Lecture 9

Programming Shared Memory III

Synchronization Primitives; Condition Variables

Ceng471 *Parallel Computing* at December 23, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Computing the value of π I

- **Computing the value of π.**
- Based on generating random numbers in a *unit length square* and counting the number of points that fall within the largest circle inscribed in the square.
- Since the area of the circle (πr^2) is equal to $\pi/4$, and the area of the square is 1×1, the fraction of random points that fall in the circle should approach $\pi/4$.
- **Threaded strategy**:
- assigns a fixed number of points to each thread.
- Each thread generates these random points and keeps track of the number of points in the circle locally.
- After all threads finish execution, their counts are combined to compute the value of π (by calculating the fraction over all threads and multiplying by 4).

Computing the value of π II

```
void *compute_pi (void *s) {
 int seed, i, *hit_pointer;
 double rand_no_x, rand_no_y;
 int local_hits;

 hit_pointer = (int *) s;
 seed = *hit_pointer;
 local_hits = 0;
 for (i = 0; i < sample_points_per_thread; i++) {
  rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
  rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
  if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
       (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
        local_hits ++;
  seed *= i;
 }
 *hit_pointer = local_hits;
 pthread_exit(0);
}
```

The *arg* field is used to pass an integer id that is used as a seed for randomization.

Computing the value of π III

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
//cat /proc/sys/kernel/threads-max
void *compute_pi (void);
int total_hits, total_misses, hits[MAX_THREADS],
    sample_points, sample_points_per_thread,
    num_threads;

main() {
    int i;
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    double computed_pi;
    double time_start, time_end;
    struct timeval tv;
    struct timezone tz;

    pthread_attr_init (&attr);
    pthread_attr_setscope (&attr,
                   PTHREAD_SCOPE_SYSTEM);
    printf("Enter number of sample points: ");
    scanf("%d", &sample_points);
    printf("Enter number of threads: ");
    scanf("%d", &num_threads);

```

Computing the value of π IV

```
    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread=sample_points/num_threads;
    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];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}
```

Computing the value of π V

- For computing the value of π,
- First read in the desired number of threads, *num_threads*, and the desired number of sample points, *sample_points*.
- These points are divided equally among the threads.
- The program uses an array, **hits**, for assigning an integer id to each thread (this id is used as a seed for randomizing the random number generator).
- The same array is used to keep track of the number of hits (points inside the circle) encountered by each thread upon return.

Computing the value of π VI

- The program creates *num_threads* threads, each invoking the same function *compute_pi*, using the **pthread_create** function.
- Once the respective *compute_pi* threads have generated assigned number of random points and computed their hit ratios, the results must be combined to determine π.
- Once all threads have joined, the value of π is computed by multiplying the combined hit ratio by 4.0.
- The use of the function *rand_r* (instead of superior random number generators such as *drand48*).
- The reason for this is that many functions (including *rand* and *drand48*) are not **reentrant**.

Producer-consumer work queues I

- **Producer-consumer work queues**
- A common use of mutex-locks is in establishing a producer-consumer relationship between threads.
- The **producer creates tasks and inserts** them into a work-queue.
- The **consumer threads pick up tasks** from the task queue and execute them.
- Consider that the task queue can hold only one task.
- In a general case, the task queue may be longer but is typically of bounded size.
- A simple (and incorrect) threaded program would associate a producer thread with creating a task
- and placing it in a **shared data structure**
- and the consumer threads with picking up tasks from this shared data structure and executing them.

Producer-consumer work queues II

- However, this simple version does not account for the following possibilities:

1. The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
2. The consumer threads must not pick up tasks until there is something present in the shared data structure.
3. Individual consumer threads should pick up tasks one at a time.

- To implement this, we can use a variable called *task_available*.
    - If this variable is 0, consumer threads must wait, but the producer thread can insert tasks into the shared data structure *task_queue*.
    - If *task_available* is equal to 1, the producer thread must wait to insert the task into the shared data structure but one of the consumer threads can pick up the task available.

Producer-consumer work queues III

All of these operations on the variable *task_available* should be protected by **mutex-locks** to ensure that only one thread is executing test-update on it.

```
pthread_mutex_t task_queue_lock;
int task_available;

/* other shared data structures here */

main() {
     /* declarations and initializations */
     task_available = 0;
     pthread_init();
     pthread_mutex_init(&task_queue_lock, NULL);
/* create and join producer and consumer threads */
}

```

Producer-consumer work queues IV

```
void *producer(void *producer_thread_data) {
    int inserted;
    struct task my_task;
    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 work queues V

```
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);
    }
}
```

Producer-consumer work queues VI

- The *create_task* and *process_task* functions are left **outside the critical region**, making the critical section as small as possible.
- but *insert_into_queue* and *extract_from_queue* functions are left **inside the critical region**.
- Inside because if the lock is **relinquished** after updating *task_available* but not inserting or extracting the task,
- other threads may gain access to the shared data structure while the insertion or extraction is in progress, resulting in errors.

Producer-consumer work queues VII

- For producer-consumer work queues
- The producer thread creates a task and waits for space on the queue.
- This is indicated by the variable *task_available* being 0.
- The test and update of this variable as well as insertion and extraction from the shared queue are protected by a mutex called *task_queue_lock*.
- Once space is available on the task queue, the recently created task is inserted into the task queue and the availability of the task is signaled by setting *task_available* to 1.
- Within the producer thread, the fact that the recently created task has been inserted into the queue is signaled by the variable *inserted* being set to 1, which allows the producer to produce the next task.
- Irrespective of whether a recently created task is successfully inserted into the queue or not, the lock is relinquished.

Producer-consumer work queues VIII

- This allows consumer threads to pick up work from the queue in case there is work on the queue to begin with.
- If the lock is not relinquished, threads would deadlock since a consumer would not be able to get the lock to pick up the task and the producer would not be able to insert its task into the task queue.
- The consumer thread waits for a task to become available and executes it when available.
- As was the case with the producer thread, the consumer relinquishes the lock in each iteration of the while loop to allow the producer to insert work into the queue if there was none.

Synchronization Primitives; Condition Variables I

- Indiscriminate use of locks can **result in idling overhead** from blocked threads.
- While the function **pthread_mutex_trylock** removes this overhead, it introduces the overhead of polling for availability of locks.
- For example, if the producer-consumer example is rewritten using *pthread_mutex_trylock* instead of *pthread_mutex_lock*,
- the producer and consumer threads would have to periodically poll for availability of lock (and subsequently availability of buffer space or tasks on queue).
- A natural solution to this problem is to **suspend the execution** of the polling thread until space becomes available.
- An **interrupt driven mechanism** as opposed to a **polled mechanism**.

Synchronization Primitives; Condition Variables II

- The availability of space is signaled by the thread that holding the space.
- The functionality to accomplish this is provided by a **condition variable**.
- A condition variable is a data object used for synchronizing threads and always used in conjunction with a mutex lock.
- While mutexes implement synchronization by **controlling thread access to data**,
- condition variables allow threads to synchronize based upon **the actual value of data**.
- This variable allows a thread to block itself until specified data reaches a predefined state.

Synchronization Primitives; Condition Variables III

- **pthread_cond_wait**

```
int pthread_cond_wait(pthread_cond_t *cond,
        pthread_mutex_t *mutex);
```

- A thread locks this mutex and tests the predicate defined on the shared variable;
- if the predicate is not true, the thread waits on the condition variable associated with the predicate using this function.
- A call to this function blocks the execution of the thread until it receives a signal from another thread or is interrupted by an OS signal.
- In addition to blocking the thread, the **pthread_cond_wait** function **releases the lock on mutex**.
- This is important because otherwise no other thread will be able to work on the shared variable and the predicate would never be satisfied.

Synchronization Primitives; Condition Variables IV

- **pthread_cond_signal**

```
int pthread_cond_signal(pthread_cond_t
                                   *cond);
```

- When the condition is signaled, **pthread_cond_signal**, one of these threads in the queue is unblocked,
- and when the mutex becomes available, it is handed to this thread (and the thread becomes runnable).
- When the thread is released on a signal, it waits to reacquire the lock on mutex before resuming execution.
- It is convenient to think of each condition variable as being associated with a queue.
- Threads performing a condition wait on the variable relinquish their lock and enter the queue.

Synchronization Primitives; Condition Variables V

- **pthread_cond_init** & **pthread_cond_destroy**

```
int pthread_cond_init(pthread_cond_t *cond,
     const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Function calls for initializing and destroying condition variables.
- Condition variables must be declared with type *pthread_cond_t*, and must be initialized before they can be used.
- There are two ways to initialize a condition variable:

1. Statically, when it is declared. For example: *pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;*
2. Dynamically, with the **pthread_cond_init()** routine.

Synchronization Primitives; Condition Variables VI

- The function **pthread_cond_init** initializes a condition variable (pointed to by *cond*).
- The ID of the created condition variable is returned to the calling thread through the condition parameter.
- This method permits setting condition variable object attributes, *attr*. (*NULL* assigns default attributes)
- If at some point in a program a condition variable is no longer required, it can be discarded using the function **pthread_cond_destroy**.

Synchronization Primitives; Condition Variables VII

<table>
<tr><td colspan="2">Main Thread
◦ Declare and initialize global data/variables which require synchronization (such as "count")
◦ Declare and initialize a condition variable object
◦ Declare and initialize an associated mutex
◦ Create threads A and B to do work</td></tr>
<tr><td>Thread A
◦ Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
◦ Lock associated mutex and check value of a global variable
◦ Call <code>pthread_cond_wait()</code> to perform a blocking wait for signal from Thread-B. Note that a call to <code>pthread_cond_wait()</code> automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
◦ When signalled, wake up. Mutex is automatically and atomically locked.
◦ Explicitly unlock mutex
◦ Continue</td><td>Thread B
◦ Do work
◦ Lock associated mutex
◦ Change the value of the global variable that Thread-A is waiting upon.
◦ Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
◦ Unlock mutex.
◦ Continue</td></tr>
<tr><td colspan="2">Main Thread
Join / Continue</td></tr>
</table>

Figure: A representative sequence for using condition variables.

Synchronization Primitives; Condition Variables VIII

- When a thread performs a condition wait, it takes itself off the runnable list consequently, it does **not use any CPU cycles** until it is woken up.
- This is in contrast to a mutex lock which **consumes** CPU cycles as it polls for the lock.
- **pthread_cond_broadcast**.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- In some cases, it may be beneficial to wake all threads that are waiting on the condition variable as opposed to a single thread.
- An example of this is in the producer-consumer scenario with large work queues and multiple tasks being inserted into the work queue on each insertion cycle.
- Another example is in the implementation of barriers.

Synchronization Primitives; Condition Variables IX

- **pthread_cond_timedwait**,

```
int pthread_cond_timedwait(pthread_cond_t *cond,
     pthread_mutex_t *mutex,
     const struct timespec *abstime);
```

- It is often useful to build time-outs into condition waits.
- Using the function a thread can perform a wait on a condition variable until a specified time expires.
- At this point, the thread wakes up by itself if it does not receive a signal or a broadcast.
- If the absolute time *abstime* specified expires before a signal or broadcast is received, the function returns an error message.
- It also reacquires the lock on mutex when it becomes available.