PRODUCERS AND CONSUMERS

1 Queues:

- A Queue is a list abstract data type in which new elements are inserted at the “rear” of the queue and elements are removed from the “front” of the queue. A queue can thus be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket. The elements of the list are the customers. Newcustomers go to the end or “rear” of the line, and the next customer to check out isthe customer standing at the “front” of the line.
- When a new entry is added to the rear of a queue, we sometimes say that the entry has been “enqueued,” and when an entry is removed from the front of a queue, we sometimes say that the entry has been “dequeued.”
- Queues occur frequently in computer science. For example, if we have a number of processes, each of which wants to store some data on a hard drive, then a natural way to insure that only one process writes to the disk at a time is to have the processes form a queue, that is, the first process that wants to write gets access to the drive first, the second process gets access to the drive next, and so on.
- A queue is also a natural data structure to use in many multithreaded applications. For example, suppose we have several “producer” threads and several “consumer” threads. The producer threads might “produce” requests for data from a server—for example, current stock prices—while the consumer threads might “consume” the request by finding or generating the requested data—the current stock prices.
- The producer threads could enqueue the requested prices, and the consumer threads could dequeue them. In this example, the process wouldn’t be completed until the consumer threads had given the requested data to the producer threads.

2. Message-passing

- Another natural application would be implementing message-passing on a sharedmemorysystem. Each thread could have a shared message queue, and when onethread wanted to “send a message” to another thread, it could enqueue the messagein the destination thread’s queue. A thread could receive a message by dequeuing themessage at the head of its message queue.

- A simple message-passing program in which eachthread generates random integer “messages” and random destinations for the messages.After creating the message, the thread enqueues the message in the appropriatemessage queue. After sending a message, a thread checks its queue to see if it hasreceived a message. If it has, it dequeues the first message in its queue and prints itout. Each thread alternates between sending and trying to receive messages.

- The user specify the number of messages each thread should send. When a thread isdone sending messages, it receives messages until all the threads are done, at whichpoint all the threads quit. Pseudocode for each thread might look something like this:

```
for (sent msgs = 0; sent msgs< send max; sent msgs++)
{
        Send msg();
        Try receive();
}
while (!Done())
        Try receive();
```

3 Sending messages:

- Accessing a message queue to ***enqueue*** a message is prob a critical section. We need to have a variable that keeps track ofthe rear of the queue. For example, if we use a singly linked list with the tail of the list corresponding to the rear of the queue, then, in order to efficiently enqueue, we wouldlike to store a pointer to the rear.

- When we ***enqueue*** a new message, we'll need tocheck and update the rear pointer. If two threads try to do this simultaneously, wemay lose a message that has been enqueued by one of the threads. The results of the two operations will conflict, and hence enqueueinga message will form a critical section.
- Pseudocode for the Send msg() function might look something like this:

```
mesg = random();
dest = random() % thread count;
# pragmaomp critical
Enqueue(queue, dest, my rank, mesg);
```

This allows a thread to send a message to itself.

4 Receiving messages

- The synchronization issues for receiving a message are a little different. Only the owner of the queue (that is, the destination thread) will ***dequeue*** from a given messagequeue.

- As long as we ***dequeue*** one message at a time, if there are at least two messages in the queue, a call to ***Dequeue*** can't possibly conflict with any calls to Enqueue, so if we keep track of the size of the queue, we can avoid any synchronization (for example, critical directives), as long as there are at least two messages.
- If we store two variables, enqueued and dequeued, then the number of messages in the queue is

```
queue_size = enqueued-dequeued
```

and the only thread that will update ***dequeued*** is the owner of the queue.

- One thread can update enqueued at the same time that another thread is using it tocompute queue size. To see this, let's suppose thread q is computing queue_size. Itwill either get the old value of enqueued or the new value. It may therefore computea queue_sizeof 0 or 1 when queue_sizeshould actually be 1 or 2, respectively, butin our program this will only cause a modest delay.

- Thread q will try again later ifqueue size is 0 when it should be 1, and it will execute the critical section directiveunnecessarily if queue size is 1 when it should be 2.Thus, we can implement Try_ receive as follows:

```
Queue_size = enqueued-dequeued;
if (queue size == 0) return;
else if (queue_size == 1)
#       pragmaomp critical
        Dequeue(queue, &src, &mesg);
else
        Dequeue(queue, &src, &mesg);
        Print_message(src, mesg);
```

5 Termination detection

- The implementation of the Done function specifies the termination. The following "obvious" implementation will have problems:

```
Queue_size = enqueued- dequeued;
if (queue_size == 0)
        return TRUE;
else
        return FALSE;
```

- If thread **u** executes this code, it's entirely possible that some thread—call it threadv—will send a message to thread u after u has computed queue size = 0. After thread **u** computes queue size = 0, it will terminate and the message sent bythread **v** will never be received.

- In our program, after each thread has completed the for loop, it won't send any new messages. Thus, if we add a counter done sending, and each threadincrements this after completing its for loop, then we can implement Done asfollows:

```
queue_size= enqueued- dequeued;
if (queue_size== 0 &&done_sending == thread_count)
return TRUE;
else
return FALSE;
```

6. The atomic directive

- After completing its sends, each thread increments done sending before proceeding to its final loop of receives. Clearly, incrementing ***done_sending***is a critical section, and we could protect it with a ***critical*** directive.

- OpenMP provides a potentially higher performance directive: the atomic directive:

  #pragmaomp atomic

- Unlike the ***critical*** directive, it can only protect critical sections that consist of a single C assignment statement. Further, the statement must have one of the followingforms:

  x <op>= <expression>;
  x++;
  ++x;
  x--;
  --x;

  Here <op> can be one of the binary operators
  +,*,-, /, &, ˆ, j ,<<, or >>.

  It's also important to remember that <expression> must not reference x.
- It should be noted that only the load and store of x are guaranteed to be protected.For example, in the code

  # pragmaomp atomic
  x += y++;

  a thread's update to x will be completed before any other thread can begin updatingx. However, the update to y may be unprotected and the results may beunpredictable.

- The idea behind the atomic directive is that many processors provide a special load-modify-store instruction, and a critical section that only does a loadmodify-
  store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general criticalsections.

7.Critical sections and locks

- Since enforcing mutual exclusion among threads serializes execution, this default behavior of OpenMP—treating all critical blocks as part of one composite critical section—can be highly detrimental to our program's performance. OpenMP does provide the option of adding a name to a critical directive:

  # pragmaomp critical(name)

- When we do this, two blocks protected with ***critical*** directives with different names can be executed simultaneously. The names are set during compilation, and we want a different critical section for each thread's queue. Weneed to set the names at run-time, and in our setting, when we want to allow simultaneous access to the same block of code by threads accessing different queues, the named***critical*** directive isn't sufficient.

- The alternative is to use ***locks***. A ***lock*** consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

- The use of a ***lock*** can be roughly described by the following pseudocode:

  /*Executed by one thread */
  Initialize the lock data structure;
  . . .
  /*Executed by multiple threads*/
  Attempt to lock or set the lock data structure;
  Critical section;
  Unlock or unset the lock data structure;
  . . .
  /*Executed by one thread */
  Destroy the lock data structure;

- The ***lock*** data structure is shared among the threads that will execute the critical section. One of the threads (e.g., the master thread) will initialize the ***lock***, and whenall the threads are done using the lock, one of the threads should destroy it.
- Before a thread enters the critical section, it attempts to set or lock the ***lock*** data structure by calling the lock function. If no other thread is executing code in the critical section, it obtains the lock and proceeds into the critical section past the callto the ***lock*** function. When the thread finishes the code in the critical section, it callsan ***unlock*** function, which relinquishes or unsets the lock and allows another threadto obtain the ***lock***.

- OpenMP has two types of locks: ***simple locks*** and ***nested locks***. A ***simple lock*** can only be set once before it is unset, while a ***nested lock*** can be set multiple times bythe same thread before it is unset. The type of an OpenMP simple lock is omp_lock_t,and the simple lock functions that we'll be using are

```
void omp_init_lock(omp_lock_t*    lock_p   /* out */);
void omp_set_lock(omp_lock_t*     lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*   lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p   /* in/out */);
```

  - The type and the functions are specified in omp.h. The first function initializes the lock so that it's unlocked, that is, no thread owns the lock. The second function attempts to set the lock. If it succeeds, the calling thread

proceeds; if it fails, the calling thread blocks until the lock becomes available. The third function unsets the lock so another thread can obtain it. The fourth function makes the lock uninitialized.

8. Using locks in the message-passing program

- If we want to insure mutual exclusion in each individual message queue, not in a particular block of source code, we need to include a data member with type ***omp_lock_t*** in our queue struct, we can simply call ***omp_set_lock*** each time we want to insure exclusive access to a message queue. So the code

```
# pragma omp critical
/* q_p = msg_queues[dest] */
Enqueue(q_p, my_rank, mesg);
```

can be replaced with

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Similarly, the code

```
# pragma omp critical
/* q_p = msg_queues[my_rank] */
Dequeue(q_p, &src, &mesg);
```

can be replaced with

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

- Now when a thread tries to send or receive a message, it can only be blocked by a thread attempting to access the same message queue, since different message queues have different locks. In our original implementation, only one thread could send at a time, regardless of the destination.

9. Some Caveats(Warnings)

1. You shouldn't mix the different types of mutual exclusion for a single critical section. For example, suppose a program contains the following two segments.

```
# pragmaomp atomic
      x += f(y);
```

```
# pragma omp critical
      x = g(x);
```

- The update to x on the right doesn't have the form required by the ***atomic*** directive, so the programmer used a ***critical*** directive. However, the ***critical*** directive won't exclude the action executed by the atomic block, and it's possible that the results will be incorrect. The programmer needs to either rewrite the function g so that its use can have the form required by the atomic directive, or she needs to protect both blocks with a ***critical*** directive.

2. There is no guarantee of fairness in mutual exclusion constructs. This means that it's possible that a thread can be blocked forever in waiting for access to a critical section. For example, in the code

```
            while(1) {
            . . .
#           pragmaomp critical
            x = g(my rank);
                    . . .
            }
```

it's possible that, for example, thread 1 can block forever waiting to execute x = g(my_rank), while the other threads repeatedly execute the assignment. Ofcourse, this wouldn't be an issue if the loop terminated. Also note that many implementations give threads access to the critical section in the order in which
they reach it, and for these implementations, this won't be an issue.

3. It can be dangerous to "nest" mutual exclusion constructs. As an example, suppose a program contains the following two segments.

```
# pragmaomp critical
        y = f(x);
        . . .
        double f(double x) {
# pragmaomp critical
        z = g(x); /* z is shared */
        . . .
}
```

- This is guaranteed to ***deadlock***. When a thread attempts to enter the second critical section, it will block forever. If thread u is executing code in the first critical block, no thread can execute code in the second block. In particular, thread u can't execute this code. However, if thread ***u*** is blocked waiting to enter the second critical block, then it will never leave the first, and it will stay blocked forever.
- In this example, we can solve the problem by using named critical sections. That is, we could rewrite the code as

```
# pragmaomp critical(one)
        y = f(x);
        . . .
        double f(double x) {
# pragmaomp critical(two)
        z = g(x); /_ z is global _/
        . . .
}
```

- However, it's not difficult to come up with examples when naming won't help. For example, if a program has two named critical sections—say ***one*** and ***two***— and threads can attempt to enter the critical sections in different orders, then deadlock can occur. For example, suppose thread ***u*** enters ***one*** at the same time that thread ***v*** enters two and ***u*** then attempts to enter ***two*** while ***v*** attempts to enter***one.***