Programming Shared-memory Platforms with Pthreads

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Threaded Programming Models

- Library-based models
  - —all data is shared, unless otherwise specified
  - —examples: Pthreads, Intel Threading Building Blocks, Java Concurrency, Boost, Microsoft .Net Task Parallel Library
- Directive-based models, e.g., OpenMP
  - —shared and private data
  - —pragma syntax simplifies thread creation and synchronization
- Programming languages
  - —Cilk Plus (Intel, GCC)
  - —CUDA (NVIDIA)
  - —Habanero-Java (Rice)

Topics for Today

- The POSIX thread API (Pthreads)
- Synchronization primitives in Pthreads
  - —mutexes
  - —condition variables
  - —reader/writer locks
- Thread-specific data

POSIX Thread API (Pthreads)

- Standard threads API supported on almost all platforms
- Concepts behind Pthreads interface are broadly applicable
  —largely independent of the API
  —useful for programming with other thread APIs as well
    – Windows threads
    – Java threads
    – …
- Threads are peers, unlike Linux/Unix processes
  —no parent/child relationship

Why Should I Care About Pthreads?

Pthreads is the foundation for multithreaded programming models

- **Used to implement higher-level threading libraries such as Boost and Intel's Threading Building Blocks**
- **Used to implement runtime systems for directive-and language-based programming models such as OpenMP and Cilk Plus**

PThread Creation

Asynchronously invoke `thread_function` in a new thread

```
#include <pthread.h>
int pthread_create(
   pthread_t *thread_handle,  /* returns handle here */
   const pthread_attr_t *attribute,
   void * (*thread_function)(void *),
   void *arg);  /* single argument; perhaps a structure */
```

`attribute` created by `pthread_attr_init:`

specifies the size for the thread's stack and how the thread should be managed by the OS

Thread Attributes

Special functions exist for getting/setting each attribute property

e.g., int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)

- Stack size
- Detach state
  - PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE
    - reclaim storage at termination (detached) or retain (joinable)
- Scheduling policy
  - SCHED_OTHER: standard round robin (priority must be 0)
  - SCHED_FIFO, SCHED_RR: real time policies
    - FIFO: re-enter priority list at head; RR: re-enter priority list at tail
- Scheduling parameters
  - only priority
- Inherit scheduling policy
  - PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED
- Thread scheduling scope
  - PTHREAD_SCOPE_SYSTEM, ~~PTHREAD_SCOPE_PROCESS~~

Wait for Pthread Termination

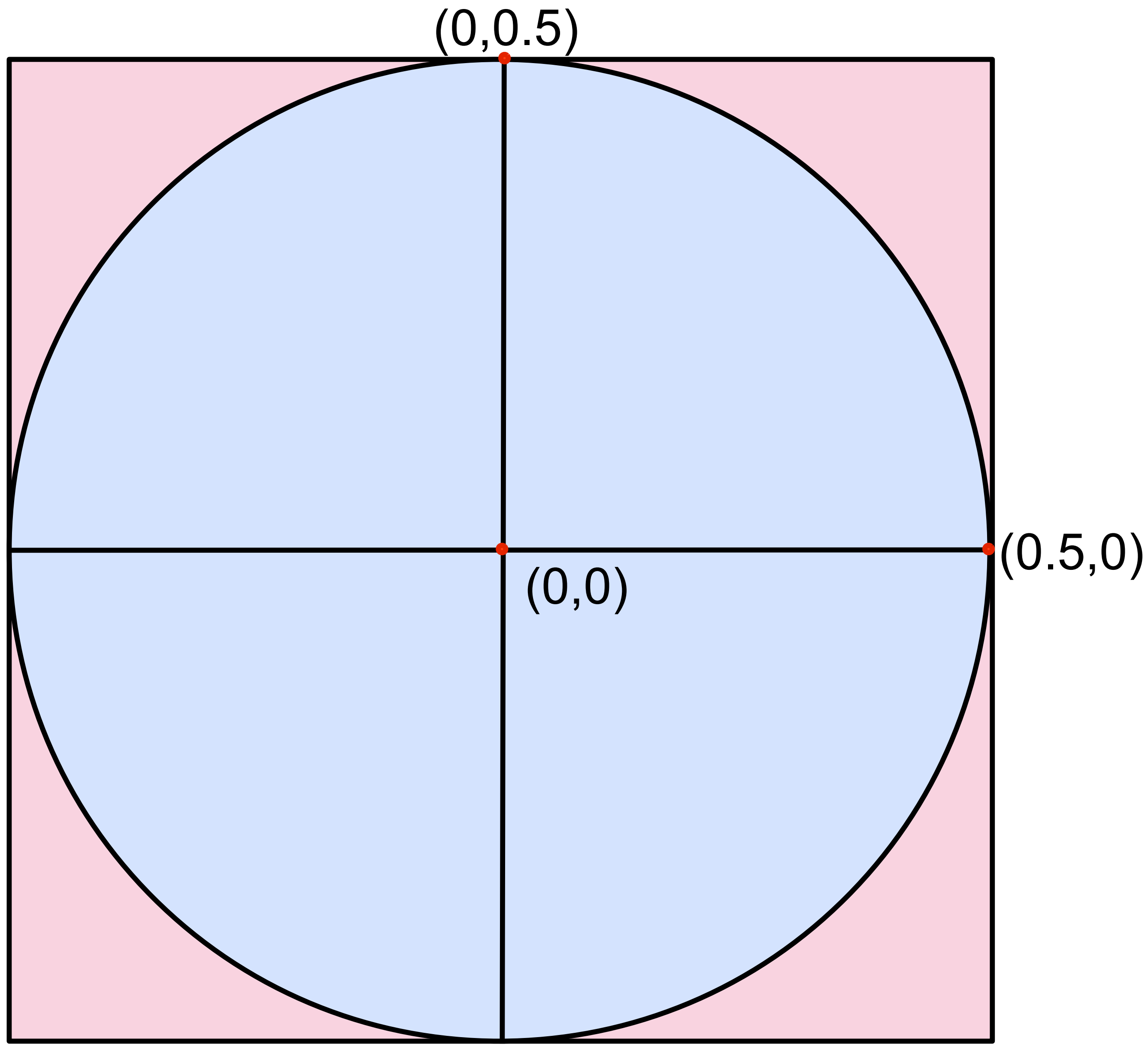
Suspend execution of calling thread until thread terminates

```
#include <pthread.h>
int pthread_join (
  pthread_t thread,  /* thread id */
  void **ptr);  /* ptr to location for return code a terminating
                   thread passes to pthread_exit */
```

Running Example: Monte Carlo Estimation of Pi

Approximate Pi

- generate random points with x, y $\in$ [-0.5, 0.5]
- test if point inside the circle, i.e., $x^2 + y^2 < (0.5)^2$
- ratio of circle to square = $\pi r^2 / 4r^2 = \pi / 4$
- $\pi \approx 4 * $ (number of points inside the circle) / (number of points total)



Example: Creation and Termination (`main`)

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
   ...
   pthread_t p_threads[NUM_THREADS];
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   for (i=0; i< NUM_THREADS; i++) {
      hits[i] = i;
      pthread_create(&p_threads[i], &attr, compute_pi,
         (void*) &hits[i]);
   }
   for (i=0; i< NUM_THREADS; i++) {
      pthread_join(p_threads[i], NULL);
      total_hits += hits[i];
   }
   ...
```

default attributes

thread function

thread argument

Example: Thread Function (`compute_pi`)

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double x_coord, y_coord;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
       x_coord = (double)(rand_r(&seed))/(RAND_MAX) - 0.5;
       y_coord =(double)(rand_r(&seed))/(RAND_MAX) - 0.5;
       if ((x_coord * x_coord + y_coord * y_coord) < 0.25)
           local_hits++;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

tally how many random points fall in a unit circle centered at the origin

`rand_r`: reentrant random number generation in [0,RAND_MAX]

Example: Thread Function (compute_pi)

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double x_coord, y_coord;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
       x_coord = (double)(rand_r(&seed))/(RAND_MAX) - 0.5;
       y_coord =(double)(rand_r(&seed))/(RAND_MAX) - 0.5;
       if ((x_coord * x_coord + y_coord * y_coord) < 0.25)
           local_hits++;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

avoid false sharing by using a local accumulator

Critical Sections and Mutual Exclusion

- Critical section = code executed by only one thread at a time

/* threads compete to update global variable best_cost */

```
if (my_cost < best_cost)
    best_cost = my_cost;
```

- Mutex locks enforce mutual exclusion in Pthreads
  - mutex lock states: locked and unlocked
  - only one thread can lock a mutex lock at any particular time
- Using mutex locks
  - request lock before executing critical sect
  - enter critical section when lock granted
  - release lock when leaving critical section
- Operations

created by
pthread_mutex_attr_init
specifies mutex type

```
int pthread_mutex_init (pthread_mutex_t *mutex ,
      const pthread_mutexattr_t *lock_attr)
int pthread_mutex_lock(pthread_mutex_t *mutex_lock)
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock)
```

atomic operation

Mutex Types

- **Normal**
  - — thread deadlocks if tries to lock a mutex it already has locked
- **Recursive**
  - — single thread may lock a mutex as many times as it wants
    - – increments a count on the number of locks
  - — thread relinquishes lock when mutex count becomes zero
- **Errorcheck**
  - — report error when a thread tries to lock a mutex it already locked
  - — report error if a thread unlocks a mutex locked by another

Example: Reduction Using Mutex Locks

```
pthread_mutex_t cost_lock;
...
int main() {
   ...
   pthread_mutex_init(&cost_lock, NULL);
   ...
}
void *find_best(void *list_ptr) {
   ...
   pthread_mutex_lock(&cost_lock);   /* lock the mutex */
   if (my_cost < best_cost)
      best_cost = my_cost;
   pthread_mutex_unlock(&cost_lock); /* unlock the mutex */
}
```

use default (normal) lock type

critical section

Producer-Consumer Using Mutex Locks

Constraints

- **Producer thread**
  - must not overwrite the shared buffer until previous task has picked up by a consumer
- **Consumer thread**
  - must not pick up a task until one is available in the queue
  - must pick up tasks one at a time

Producer-Consumer Using Mutex Locks

```
pthread_mutex_t task_queue_lock;
int task_available;
...
main() {
    ...
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ...
}
void *producer(void *producer_thread_data) {
    ...
    while (!done()) {
       inserted = 0;
       create_task(&my_task);
       while (inserted == 0) {
           pthread_mutex_lock(&task_queue_lock);
           if (work_available == 0) {
              consumer_work = my_task; work_available = 1;
              inserted = 1;
           }
           pthread_mutex_unlock(&task_queue_lock);
       }
    }
}
```

critical section

Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
       extracted = 0;
       while (extracted == 0) {
           pthread_mutex_lock(&task_queue_lock);
           if (work_available == 1) {
              my_task = consumer_work;
              work_available = 0;
              extracted = 1;
           }
           pthread_mutex_unlock(&task_queue_lock);
       }
       process_task(my_task);
    }
```

critical section

Overheads of Locking

- Locks enforce serialization
  - threads must execute critical sections one at a time
- Large critical sections can seriously degrade performance
- Reduce overhead by overlapping computation with waiting

```
int pthread_mutex_trylock(pthread_mutex_t *mutex_lock)
```

  - acquires lock if available
  - returns EBUSY if not available
  - enables a thread to do something else if a lock is unavailable

Condition Variables for Synchronization

Condition variable: associated with a predicate and a mutex

- Using a condition variable
  - thread can block itself until a condition becomes true
    - thread locks a mutex
    - tests a predicate defined on a shared variable
      - if predicate is false, then wait on the condition variable
      - waiting on condition variable unlocks associated mutex
  - when some thread makes a predicate true
    - that thread can signal the condition variable to either
      - wake one waiting thread
      - wake all waiting threads
    - when thread releases the mutex, it is passed to first waiter

Pthread Condition Variable API

```
/* initialize or destroy a condition variable */
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/* block until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *wtime);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

abort wait if time exceeded

wake one

wake all

Condition Variable Producer-Consumer

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
}
```

default initializations

Producer Using Condition Variables

```
void *producer(void *producer_thread_data) {
    int inserted; task_t *t;
    while (!done()) {
      t = create_task();
      pthread_mutex_lock(&task_queue_cond_lock);
      while (work_available == 1)
          pthread_cond_wait(&cond_queue_empty,
              &task_queue_cond_lock);
      consumer_work = t;
      work_available = 1;
      pthread_cond_signal(&cond_queue_full);
      pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

note loop

releases mutex on wait

reacquires mutex when woken

Why Loop When Awaiting A Condition?

note loop

```
...
pthread_mutex_lock(&task_queue_cond_lock);
while (work_available == 1)
      pthread_cond_wait(&cond_queue_empty,
            &task_queue_cond_lock);
...
pthread_cond_signal(&cond_queue_full);
pthread_mutex_unlock(&task_queue_cond_lock);
```

When using condition variables there is always a **boolean predicate** that indicates if the thread should proceed or wait

Spurious wakeups may occur when waiting on condition variables.

Thus, waking up from a wait on a condition variable doesn't imply anything about the value of the boolean predicate; the predicate must be re-evaluated when a conditional wait completes

Why Allow Spurious Wakeups?

- **Defining condition variable waits to permit spurious forces correct/robust code by requiring predicate loops.**
  - **"Religiously" using a loop protects the application against its own imperfect coding practices.**
- **Making condition wakeup completely predictable might substantially slow all condition variable operations.**
  - **It isn't difficult to imagine machines and implementation code that could exploit this semantics to improve the performance of average condition wait operations.**

-- David R. Butenhof - author of "Programming with POSIX Threads"

Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (work_available == 0)
            pthread_cond_wait(&cond_queue_full,
                &task_queue_cond_lock);
        my_task = consumer_work;
        work_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```

note loop

releases mutex on wait

reacquires mutex when woken

Reader-Writer Locks

- Purpose: access to data structure when
  - frequent reads
  - infrequent writes
- Acquire read lock
  - OK to grant when other threads already have acquired read locks
  - if write lock on the data or queued write locks
    - reader thread performs a condition wait
- Acquire write lock
  - if multiple threads request a write lock
    - must perform a condition wait

Read-Write Lock Sketch

- While pthreads provides a pthread_rwlock, you could build your own using basic primitives
- Use a data type with the following components
  - —a count of the number of active readers
  - —0/1 integer specifying whether a writer is active
  - —a condition variable readers_proceed
    - – signaled when readers can proceed
  - —a condition variable writer_proceed
    - – signaled when one of the writers can proceed
  - —a count pending_writers of pending writers
  - —a mutex read_write_lock
    - – controls access to the reader/writer data structure

Thread-Specific Data

Goal: associate some state with a thread

- **Choices**
  - — pass data as argument to each call thread makes
    - – not always an option, e.g. when using predefined libraries
  - — store data in a shared variable indexed by thread id
  - — using thread-specific keys
- **Why thread-specific keys?**
  - — libraries want to maintain internal state
  - — don't want to require clients to know about it and pass it back
  - — substitute for static data in a threaded environment
- **Operations**

associate NULL with key in each active thread

```
int pthread_key_create(pthread_key_t *key, void (*destroy)(void *))

int pthread_setspecific(pthread_key_t key, const void *value)

void *pthread_getspecific(pthread_key_t key)
```

retrieve value for current thread from key

associate (key,value) with current thread

Thread-Specific Data Example: Key Creation

Example: remember performance information for a thread

```
#include <pthread.h>
static pthread_key_t profiler_state;
initialize_profiler_state() {
   …
   pthread_key_create(&profiler_state,
                       (void *) free_profile);
   …
}
void free_profile(profile *my_profile) {
  free(my_profile);
}
```

opaque handle used to locate thread-specific data

destructor for key value

Thread-Specific Data Example: Specific Data

Example: remember profiler state for a thread

```
void init_thread_profile(…) {
   profile *my_profile = (profile *) malloc(…);
   pthread_setspecific(profiler_state, (void *) my_profile);
   …
}
void update_thread_profile(...) {
  profile *my_profile = (profile *)
                         pthread_getspecific(profiler_state);
  // update profile
}
```

References

- Adapted from slides “Programming Shared Address Space Platforms” by Ananth Grama.
- Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell. “Pthreads Programming: A POSIX Standard for Better Multiprocessing.” O'Reilly Media, 1996.
- Chapter 7. “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003