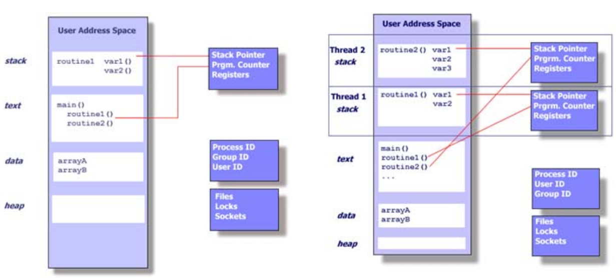
Programming Shared-memory Machines

Some slides adapted from Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar ``Introduction to Parallel Computing'', Addison Wesley, 2003.

Overview

- Thread Basics
- The POSIX Thread API
- Synchronization primitives in Pthreads
  - locks
  - try-locks
- Deadlocks and how to avoid them
- Composite synchronization constructs
- Controlling Thread and Synchronization Attributes
- OpenMP: a Standard for Directive Based Parallel Programming

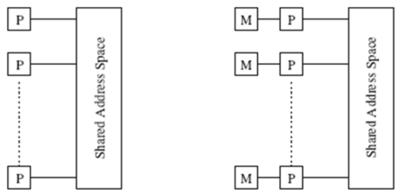
Process vs Threads



Thread Basics

- Each thread has its own stack, SP, PC, registers, etc.
- Threads share global variables and heap.
- Caveat: writing programs in which shared space is treated as a "flat" address space may give poor performance
  - Locality is just as important in shared-memory machines as it is in distributed-memory machines

Thread Basics



- The logical machine model of a thread-based programming paradigm.

The POSIX Thread API

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here 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.

Thread Basics: Creation and Termination

- Creating Pthreads:

```
#include <pthread.h>
int pthread_create (
   pthread_t *thread_handle,
   const pthread_attr_t *attribute,
   void * (*thread_function)(void *),
   void *arg);
```

- Thread is created and it starts to execute thread_function with parameter arg
- Thread handle: name for thread

Terminating threads

- Thread terminated when:
  - o it returns from its starting routine, or
  - o it makes a call to pthread_exit()
- Main thread
  - exits with pthread_exit(): other threads will continue to execute
  - Otherwise: other threads automatically terminated
- Cleanup:
  - pthread_exit() routine does not close files
  - any files opened inside the thread will remain open after the thread is terminated.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
  printf("\n%d: Hello World!\n", threadid);
  pthread_exit(NULL);
 }

int main(int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0;t<NUM_THREADS;t++){
       printf("Creating thread %d\n", t);
       rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
       if (rc){ printf("ERROR; return code from pthread_create() is %d\n", rc);
              exit(-1);
             }
  }
  pthread_exit(NULL);
}
```

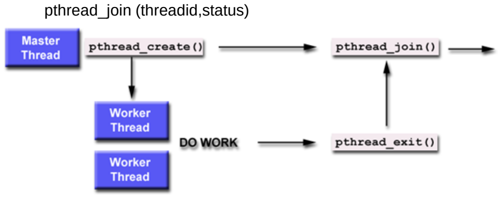
Output

```
Creating thread 0
Creating thread 1

0: Hello World!

1: Hello World!
Creating thread 2
Creating thread 3

2: Hello World!

3: Hello World!
Creating thread 4

4: Hello World!
```

Synchronizing threads

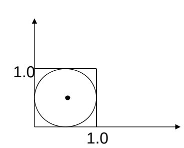
•"Joining" is one way to synchronize threads (not used very often)

pthread_join (threadid,status)



•The pthread_join() function blocks the calling thread until the specified thread terminates.
•The programmer can obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

Threads: Example 2



- Area of circle = pi*0.25
- Area of square = 1
- So if we shoot randomly into square, probability of hitting circle is pi*0.25
- Estimating value of pi:
  - generate a large number of random values inside the unit square
  - see what fraction of them fall inside circle and multiply by 4
- Simple example of Monte Carlo method: estimate some value by repeated sampling of some space
- Monte Carlo method can be easily parallelized provided each parallel thread generates independent random numbers

Threads: Example2

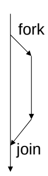
```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}
```

Threads: Example2 (contd.)

```
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);
}
```

Synchronizing threads

- Style of computing shown in Example 2 is sometimes called fork-join parallelism



- This style of parallel execution in which threads only synchronize at the end is quite rare
- Usually, threads need to synchronize during their execution

Need for synchronization

- Two common scenarios:
  - Mutual exclusion
    - Shared “resource” such as variable or device
    - Only one thread at a time can access resource
    - Critical section: portion of code that should be executed by only thread at a time
  - Producer-consumer
    - One thread (producer) generates a sequence of values
    - Another thread (consumer) reads these values
    - Values are communicated by writing them into a shared buffer
    - Producer must block if buffer is full
    - Consumer must block if buffer is empty

Need for Mutual Exclusion

- When multiple threads attempt to manipulate the same data item, the results can often be incorrect if proper care is not taken to synchronize them.
- Consider:

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

- 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.
- Depending on the schedule of the threads, the value of `best_cost` could be 50 or 75!
  - Thread 1 reads best_cost (100)
  - Thread 2 reads best_cost (100)
  - Thread 1 writes best_cost (50)
  - Thread 2 writes best_cost (75)
- The value 75 does not "seem right" because it would not arise in a sequential execution of the same algorithm

General problem

- The code in the previous example is called a critical section
  - Several threads may try to execute code in critical section but only one should succeed at a time
- Problem arises very often when writing threaded code
  - Thread A want to read and write one or more variables in critical section
  - While it is doing that, other threads should be excluded from accessing those variables
- Solution: lock
  - Threads compete for "acquiring" lock
  - Pthreads implementation guarantees that only one thread will succeed in acquiring lock
  - Successful thread enters critical section, performs its activity
  - When critical section is done, lock is "released"

Mutex in Pthreads

- The Pthreads API provides the following functions for handling mutex-locks:
  - Lock creation

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

  - Acquiring lock

```
int pthread_mutex_lock (
    pthread_mutex_t *mutex_lock);
```

  - Releasing lock

```
int pthread_mutex_unlock (
    pthread_mutex_t *mutex_lock);
```

Implementation (see next time)

- Lock is implemented by
  - variable with two states: *available* or *not_available*
  - queue that can hold ids of threads waiting for the lock
- Lock acquire:
  - If state of lock is *available*, its state is changed to *not_available*, and control returns to application program
  - If state of lock is *not_available*, thread-id is queued up at the lock, and control returns to application program only when lock is acquired by that thread
  - Key invariant: once a thread tries to acquire lock, control returns to thread only after lock has been awarded to that thread
- Lock release:
  - next thread in queue is informed it has acquired lock, and it can proceed
- "Fairness": any thread that wants to acquire a lock can succeed ultimately even if other threads want to acquire the lock an unbounded number of times

Correct Mutual Exclusion

- We can now write our previously incorrect critical section as:

```
pthread_mutex_t minimum_value_lock;
...
main() {
    ....
    pthread_mutex_init(&minimum_value_lock, NULL);
    ....
}
void *find_min(void *list_ptr) {
    ....
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
    minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
}
```

critical section

Critical sections

- For performance, it is important to keep critical sections as small as possible
- While one thread is within critical section, all others threads that want to enter the critical section are blocked
- It is up to the programmer to ensure that locks are used correctly to protect variables in critical sections

Thread A	Thread B	Thread C
lock(l)	lock(l)	
x:= ..x..	x:= ..x..	x: = …x
unlock(l)	unlock(l)	

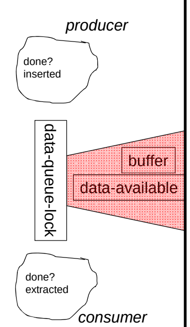
This program may fail to execute correctly because programmer forgot to use locks in Thread C

Producer-Consumer Using Locks

- Two threads
  - Producer: produces data
  - Consumer: consumes data
- Shared buffer is used to communicate data from producer to consumer
  - Buffer can contain one data value (in this example)
  - Flag is associated with buffer to indicate buffer has valid data
- Consumer must not read data from buffer unless there is valid data
- Producer must not overwrite data in buffer before it is read by consumer

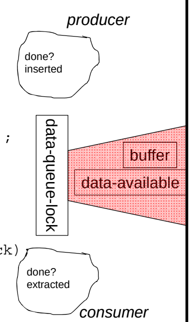
Producer-Consumer Using Locks

```
pthread_mutex_t data_queue_lock;
int data_available; //1 if buffer is full
...
main() {
    ....
    data_available = 0;
    pthread_mutex_init(&data_queue_lock, NULL);
    ....
}
void *producer(void *producer_thread_data) {
    ....
    while (!done()) {
        create_data(&my_data);
        inserted = 0;
        while (inserted == 0) {
            pthread_mutex_lock(&data_queue_lock);
            if (data_available == 0) {
                insert_data(my_data);
                data_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&data_queue_lock);
        }
    }
}
```



Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct data my_data;
    /* local data structure declarations */
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&data_queue_lock);
            if (data_available == 1) {
                extract_data(&my_data);
                data_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&data_queue_lock);
        }
        process_data(my_data);
    }
}
```



Types of Mutexes

- Pthreads supports three types of mutexes - normal, recursive, and error-check.
- A normal mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A recursive mutex allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
- An error check mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).
- The type of the mutex can be set in the attributes object before it is passed at time of initialization.

Reducing lock overhead

- Another kind of lock: trylock.

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

- If lock is available, acquire it; otherwise, return a "busy" error code (EBUSY)
- 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.

Alleviating Locking Overhead (Example)

```
/* Finding k matches in a list */
void *find_entries(void *start_pointer) {
    /* This is the thread function */
    struct database_record *next_record;
    int count;
    current_pointer = start_pointer;
    do {
        next_record = find_next_entry(current_pointer);
        count = output_record(next_record);
    } while (count < requested_number_of_records);
}
int output_record(struct database_record *record_ptr) {
    int count;
    pthread_mutex_lock(&output_count_lock);
    output_count ++;
    count = output_count;
    pthread_mutex_unlock(&output_count_lock);
    if (count <= requested_number_of_records)
        print_record(record_ptr);
    return (count);
}
```

Alleviating Locking Overhead (Example)

```
/* rewritten output_record function */
int output_record(struct database_record
   *record_ptr) {
    int count;
    int lock_status;
    lock_status=pthread_mutex_trylock(&output_count_lock);
    if (lock_status == EBUSY) {
        insert_into_local_list(record_ptr);
        return(0);
    }
    else {
        count = output_count;
        output_count += number_on_local_list + 1;
        pthread_mutex_unlock(&output_count_lock);
        print_records(record_ptr, local_list,
            requested_number_of_records - count);
        return(count + number_on_local_list + 1);
    }
}
```

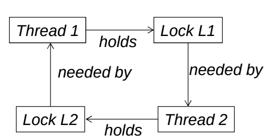
Problems with locks

- Locks are most dangerous when a thread needs to acquire multiple locks before releasing locks
- Two main problems:
  - deadlock
  - livelock
- Deadlock:
  - Threads A and B need locks L1 and l2
  - Thread A acquires L1 and wants L2
  - Thread B acquires L2 and wants L1
  - In general, there will be a cycle of threads in which each thread holds some locks and is waiting for locks held by other threads in the cycle
- Livelock:
  - may arise in some solutions to deadlock

Deadlock

- Code snippet shows example of possible deadlock
- Subtle point:
  - deadlock may happen in some executions and not in others!
- "Deadly embrace": Dijkstra
- How do we ensure deadlocks cannot occur?

Thread 1:	*Thread 2:*
…	…
lock(L1);	lock(L2);
lock(L2);	lock(L1);
….	…



Deadlock: four conditions

- Mutual exclusion:
  - thread has exclusive control over resource it acquires
- Hold-and-wait:
  - thread does not release resource it holds if it is waiting for another resource
- No pre-emption:
  - No external agency forces a thread to release resources if thread is waiting for another resource
- Circular wait:
  - There is a cycle of threads such that each thread holds one or more resources needed by the next thread in the cycle

You prevent deadlocks by ensuring that one or more of these conditions cannot arise in your program.

Prevent circular wait

- Assign a logical total order to locks
  - (eg) name them L1,L2,L3,…
- Ensure that threads will never try to acquire a lower numbered lock while holding a higher numbered lock
  - (eg) if thread owns L3, it can try to acquire L4, L5, L6,… but it cannot try to acquire locks L1 or L2 (unless it already owns them and locks are re-entrant)
- Useful software engineering principle when you have control over the entire code base and you know what locks are required where
- However
  - easy to make mistakes
  - tension with encapsulation:
    - requires detailed knowledge of entire code base

Prevent hold-and-wait

- Try to acquire all locks atomically
- One implementation:
  - single global lock to get permission to acquire locks you need
- Problem:
  - not scalable
  - conflicts with modularity and encapsulation
- You might encounter a hidden version of this problem if thread has to enter the kernel to perform some function like storage allocation
  - kernel lock is like the global-lock in our example

```
…
lock(global-lock);
lock(l1);
lock(l2);
unlock(global-lock);
…
```

Self-preemption

- Coding discipline:
  - Use only try-locks
  - If a thread cannot acquire a lock while it is holding other locks, it releases all locks it holds and tries again
  - Variation: OS or some other agency steps in and preempts a thread
- Problems:
  - Encapsulation
  - Livelock: threads can keep on acquiring and releasing locks without making progress because no thread ever gets all the locks it needs
  - One solution to livelock: (Ethernet) backoff: thread does not retry until some randomly chosen amount of time has passed

```
loop:
    //start of lock acquires
    ….
    if (trylock(Lj) == EBUSY) {
    //unlock all locks you hold
    goto loop;
    }
    ….
endloop:

//compute with resources
//release locks
```

Lock-free synchronization

- Use more powerful hardware instructions that perform atomic computations on variables
  - no notion of "holding" resources like locks
  - these atomic computations are enough for many applications but in general, they need to be composed and this can be tricky
- Example: CompareAndSwap instruction

```
int CompareAndSwap(int *address, int expected, int new)
        if (*address == expected) {
            *address = new;
            return SUCCESS;
            }
        else return FAIL;

void AtomicIncrement(int *value; int amount) {
    do {int old = *value;
        } while (CompareAndSwap(value,old,old+amount) == FAIL)
```

Composite Synchronization Constructs

- By design, Pthreads provide support for a basic set of operations.
- Higher level constructs can be built using basic synchronization constructs.
- We discuss two such constructs - read-write locks and barriers.

Read-Write Locks

- In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.
- A read lock is granted when there are other threads that may already have read locks.
- If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait.
- If there are multiple threads requesting a write lock, they must perform a condition wait.
- With this description, we can design functions for read locks `mylib_rwlock_rlock`, write locks `mylib_rwlock_wlock`, and unlocking `mylib_rwlock_unlock`.

Read-Write Locks

- The lock data type `mylib_rwlock_t` holds the following:
  - a count of the number of readers,
  - the writer (a 0/1 integer specifying whether a writer is present),
  - a condition variable `readers_proceed` that is signaled when readers can proceed,
  - a condition variable `writer_proceed` that is signaled when one of the writers can proceed,
  - a count `pending_writers` of pending writers, and
  - a mutex `read_write_lock` associated with the shared data structure

Read-Write Locks

```
typedef struct {
   int readers;
   int writer;
   pthread_cond_t readers_proceed;
   pthread_cond_t writer_proceed;
   int pending_writers;
   pthread_mutex_t read_write_lock;
} mylib_rwlock_t;
void mylib_rwlock_init (mylib_rwlock_t *l) {
   l -> readers = l -> writer = l -> pending_writers
     = 0;
   pthread_mutex_init(&(l -> read_write_lock),
     NULL);
   pthread_cond_init(&(l -> readers_proceed), NULL);
   pthread_cond_init(&(l -> writer_proceed), NULL);
}
```

Read-Write Locks

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
   /* if there is a write lock or pending writers, perform
     condition wait.. else increment count of readers and grant
     read lock */
   pthread_mutex_lock(&(l -> read_write_lock));
   while ((l -> pending_writers > 0) || (l -> writer > 0))
      pthread_cond_wait(&(l -> readers_proceed),
      &(l -> read_write_lock));
   l -> readers ++;
   pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Read-Write Locks

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
   /* if there are readers or writers, increment pending
     writers count and wait. On being woken, decrement
     pending writers count and increment writer count */

   pthread_mutex_lock(&(l -> read_write_lock));
   l -> pendingwriters ++;
   while ((l -> writer > 0) || (l -> readers > 0)) {
      pthread_cond_wait(&(l -> writer_proceed),
        &(l -> read_write_lock));
      }
   l -> pending_writers --;
   l -> writer ++;
   pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Read-Write Locks

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
/* if there is a write lock then unlock, else if there are read
   locks, decrement count of read locks. If the count is 0 and
   there is a pending writer, let it through, else if there are
   pending readers, let them all go through */
pthread_mutex_lock(&(l -> read_write_lock));
if (l -> writer > 0)
    l -> writer = 0;
else if (l -> readers > 0)
    l -> readers --;

if ((l -> readers == 0) && (l -> pending_writers > 0))
    pthread_cond_signal(&(l -> writer_proceed));
else //no pending writers
    pthread_cond_broadcast(&(l -> readers_proceed));

pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Barriers

- As in MPI, a barrier holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex and a condition variable.
- A single integer is used to keep track of the number of threads that have reached the barrier.
- If the count is less than the total number of threads, the threads execute a condition wait.
- The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.

Barriers

```
typedef struct {
   pthread_mutex_t count_lock;
   pthread_cond_t ok_to_proceed;
   int count;
} mylib_barrier_t;
void mylib_init_barrier(mylib_barrier_t *b) {
   b -> count = 0;
   pthread_mutex_init(&(b -> count_lock), NULL);
   pthread_cond_init(&(b -> ok_to_proceed), NULL);
}
```

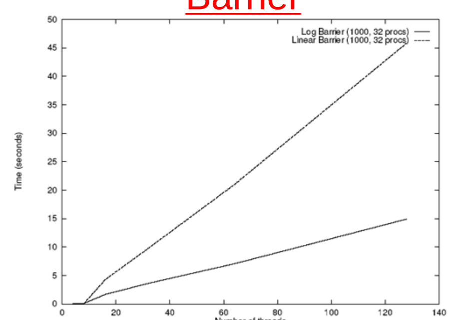
Barriers

```
void mylib_barrier (mylib_barrier_t *b, int
  num_threads) {
   pthread_mutex_lock(&(b -> count_lock));
   b -> count ++;
   if (b -> count == num_threads) {
      b -> count = 0;
      pthread_cond_broadcast(&(b -> ok_to_proceed));
   }
   else
      while (pthread_cond_wait(&(b -> ok_to_proceed),
         &(b -> count_lock)) != 0);
   pthread_mutex_unlock(&(b -> count_lock));
}
```

Barriers

- The barrier described above is called a linear barrier.
- The trivial lower bound on execution time of this function is therefore ***O(n)*** for ***n*** threads.
- This implementation of a barrier can be speeded up using multiple barrier variables organized in a tree.
- We use ***n/2*** condition variable-mutex pairs for implementing a barrier for ***n*** threads.
- At the lowest level, threads are paired up and each pair of threads shares a single condition variable-mutex pair.
- Once both threads arrive, one of the two moves on, the other one waits.
- This process repeats up the tree.
- This is also called a log barrier and its runtime grows as ***O(log p)***.

Barrier



- Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

Condition Variables

- Condition variables are another construct for more efficient synchronization: permit a thread to be woken up when some predicate on the data is satisifed
- Example: one thread produces a sequence of data items, and consumer thread must wait till there are more than n items in buffer
- Busy waiting is inefficient
  - Better to let waiting thread sleep and get notified when predicate is satisifed
  - Solution: condition variables
- Basic operations using condition variables
  - Thread can wait on condition variable: intuitively, thread blocks until some other thread signals that condition variable
  - Thread can signal condition variable: release one thread waiting on condition variable
  - Condition variables are not boolean variables!
- Correct operation of condition variables requires an associated mutex as we will see later

Condition Variable Constructs

- Pthreads provides the following functions for condition variables:

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

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

Locks associated with condition variables

- Correct operation with condition variable requires an associated lock
  - Wait and signal must be performed while holding lock
- Problem:
  - If thread A holds lock, calls wait on a condition variable, and then goes to sleep, how does thread B acquire lock to signal this condition variable?
- Solution:
  - When thread A calls wait and goes to sleep, pthreads implementation automatically releases associated lock
  - When thread A needs to be woken up in response to signal, pthreads implementation tries to reacquire lock and returns control to application program only after lock has been reacquired
  - Signal and lock reacquire are separate events, so it is good practice to re-check that data predicate after control returns from wait

    => Use a loop around wait (shown in examples)

Producer-Consumer Using Condition Variables

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t data_queue_cond_lock;
int data_available;
/* other data structures here */
main() {
   /* declarations and initializations */
   data_available = 0;
   pthread_init();
   pthread_cond_init(&cond_queue_empty, NULL);
   pthread_cond_init(&cond_queue_full, NULL);
   pthread_mutex_init(&data_queue_cond_lock, NULL);
   /* create and join producer and consumer threads */
}
```

Producer-Consumer Using Condition Variables

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

Producer-Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
    pthread_mutex_lock(&data_queue_cond_lock);
    while (data_available == 0)
        pthread_cond_wait(&cond_queue_full,
            &data_queue_cond_lock);
    my_data = extract_from_queue();
    data_available = 0;
    pthread_cond_signal(&cond_queue_empty);
    pthread_mutex_unlock(&data_queue_cond_lock);
    process_data(my_data);
    }
}
```

Controlling Thread and Synchronization Attributes

- The Pthreads API allows a programmer to change the default attributes of entities using *attributes objects*.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- Use `pthread_attr_init` to create an attributes object.
- Individual properties associated with the attributes object can be changed using the following functions:
  `pthread_attr_setdetachstate`,
  `pthread_attr_setguardsize_np`,
  `pthread_attr_setstacksize`,
  `pthread_attr_setinheritsched`,
  `pthread_attr_setschedpolicy`, and
  `pthread_attr_setschedparam`

Attributes Objects for Mutexes

- Initialize the attrributes object using function: `pthread_mutexattr_init`.
- The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object.

```
pthread_mutexattr_settype_np (
pthread_mutexattr_t *attr,
int type);
```

- Here, `type` specifies the type of the mutex and can take one of:
  - `PTHREAD_MUTEX_NORMAL_NP`
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`

Types of threads

- Thread implementations:
  - User-level threads:
    - Implemented by user-level runtime library
    - OS is unaware of threads
    - Portable, thread scheduling can be tuned to application requirements
    - Problem: cannot leverage multiprocessors, entire process blocks when one thread blocks
  - Kernel-level threads:
    - OS is aware of each thread and schedules them
    - Thread operations are performed by OS
    - Can leverage multiprocessors
    - Problem: higher overhead, usually not quite as portable
  - Hybrid-level threads: Solaris
    - OS provides some number of kernel level threads, and each of these can create multiple user-level threads
    - Problem: complexity

OpenMP: a Standard for Directive Based Parallel Programming

- OpenMP is a directive-based API that can be used with FORTRAN, C, and C++ for programming shared address space machines.
- OpenMP directives provide support for concurrency, synchronization, and data handling while obviating the need for explicitly setting up mutexes, condition variables, data scope, and initialization.

OpenMP Programming Model

- OpenMP directives in C and C++ are based on the `#pragma` compiler directives.
- A directive consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

- OpenMP programs execute serially until they encounter the `parallel` directive, which creates a group of threads.

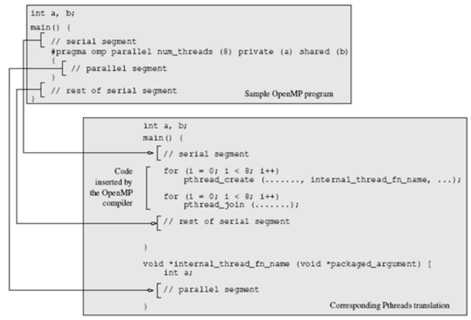
```
#pragma omp parallel [clause list]
/* structured block */
```

- The main thread that encounters the `parallel` directive becomes the *master* of this group of threads and is assigned the thread id 0 within the group.

OpenMP Programming Model

- The clause list is used to specify conditional parallelization, number of threads, and data handling.
  - **Conditional Parallelization:** The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads.
  - **Degree of Concurrency:** The clause `num_threads(integer expression)` specifies the number of threads that are created.
  - **Data Handling:** The clause `private (variable list)` indicates variables local to each thread. The clause `firstprivate (variable list)` is similar to the `private`, except values of variables are initialized to corresponding values before the parallel directive. The clause `shared (variable list)` indicates that variables are shared across all the threads.

OpenMP Programming Model



- A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

OpenMP Programming Model

```
#pragma omp parallel if (is_parallel== 1) num_threads(8) \
  private (a) shared (b) firstprivate(c) {
 /* structured block */
}
```

- If the value of the variable is_parallel equals one, eight threads are created.
- Each of these threads gets private copies of variables a and c, and shares a single value of variable b.
- The value of each copy of c is initialized to the value of c before the parallel directive.
- The default state of a variable is specified by the clause default (shared) or default (none).

Reduction Clause in OpenMP

- The `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the `reduction` clause is `reduction (operator: variable list).`
- The variables in the list are implicitly specified as being private to threads.
- The `operator` can be one of `+, *, -, &, |, ^, &&`, and `||`.

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

OpenMP Programming: Example

```
/* ******************************************************
An OpenMP version of a threaded program to compute PI.
***************************************************** */
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 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)
            sum ++;
    }
}
```

Specifying Concurrent Tasks in OpenMP

- The `parallel` directive can be used in conjunction with other directives to specify concurrency across iterations and tasks.
- OpenMP provides two directives - `for` and `sections` - to specify concurrent iterations and tasks.
- The `for` directive is used to split parallel iteration spaces across threads. The general form of a for directive is as follows:

```
#pragma omp for [clause list]
    /* for loop */
```

- The clauses that can be used in this context are: `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, `nowait`, and `ordered`.

Specifying Concurrent Tasks in OpenMP: Example

```
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; 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)
            sum ++;
    }
}
```

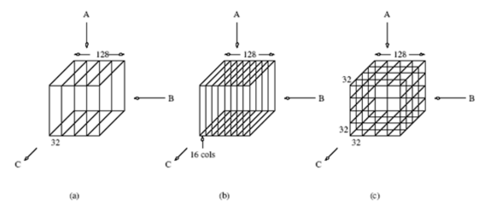
Assigning Iterations to Threads

- The `schedule` clause of the `for` directive deals with the assignment of iterations to threads.
- The general form of the `schedule` directive is `schedule(scheduling_class[, parameter])`.
- OpenMP supports four scheduling classes: `static`, `dynamic`, `guided`, and `runtime`.

Assigning Iterations to Threads: Example

```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
    num_threads(4)
    #pragma omp for schedule(static)
    for (i = 0; i < dim; i++) {
        for (j = 0; j < dim; j++) {
            c(i,j) = 0;
            for (k = 0; k < dim; k++) {
                c(i,j) += a(i, k) * b(k, j);
            }
        }
    }
```

Assigning Iterations to Threads: Example



- Three different schedules using the static scheduling class of OpenMP.

Parallel For Loops

- Often, it is desirable to have a sequence of `for`-directives within a parallel construct that do not execute an implicit barrier at the end of each `for` directive.
- OpenMP provides a clause - `nowait`, which can be used with a for directive.

Parallel For Loops: Example

```
#pragma omp parallel
{
    #pragma omp for nowait
        for (i = 0; i < nmax; i++)
            if (isEqual(name, current_list[i])
                processCurrentName(name);
    #pragma omp for
        for (i = 0; i < mmax; i++)
            if (isEqual(name, past_list[i])
                processPastName(name);
}
```

The `sections` Directive

- OpenMP supports non-iterative parallel task assignment using the `sections` directive.
- The general form of the `sections` directive is as follows:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

The `sections` Directive: Example

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Nesting `parallel` Directives

- Nested parallelism can be enabled using the `OMP_NESTED` environment variable.
- If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.

Synchronization Constructs in OpenMP

- OpenMP provides a variety of synchronization constructs:

```
#pragma omp barrier
#pragma omp single [clause list]
    structured block
#pragma omp master
    structured block
#pragma omp critical [(name)]
    structured block
#pragma omp ordered
    structured block
```

OpenMP Library Functions

- In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

```
/* thread and processor count */
void omp_set_num_threads (int
  num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

- In addition, all lock routines also have a nested lock counterpart
- for recursive mutexes.

Environment Variables in OpenMP

- `OMP_NUM_THREADS`: This environment variable specifies the default number of threads created upon entering a parallel region.
- `OMP_SET_DYNAMIC`: Determines if the number of threads can be dynamically changed.
- `OMP_NESTED`: Turns on nested parallelism.
- `OMP_SCHEDULE`: Scheduling of for-loops if the clause specifies runtime

Explicit Threads versus Directive Based Programming

- Directives layered on top of threads facilitate a variety of thread-related tasks.
- A programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.
- There are some drawbacks to using directives as well.
- An artifact of explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention.
- Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations.
- Finally, since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs are easier to find.