1 Programming Shared Memory

1.1 Why Threads?

Threaded programming models offer significant advantages over message-passing programming models along with some disadvantages as well.

- **Software Portability;** Threaded applications can be developed on serial machines and run on parallel machines without any changes. This ability to migrate programs between diverse architectural platforms is a very significant advantage of threaded APIs.

- **Latency Hiding;** One of the major overheads in programs (both serial and parallel) is the access latency for memory access, I/O, and communication. By allowing multiple threads to execute on the same processor, threaded APIs enable this latency to be hidden. In effect, while one thread is waiting for a communication operation, other threads can utilize the CPU, thus masking associated overhead.

- **Scheduling and Load Balancing;** While writing shared address space parallel programs, a programmer must express concurrency in a way that minimizes overheads of remote interaction and idling. While in many *structured* applications the task of allocating equal work to processors is easily accomplished, in *unstructured* and *dynamic* applications (such as game playing and discrete optimization) this task is more difficult. Threaded APIs allow the programmer to specify a large number of concurrent tasks and support system-level dynamic mapping of tasks to processors with a view to minimizing idling overheads. By providing this support at the system level, threaded APIs rid the programmer of the burden of explicit scheduling and load balancing.

- *Ease of Programming, Widespread Use* Due to the mentioned advantages, threaded programs are significantly easier to write than corresponding programs using message passing APIs. With widespread acceptance of the POSIX thread API, development tools for POSIX threads are more widely available and stable. These issues are important from the program development and software engineering aspects.

A number of vendors provide vendor-specific thread APIs. The IEEE specifies a standard 1003.1c-1995, POSIX API. Also referred to as *Pthreads*, POSIX has emerged as the standard threads API, supported by most vendors. The concepts themselves are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

1.2 Thread Basics: Creation and Termination

- A simple threaded program for computing the value of π.
- The method we use here is 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 $\pi/4$.
- A simple threaded strategy for generating the value of π assigns a fixed number of points to each thread. Each thread generates these random points and keeps track of the number of points that land 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).
- **pthread_create**

```
1   #include <pthread.h>
2   int
3   pthread_create (
4       pthread_t   *thread_handle,
5       const pthread_attr_t   *attribute,
6       void *   (*\textit{thread\_function})(void *),
7       void   *arg);
```

  - The **pthread_create** function creates a single thread that corresponds to the invocation of the function *thread_function* (and any other functions called by *thread_function*).
  - On successful creation of a thread, a unique identifier is associated with the thread and assigned to the location pointed to by *thread_handle*.
  - The thread has the attributes described by the *attribute* argument. When this argument is *NULL*, a thread with default attributes is created.
  - The *arg* field specifies a pointer to the argument to function *thread_function*. This argument is typically used to pass the workspace and other *thread-specific* data to a thread. In the **compute_pi** example, it is used to pass an integer id that is used as a seed for randomization.

- The *thread_handle* variable is written before the function **pthread_create** returns; and the new thread is ready for execution as soon as it is created.
- If the thread is scheduled on the same processor, the new thread may, in fact, *preempt* its creator. This is important to note because all thread initialization procedures must be completed before creating the thread. Otherwise, errors may result based on thread scheduling. This is a very common class of errors caused by race conditions for data access that shows itself in some execution instances, but not in others.
- On successful creation of a thread, **pthread_create** returns 0; else it returns an error code.

- 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.
  - The program creates *num_threads* threads, each invoking the 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.
  - The main program must wait for the threads to run to completion. This is done using the function **pthread_join** which suspends execution of the calling thread until the specified thread terminates.

```
1   int
2   pthread_join (
3       pthread_t thread,
4       void **ptr);
```

  - A call to this function waits for the termination of the thread whose id is given by thread.

– On a successful call to **pthread_join**, the value passed to **pthread_exit** is returned in the location pointed to by *ptr*. On successful completion, **pthread_join** returns 0, else it returns an error-code.

– Once all threads have joined, the value of π is computed by multiplying the combined hit ratio by 4.0.

```
1   #include <pthread.h>
2   #include <stdlib.h>
3
4   #define MAX_THREADS 512
5   void *compute_pi (void *);
6
7   int total_hits, total_misses, hits[MAX_THREADS],
8       sample_points, sample_points_per_thread, num_threads;
9
10  main() {
11      int i;
12      pthread_t p_threads[MAX_THREADS];
13      pthread_attr_t attr;
14      double computed_pi;
15      double time_start, time_end;
16      struct timeval tv;
17      struct timezone tz;
18
19      pthread_attr_init (&attr);
20      pthread_attr_setscope (&attr,PTHREAD_SCOPE_SYSTEM);
21      printf("Enter number of sample points: ");
22      scanf("%d", &sample_points);
23      printf("Enter number of threads: ");
24      scanf("%d", &num_threads);
25
26      gettimeofday(&tv, &tz);
27      time_start = (double)tv.tv_sec +
28                   (double)tv.tv_usec / 1000000.0;
29
30      total_hits = 0;
31      sample_points_per_thread = sample_points / num_threads;
32      for (i=0; i< num_threads; i++) {
33          hits[i] = i;
34          pthread_create(&p_threads[i], &attr, compute_pi,
35              (void *) &hits[i]);
36      }
37      for (i=0; i< num_threads; i++) {
```

```
38          pthread_join(p_threads[i], NULL);
39          total_hits += hits[i];
40      }
41      computed_pi = 4.0*(double) total_hits /
42          ((double)(sample_points));
43      gettimeofday(&tv, &tz);
44      time_end = (double)tv.tv_sec +
45                 (double)tv.tv_usec / 1000000.0;
46
47      printf("Computed PI = %lf\n", computed_pi);
48      printf(" %lf\n", time_end - time_start);
49  }
50
51  void *compute_pi (void *s) {
52      int seed, i, *hit_pointer;
53      double rand_no_x, rand_no_y;
54      int local_hits;
55
56      hit_pointer = (int *) s;
57      seed = *hit_pointer;
58      local_hits = 0;
59      for (i = 0; i < sample_points_per_thread; i++) {
60          rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
61          rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
62          if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
63              (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
64              local_hits ++;
65          seed *= i;
66      }
67      *hit_pointer = local_hits;
68      pthread_exit(0);
69  }
```

- 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**. Reentrant functions are those that can be safely called when another instance has been suspended in the middle of its invocation.

- It is easy to see why all thread functions must be reentrant because a thread can be preempted in the middle of its execution. If another thread starts executing the same function at this point, a non-reentrant function might not work as desired.

1.3 Synchronization Primitives in Pthreads

While communication is implicit in shared-address-space programming, much of the effort associated with writing correct threaded programs is spent on synchronizing concurrent threads with respect to their data accesses or scheduling.

1.3.1 Mutual Exclusion for Shared Variables

- Using **pthread_create** and **pthread_join** calls, we can create concurrent tasks. These tasks work together to manipulate data and accomplish a given task. When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.
- Consider the following code fragment being executed by multiple threads. The variable *my_cost* is thread-local and *best_cost* is a global variable shared by all threads.

```
1   /* each thread tries to update variable best_cost as follows */
2   if (my_cost < best_cost)
3       best_cost = my_cost;
```

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
  - Assume that there are two threads, the initial value of *best_cost* is 100, and the values of *my_cost* are 50 and 75 at threads t1 and t2, respectively.
  - If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement. Depending on which thread executes first, the value of *best_cost* at the end could be either 50 or 75.
  - There are two problems here: the first is the non-deterministic nature of the result; second, and more importantly, the value 75 of *best_cost* is inconsistent in the sense that no serialization of the two threads can possibly yield this result.
  - This is an undesirable situation, sometimes also referred to as a race condition (so called because the result of the computation depends on the race between competing threads).

- The aforementioned situation occurred because the test-and-update operation illustrated above is an atomic operation; i.e., the operation should not be broken into sub-operations.

- Furthermore, the code corresponds to a critical segment; i.e., a segment that must be executed by only one thread at any time. Many statements that seem atomic in higher level languages such as C may in fact be non-atomic; for example, a statement of the form $global_count+ = 5$ may comprise several assembler instructions and therefore must be handled carefully.

- Threaded APIs provide support for implementing critical sections and atomic operations using mutex-locks (mutual exclusion locks).

  - Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock.
  - A lock is an atomic operation generally associated with a piece of code that manipulates shared data. To access the shared data, a thread must first try to acquire a mutex-lock.
  - If the mutex-lock is already locked, the process trying to acquire the lock is blocked. This is because a locked mutex-lock implies that there is another thread currently in the critical section and that no other thread must be allowed in.
  - When a thread leaves a critical section, it must unlock the mutex-lock so that other threads can enter the critical section.
  - All mutex-locks must be initialized to the unlocked state at the beginning of the program.

- The function **pthread_mutex_lock** can be used to attempt a lock on a mutex-lock.

```
1   int
2   pthread_mutex_lock (
3       pthread_mutex_t *mutex_lock);
```

- A call to this function attempts a lock on the mutex-lock *mutex_lock*. (The data type of a *mutex_lock* is predefined to be *pthread_mutex_t*.)

- If the mutex-lock is already locked, the calling thread blocks; otherwise the mutex-lock is locked and the calling thread returns. A successful return from the function returns a value 0. Other values indicate error conditions such as deadlocks.

- On leaving a critical section, a thread must unlock the mutex-lock associated with the section. If it does not do so, no other thread will be able to enter this section, typically resulting in a deadlock. The Pthreads function **pthread_mutex_unlock** is used to unlock a mutex-lock.

```
1   int
2   pthread_mutex_unlock (
3       pthread_mutex_t *mutex_lock);
```

- On calling this function, in the case of a normal mutex-lock, the lock is relinquished and one of the blocked threads is scheduled to enter the critical section.

- The specific thread is determined by the scheduling policy. There are other types of locks (other than normal locks).

- If a programmer attempts a **pthread_mutex_unlock** on a previously unlocked mutex or one that is locked by another thread, the effect is undefined.

- We need one more function before we can start using mutex-locks, namely, a function to initialize a mutex-lock to its unlocked state. The Pthreads function for this is **pthread_mutex_init**.

```
1   int
2   pthread_mutex_init (
3       pthread_mutex_t *mutex_lock,
4       const pthread_mutexattr_t *lock_attr);
```

  This function initializes the mutex-lock *mutex_lock* to an unlocked state. The attributes of the mutex-lock are specified by *lock_attr*. If this argument is set to *NULL*, the default mutex-lock attributes are used (normal mutex-lock).

- **Computing the minimum entry in a list of integers**
  - A threaded program to compute the minimum of a list of integers. The list is partitioned equally among the threads.
    - The size of each thread's partition is stored in the variable *partial_list_size*.
    - The pointer to the start of each thread's partial list is passed to it as the pointer *list_ptr*.

```
1   #include <pthread.h>
2   void *find_min(void *list_ptr);
3   pthread_mutex_t minimum_value_lock;
4   int minimum_value, partial_list_size;
5
6   main() {
7       /* declare and initialize data structures and list */
8       minimum_value = MIN_INT;
9       pthread_init();
10      pthread_mutex_init(&minimum_value_lock, NULL);
11
12      /* initialize lists, list_ptr, and partial_list_size */
13      /* create and join threads here */
14  }
15
16  void *find_min(void *list_ptr) {
17      int *partial_list_pointer, my_min, i;
18      my_min = MIN_INT;
19      partial_list_pointer = (int *) list_ptr;
20      for (i = 0; i < partial_list_size; i++)
21          if (partial_list_pointer[i] < my_min)
22              my_min = partial_list_pointer[i];
23      /* lock the mutex associated with minimum_value and
24      update the variable as required */
25      pthread_mutex_lock(&minimum_value_lock);
26      if (my_min < minimum_value)
27          minimum_value = my_min;
28      /* and unlock the mutex */
29      pthread_mutex_unlock(&minimum_value_lock);
30      pthread_exit(0);
31  }
```

- In this example, the test-update operation for *minimum_value* is protected by the mutex-lock *minimum_value_lock*.
- Threads execute *pthread_mutex_lock* to gain exclusive access to the variable *minimum_value*. Once this access is gained, the value is updated as required, and the lock subsequently released. Since at any point of time, only one thread can hold a lock, only one thread can test-update the variable.

- **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.
- Let us consider a simple instance of this paradigm in which 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. However, this simple version does not account for the following possibilities:
  * The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
  * The consumer threads must not pick up tasks until there is something present in the shared data structure.
  * 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.
- 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.

```
1   pthread_mutex_t task_queue_lock;
2   int task_available;
3
4   /* other shared data structures here */
5
6   main() {
7       /* declarations and initializations */
8       task_available = 0;
9       pthread_init();
10      pthread_mutex_init(&task_queue_lock, NULL);
11     /* create and join producer and consumer threads */
12  }
```

```
13
14  void *producer(void *producer_thread_data) {
15      int inserted;
16      struct task my_task;
17      while (!done()) {
18          inserted = 0;
19          create_task(&my_task);
20          while (inserted == 0) {
21              pthread_mutex_lock(&task_queue_lock);
22              if (task_available == 0) {
23                  insert_into_queue(my_task);
24                  task_available = 1;
25                  inserted = 1;
26              }
27              pthread_mutex_unlock(&task_queue_lock);
28          }
29      }
30  }
31
32  void *consumer(void *consumer_thread_data) {
33      int extracted;
34      struct task my_task;
35      /* local data structure declarations */
36      while (!done()) {
37          extracted = 0;
38          while (extracted == 0) {
39              pthread_mutex_lock(&task_queue_lock);
40              if (task_available == 1) {
41                  extract_from_queue(&my_task);
42                  task_available = 0;
43                  extracted = 1;
44              }
45              pthread_mutex_unlock(&task_queue_lock);
46          }
47          process_task(my_task);
48      }
49  }
```

– In this example, 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. 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.
- *Overheads of Locking;*
    - Locks represent serialization points since critical sections must be executed by threads one after the other.
    - Encapsulating large segments of the program within locks can, therefore, lead to significant performance degradation. It is important to minimize the size of critical sections.
    - For instance, in the above example, the *create_task* and *process_task* functions are left outside the critical region, but *insert_into_queue* and *extract_from_queue* functions are left inside the critical region.
    - The former is left out in the interest of making the critical section as small as possible.
    - The *insert_into_queue* and *extract_from_queue* functions are left 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.

  - It is therefore important to handle critical sections and shared data structures with extreme care.

- Facilitating Locking Overheads
  - It is often possible to reduce the idling overhead associated with locks using an alternate function, *pthread_mutex_trylock*.
  - This function attempts a lock on *mutex_lock*. If the lock is successful, the function returns a zero. If it is already locked by another thread, instead of blocking the thread execution, it returns a value *EBUSY*. This allows the thread to do other work and to poll the mutex for a lock.
  - Furthermore, *pthread_mutex_trylock* is typically much faster than *pthread_mutex_lock* on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.

```
1   int
2   pthread_mutex_trylock (
3       pthread_mutex_t *mutex_lock);
```

- **Finding k matches in a list**
  - We consider the example of finding k matches to a query item in a given list. The list is partitioned equally among the threads. Assuming that the list has n entries, each of the p threads is responsible for searching n/p entries of the list.

```
1   void *find_entries(void *start_pointer) {
2
3       /* This is the thread function */
4
5       struct database_record *next_record;
6       int count;
7       current_pointer = start_pointer;
8       do {
9           next_record = find_next_entry(current_pointer);
10          count = output_record(next_record);
11      } while (count < requested_number_of_records);
12  }
13
14  int output_record(struct database_record *record_ptr) {
15      int count;
```

```
16      pthread_mutex_lock(&output_count_lock);
17      output_count ++;
18      count = output_count;
19      pthread_mutex_unlock(&output_count_lock);
20
21      if (count <= requested_number_of_records)
22          print_record(record_ptr);
23      return (count);
24  }
```

- This program segment finds an entry in its part of the database, updates the global count and then finds the next entry.
- If the time for a lock-update count-unlock cycle is t1 and the time to find an entry is t2, then the total time for satisfying the query is $(t1 + t2) * n_{max}$, where n_{max} is the maximum number of entries found by any thread.
- If t1 and t2 are comparable, then locking leads to considerable overhead. This locking overhead can be handled by using the function *pthread_mutex_trylock*.
- Each thread now finds the next entry and tries to acquire the lock and update count. If another thread already has the lock, the record is inserted into a local list and the thread proceeds to find other matches.
- When it finally gets the lock, it inserts all entries found locally thus far into the list (provided the number does not exceed the desired number of entries).

```
1   int output_record(struct database_record *record_ptr) {
2       int count;
3       int lock_status;
4       lock_status = pthread_mutex_trylock(&output_count_lock);
5       if (lock_status == EBUSY) {
6           insert_into_local_list(record_ptr);
7           return(0);
8       }
9       else {
10          count = output_count;
11          output_count += number_on_local_list + 1;
12          pthread_mutex_unlock(&output_count_lock);
13          print_records(record_ptr, local_list,
14                requested_number_of_records - count);
15          return(count + number_on_local_list + 1);
16      }
17  }
```

- Notice that if the lock for updating the global count is not available, the function inserts the current record into a local list and returns.
- If the lock is available, it increments the global count by the number of records on the local list, and then by one (for the current record). It then unlocks the associated lock and proceeds to print as many records as are required using the function *print_records*.
- The time for execution of this version is less than the time for the first one on two counts: First, as mentioned, the time for executing a *pthread_mutex_trylock* is typically much smaller than that for a *pthread_mutex_lock*. Second, since multiple records may be inserted on each lock, the number of locking operations is also reduced.

1.3.2 Condition Variables for Synchronization

- As it is noted in the previous section, indiscriminate use of locks can result in idling overhead from blocked threads. While the function *pthread_mutex_trylock* alleviates 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 producer until space becomes available (an interrupt driven mechanism as opposed to a polled mechanism).
- The availability of space is signaled by the consumer thread that consumes the task. The functionality to accomplish this is provided by a *condition variable*.
- A condition variable is a data object used for synchronizing threads. This variable allows a thread to block itself until specified data reaches a predefined state.
- In the producer-consumer case, the shared variable *task_available* must become 1 before the consumer threads can be signaled.
- The boolean condition *task_available == 1* is referred to as a predicate. A condition variable is associated with this predicate.
- When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.

- A single condition variable may be associated with more than one predicate. However, this is strongly discouraged since it makes the program difficult to debug.

- A condition variable always has a mutex associated with it. A thread locks this mutex and tests the predicate defined on the shared variable (in this case: *task_available*); if the predicate is not true, the thread waits on the condition variable associated with the predicate using the function **pthread_cond_wait**.

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

- 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 *task_available* and the predicate would never be satisfied.

- 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.

- When the condition is signaled (using **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).

- In the context of our producer-consumer example, the producer thread produces the task and, since the lock on mutex has been relinquished (by waiting consumers), it can insert its task on the queue and set *task_available* to 1 after locking mutex. Since the predicate has now been satisfied, the producer must wake up one of the consumer threads by signaling it.

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

- The function unblocks at least one thread that is currently waiting on the condition variable *cond*. The producer then relinquishes its lock on mutex by explicitly calling *pthread_mutex_unlock*, allowing one of the blocked consumer threads to consume the task.

- Before our producer-consumer example is rewriten using condition variables, we need to introduce two more function calls for initializing and destroying condition variables, *pthread_cond_init* and *pthread_cond_destroy*; respectively.

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

- The function *pthread_cond_init* initializes a condition variable (pointed to by *cond*) whose attributes are defined in the attribute object *attr*. Setting this pointer to *NULL* assigns default attributes for condition variables.

- If at some point in a program a condition variable is no longer required, it can be discarded using the function *pthread_cond_destroy*.

- These functions for manipulating condition variables enable us to rewrite our producer-consumer segment.

- **Producer-consumer using condition variables**

  - Condition variables can be used to block execution of the producer thread when the work queue is full and the consumer thread when the work queue is empty.
  - We use two condition variables *cond_queue_empty* and *cond_queue_full* for specifying empty and full queues respectively.
  - The predicate associated with *cond_queue_empty* is *task_available == 0*, and *cond_queue_full* is asserted when *task_available == 1*.
  - The producer queue locks the mutex *task_queue_cond_lock* associated with the shared variable *task_available*.
  - It checks to see if *task_available* is 0 (i.e., queue is empty). If this is the case, the producer inserts the task into the work queue and signals any waiting consumer threads to wake up by signaling the condition variable *cond_queue_full*. It subsequently proceeds to create additional tasks.
  - If *task_available* is 1 (i.e., queue is full), the producer performs a condition wait on the condition variable *cond_queue_empty* (i.e., it waits for the queue to become empty).

– The reason for implicitly releasing the lock on *task_queue_cond_lock* becomes clear at this point. If the lock is not released, no consumer will be able to consume the task and the queue would never be empty. At this point, the producer thread is blocked.

– Since the lock is available to the consumer, the thread can consume the task and signal the condition variable *cond_queue_empty* when the task has been taken off the work queue.

– The consumer thread locks the mutex *task_queue_cond_lock* to check if the shared variable *task_available* is 1. If not, it performs a condition wait on *cond_queue_full*. (Note that this signal is generated from the producer when a task is inserted into the work queue.)

– If there is a task available, the consumer takes it off the work queue and signals the producer. In this way, the producer and consumer threads operate by signaling each other. It is easy to see that this mode of operation is similar to an interrupt-based operation as opposed to a polling-based operation of *pthread_mutex_trylock*.

```
1   pthread_cond_t cond_queue_empty, cond_queue_full;
2   pthread_mutex_t task_queue_cond_lock;
3   int task_available;
4
5   /* other data structures here */
6
7   main() {
8       /* declarations and initializations */
9       task_available = 0;
10      pthread_init();
11      pthread_cond_init(&cond_queue_empty, NULL);
12      pthread_cond_init(&cond_queue_full, NULL);
13      pthread_mutex_init(&task_queue_cond_lock, NULL);
14      /* create and join producer and consumer threads */
15  }
16
17  void *producer(void *producer_thread_data) {
18      int inserted;
19      while (!done()) {
20          create_task();
21          pthread_mutex_lock(&task_queue_cond_lock);
22          while (task_available == 1)
23              pthread_cond_wait(&cond_queue_empty,
24                  &task_queue_cond_lock);
25          insert_into_queue();
26          task_available = 1;
```

```
27          pthread_cond_signal(&cond_queue_full);
28          pthread_mutex_unlock(&task_queue_cond_lock);
29      }
30  }
31
32  void *consumer(void *consumer_thread_data) {
33      while (!done()) {
34          pthread_mutex_lock(&task_queue_cond_lock);
35          while (task_available == 0)
36              pthread_cond_wait(&cond_queue_full,
37                  &task_queue_cond_lock);
38          my_task = extract_from_queue();
39          task_available = 0;
40          pthread_cond_signal(&cond_queue_empty);
41          pthread_mutex_unlock(&task_queue_cond_lock);
42          process_task(my_task);
43      }
44  }
```

- An important point to note about this program segment is that the predicate associated with a condition variable is checked in a loop.
- One might expect that when *cond_queue_full* is asserted, the value of *task_available* must be 1. However, it is a good practice to check for the condition in a loop because the thread might be woken up due to other reasons (such as an OS signal).
- In other cases, when the condition variable is signaled using a condition broadcast (signaling all waiting threads instead of just one), one of the threads that got the lock earlier might invalidate the condition.
- In the example of multiple producers and multiple consumers, a task available on the work queue might be consumed by one of the other consumers.
- 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.
- In the above example, each task could be consumed by only one consumer thread. Therefore, we choose to signal one blocked thread at a time.

– In some other computations, it may be beneficial to wake all threads that are waiting on the condition variable as opposed to a single thread. This can be done using the function *pthread_cond_broadcast*.

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

– 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 of the use of *pthread_cond_broadcast* is in the implementation of barriers.

– It is often useful to build time-outs into condition waits. Using the function **pthread_cond_timedwait**, 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.

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

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.