Programming with Threads

- Overview
- Example: Compute π
- Threads – mutexes, condition variables
- Example: Producer-consumer
- Example: Barrier

Threads

- A thread is an independent stream of instruction that can be scheduled to run as such by the operating system
- Threads exist within a process and use process resources
  - A process needs a process id, environment, working directory, program instructions, registers, stack, heap, file descriptors, signal actions, shared libraries, inter-process communication tools (queues, pipes, semaphores, shared memory)
- A thread duplicates only essential resources such as program counter, stack pointer, registers, etc.
- All threads in a process can access and modify the process address space
- Threads are “lightweight” compared to a process, and can be scheduled very fast

Advantages

- Software portability
- Latency hiding
- Scheduling and load balancing
- Serial performance
- Parallel computation by concurrent scheduling of threads by the system
- Ease of programming, widespread use

POSIX Thread API

- POSIX: Portable Operating System Interface
- POSIX threads: IEEE specified standard 1003.1c-1995
- Also referred to as pthreads
- Vendors support this standard

Thread Management

- Threads can create threads
- Once created, threads are peers with no implied hierarchy or dependence
- The new thread may preempt its creator on a single processor
- All thread initialization procedures must be completed before creating the thread

Basic Routines

- *pthread_create*: create a thread
- *pthread_join*: wait for termination of thread
- *pthread_equal*: compares thread ids of two threads
- *pthread_exit*: terminates currently running thread
- *pthread_self*: returns own thread id

Example: Howdy

```
#include <pthread.h>
#include <stdio.h>
#define MAX_THREADS 8

void *start_routine(void *id)
{
   printf(”Howdy from thread #%d\n",id);
   pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
   pthread_t threads[MAX_THREADS];
   int status, i;
   for(i=0; i < MAX_THREADS; i++){
      status = pthread_create(&threads[i], NULL,
                              start_routine, (void *)i);
      if (status) {
         printf("Error creating thread \n"); exit(-1);
      }
    }
    pthread_exit(NULL);
}
```

`pthread_exit(status)`

Called to exit a thread; typically used when thread is not required to exist any more. Does not close any files!

`pthread_create(thread_id, attr,start_routine,arg)`

`attr`: object to set thread attributes, can be NULL

`start_routine`: function executed by thread upon creating

`arg`: single argument that is passed to the `start_routine`; can be NULL

This call ensures that main does not exit before the threads that it created are done. All threads created by main terminate when main exits.

Example: Howdy on Grace

```
$ module load intel/2020a
$ icc -o howdy.exe howdy.c -lpthread
$ ./howdy.exe
Howdy from thread #0
Howdy from thread #1
Howdy from thread #2
Howdy from thread #6
Howdy from thread #4
Howdy from thread #3
Howdy from thread #5
Howdy from thread #7
```

Grace Job File

- Create job file `howdy.job`
- Submit job file (submits job and returns a job id)

  `sbatch howdy.job`
- Get information about a submitted job

  `squeue --job <jobid>`
- Cancel job

  `scancel <jobid>`

Sample Job File Parameters

```
#!/bin/bash
#SBATCH --job-name=Howdy            #Set the job name to ”Howdy"
#SBATCH --time=1:30:00              #Set the wall clock limit to 1hr and 30min
#SBATCH --nodes=1                   #Request 1 node
#SBATCH --ntasks-per-node=8         #Request 8 tasks/cores per node
#SBATCH --mem=8G                    #Request 8GB per node
#SBATCH --output=output.%j          #Send stdout/err to "output.[jobID]"

module load intel/2020a
./howdy.exe
```

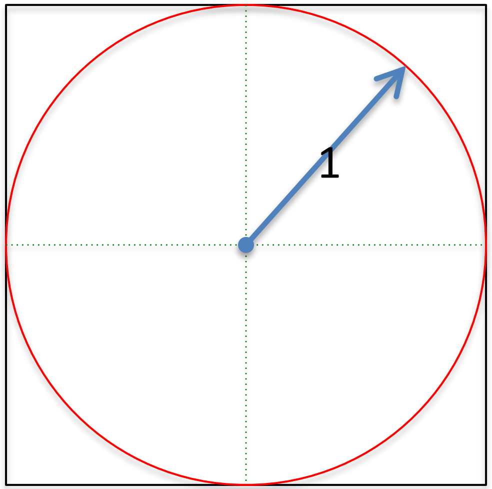
Compute π

Algorithm

- Generate random points (x,y) in a unit length square
- Keep track of points that fall within a circle of unit diameter
- Since ratio of area of circle and area of square equals $\pi/4$ the fraction of points falling within the circle approaches $\pi/4$ for large number of random trials

Implementation with Threads

- Assign fixed number of points to each thread
- Each thread generates random points and keeps track of the fraction falling within the circle
- When all threads have finished, the fractions are combined to get the value of π



Compute π …

Whether threads are created in a joinable state by default is implementation dependent. This code explicitly creates joinable threads by setting `attr`

```
#include <pthread.h>
#define MAX_THREADS     512
int total_hits, hits[MAX_THREADS], num_threads,
    sample_points, sample_points_per_thread;
int main(int argc, char *argv[]) {
  ...
  pthread_t p_threads[MAX_THREADS];
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
  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]);
  }
  pthread_attr_destroy(&attr);
  for (i=0; i< num_threads; i++) {
      pthread_join(p_threads[i], NULL);
      total_hits += hits[i];
  }
  computed_pi = 4.0*total_hits/sample_points;
}
```

Example: Compute π …

```
void *compute_pi (void *s) {
  int i, hits;
  double x, y, d;
  int *hit_pointer = (int *) s;
  unsigned int seed = *hit_pointer;
  hits = 0;
  for (i = 0; i < sample_points_per_thread; i++){
    x =(double)(rand_r(&seed))/(double)(RAND_MAX);
    y =(double)(rand_r(&seed))/(double)(RAND_MAX);
    d = (x-0.5)*(x-0.5)+(y-0.5)*(y-0.5);
    if (d < 0.25) hits ++;
    seed *= (i+1);
  }
  *hit_pointer = hits;
  pthread_exit(NULL);
}
```

`rand_r` is a thread-safe routine; drand48() is a better random number generator, but is not thread-safe

One must use thread-safe routines. Internal state of these routines is preserved correctly in a multithreaded environment.

Thread Synchronization: Locks

- *Race condition*: a situation where the result of a computation depends on a race between competing entities
  - E.g., two threads concurrently writing to a shared variable produce indeterminate result
- Mutex (*mut*ual *ex*clusion) variables are used to
  - Protect shared data in the event of concurrent writes by threads
  - Synchronize between threads
- Mutex-locks are used to provide critical sections and atomic operations
  - Only one thread can lock a mutex-lock
  - To access shared data, a thread must do the following
    - Thread tries to acquire a mutex-lock
    - Thread blocks until mutex-lock becomes available
    - Upon acquiring the mutex-lock, thread operates on data
    - Thread releases lock when finished

Mutual Exclusion for Shared Variables

- *pthread_mutex_init(&m,&attr)*: create a new mutex
- *pthread_mutex_destroy(&m)*: destroys the mutex
- *pthread_mutex_lock(&m)*:
  - Attempt to acquire the mutex *m*
  - Block until mutex *m* becomes available
  - Upon return from the call, mutex *m* has been acquired
- *pthread_mutex_trylock(&m)*:
  - Attempt to acquire the mutex *m*
  - Return immediately
  - Has acquired the mutex *m* only if successful
- *pthread_mutex_unlock(&m)*: release the mutex

m: mutex

attr: mutex attributes, NULL sets default attributes

Types of Locks

POSIX API supports three different kinds of locks:

- *Normal Mutex*: Only a single thread is allowed to lock a normal mutex once at any point of time. If the same thread attempts to lock the mutex, it results in a deadlock.
- *Recursive Mutex*: A recursive mutex allows a single thread to lock a mutex multiple times. Each time a thread locks the mutex, a lock counter is incremented. Each unlock decrements the counter. Recursive mutex is useful when a thread function needs to call itself recursively. For any other thread to be able to successfully lock a recursive mutex, the lock counter must be zero
- *Errorcheck Mutex*: Unlike normal mutex, when a thread attempts a lock on a errorcheck mutex it has already locked, instead of dead-locking, it returns an error

Example: Minimum of List Using Mutex

- Partition list equally among the threads
- Each thread computes minimum of its sublist, then updates global minimum value
- Access to the global minimum variable is protected by a mutex
- To update the global minimum:
  - A thread executes *pthread_mutex_lock* to gain exclusive access to the mutex
  - The thread blocks if the mutex is not available
  - When the mutex becomes available, the thread returns from the call after having acquired the mutex
  - The thread updates the global minimum variable
  - The thread executes *pthread_mutex_unlock* to release the mutex

Minimum of List …

```
pthread_mutex_t minimum_value_lock;
int minimum_value;
...
void *find_min(void *list_ptr) {
 partial_list_pointer = (int *) list_ptr;
 for (i = 0; i < partial_list_size; i++) {
   if (partial_list_pointer[i] < my_min)
      my_min = partial_list_pointer[i];
   }
 pthread_mutex_lock(&minimum_value_lock);
   if (my_min < minimum_value)
       minimum_value = my_min;
 pthread_mutex_unlock(&minimum_value_lock);
 pthread_exit(0);
}
main() {
/* initialize data structures and list */
 pthread_init();
 pthread_mutex_init(&minimum_value_lock, NULL);
/* create and join threads here */
}
```

Thread must

- acquire mutex before it updates `minimum_value`
- Release mutex after updating `minimum_value`

Assumes that `minimum_value` has been initialized correctly with a value from the list

Discussion – minimum of list

```
void *find_min(void *list_ptr) {
 partial_list_pointer = (int *) list_ptr;
 for (i = 0; i < partial_list_size; i++) {
   if (partial_list_pointer[i] < my_min)
      my_min = partial_list_pointer[i];
   pthread_mutex_lock(&minimum_value_lock);
     if (my_min < minimum_value)
       minimum_value = my_min;
   pthread_mutex_unlock(&minimum_value_lock);
 }
 pthread_exit(0);
}
```

- What do you expect when a large number of processors are used?
- How can the execution time be reduced?

Discussion – minimum of list …

```
void *find_min(void *list_ptr) {
 partial_list_pointer = (int *) list_ptr;
 for (i = 0; i < partial_list_size; i++)
   if (partial_list_pointer[i] < my_min) {
     my_min = partial_list_pointer[i];
     pthread_mutex_lock(&minimum_value_lock);
       if (my_min < minimum_value)
         minimum_value = my_min;
     pthread_mutex_unlock(&minimum_value_lock);
   }
 pthread_exit(0);
}
```

- What happens if the values in the list are in decreasing order?
- How can the execution time be reduced?
- Does mutex provide the ideal mechanism for accessing `minimum_value`?

Example: Producer Consumer Using Mutex

- Producer thread creates tasks and inserts them into a task queue
- Consumer threads pick up tasks from the task queue and accomplish them
- The producer thread must not overwrite shared data structure when the previous task has not been picked up by consumer threads
- The consumer threads must not pick up tasks until there is something present in the shared data structure
- Only one consumer thread should pick up a task
- Use variable called `task_available` to signal availability of task
- All operations on `task_available` should be protected by mutexes to ensure exclusive reads and writes

Producer Consumer Using Mutex …

```
pthread_mutex_t task_queue_lock;
int task_available;
main() {
  task_available = 0;
  pthread_init();
  pthread_mutex_init(&task_queue_lock, NULL);
  /* create & join producer & consumer threads */
  pthread_mutex_destroy(&task_queue_lock
}
```

Producer Consumer Using Mutex …

```
void *consumer(void *consumer_thread_data) {
  int extracted;
  while (!done()) {
    extracted = 0;
    while (extracted == 0) {
      pthread_mutex_lock(&task_queue_lock);
      if (task_available == 1) {
        my_task = extract_from_queue();
        extracted = 1;
        task_available = 0;
      }
      pthread_mutex_unlock(&task_queue_lock);
    }
    process_task(my_task);
  }
}
```

```
void *producer(void *producer_thread_data) {
  int inserted;
  while (!done()) {
    inserted = 0;
    create_task();
    while (inserted == 0) {
      pthread_mutex_lock(&task_queue_lock);
      if (task_available == 0) {
        insert_into_queue();
        inserted = 1;
        task_available = 1;
      }
      pthread_mutex_unlock(&task_queue_lock);
    }
  }
}
```

Thread Synchronization: Condition Variables

- Mutex locks are not ideal for synchronization due to idling overhead from blocked threads
- *Condition variable* is a data object for synchronizing threads
- Condition variable allows a thread to block itself until a specified data reaches a predefined state. When data reaches predefined state, condition variable signals one or more waiting threads.

Condition Variables for Synchronization

- *pthread_cond_init(&v,&attr)*: creates a new condition variable
- *pthread_cond_destroy(&v)*: destroys the condition variable
- *pthread_cond_wait(&v,&m)*:
  - Called after the thread has acquired mutex *m* and has tested that the condition is not satisfied, i.e., predicate is not true
  - Atomically blocks the thread and releases mutex *m*
  - Returns from the call when another thread signals *v*
  - Upon return, the thread has reacquired the mutex *m*, but should re-test the predicate
- *pthread_cond_timedwait(&v,&m,&t)*: same as *pthread_cond_wait* except the thread is blocked until time *t* unless the condition variable signal occurs first
- *pthread_cond_signal(&v)*: unblocks at least one waiting thread
- *pthread_cond_broadcast(&v)*: unblocks all waiting threads

v: condition variable, *m*: mutex, *t*: time (not duration)

attr: condition variable attributes, NULL sets default attributes

Using Condition Variables – Good Practices

1. Acquire the mutex before testing the predicate.
2. Call *pthread_cond_wait* if the predicate is false.
3. Retest the predicate after returning from *pthread_cond_wait* since the return may be from a *pthread_cond_signal* that did not cause the predicate to become true.
4. Acquire the mutex before changing any of the variables appearing in the predicate; this is done implicitly if thread returns from a *pthread_cond_wait*
5. Hold the mutex only for a short period of time, usually for testing the predicate or modifying shared variables
6. Release the mutex explicity by *pthread_mutex_unlock* or implicitly by *pthread_cond_wait*

See `producer` and `consumer` routines in subsequent slides on "Producer Consumer Using Condition Variables"

Example: Producer Consumer via 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 can use two condition variables `cond_queue_empty` and `cond_queue_full` for specifying empty and full queues respectively.
- The condition associated with `cond_queue_empty` is asserted when the variable `task_available` becomes 0 and `cond_queue_full` is asserted when `task_available` becomes 1.
- Producer locks mutex `task_queue_cond_lock` associated with the shared variable `task_available` and checks to see if `task_available` is 0 (i.e. queue is empty). If so, producer inserts task into task-queue and signals any waiting consumer threads to wake up using condition variable `cond_queue_full` . If not so, (i.e. queue is full), the producer waits for queue to become empty by performing a condition wait on the condition variable `cond_queue_empty`
- 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` (Eventually, this signal is generated from producer). If task is available, the consumer takes it off the work queue and signals the producer.

Producer Consumer …

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
main() {
  task_available = 0;
  pthread_init();
  pthread_cond_init(&cond_queue_empty, NULL);
  pthread_cond_init(&cond_queue_full, NULL);
  pthread_mutex_init(&task_queue_cond_lock, NULL);
/* create & join producer & consumer threads */
}
```

Producer Consumer …

```
void *producer(void *producer_thread_data) {
  while (!done()) {
    create_task();
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 1)
      pthread_cond_wait(&cond_queue_empty,
        &task_queue_cond_lock);
    insert_into_queue(); task_available = 1;
    pthread_cond_signal(&cond_queue_full);
    pthread_mutex_unlock(
         &task_queue_cond_lock);
  }
}
```

```
void *consumer(void *consumer_thread_data) {
  while (!done()) {
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 0)
        pthread_cond_wait(&cond_queue_full,
          &task_queue_cond_lock);
    my_task = extract_from_queue();
    task_available = 0;
    pthread_cond_signal(&cond_queue_empty);
    pthread_mutex_unlock(
         &task_queue_cond_lock);
    process_task(my_task);
}
```

Example: Constructing Barrier

- We will construct a logarithmic barrier for $n=2^k$ threads using n-1 condition variable-mutex pairs
- At the first level, thread 2i is responsible for assuring synchronization with thread 2i+1 (i=0,...,(n-1)/2) using a condition variable-mutex pair
- At the second level, thread 4i is responsible for assuring synchronization with thread 4i+2 (i=0,...,(n-1)/4) using a condition variable-mutex pair
- This process continues for k levels, where only two threads 0 and n/2 need to synchronize
- Releasing the threads requires the above process in reverse using condition variables to signal the other thread in each pair that synchronized at each level

Barrier …

```
typedef struct barrier_node {
        pthread_mutex_t count_lock;
        pthread_cond_t sync;
        pthread_cond_t release;
        int count;
} mylib_barrier_t_internal;
typedef struct barrier_node
               mylog_logbarrier_t[MAX_THREADS];
pthread_t p_threads[MAX_THREADS];
pthread_attr_t attr;
void mylib_init_barrier(mylog_logbarrier_t b) {
  int i;
  for (i = 0; i < MAX_THREADS; i++) {
    b[i].count = 0;
    pthread_mutex_init(&(b[i].count_lock), NULL);
    pthread_cond_init(&(b[i].sync), NULL);
    pthread_cond_init(&(b[i].release), NULL);
  }
}
```

Barrier …

```
i = 2; base = 0;
do { /* Synchronizing phase */
  index = base + thread_id / i;
  if (thread_id % i == 0) {
    pthread_mutex_lock(&(b[index].count_lock));
    b[index].count ++;
    while (b[index].count < 2)
      pthread_cond_wait(
        &(b[index].sync),&(b[index].count_lock));
    pthread_mutex_unlock(&(b[index].count_lock)); }
  else {
    pthread_mutex_lock(&(b[index].count_lock));
    b[index].count ++;
    if (b[index].count == 2)
      pthread_cond_signal(&(b[index].sync));
    while (pthread_cond_wait(&(b[index].release),
                &(b[index].count_lock)) != 0);
    pthread_mutex_unlock(&(b[index].count_lock));
    break;}
  base = base + num_threads/i; i = i * 2;
} while (i <= num_threads);
```

```
i = i / 2;
/* Releasing phase */
 for (; i > 1; i = i / 2) {
   base = base - num_threads/i;
   index = base + thread_id / i;
   pthread_mutex_lock(
        &(b[index].count_lock));
   b[index].count = 0;
   pthread_cond_signal(
        &(b[index].release));
   pthread_mutex_unlock(
        &(b[index].count_lock));
 }
}
```

Thread Scheduling

Scheduling Routines

- *pthread_getschedparam*: gets the current scheduling policy and priority of thread
- *pthread_setschedparam*: sets the current scheduling policy and priority of thread

Scheduling Policies

- SCHED_FIFO (first-in/first-out): Thread with the highest priority executes until it is blocked or another thread with a higher priority becomes ready. In the latter case, the currently running thread is pre-empted and the new thread is scheduled. If multiple threads with highest identical priorities exist, then the first thread continues execution until it blocks
- SCHED_RR (round-robin): Identical to SCHED_FIFO, except, when multiple threads exist with identical highest priority, they are time-sliced.
- SCHED_FG_NP (default): Threads are time-sliced irrespective of their priority. Higher priority threads still get more execution time than lower priority threads. However, if SCHED_FIFO or SCHED_RR threads exist in the system, they may starve SCHED_FG_NP threads.
- SCHED_BG_NP (Background): Similar to SCHED_FG_NP, however, threads scheduled as SCHED_BG_NP get less execution time in the presence of SCHED_FIFO or SCHED_RR threads than SCHED_FG_NP.
- Priority of a thread can be specified by programmer.

Other Thread Routines

Thread Attribute Routines

- Thread attributes are set via these routines.
  - *pthread_attr_init*: creates attribute variable
  - *pthread_attr_destroy*: destroys attribute variable
  - *pthread_attr_getstacksize*
  - *pthread_attr_setstacksize*
  - *pthread_attr_getdetachstate*
  - *pthread_attr_setdetachstate*
- Detach state of thread determines whether a thread is joinable or not.

IO Routines

- *flockfile*
- *ftrylockfile*
- *funlockfile*
- *getc_unlocked*
- *getchar_unlocked*
- *putc_unlocked*
- *putchar_unlocked*

Other Important Issues

Thread-safeness

- Shared data accessed by different threads concurrently can cause race conditions
- Library routines need to be carefully designed to ensure internal data is preserved correctly in a multithreaded environment

Thread limits

- Some limits may be implementation dependent which limits portability
  - E.g., thread stack size and maximum number of threads