CS529 Lecture 03: POSIX Threads

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

February 22, 2011

Outline

Sources of material

- "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.
- "Programming Shared Memory Platforms with Pthreads", by John Mellor Crummey

Outline

- Shared-address space programming taxonomy
- The POSIX threads API (Pthreads)
- Synchronization primitives in Pthreads
  - Mutexes
  - Condition variables
  - Reader/writer locks

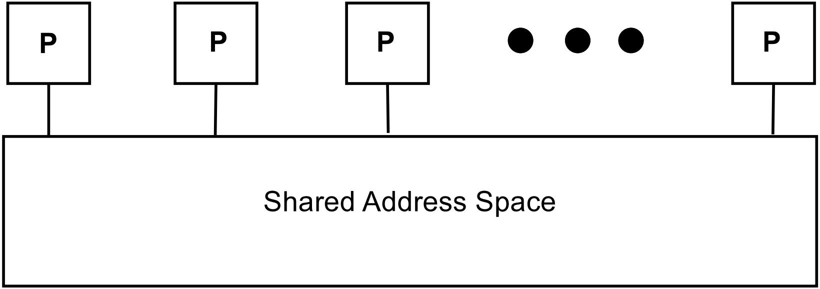
Outline

Shared Address Space Programming Models

- Lightweight processes and threads
  - all memory is global and shared
  - examples: Pthreads, Cilk (lazy, lightweight threads)
- Process-based models
  - each process' data is private, unless otherwise specified
  - example: Linux `shget, shmat, shmdt` API
- Directive-based models (e.g. OpenMP)
  - shared and private data
  - logically shared address space
  - simplify decomposition, scheduling, synchronization
- Global Address Space programming languages
  - shared and private data
  - hardware based on distributed memory, often with shared-memory nodes
  - Unified Parallel C, Co-array Fortran

Thread

- A single, sequential stream of control in a program
- Logical machine model
  - Flat global memory shared among all threads
  - Local stack of frames for each thread's active procedures



Why Threads?

- Portable, widely available programming model
  - Used on both serial machines (latency overlap) and parallel machines (concurrency)
- Useful for hiding latency
  - Overlap I/O, communication, or memory latency with the execution of threads other than the stalled ones
- Scheduling and load balancing
  - Can implement dynamic concurrency (N-to-M execution model)
- Relatively easy to program
  - Significantly easier than message passing (no naming of processors, no explicit communication)

Outline

POSIX Threads API (Pthreads)

- Standard threads API supported by vendors (software, with architecture-dependent implementation)
- Concepts behind POSIX threads interface are broadly applicable
  - Concurrency and synchronization abstractions relatively independent of the API
  - Useful for programming with other thread APIs
    - NT threads
    - Java threads
- Threads are peers, unlike processes
  - no parent/child relationship
  - inherit parent/child properties of process address space

POSIX Thread 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` contains details about
  - whether scheduling policy is inherited or explicit
  - scheduling policy, scheduling priority
  - stack size, stack guard region size

Thread Attributes

- Detach state
  - PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE
  - reclaim storage at termination (detached) or join (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
- Stack size

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 */
```

Example: Thread Creation and Termination

```
#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];
  }
...
```

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))/((1<<15)-1) - 0.5;
    y_coord =(double)(rand_r(&seed))/((1<<15)-1) - 0.5;
    if ((x_coord * x_coord + y_coord * y_coord) < 0.25)
      local_hits++;
  }
  *hit_pointer = local_hits;
  pthread_exit(0);
}
```

Programming for Performance Note

- Code carefully minimizes false-sharing of cache lines
  - false sharing
    - multiple processors access words in the same cache line
    - at least one processor updates a word in the cache line
    - no word updated by one processor is accessed by another
- False sharing resolved in code by localizing (privatizing) variables

Example: Thread function (compute_pi) with false sharing prevention

```
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))/((1<<15)-1) - 0.5;
    y_coord =(double)(rand_r(&seed))/((1<<15)-1) - 0.5;
    if ((x_coord * x_coord + y_coord * y_coord) < 0.25)
      local_hits++; // avoid false sharing!
  }
  *hit_pointer = local_hits;
  pthread_exit(0);
}
```

Outline

Data Races in Pthreads Programs

- Consider

```
/* threads compete to update global variable best_cost */
if (my_cost < best_cost)
  best_cost = my_cost;
```

  - two threads
  - initial value of `best_cost` is 100
  - values of `my_cost` are 50 and 75 for threads t1 and t2
- After execution, `best_cost` could be 50 or 75
- 75 does not correspond to any serialization of the threads

Critical Sections and Mutual Exclusion

- Critical section = must execute code 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 critical sections 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 section
  - enter critical section when lock granted
  - release lock when leaving critical section
- Operations

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

Mutex Types

- Normal
  - thread deadlocks if it 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
- Error check
  - 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 */
}
```

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 (task_available == 0) {
        insert_into_queue(my_task);
        task_available = 1;
        inserted = 1;
      }
      pthread_mutex_unlock(&task_queue_lock);
    }
  }
}
```

Producer Consumer using Mutex 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 (task_available == 1) {
        extract_from_queue(&my_task);
        task_available = 0;
        extracted = 1;
      }
      pthread_mutex_unlock(&task_queue_lock);
    }
  process_task(my_task);
}
```

Overheads of Locking

- Locks enforce serialization
  - threads must execute critical sections one at a time
  - many critical sections may co-exist, one convoy of threads per critical section
- Large critical sections can seriously degrade performance
  - Long periods of serialization
- Reduce overhead by overlapping computation with waiting

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

  - acquire lock if available
  - return EBUSY if not available
  - enables a thread to do something else if lock 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 or 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);
```

Condition Variable Producer Consumer (main)

```
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 */
}
```

Producer using Condition Variables

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

Consumer using Condition Variables

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

Composite Synchronization Constructs

- POSIX threads provides only basic synchronization constructs
- Build higher-level constructs from basic ones
  - e.g. readers-writer locks

Readers-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 lock
  - 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

Readers-Writer Lock Sketch

- Use a data type with the following components
  - a count of the number of active readers
  - a count of the number of waiting 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 `waiting_writers` of waiting writers
  - a mutex `read_write_lock`
    - controls access to the reader/writer data structure

Readers-writer Lock with Writer Priority

```
void *reader_start() {
  pthread_mutex_lock(&read_write_lock);
  while (waiting_writers + active_writer > 0)) {
    waiting_readers++;
    pthread_cond_wait(&readers_proceed, &read_write_lock);
    waiting_readers--;
  }
  active_readers++;
  pthread_mutex_unlock(&read_write_lock);
}
```

Readers-writer Lock with Writer Priority

```
void *reader_finish() {
  pthread_mutex_lock(&read_write_lock);
  active_readers--;
  if (active_readers == 0 && waiting_writers > 0) {
    pthread_cond_signal(&writer_proceed);
  }
  pthread_mutex_unlock(&read_write_lock);
}
```

Readers-writer Lock with Writer Priority

```
void *writer_start() {
  pthread_mutex_lock(&read_write_lock);
  while ((active_writers + active_readers) > 0) {
    waiting_writers++;
    pthread_cond_wait(&writer_proceed,&read_write_lock);
    waiting_writers--;
  }
  active_writers++;
  pthread_mutex_unlock(&read_write_lock);
}
```

Readers-writer Lock with Writer Priority

```
void *writer_finish() {
  pthread_mutex_lock(&read_write_lock);
  active_writers--;
  if (waiting_writers > 0} {
    pthread_cond_signal(&writer_proceed);
  }
  else if (waiting_readers > 0) {
    pthread_cond_broadcast(&reader_proceed);
  }
  pthread_mutex_unlock(&read_write_lock);
}
```