Programming Shared Address Space Platforms using Pthreads

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Topic Overview

- Shared address space programming models
- The POSIX Thread API
- Synchronization Primitives in Pthreads
  - Mutex
  - Condition variable
  - Etc.
A Shared Address Space System

- The logical machine model of a thread-based programming paradigm
Shared Address Space Programming

- **Multiple threads (processes) on shared address space**
  - Communication is implicitly specified
  - Focus on constructs for expressing concurrency and synchronization
    - Careful control required when shared data are accessed

- **Programming models**
  - Threads libraries (classes): Pthreads, Java threads
  - New programming languages: Ada
  - Modifying syntax of existing languages: UPC (Berkeley Unified Parallel C), Cilk Plus, C++11
  - Compiler directives: OpenMP
Overview of Programming Models

- A single-thread process:
  
  ```
  for (row = 0; row < n; row++)
      for (column = 0; column < n; column++)
          c[row][column] = dot_product(get_row(a, row),
                                      get_col(b, col));
  ```

- A multi-thread process:
  
  ```
  for (row = 0; row < n; row++)
      for (column = 0; column < n; column++)
          c[row][column] = create_thread(dot_product(get_row(a, row),
                                                  get_col(b, col)));
Threads vs. Processes

**Process**
- One address space per process
- Each process has its own data (global variables), stack, heap

**Thread**
- Multiple threads share on address space
  - But its own stack and register context
- Threads within the same address space share data (global variables), heap
P(POSIX®)Threads

- POSIX: Portable Operating System Interface
- Standard threads API supported by most vendors
  - Created by IEEE
  - Called POSIX 1003.1c in 1995
- A library that can be linked with C programs.
- Concepts are largely independent of the API
- Useful for programming with other thread APIs
  - Windows NT threads
  - Solaris threads
  - Java threads
  - ...

SSE3054: Multicore Systems | Spring 2016 | Jinkyu Jeong (jinkyu@skku.edu)
Creating a Pthread

```c
int pthread_create(
    pthread_t* thread /* out */ ,
    const pthread_attr_t* attr /* in */ ,
    void* (*start_routine) (void*) /* in */ ,
    void* arg /* in */ ) ;
```

- Specify thread handle (allocated before calling)
- Specify the attributes of a creating thread
- The function that the thread is to run
- Pointer to the argument passed to the function `start_routine`
Pthread Attributes

- **Stack size**
- **Detach state**
  - PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_ATTACHED
    - Release resources at termination (detached) or retain (joinable)
- **Scheduling policy**
  - SCHED_OTHER: standard policy
  - SCHED_FIFO, SCHED_RR: real-time policy
- **Scheduling parameters**
  - Priority only
- **Inherit scheduling policy**
  - PTHREAD_INHERIT_SCHED, PTHREAD_EXPICIT_SCHED
- **Thread scheduling scope**
  - PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS
- **Special functions exist for getting/setting each attribute**
  - int pthread_attr_setstack_size(pthread_attr_t* attr, size_t stacksize)
Pthreads – detached thread

- `pthread_attr_setdetachstate`

```c
main()

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)

pthread_create(&th_id1, &attr, proc1, NULL)

proc1 {
    ...
}

pthread_create(&th_id2, &attr, proc2, NULL)

proc2 {
    ...
}

terminates!

terminates!
```
Exiting a Pthread

- **4 ways to exit threads**
  - Thread will naturally exit after starting thread function returns
  - Thread itself can exit by calling `pthread_exit()`
  - Other threads can terminate a thread by calling `pthread_cancel()`
    - A specified thread terminates when it reaches a cancelation point
  - Thread exits if the process that owns the thread exits

- **APIs**
  - `void pthread_exit (void *retval);`
  - `int pthread_cancel (pthread_t thread)`
Waiting a Pthread

```c
int pthread_join (  
    pthread_t thread, void **ptr);  
• Returns after a specified thread terminates  
• ptr stores return code of a terminating thread

main()
pthread_create(&th_id, NULL, proc1, &arg)
pthread_join(th_id, *status)

proc1(*arg)  
{  
    return (&status);  
}
```
Example: estimating Pi

- Estimating Pi using Monte Carlo method

Image credit: wikipedia.org
Example: Creation and Termination

```c
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi(void *);
....
main() {
    ...
pthread_t p_threads[MAX_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

```c
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double x, y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        x = (double)rand_r(&seed)/RAND_MAX;
        y = (double)rand_r(&seed)/RAND_MAX;
        if ( x * x + y * y < 1.0 )
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```
Pthreads – compilation

- Pthreads are supported by almost all compilers
  - GNU Compiler
    - `gcc -Wall -o hello hello.c -lpthread`
    - `-lxxx` : specifies which static library to link
    - `-Wall` : specifies to print out all types of warnings
False sharing (1)

- Unnecessary cache coherence packets (miss + invalidate) due to unintended cache line sharing

```c
int i, j, m, n;
double y[m];
...
for ( i=0; i<m; i++ )
    for ( j=0; j<n; j++ )
y[i] += f(i, j);
```

```c
int i, j, m, n;
double y[m];
...
for ( i=0; i<m/2; i++ )
    for ( j=0; j<n; j++ )
y[i] += f(i, j);
```

Core 0

Core 1
False sharing (2)

- Unnecessary cache coherence packets (miss + invalidate) due to unintended cache line sharing

```
int i, j, m, n;
double y[m];
...
for ( i=0; i<m/2; i++ )
    for ( j=0; j<n; j++ )
        y[i] += f(i, j);
for ( i=m/2; i<m; i++ )
    for ( j=0; j<n; j++ )
        y[i] += f(i, j);
```
Synchronization

- Accessing shared data
  - Example: two threads increase the same variable x

```c
int x = 0;
inc ()
{   x = x + 1; }
main()
{
    pthread_create(&th1, NULL, inc, NULL);
pthread_create(&th2, NULL, inc, NULL);

    pthread_join(th1, NULL);
pthread_join(th2, NULL);

    printf("x = %d\n", x);
}
```

Thread 1
- load r1 ← x
- add r1 ← r1, 1
- store r1 → x

Thread 2
- load r1 ← x
- add r1 ← r1, 1
- store r1 → x

Time
Critical Sections

- **Critical section**
  - Need to guarantee that one process (thread) can access a certain resource at a time
  - Implemented mechanism is known as "mutual exclusion"

- **Locks**
  - Simple mechanism for mutual exclusion
  - A lock can have only two values
    - 1 – a thread entered the critical section
    - 0 – no thread is in the critical section
  - Acquire the lock before entering the critical section (set to 1)
  - Release the lock after leaving the critical section (set to 0)
Acquiring Locks

- Simple C code is adequate?

```c
lock(lock_var) {
    if (lock_var == 0) // lock is free
        lock_var = 1;
}

unlock(lock_var) {
    lock_var = 0;
}
```

- Special atomic instruction should be used
  - Pthread library provides APIs
Pthreads - Lock

- Use a special mutex variable & APIs

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
/* or */ pthread_mutex_init(&lock, &attr);
pthread_mutex_lock( &lock );
    // critical section
pthread_mutex_unlock( &lock );
```

- `pthread_mutex_lock`
  - A thread will wait until it can acquire the lock

- `pthread_mutex_unlock`
  - If multiple threads are waiting, only one thread is selected to receive the lock
  - Only the thread that acquires the lock can unlock it
Mutex Attributes

- Specifying attribute of a mutex

```c
pthread_mutex_t lock;
Pthread_mutexattr_t attr;

pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_FAST_NP)
pthread_mutex_init(&lock, &attr)

pthread_mutex_lock( &lock );
    // critical section
pthread_mutex_unlock( &lock );
```

- Attributes
  - PTHREAD_MUTEX_FAST_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

Non-portable to other systems
Producer-Consumer Using Locks

```c
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);
        }
    }
}
```
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);
        }
        pthread_mutex_unlock(&task_queue_lock);
    }
    process_task(my_task);
}

Serialization

- Critical sections serialize the code execution
  - Too many or large critical sections can slow down the performance – sequential code may run faster
Alleviating Locking Overhead

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

- Reduce overhead by overlapping computation with waiting
  - Acquires lock if unlocked
  - Returns EBUSY if locked
### Condition Variables

- **Wait until a condition is satisfied**
  - A global variable is used to indicate condition (predicate value)

- **Three variables are linked all together**
  - mutex lock, condition variable, predicate

```c
thread1

action() {
    ...
    mutex_lock(&lock);
    while (predicate == 0) // test predicate
        cond_wait(&cond, &lock);
    mutex_unlock(&lock);
    // perform action
    ...
}

thread2

signal() {
    ...
    mutex_lock(&lock);
    predicate = 1; // set predicate
    cond_signal(&cond);
    mutex_unlock(&lock);
    ...
}
```

- When a thread waits using `cond_wait`, associated mutex is unlocked
- If a thread is signalled, returns after acquiring the mutex lock
Pthread Condition Variable API

- Initialize and destroy

\[
\begin{align*}
\text{int } & \text{pthread_cond_init}(\text{pthread_cond_t } *\text{cond}, \\
& \quad \text{const pthread_condattr_t } *\text{attr}); \\
\text{int } & \text{pthread_cond_destroy}(\text{pthread_cond_t } *\text{cond});
\end{align*}
\]

- Wait for a condition

\[
\begin{align*}
\text{int } & \text{pthread_cond_wait}(\text{pthread_cond_t } *\text{cond}, \\
& \quad \text{pthread_mutex_t } *\text{mutex}); \\
\text{int } & \text{pthread_cond_timedwait}(\text{pthread_cond_t } *\text{cond}, \\
& \quad \text{pthread_mutex_t } *\text{mutex}, \\
& \quad \text{const struct timespec } *\text{wtime});
\end{align*}
\]

- Signal one or all waiting threads

\[
\begin{align*}
\text{int } & \text{pthread_cond_signal}(\text{pthread_cond_t } *\text{cond}); \\
\text{int } & \text{pthread_cond_broadcast}(\text{pthread_cond_t } *\text{cond});
\end{align*}
\]
Producer-Consumer Using Condition Variables

```c
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-Consumer Using Condition Variables

```c
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);
    }
}
```
Producer-Consumer Using Condition Variables

```c
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_queueCond_lock);
        }
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queueCond_lock);
        process_task(my_task);
    }
}
```
Thread-Specific Data

- **Goal:** associate some data with a thread
- **Choices**
  - Pass data as arguments to functions
  - Store data in a shared array indexed by thread id
  - Use thread-specific data API
- **API**
  ```c
  int pthread_key_create(pthread_key_t *key, 
                          void (*destr_function)(void *));
  int pthread_key_delete(pthread_key_t key);
  int pthread_setspecific(pthread_key_t key, 
                          const void *pointer);
  void * pthread_getspecific(pthread_key_t key);
  ```
Example: Thread-Specific Data

/* Key for the thread-specific buffer */
static pthread_key_t buffer_key;

/* Initialize key, only once */
initialize_buffer() {
    ...
    pthread_key_create(&buffer_key, buffer_destroy);
    ...
}

/* Free each thread-specific buffer when associated thread terminates */
static void buffer_destroy(void * buf) {
    free(buf);
}
Example: Thread-Specific Data

/* Allocate a thread-specific buffer */
void buffer_alloc(void)
{
    pthread_setspecific(buffer_key, malloc(100));
}

/* Return the thread-specific buffer */
char * get_buffer(void)
{
    return (char *) pthread_getspecific(buffer_key);
}
Other Useful APIs in Pthread

- Read-write lock
- Spinlock
- Barrier

Caveat
- Not all Pthread libraries provide those
- One can implement those functions using mutexes, condition variables, and other variables
Read-Write Locks

- Useful for applications having a frequently read but infrequently written data structure
- Provide a critical section that
  - Multiple reader can be in the critical section simultaneously
    - Increasing concurrency of execution
  - One writer can be in the critical section at a time
    - Avoiding race condition
Pthread Read-Write Mutex API

- **Initialize and destroy a rwlock**
  
  ```c
  pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER
  pthread_rwlock_init(&rwlock, &attr)
  pthread_rwlock_destroy(&rwlock)
  ```

- **Read locking**
  
  ```c
  pthread_rwlock_rdlock(&rwlock)
  ```

- **Write locking**
  
  ```c
  pthread_rwlock_wrlock(&rwlock)
  ```

- **Unlocking**
  
  ```c
  pthread_rwlock_unlock(&rwlock)
  ```
Pthread Spinlock API

- **Pthread mutex is blocking-based lock**
  - Inefficient when a critical section is short
    - Due to management cost for blocking/waking-up a thread

- **Busy-waiting lock (e.g., spinlock)**
  - More efficient when a critical section is short
    - Spinning a few cycles is faster than blocking and waking up a thread

- **Pthread library provides spinlock-based mutual exclusion**

  ```c
  pthread_spinlock_t
  pthread_spin_init(pthread_spinlock_t* spinlock, int nr_shared)
  pthread_spin_lock(pthread_spinlock_t* spinlock)
  pthread_spin_unlock(pthread_spinlock_t* spinlock)
  ```
Pthread Barrier API

**Barrier**
- A execution synchronization point of threads
- Wait until all thread reach the point

**API**
- `pthread_barrier_init(pthread_barrier_t* barrier, pthread_barrierattr_t* attr, int value)`
  - Initialize a barrier
  - The integer value specifies the number of threads to synchronize
- `pthread_barrier_wait(pthread_barrier_t* barrier)`
  - Waits until the specified number of threads arrives at the barrier
References

- “COMP422: Parallel Computing” by Prof. John Mellor-Crummey at Rice Univ.