a new entry into a collection ONLY if it is *not* already there.
- This can be achieved using recursive mutexes by providing methods for acquiring and releasing the ADT lock.
- Once a thread possesses the lock, it may invoke as many of the other methods as it wishes before releasing the lock.
- Use of a recursive mutex enables each of our “synchronized” methods to be invoked in the scope of one of these larger transactions.
Outline of a generic, thread-safe container interface

#ifndef _TSFOO_H_
#define _TSFOO_H_

/* interface definition for generic, thread-safe Foo container */
#include “tsiterator.h”

typedef struct tsfoo TSFoo;
TSFoo *tsfoo_create(/* appropriate arguments */);
void tsfoo_lock(TSFoo *f);
void tsfoo_unlock(TSFoo *f);
void tsfoo_destroy (TSFoo *f, void (*freeFxn)(void *element));
void tsfoo_purge(TSFoo *f, void (*freeFxn)(void *element));
int tsfoo_put(TSFoo *f, void *element);
int tsfoo_get(TSFoo *f, void **element);
int tsfoo_isEmpty(TSFoo *st);
long tsfoo_size(TSFoo *f);
void **tsfoo_toArray(TSFoo *f, long *len);
TSIterator *tsfoo_it_create(TSFoo *f);
#endif /* _TSFOO_H_ */
What does each line mean?

- `typedef struct tsfoo TSFoo;` opaque data type for TSFoo's
- `TSFoo *tsfoo_create(/* appropriate arguments */);` this is called to create a new instance of a TSFoo
- `void tsfoo_lock(TSFoo *f);` acquires lock for ‘f’ to support multi-call transaction
- `void tsfoo_unlock(TSFoo *f);` releases lock for ‘f’ after completion of multi-call transaction
What does each line mean?

- `void tsfoo_destroy (TSFoo *f, void (*freeFxn)(void *e));`
  this destroys the TSFoo instance
- `void tsfoo_purge (TSFoo *f, void (*freeFxn)(void *e));`
  purges all elements from the TSFoo
- `int tsfoo_put(TSFoo *f, void *element);`
  adds an element to ‘f’
- `int tsfoo_get(TSFoo *f, void **element);`
  destructively fetches an element from ‘f’, returning the element in *element
What does each line mean?

- `int tsfoo_isEmpty(TSFoo *f);` returns 1 if the TSFoo is empty, returns 0 if not
- `long tsfoo_size(TSFoo *f);` returns the number of elements in the TSFoo
- `void **tsfoo_toArray(TSFoo *f, long *len);` returns an array of pointers to the elements in the TSFoo in the natural order defined by Foo’s; the number of elements in the array is returned in *len
- `TSIterator *tsfoo_it_create(TSFoo *f);` creates generic, thread-safe iterator for this TSFoo instance; successive calls to tsit_next() on the iterator returned will return the elements of the TSFoo in the natural order defined by Foo’s
A note on thread-safe iterators

- there is a general problem when using iterators if the structure can change while you are traversing it
- when you are given a thread-safe iterator by the factory method of one of these ADTs, the calling thread also possesses the lock on the ADT, and will retain that lock until you invoke tsit_destroy() on that iterator
Generic, thread-safe iterator – tsiterator.h

#ifndef _TSITERATOR_H_
#define _TSITERATOR_H_

/*
 * interface definition for thread safe generic iterator
 */

#include <pthread.h>

typedef struct tsiterator TSIterator;

TSIterator *tsit_create(pthread_mutex_t *lock, long size, void **elements);

int tsit_hasNext(TSIterator *it);

int tsit_next(TSIterator *it, void **element);

void tsit_destroy(TSIterator *it);

#endif /* _TSITERATOR_H_ */
```
#include "tsiterator.h"
#include <stdlib.h>
#include <pthread.h>

struct tsiterator {
    long next;
    long size;
    void **elements;
    pthread_mutex_t *lock;
};

TSIterator *tsit_create(pthread_mutex_t *lock, long size, void **elements) {
    TSIterator *it = (TSIterator *)malloc(sizeof(TSIterator));

    if (it != NULL) {
        it->next = 0L;
        it->size = size;
        it->elements = elements;
        it->lock = lock;
    }
    return it;
}
```
```c
int tsit_hasNext(TSIterator *it) {
    return (it->next < it->size) ? 1 : 0;
}

int tsit_next(TSIterator *it, void **element) {
    int status = 0;
    if (it->next < it->size) {
        *element = it->elements[it->next++];
        status = 1;
    }
    return status;
}

void tsit_destroy(TSIterator *it) {
    free(it->elements);
    pthread_mutex_unlock(it->lock);
    free(it);
}
```
Generic, thread-safe stack – tsstack.h

#ifndef _TSSTACK_H_
define _TSSTACK_H_

/*
 * interface definition for generic stack implementation
 */
#include “tsiterator.h”

typedef struct tsstack TSStack; /* opaque type definition */

TSStack *tsstack_create(long capacity);
void tsstack_destroy(TSStack *st, void (*freeFxn)(void *element));
void tsstack_purge(TSStack *st, void (*freeFxn)(void *element));
void tsstack_lock(TSStack *st);
void tsstack_unlock(TSStack *st);
int tsstack_push(TSStack *st, void *element);
int tsstack_pop(TSStack *st, void **element);
int tsstack.peek(TSStack *st, void **element);
int tsstack.isEmpty(TSStack *st);
long tsstack.size(TSStack *st);
void **tsstack.toArray(TSStack *st, long *len);
TSLiterator *tsstack_it_create(TSStack *st);

#endif /* _TSSTACK_H_ */
#include "tsstack.h"
#include "stack.h"
#include <stdlib.h>
#include <pthread.h>

#define LOCK(st) &((st)->lock)

struct tsstack {
    Stack *st;
    pthread_mutex_t lock;
};
TSStack *tsstack_create(long capacity) {
    TSStack *tsst = (TSStack *)malloc(sizeof(TSStack));

    if (tsst != NULL) {
        Stack *st = stack_create(capacity);

        if (st == NULL) {
            free(tsst);
            tsst = NULL;
        } else {
            pthread_mutexattr_t ma;
            pthread_mutexattr_init(&ma);
            pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_RECURSIVE);
            tsst->st = st;
            pthread_mutex_init(LOCK(tsst), &ma);
            pthread_mutexattr_destroy(&ma);
        }
    }

    return tsst;
}
void tsstack_destroy(TSSStack *st, void (*freeFxn)(void*)) {
    pthread_mutex_lock(LOCK(st));
    stack_destroy(st->st, freeFxn);
    pthread_mutex_unlock(LOCK(st));
    pthread_mutex_destroy(LOCK(st));
    free(st);
}

void tsstack_purge(TSSStack *st, void (*freeFxn)(void*)) {
    pthread_mutex_lock(LOCK(st));
    stack_purge(st->st, freeFxn);
    pthread_mutex_unlock(LOCK(st));
}

void tsstack_lock(TSSStack *st) {
    pthread_mutex_lock(LOCK(st));
}

void tsstack_unlock(TSSStack *st) {
    pthread_mutex_unlock(LOCK(st));
}
```c
int tsstack_push(TSStack *st, void *element) {
    int result;
    pthread_mutex_lock(LOCK(st));
    result = stack_push(st->st, element);
    pthread_mutex_unlock(LOCK(st));
    return result;
}

int tsstack_pop(TSStack *st, void **element) {
    int result;
    pthread_mutex_lock(LOCK(st));
    result = stack_pop(st->st, element);
    pthread_mutex_unlock(LOCK(st));
    return result;
}

int tsstack_peek(TSStack *st, void **element) {
    int result;
    pthread_mutex_lock(LOCK(st));
    result = stack_peek(st->st, element);
    pthread_mutex_unlock(LOCK(st));
    return result;
}
```
int tsstack_isEmpty(TSStack *st) {
    int result;
    pthread_mutex_lock(LOCK(st));
    result = stack_isEmpty(st->st);
    pthread_mutex_unlock(LOCK(st));
    return result;
}

long tsstack_size(TSStack *st) {
    long result;
    pthread_mutex_lock(LOCK(st));
    result = stack_size(st->st);
    pthread_mutex_unlock(LOCK(st));
    return result;
}

void **tsstack_toArray(TSStack *st, long *len) {
    void **result;
    pthread_mutex_lock(LOCK(st));
    result = stack_toArray(st->st, len);
    pthread_mutex_unlock(LOCK(st));
    return result;
}
TSIterator *tsstack_it_create(TSSStack *st) {
    TSIterator *it = NULL;
    void **tmp;
    long len;

    pthread_mutex_lock(LOCK(st));
    tmp = stack_toArray(st->st, &len);
    if (tmp != NULL) {
        it = tsit_create(LOCK(st), len, tmp);
        if (it == NULL)
            free(tmp);
    }
    if (it == NULL)
        pthread_mutex_unlock(LOCK(st));
    return it;
}